وادي التقنية

تأليف
ديف توماس اندي هانت

المبرمج العملي

رحلتك إلى الإتقان

ترجمة
علي هلال عيسى

إشراف و مراجعة
زايد السعيدي

المبرمج العملي

رحلتك إلى ا لإتقان

تَرْجَمَة

علي هلال عيسى

الإعداد والإشراف

زايد بن عامر السعيدي

أصل هذا العمل تَرْجَمَة متصرفة لكتاب The Pragmatic Programmer الطبعة الثانية، تأليف ديف توماس و اندي هانت، ترجمه إلى العربية موقع وادي التقنية وإشراف عام ومراجعة زايد السعيدي.

ثناء للطبعة الثانية من كتاب المبرمج العملي

يقول البعض مع كتاب المبرمج العملي، جمعَ آندي وديف البرقَ في زجاجة. وأنّه من غير المحتمل أن يكتب أيّ شخص قريبًا كتابًا يمكنه تحريك الصناعة بِزُمَّتها كما فعل. مع ذلك، يلمعُ البرق مرّتين، وهذا الكتاب دليل على ذلك. يضمن المحتوى المحدّث أن يبقى على قمة قوائم "أفضل الكتب في تطوير البرمجيات" لمدة 20 عامًا أخرى، تمامًا حيث ينتمي.

← **(فيكي) براسور**

مدير إستراتيجية المصادر المفتوحة بشبكات Juniper

إذا كنت تريد لبرنامَجك أن يكون سهل التحديث والصيانة، فاحتفظ بنسخةٍ من المبرمج العملي قريبًا منك. إنها مليئة بالمشورة العملية، سواء التقنية أم المهنية، التي من شانها أن تخدمك أنت ومشروعاتك خدمةً جيدةً لسنوات قادمة.

← **أندريا جوليه**

المدير التنفيذي لشركة Corgibytes ؛ مؤسس LegacyCode.Rocks

المبرمج العملي هو الكتاب الوحيد الذي يمكنني أن أشير إليه على أنه غيّر تمامًا المسار الحالي لمسيرتي في مجال البرمجيات وأرشدني نحو النجاح. لقد فتحت قراءته ذهني لاحتمالات أن أكون حِرفيًا، وليس مجرّد ترسٍ في آلةٍ كبيرة. إنه واحد من أكثر الكتب أهميّة في حياتي.

← **اوبى ميديا فرنانديز**

مؤلف، السكك الحديد

يمكن للقرّاء للمرّة الأولى أن يتطلعوا إلى استقراء آسر للعالم الحديث لممارسة البرمجيات، وهو العالم الذي لعبت الطبعة الأولى دورًا رئيساً في تشكيله. سيُعيد قرّاء الإصدار الأول اكتشاف الرؤى والحكمة العمليّة التي جعلت من الكتاب ذا تأثيرٍ كبيرٍ في المقام الأول، مُنسَقاً ومُحدّثا بخبرة، إلى جانب الكثير من الميزات الجديدة.

← **ديفيد أ. بلاك**

مؤلّف مبرمج لغة روبي المتمكّن

لدي نسخة ورقيّة قديمة من المبرمج العملي الأصلي على رف الكتب الخاص بي لقد قراءتها مرّة تلو أخرى، ومنذ وقت طويل مضى غيّر هذا الكتاب كل شيء عن كيفيّة تعاملي مع وظيفتي كمبرمج. في الطبعة الجديدة، تغيّر كل شيء ولم يتغير أي شيء: أقرأه الآن على جهازي الأي باد وأمثلة الشِفرة البرمجيّة تَستخدِم لغات البرمجة الحديثة - ولكن المفاهيم المضمّنة والأفكار والطرائق الأساسيّة غير محدودة بالزمن وقابلة للتطبيق عالميًا. وبعد مرور عشرين عامًا، أصبح الكتاب وثيق الصّلة بي كما كان دائمًا. يسعدني أن أعرف أن المطوّرين الحاليين والمستقبليين سيحصلون على نفس الفرصة للتعلم من رؤى آندي وديف العميقة كما فعلت ذات يوم.

← **ساندي مامولي**

مدرب منهجية التطوير الرشيق "أجايل"، مؤلّف كتاب كيف يتيح الاختيار الذاتي للنّاس التفوق

قبل عشرين عامًا، غيّرت النسخة الأولى من المبرمج العملي مسار مسيرتي المهنية تمامًا. هذا الإصدار الجديد يمكن أن يفعل الشيء نفسه بالنسبة لك.

← **مايك كوهن**

مؤلف النجاح في منهجية التطوير الرشيق" أجايل"، الاستنتاج والتخطيط في "أجايل"، وقصص المستخدم العملية.

لجولييت وإيلي

وزكاري وإليزابيث

وهنري وستيوارت

جدول المحتويات

تقديم محرّر الترجمة العربية

نحمد الله حق حمده، وبه نستعين على جميع أمورنا، فلك اللهم الحق الحمد على كل النعم التي أسبلتها علينا، وصلى الله وسلم على سيد المرسلين و خاتم النبيين محمد صلى الله عليه وسلم، أما بعد فيسعدنا في وادي التقنية أن نخرج هذا الكتاب المختص في أساليب تطوير البرمجيات، كتاب المبرمج العملي كسب قَبُول واسع وثناء كبير من جَمهور المبرمجين، وذلك لابتعاده عن الحشو و سهولة مأخذه في الشرح.

كتاب المبرمج العملي هو الكتاب الثاني الذي نترجمه إلى العربية بعد كتاب الشفرة الكاملة الذي بدوره حضي بقبول جيد في أوساط المبرمجين العرب، وبهذين الكتابين نكون قدمنا ذخيرة جيده للمبرمجين على مختلف لغاتهم في أساليب تطوير البرمجيات.

في هذا الكتاب بذلنا جهدنا في إخراجه بلغة عربية سليمة، ومصطلحات دقيقة قدر الجهد والوقت المتاح، لا ندعي الكمال، وهناك تقصير، لذا نشرنا نسخة قابلة للتعديل ليتمكن من يكتشف هذه الأخطاء تصحيحها و نشر الكتاب من جديد.

وختاما أود أن أشكر المهندس علي هلال عيسى على ترجمته هذا الكتاب بصبر محافظا على جدول التسليم بالرغم من زهدة التعويض، و أشكر أخي فهد السعيدي تصميمه الغلاف الرائع و قيامه بموقع وادي التقنية، وأشكر الدكتور طه زروقي و الدكتور صهيب عفيفي على دعم اللغة العربية في برنامَج المدقق الإملائي و النحوي languagetool، و أشكر كل من ساهم في موقع وادي التقنية.

أخوكم زايد السعيدي

مسقط، سلطنة عمان

غرة ربيع الأول سنة 1443هجري

18 من أكتوبر 2020م.

تقديم المترجم

"رحلتك نحو الإتقان" هذه الكلمات البسيطة المسطرة أسفل هذا العنوان الكبير "المبرمج العملي"، تعبّر تعبـيرًا صـادقًا عن فحواه ومضمونه. فهو الذي سيكون رفيق دربك في رحلتك لتصبح مبرمجًا أفضل، وربما شخصًا أفضل!

مُقدّمًا لك بمحتـوىً عصـريّ مُحـدّث، أفكـارًا جديـدة وإرشـادات علميّـة وأمثلـة واقعيّـة تسـاعدك على ضـبط بوصـلتك في الاتجـاه الصحيح.

حاولنا خلال التَرْجَمَة جاهدين إيصال المعنى سليمًا واضحًا، والحفاظ على أسلوب المؤلف قدر المستطاع. ونظرًا لاحتـواء الكتـاب على كمّ كبيرٍ من المصطلحات فقد عمدنا إلى إنشاء مسردٍ للمصطلحات خاص بـه، للمسـاعدة على فهم المعـنى المقصـود فهمًـا صـحيحًا، واعتمــدنا في ذلــك بشــكل رئيس على معجمــات وقــواميس لغويّــة متخصّصــة، مثــل قــاموس عربــآيز، وقــاموس المعــاني، و wordreference، وغيرهم، وعملنا على ذكر المصطلح بالإنجليزية عند وروده للمرة الأولى في النص أو حيث وجدنا ذلك ضروريًا.

المؤلفان غزيريّ المعرفة وواسعي الثقافة وقد طرحا بين طيات هذا الكتاب كثـيرًا من المواضـيع والأفكـار الـتي قـد تتطلب بحثًـا إضافيًا من القارئ الجديد لاستكشافها وللاستيضـاح عنهـا، لـذا أضـفنا هـوامشٌ أسـفل النص لشـرح هـذه المفـاهيم بشـكل مبسّـط تجنبًا لتشتيت القارئ، وسبقناها بالوسم "[المترجم]" تمييزًا لها عن هوامش الكتاب الأصليّة. كما أضفنا بعضًا من الروابـط الإضـافيّة للتوسع، حيث وجدنا ذلك مناسبًا.

في النهاية إنه لشرفٌ كبيرٌ لي أن أحظى بتَرْجَمَة واحـدٍ من أهم الكتب في عـالم البرمجـة، وهـو الكتـاب الثـاني الـذي أترجمـه بعـد مرجع الشِفرة الكاملة "Code Complete - 2nd Edition" وما كنت لأحظى بهذا الشرف لولا دعم وإشراف موقع وادي التقنية، هذا الموقع الذي لا يدّخر جهدًا في سبيل دعم المحتوى النوعي التقني العربي على الإنترنت والمساهمة في نشره وتطويره.

كل الشكر لإدارة الموقع وأخص بالذكر الأستاذ العزيز: زايد بن عامر السعيدي على رعايته وإعداده وإشرافه على هـذا الكتـاب، كما لا يفوتني أن أشكر أخوتي على دعمهم ومساعدتهم لي في تدقيق هنـا أو تصـويبٍ هنـاك (الـدكتور مهنـد والمهندسـة مي والأسـتاذ علاء والمهندس محمد)، أطيب الأمنيات لكم جميعًا.

علي هلال عيسى[1]

موسكو – جمهورية روسيا الاتحادية

الخميس: 14 صفر 1442 هـ

1 أكتوبر 2020 م

1 خريج ماجستير علوم الويب (MWS)ـ الجامعة الافتراضية السـورية- دمشـق، طـالب دراسـات عليـا في معهـد الهندسـة الصـناعية وتكنولوجيـا المعلومات والميكاترونكس- موسكو. www.ali-i.com

تمهيد

أتذكّر عندما غرّد ديف وأندي لأول مرّة عن الطبعة الجديدة من هذا الكتاب. كان خبرًا مهمًا. شعرتُ برضىً متزايد وأنا أراقب مجتمع كتابة الشِفرة يستجيب بحماسة. بعد عشرين عامًا ما زال "المبرمج العملي" ساري المفعول الآن كما كان في السابق.

كتابٌ لديه تاريخ كهذا يعني الكثير. لقد حصلت على امتيازِ قراءة نسخةٍ غير محرّرة من أجل كتابة هذه المقدّمة، وفهمت لماذا أحدث مثل هذه الضجّة. مع أنّه كتاب تقني، تسميته كذلك تسيء إليه. الكتب التقنية عادةً مهولة، إنها مكتظة بالكلمات الكبيرة، والصيغ الغامضة والأمثلة المعقّدة التي تشعرك بالغباء. كلما كان الكاتب أكثر خبرةً، كان من الأسهل عليه نسيان ماذا يعني تعلّم مفاهيم جديدة، ومعنى أن تكون مبتدئًا.

مع عقودٍ من الخبرة في البرمجة، تغلّب ديف واندي على التحدي الصعب بالكتابة بنفس متعة أناس تعلّموا للتو هذه الدروس. لا يتكلمان معك باستعلاء، لا يعدّانك خبيرًا، حتى أنهما لا يضعان في الاعتبار أنك قرأت الإصدار الأول. يأخذان بيدك كما أنت - كمبرمجٍ كل ما يريده أن يصبح أفضل. أمضيا صفحات هذا الكتاب يساعدانك لتصبح هناك، خطوةً قابلةً للتنفيذ في كل مرّة.

لأكون مُنصفًا، لقد قاما فعلًا بهذا مُسبقًا. كان الإصدار الأصلي مليئاً بالأمثلة الملموسة، والأفكار الجديدة والإرشادات العمليّة لبناء عضلات كتابة الشِفرة وتطوير ذهنيّة كتابة شِفرةٍ تبقي مُطبَقةً لليوم. لكن هذا الإصدار المحدّث قدم تطويرين على هذا الكتاب:

الأول هو الأوضح، حذفَ بعض المراجع القديمة، والأمثلة القديمة، واستبدلهم بمحتويات حديثة وعصريّة. لن تجد أمثلة عن ثوابت الحلَقات أو آليات البناء. لقد أخذ ديف وأندي محتواهم القوي، وتأكّدوا من أنّ الدروس تؤدي عملها، وخالية من انحرافات الأمثلة القديمة. إنها تمسح الغبار عن الأفكار القديمة مثل "DRY" (لا تكرر نفسك) وتهبها حلّة جديد تجعلها مُتألّقةً حقًّا.

لكن التطوير الثاني هو ما يجعل هذا الإصدار مدهشًا حقًا. بعد كتابة الطبعة الأولى، أتيحت لهم الفرصة للتفكير في ما كانوا يحاولون قوله، وما أرادوا من قُرّائهم استبعاده، وكيف تم تلقيه. لقد حصلوا على ملاحظات على تلك الدروس. ورأوا العالق، وما يحتاج إلى تحسين، وما الذي أُسيء فهمه. خلال العشرين عامًا التي شقّ فيها هذا الكتاب طريقه عبر ألاف أيدي وقلوب المبرمجين حول العالم، قد درس ديف واندي هذه الاستجابة وصاغوا أفكارًا جديدة، ومفاهيم جديدة.

لقد تعلموا أهميّة الوكالة وأدركوا أن لدى المطوّرين وكالة أكثر من معظم المحترفين الآخرين. لقد بدؤوا الكتاب برسالة بسيطة ولكنها عميقة: "إنّها حياتك" إنّه يذكرنا بقوتنا الخاصّة في قاعدة شِفراتنا الخاصّة، في وظائفنا، في مهننا. فهو يحدّد الإيقاع لكل شيء آخر في الكتاب، أنه أكثر من مجرّد كتاب تقني آخر مملوء بأمثلة الشِفرات البرمجيّة.

إنّ ما يجعله يبرز حقا بين رفوف الكتب التقنية هو أنّه يفهم ما يعنيه أن تكون مبرمجًا. البرمجة هي محاولة لجعل المستقبل أقلّ إيلامًا. الأمر يتعلق بتسهيل الأمور على زملائنا في الفريق. ارتكاب الأخطاء والقدرة على التراجع. تكوين عادات جيّدة. فهم مجموعة أدواتك. كتابة الشِفرة هي مجرّد جزء من عالم كونك مبرمجًا، وهذا الكتاب يستكشف هذا العالم.

أقضي الكثير من الوقت بالتفكير في رحلة كتابة الشِفرة. لم أنشأ على كتابة الشِفرة، لم أدرسها في الجامعة. لم أصرف سنين مراهقتي عابثًا مع التقنية. دخلت عالم كتابة الشِفرة في منتصف العشرينيات، وكان عليّ تعلّم معنى أن تكون مبرمجًا. يختلف هذا المجتمع جدًا عمّا اعتدت أن أكون جزءاً منه. هناك تفاني فريد للتعلم والتطبيق العملي منعش ومهول على حد سواء.

بالنسبة إلي، يبدو حقًّا كدخول عالم جديد. مدينة جديدة في الأقل. كان عليّ أن أتعرف إلى الجيران، وأختار متجر البقالة الخاص بي، وأجد أفضل المقاهي. استغرق الأمر بعض الوقت للاستقرار، للعثور على الطرق الأكثر كفاءة، لتجنب الشوارع المزدحمة، لمعرفة متى من المرجح أن تزدحم حركة المرور. الطقس مختلف، كنتُ بحاجةٍ إلى خزانة ملابس جديدة.

الأسابيع القليلة الأولى، حتى الأشهر، في بلدة جديدة يمكن أن تكون مخيفة. ألن يكونَ رائعًا أن يكون لديك جارًا ودودا ومطّلعا كان يعيش هناك لمدّة؟ بإمكانه أخذك في جولة، يريك فيها هذه المقاهي؟ شخص ما كان موجودًا هناك لمدّة كافية ليعرف الثقافة، وليفهَمَ نبض المدينة. بذلك لا تشعر فقط أنك في منزلك، بل تصبح عضوًا مساهمًا أيضًا؟ ديف وآندي هما هذان الجاران.

كوافدٍ جديدٍ نسبيًا، من السهل بما كان أن تشعر بالإرباك، ليس فقط بفعل البرمجة بل بعملية أن تصبح مبرمجًا. هناك تحوّل كامل في العقليّة يجب أن تحدث، تغيّر في العادات والسلوكيات والتوقعات. لا تحصل عملية أن تصبح مبرمجًا أفضل فقط لأنك تعرف كيف تكتب الشِفرة؛ يجب أن تُقابَل بالنيّة والممارسة المتعمّدة. هذا الكتاب هو دليل لتصبح مبرمجًا أفضل بكفاءة.

لكن لا تخطئ، إنه لا يخبرك كيف ينبغي أن تكون البرمجة. إنه ليس فلسفي أو جدلي على تلك الطريقة. إنه يخبرك، بوضوحٍ وبساطة، ما المبرمج العملي - كيف يتصرف، وكيف تتم مقاربة الشِفرة. يترك الأمر لك لتقرّر ما إذا كنت تريد أن تكون واحدا منهم. إذا شعرت أنه ليس لك، فلن يقفوا في طريقك. لكن إذا قرّرت ذلك، فجاراك الودودان هناك ليُرشدانك إلى الطريق.

سارون يتبريك

المؤسس والرئيس التنفيذي لـ

CodeNewbie Host of Command Line Heroes

تمهيد للإصدار الثاني

في التسعينيات، عملنا مع شركات كانت مشروعاتها تواجه مشكلات. وجدنا أنفسنا نقول الأشياء نفسها لكل منها: ربما عليك اختباره قبل شرائه، لماذا تُبنى الشِفرة على "آلة ماري" حصرًا؟ لماذا لا يسأل أحدٌ المستخدمين؟

بدأنا تدوين الملاحظات لتوفير الوقت مع العملاء الجدد. وأصبحت هذه الملاحظات لاحقًا "المبرمج العملي". لدهشتنا يبدو أن الكتاب ضرب على الوتر، واستمر شعبيًا طيلةَ هذه السنوات العشرين الماضية.

لكن عشرون عامًا مدّة حياةٍ طويلة بعمر البرمجيات. خذ مطوّرا من عام 1999 وضمهُ مع فريق اليوم، وسيصارع في هذا العالم الجديد الغريب. لكن عالم التسعينيات غريبٌ بنفس القدر بالنسبة لمطوّر اليوم. كانت إشارات الكتاب إلى أشياء مثل CORBA وأدوات CASE والحلقات المفهرسة غريبةً في أحسن الأحوال وأكثر إثارة للحيرة.

في الوقت نفسه، لم يكن لعشرين عامًا أيّ تأثيرٍ على الإطلاق على الحسّ العام. ربما تغيرت التِقَانَة، لكن النّاس لم يتغيروا. التمارين والنُهج التي كانت جيّدة بقيت جيّدة للآن. عمّرت هذه الجوانب من الكتاب طويلًا.

لذا كان علينا اتخاذ قرار عندما حان وقت إصدار طبعة الذكرى العشرين. كان باستطاعتنا أن نمرّ عليه ونعدّل التقنيات التي أشرنا إليها ونوقف العمل. أو إعادة النظر بالافتراضات الكامنة وراء التمارين التي أوصينا بها في ضوء خبرةٍ إضافيّة بقيمة عقدين من الزمن.

في النهاية، قمنا بكلا الأمرين.

نتيجةً لذلك، هذا الكتاب إلى حد ما سفينة سيزيوس[2]، تُصنف ثلث المواضيع تقريباً في الكتاب جديدة. البقيّة، الأغلبية أُعيدت كتابتها، جزئيًا أو كليًا. نيّتنا كانت جعل الأمور أوضح، وأكثرَ أهميّةً، ونأمل كونها غير محدودة بالزمن.

2 إذا غيرت كل مكونات السفينة عند تعطلها عبر السنين، فهل المركبة الناتجة هي نفس السفينة؟

لقد اتخذنا بعض القرارات الصعبة. لقد أسقطنا ملحق المصادر، لأنه سيكون من المستحيل مواكبة المستجدات ولأنه من الأسهل البحث عما تريد. نظّمنا وأعدنا كتابة المواضيع لنتعامل مع التساير (Concurrency)، نظرًا للوفرة الحاليّة من الأجهزة الموازية وندرة الطرق الجيّدة للتعامل معها. أضفنا محتوى ليظهر المواقف والبيئات المُتغيّرة، بدءًا من حركة التطوير السريع التي ساعدنا بطرحها، إلى القبول المتزايد لصيغ البرمجة الوظائفية والحاجة المتنامية لاعتبارات الخصوصيّة والأمن.

من المثير للاهتمام، كان النقاش بيننا أقل بكثير حول محتوى هذه الطبعة مما كان عليه عندما كتبنا الأولى. شعرَ كلانا أن الأشياء التي كانت مهمّة كانت أسهل في التحديد.

على أي حال، هذا الكتاب هو النتيجة. فضلًا، تمتع به. ربما تتبنى بعض الممارسات الجديدة. ربما تقرّر أن بعض الأشياء التي اقترحناها خاطئة. خذ دورك. أخبرنا عن ملاحظاتك.

ولكن، الأهم من ذلك، تذكّر أن تجعله ممتعًا.

كيف نُظّم هذا الكتاب

كُتب هذا الكتاب كمجموعة من المواضيع القصيرة. كل موضوع قائم بذاته، ويتناول موضوعًا معينًا. ستجد العديد من المراجع المتقاطعة التي تساعد على وضع كل موضوع في السياق. لا تتردد في قراءة الموضوعات بأي ترتيب - هذا ليس كتابًا تحتاج إلى قراءته من الأول إلى الأخير.

في بعض الأحيان سوف تصادف صندوق معنون ب نصيحة **ن** (مثل نصيحة 1- "اهتم بمهنتك" في الصفحة xxi) بالإضافة إلى التأكيد على النِّقَاط الواردة في النص، نشعر أن النصائح لها حياة خاصّة بها - فنحن نتعامل معها يوميًا.

لقد ضمّنا التمارين والتحديات عند الاقتضاء. عادةً ما تكون إجابات التمارين سهلة نسبيًا، في حين أن التحديات ذات إجابات مفتوحة. لإعطائك فكرةً عن تفكيرنا، ضمّنا إجاباتنا على التمارين في ملحق، ولكن القليل جدًا منها لديها حل واحد صحيح. قد تصلح التحدّيات لتكون أساساً لمناقشات جماعية أو مقالات في دورات البرمجة المتقدمة.

هناك أيضًا فِهرِس قصير يسرد الكتب والمقالات التي نشير إليها صراحة.

ما في الاسم

"عندما أستخدم كلمة"، قالها هامبتي دامبتي في لهجة ازدراء، "إنها تعني فقط ما أختار أن أعنيه لا أكثر ولا أقل ".

➤ لويس كارول، *عبر النظارة* (Through the Looking-Glass)

ستجد العديد من المقاطع الاصطلاحيّة الدارجة المختلفة تمامًا، كلمات انجليزيّة جيّدة حُرّفت لتعني شيئا ما تقنيًا، أو كلمات مُصاغة بطريقة مريعة حوّلت معانيها من قبل علماء الحاسوب مع ضغينةٍ ضدّ اللغة، مبعثرةً في جميع أنحاء الكتاب. سنحاول في الاستخدام الأول لهذه الكلمات الاصطلاحية تحديدها، أو في الأقل الإشارة لمعانيها. ومع ذلك، نحن متيقِّنون من أن بعضها قد سقط سهواً، والبعض الآخر، مثل **الكائن** و**قاعدة البيانات العلائقية** هي في الاستخدام الشائع بما فيه الكفاية بحيث أن إضافة تعريفٍ لها سيكون مُملًا. إذا مررت فعلاً بمصطلح لم تسمع عنه مسبقًا، لا تتخطّاه رجاءً. خذ وقتاً للبحث عنه، ربما على الشبكة، أو في كتاب تعليمي مختص بالحاسوب. وإذا سنحت لك الفرصة، اترك لنا رسالة إلكترونية واشتكي، وبذا يكون بإمكاننا إضافة تعريف للطبعة القادمة.

بناءً على ما قلنا، فقد قرّرنا الانتقام من علماء الحاسوب. في بعض الأحيان، هنالك كلمات اصطلاحيّة جيّدة تمامًا للمفاهيم، قد قرّرنا تجاهلها. لمَ؟ لأن المصطلح الموجود عادةً ما يقتصر على نطاق مشكلة محدّدة، أو مرحلة محدّدة من التطور. على أي حال، فإن واحدة من الفلسفات الأساسية لهذا الكتاب هو أن معظم التقنيات التي نوصي بها عالمية: تُطبّق قابلية التركيب (modularity) على الشِفرة البرمجيّة والتصميمات والتوثيق وتنظيم الفريق، على سبيل المثال. عندما أردنا استخدام كلمة المصطلحات التقليدية في سياق أوسع، أصبح الأمر مُحيّرًا - لم نتمكن من التغلب على الأثقال التي جلبها المصطلح الأصلي معه. وعندما حدث ذلك، ساهمنا في تراجع اللغة بواسطة اختراع مصطلحاتنا الخاصّة.

الشِفرة المصدريّة والموارد الأخرى

تُستخرج معظم الشفرات البرمجيّة الموضّحة في هذا الكتاب من المِلَفّات المصدريّة القابلة للتَرْجَمَة، وهي متاحة للتنزيل من موقعنا على الويب[3].

هناك ستجد أيضًا وصلات إلى المراجع التي نجدها مفيدة، جنبا إلى جنب مع تحديثات للكتاب والأخبار عن تطويرات المبرمجين العمليين الآخرين.

أخبرنا عن ملاحظاتك.

نحن نثمن سماع رأيك، راسلنا على ppbook@pragprog.com.

ثناء الطبعة الثانية

لقد استمتعنا حرفيًا بآلاف المحادثات الشيّقة حول البرمجة العملية على مدى السنوات العشرين الماضية، والاجتماع بالنّاس في المؤتمرات، وفي الدورات، وأحيانًا حتى على متن الطائرة. قد أضاف كل واحدٍ من هؤلاء إلى فهمنا لعملية التطوير، وساهم في التحديثات في هذه الطبعة. شكرًا لكم جميعًا (واستمروا في إخبارنا عندما نكون مخطئين).

شكرًا للمشاركين في النسخة التجريبيّة. لقد ساعدتنا أسئلتكم وتعليقاتكم على شرح الأمور بشكلٍ أفضل.

3 https://pragprog.com/titles/tpp20

قبل أن نبدأ بالنسخة التجريبية، شارَكنا الكتاب مع بعض الناس للتعليق عليه. شكرًا فيكي براسيور وجيف لانغر وكيم شرييير على تعليقاتكم المفصّلة.

وإلى خوسيه فاليم ونيك كوثبرت لمراجعتكم التقنية.

شكرًا رون جيفريز للسماح لنا باستخدام مثال سودوكو.

الكثير من الامتنان للأصدقاء في دار النشر بيرسون Pearson الذين وافقوا على السماح لنا بإنشاء هذا الكتاب بطريقتنا.

شكر خاص لجانيت فورلو التي لا غنى لنا عنها، التي تتقن أي شيء تتولى أمره، وتبقينا على اطلاع.

وأخيرًا، شكرًا لجميع المبرمجين العمليين هناك، الذين جعلوا البرمجة أفضل للجميع طوال السنوات العشرين الماضية. وهنا لعشرين أخرى.

من مقدمة الإصدار الأول

سوف يساعدك هذا الكتاب لتصبح مبرمجًا أفضل.

يمكنك أن تكون مطورًا منفردًا أو عضوًا في فريق مشروع كبير أو مستشارًا يعمل مع العديد من العملاء في وقتٍ واحد. لا يهم، سوف يساعدك هذا الكتاب كفرد للقيام بعملٍ أفضل. هذا الكتاب ليس نظريًا - فنحن ننظر في موضوعات عمليّة، وعلى استخدام تجربتك لاتخاذ قرارات أكثر استنارة. تأتي كلمة عملي من البراغماتية[4] اللاتينية - "ماهر في العمل "- وهي بدورها مشتقّة من πραγματικός اليونانية، بمعنى" مناسب للاستخدام."

هذا كتاب عن العمل.

البرمجة حرفة. في أبسط حالاتها، يتعلق الأمر بالحصول على جهاز حاسوب ليقوم بما تريد منه أن ينفّذه (أو ما يريد المستخدم أن ينفّذه). بصفتك مبرمجًا، فأنت في جانبٍ منك مستمعًا، وجانب مستشاراً، وجانب مفسراً، وجانب ديكتاتوراً. تحاول التقاط متطلباتٍ محيّرة وإيجاد طريقة للتعبير عنها بحيث تتمكن الآلة وحدها أن تعدل بينها. تحاول توثيق عملك حتى يتمكن الآخرون من فهمه، وتحاول هندسة عملك حتى تمكّن الآخرين من البناء عليه. الأكثر من ذلك، تحاول عمل كل هذا ضدّ دقاتٍ لا هوادة فيها لساعة المشروع. أنت تصنع معجزات صغيرة كل يوم.

إنّه عملٌ صعب.

هناك الكثير من النّاس يعرضون عليك المساعدة. يروّج موردو الأدوات المعجزات التي تصنعها منتجاتهم. يَعِد معلمو المنهجيّة بأن تقنياتهم تضمن النتائج. يدّعي الجميع أن لغتهم البرمجيّة هي الأفضل، وكل نظام تشغيل هو الحل لجميع العلل التي يمكن تصوّرها.

بالطبع، لا شيء من هذا صحيح. لا توجد إجابات سهلة. لا يوجد حل أفضل. سواء كان ذلك أداةً أو لغةً أو نظام تشغيلٍ. لا يمكن أن يكون هناك سوى أنظمة أكثر ملاءمة في مجموعة معيّنة من الظروف.

هنا تأتي العمليّة "الواقعيّة". يَجِبُ ألاّ تكون مُتشبثًا بأي تقنية معينة، ولكن يجب أن يكون لديك خلفية واسعة وقاعدة خبرة كافية تسمح لك باختيار حلول جيّدة في حالات معيّنة. تنشأ خلفيتك من فهم المبادئ الأساسيّة لعلوم الحاسوب، وتأتي تجربتك من مجموعةٍ واسعة من المشروعات العمليّة. تجتمع النظرية والتطبيق لتجعلك قويًا.

يمكنك تعديل نهجك ليناسب الظروف الحاليّة والبيئة. أنت تحكم على الأهميّة النسبيّة لجميع العوامل التي تؤثر على المشروع وتستخدم تجربتك لإنتاج الحلول المناسبة. وتفعل ذلك باستمرار مع تقدم العمل. المبرمجون العمليون ينجزون المهمّة، ويقومون بذلك بشكلٍ جيّد.

من ينبغي له قراءة هذا الكتاب؟

يستهدف هذا الكتاب الأشخاص الذين يريدون أن يصبحوا مبرمجين أكثر فعاليّة وإنتاجيّة. ربما تشعر بالإحباط لأنه لا يبدو أنك تحقّق إمكاناتك. ربما تنظر إلى الزملاء الذين يبدو أنهم يستخدمون أدوات تجعلهم أكثر إنتاجيّةً منك. ربما تستخدم وظيفتك الحاليّة تقنيات قديمة، وتريد أن تعرف كيف يمكن تطبيق الأفكار الجديدة على ما تفعله.

4 **[المترجم]** عملية (ذرائعية): اسم مشتق من اللفظ اليوناني" براغما" ومعناه العمل، وهي مذهب فلسفي – سياسي يعتبر نجاح العمل المعيار الوحيد للحقيقة؛ فالسياسي العملي يدّعي دائماً بأنه يتصرف ويعمل بواسطة النظر إلى النتائج العملية المثمرة التي قد يؤدي إليها قراره، وهو لا يتخذ قراره بوحي من فكرةٍ مُسبقة أو أيديولوجية سياسيّة محدّدة، وإنما بواسطة النتيجة المتوقعة لعمل.

نحن لا ندّعي أن لدينا جميع الإجابات (أو حتى معظمها)، ولا تنطبق جميع أفكارنا في جميع المواقف. كل ما يمكننا قوله هو أنك إذا اتبعت نهجنا، ستكتسب الخبرة بسرعة، ستزيد إنتاجيتك، وسيكون لديك فهم أفضل لعملية التطوير بِرُمَّتها. وستكتب برامج أفضل.

مالذي يصنع المبرمج العملي

كل مطوّر فريد من نوعه، مع نِقَاط القوة والضعف الفرديّة والتفضيلات والمكروهات. مع مرور الوقت، سيصوغ كل منهم بيئته الشخصيّة. تلك البيئة ستظهر شخصيّة المبرمج بنفس قوة هواياته أو ملابسه أو قَصّة شعره. ومع ذلك، إذا كنت مبرمجًا عمليًا، ستشارك العديد من الخصائص التالية:

مستخدم أول / متكيف سريع

لديك حدس في التِقَانَة والتقنيات، وتحبّ تجرِبة الأشياء. عندما تُعطى شيئًا جديدًا، يمكنك فهمه بسرعة ودمجه مع بقية معرفتك. تولد ثقتك من التجربة.

فضولي

تميل إلى طرح الأسئلة. هذا جميل - كيف فعلت ذلك؟ هل واجهتك مشكلات مع تلك المكتبة؟ ما هذه الحوسبة الكموميّة التي سمعت عنها؟ كيف تُنفَّذ الروابط الرمزيّة؟ أنت مُجمّع للحقائق الصغيرة، قد يؤثر كل منها على القرار لسنوات من الآنَ.

مفكر ناقد

نادرًا ما تأخذ الأمور كما هي دون أن تتحق الحقائق أولاً. عندما يقول زملاؤنا "لأن هذه هي الطريقة التي يتم بها ذلك"، أو إذا وعد مورد بحل لجميع مشكلاتك، فأنت تَشتَمُ تحديًا.

واقعي

أنت تحاول فهم الطبيعة الأساسيّة لكل مشكلة تواجهها. تمنحك هذه الواقعية إحساسًا جيدًا بمدى صعوبة الأشياء، والوقت الذي ستستغرقه. إن فهمك العميق أنّ العملية يجب أن تكون صعبةً أو ستستغرق بعض الوقت حتى تكتمل يمنحك القدرة على الاستمرار في ذلك.

متعدد المهارات

أنت تحاول جاهدًا أن تكون على درايةٍ بمجموعةٍ واسعةٍ من التقنيات والبيئات، وتعمل على مواكبة التطورات الجديدة. مع أنّ وظيفتك الحاليّة قد تتطلب منك أن تكون متخصّصًا، فستكون دائمًا قادرًا على الانتقال إلى مجالات جديدة وتحدّيات جديدة.

لقد تركنا أبسط الخصائص حتى الأخير. كل المبرمجين العمليين يشتركون بها. إنها رئيسة بما يكفي لنصيغها كنصائح:

نصيحة ١	اهتم بحرفتك

نشعر أنه لا فائدة من تطوير البرمجيات ما لم تهتم بعملها بشكل جيّد.

نصيحة ٢	فكّر في عملك.

لكي تكون مبرمجًا عمليًا، فإننا نتحداك أن تفكر فيما تفعله في أثناء قيامك به. هذه ليست مراجعة لمرّة واحدة للممارسات الحاليّة - إنها تقييمٌ نقدي مستمر لكل قرار تتخذه، كل يوم، وعلى كل مشروع. لا تعمل أبداً على وضعيّة الطيار الآلي. فكّر باستمرار وانتقد عملك في الوقت الحقيقي. شعار شركة IBM القديم، *فكّر!*، هو شعار المبرمج العملي.

إذا كان هذا يبدو عملًا شاقًا بالنسبة لك، فأنت تُظهر الخاصيّة الواقعيّة. سيستغرق هذا بعضًا من وقتك الثمين - الوقت الذي ربما يكون فعلًا تحت ضغط هائل. المكافأة هي مشاركة أكثر نشاطًا في الوظيفة التي تحبّها، والشعور بالسيطرة على مجموعة متزايدة من الموضوعات، ومتعة الشعور بالتحسّن المستمر. على المدى الطويل، سيتم سداد وقتك عندما تصبح أنت وفريقك أكثر كفاءة، وعندما تكتب شِفرة يسهل الحفاظ عليها، وتقضي وقتًا أقل في الاجتماعات.

العمليون الفرديون، الفِرق الكبيرة

يشعر بعض النّاس أنه لا يوجد مكان للفرديّة في الفرق الكبيرة أو المشروعات المعقدة. ويقولون: "البرمجيات هي تخصص هندسيّ، ينهار إذا اتخذ أعضاء الفريق قراراتهم بأنفسهم".

نحن نختلف معهم بشدة.

يجب أن تكون هناك هندسة في بناء البرمجيات. ومع ذلك، هذا لا يمنع الحرفيّة الفرديّة. فكّر في الكاتدرائيات الكبيرة التي بُنيت في أوروبا خلال العصور الوسطى. استغرق كل منها آلاف السنين من الجهد المبذول على مدى عقود عدّة. مُرّرت الدروس المستفادة إلى المجموعة التالية من البنائين، الذين تقدموا في حالة الهندسة الإنشائية بإنجازاتهم. لكن النجارين، وقاطعي الأحجار، والنحاتين، والعاملين في الزجاج كانوا جميعاً حرفيين، فسّروا المتطلبات الهندسية لإنتاج كل ما تجاوز الجانب الميكانيكي البحت من البناء. كان إيمانهم بمساهماتهم الفرديّة هو الذي استمر في المشروعات: نحن الذين قطعنا الأحجار فقط علينا تصوّر الكاتدرائيات دائمًا.

ضمن الهيكل العام للمشروع، هناك دائمًا مساحة للفرديّة والحرفيّة. هذا صحيح بشكل خاص بالنظر إلى الوضع الحالي لهندسة البرمجيات. بعد مئة عام من الآنَ، قد تبدو هندستنا قديمةً مثل التقنيات المستخدمة من قبل بناة الكاتدرائية في العصور الوسطى بالنسبة للمهندسين المدنيين اليوم، في حين تبقى حِرفيتنا موضع تقدير.

إنها عمليّة مستمرة

سأل سائح يزور كليّة إيتون في إنجلترا البستاني كيف جعل المروج مثالية للغاية. أجاب: "هذا أمر سهل، فأنت تزيل الندى كل صباح، وتقصها كل يومين، وتمهدها مرة واحدة في الأسبوع".

سأل السائح "هل هذا كل شيء؟". أجابه البستاني "بالتأكيد". "افعل ذلك لمدة 500 عام وستحصل على حديقة جميلة أيضًا."

تحتاج المروج الكبيرة إلى كميات صغيرة من الرعاية اليوميّة، وكذلك يحتاج المبرمجون الكبار. يحبّ مستشارو الإدارة إسقاط كلمة كايزن في المحادثات. "كايزن" هو مصطلح ياباني يستوعب مفهوم إجراء العديد من التحسينات الصغيرة باستمرار. كان يعد أحد الأسباب رئيسة للمكاسب الكبيرة في الإنتاجيّة والجودة في التصنيع الياباني، وتم نسخه على نطاق واسع في جميع أنحاء العالم. ينطبق كايزن على الأفراد أيضًا. كل يوم، اعمل على صقل المهارات التي لديك ولإضافة أدوات جديدة إلى ذخيرتك. على عكس مروج إيتون، ستبدأ في رؤية النتائج في غضون أيام. على مرّ السنين، ستندهش من كيفيّة ازدهار تجربتك وكيفيّة نمو مهاراتك.

الفصل الأول:

فلسفة عمليّة

هذا الكتاب عنك أنت.

لا تخطئ، إنّه عملك، والأهمّ من ذلك، إنها حياتك. أنت من يملكها. أنت هنا لأنّك تعلم أنّه بإمكانك أن تصبح مطوّرا أفضل وأن تساعد الآخرين على أن يصبحوا كذلك أيضاً. يمكنك أن تصبح *مبرمجًا عملياً*.

ما الذي يميّز المبرمجين العمليين؟ نرى أنه طريقة أو أسلوب أو فلسفة تعامل مع المشكلات وحلولها فهم يفكرون فيما وراء المشكلة الآنيّة، ويضعونها في سياقها الأعم وينشدون الصورة الأكبر. بعد كل هذا، كيف لك أن تكون *عملياً* دون هذا السياق الأعم؟ كيف يمكنك تقديم تسويات ذكيّة وقرارات رشيدة؟

إنّ تحملهم مسؤولية ما يقومون به هو مفتاح آخر من مفاتيح نجاح المبرمجين العمليين، التي نناقشها في باب "القطة أكلت شِفرتي المصدريّة" باعتبارهم مسؤولين، لن يجلس المبرمجون العمليون مكتوفي الأيدي يشاهدون مشروعاتهم وهي تنهار بسبب الإهمال. سنخبرك بكيفيّة الاحتفاظ بمشروعاتك سليمةً في باب "انتروبيا البرمجيات".

يجد معظم النّاس صعوبةً في التغيير، لأسبابٍ وجيهةٍ حيناً، وحيناً بسبب الخمول الاعتيادي القديم. سنطّلع على استراتيجية للحث على التغيير في باب "حَساء الحصى والضفادع المغلية"، و (من أجل التوازن) سنعرض القصّة التحذيريّة لبرمائي يتجاهل مخاطر التغيير التدريجي.

تتمثّل إحدى فوائد فهم السياق الذي تعمل فيه في تسهيل معرفة مدى الجودة التي ستكون عليها برمجيتك. في بعض الأحيان يكون شبه الكمال هو الخِيار الوحيد، ولكن غالبًا ما تكون هناك مقايضات سنستكشف ذلك في "البرمجيّات الجيّدة كفايةً"

يجب أن يكون لديك بالطبع قاعدة واسعة من المعرفة والخبرة لتنجح بكل هذا. التعلّم عمليّة مستمرّة ومتجدّدة سنناقش بعض الاستراتيجيات في قسم "محفظتك المعرفيّة" للحفاظ على النشاط.

أخيرًا، لا أحد منا يعمل في الفراغ ننفق جميعًا قدرًا كبيرًا من الوقت في التفاعل مع الآخرين. "تواصل" يعرض الطرق التي يمكننا بواسطتها العمل بشكلٍ أفضل.

تنبع البرمجة العملية من فلسفة التفكير العملي. يضع هذا الفصل الأساس لتلك الفلسفة.

الموضوع 1. إنها حياتك

> أنا لست في هذا العالم لأرتقي إلى مستوى توقعاتك وأنت لست في هذا العالم لكي ترقى إلى مستوى توقعاتي
>
> ← بروس لي

إنّها حياتك أنت من يملكها أنت تديرها أنت تنشئها.

العديد من المطوّرين الذين نتحدّث إليهم محبطون. مخاوفهم متنوعة. يشعر البعض أنّهم في حالة ركود في وظائفهم، والبعض الآخر أنّ التِقَانَة قد تجاوزتهم. يشعر النّاس أنهم لا يحصلون على تقدير كافٍ، أو يحصلون على رواتب منخفضة، أو أنّ فرقهم سامة. ربما يريدون الانتقال للعمل في آسيا أو أوروبا أو العمل من المنزل.

والجواب الذي نقدّمه هو نفسه دائمًا.

"لماذا لا يمكنك تغييره؟"

يجب أن يظهر تطوير البرامج تقريباً في قمّة أيّ قائمةٍ من الوظائف الممكن التحكم فيها. مهاراتنا مطلوبة، ومعرفتنا تتجاوز الحدود الجغرافية، ويمكننا العمل عن بعد. نحن نتقاضى أجرًا جيّدًا. يمكننا حقًّا فعل أي شيء نريده.

لكن ولسببٍ ما، يبدو أن المطوّرين يقاومون التغيير. إنهم يتقاعسون، ويأملون أن تتحسّن الأمور إنهم يترقبون، بشكل سلبي، ثم أن مهاراتهم أصبحت قديمة ويشكون من أن شركاتهم لم تدربهم. إنهم ينظرون إلى الإعلانات الخاصّة بالمواقع الغريبة في الحافلة، ثم ينزلون إلى المطر البارد ويتجهون إلى العمل.

إذاً إليك أهمّ نصيحة في الكتاب.

نصيحة ٣ لديك وكالة

هل تستنزفك بيئة عملك؟ هل عملك مُمِل؟ حاول إصلاحها لكن لا تستمرّ في المحاولة إلى الأبد. كما يقول مارتن فاولر، "يمكنك إمّا تغيير مؤسستك أو تغيير مؤسستك[5].

إذا كانت التِقانة تتجاوزك، فخصّص مدّة (من وقتك الخاص) لدراسة أشياء جديدة تبدو مثيرةً للاهتمام. إنك تستثمر في نفسك، لذا فإن فعل ذلك خارج ساعة العمل أمر معقول آخر الأمر.

تريد العمل عن بعد؟ هل طلبت ذلك؟ إذًا قالوا لا، عندها ابحث عن شخصٍ يقول نعم.

تمنحك هذه المثابرة مجموعة رائعة من الفرص كن استباقيًا، واغتنمها.

الأقسام ذات الصلة

- الموضوع 4، حساء الحصى والضفادع المغليّة، في الصفحة 34
- الموضوع 6، محفظتك المعرفيّة، في الصفحة 38

5 http://wiki.c2.com/?ChangeYourOrganization

الموضوع 2. القطّة أكلت شِفرتي المصدريّة

أعظم نِقَاط الضعف هو الخوف من أن تبدو ضعيفاً

← J.B بوسويت، سياسة من الأسفار المقدسة

إنّ فكرة تحمّل المسؤوليّة عن نفسك وأفعالك فيما يخصّ تقدّمك المهني، وتعلّمك وتعليمك، ومشروعك، وعملك اليوميّ هي أحد أحجار الزاوية في الفلسفة العملية يتحمّل المبرمجون العمليون مسؤولية حياتهم المهنيّة، ولا يخشون الاعتراف بعدم المعرفة أو الخطأ. كن أكيدًا، ليس هذا الجانب الأكثر إشراقًا في البرمجة. ولكنه سيحدث— حتى في أفضل المشروعات. فعلى الرغم من الاختبار الشامل والتوثيق الجيّد والأتمتة القويّة، إلاّ أن الأمور تسوء. تتأخر عمليات التسليم. وتظهر مشكلات تقنية غير متوقعة.

تحدث هذه الأشياء، ونحاول التعامل معها بأكبر قدرٍ ممكنٍ من الاحترافية. هذا يعني أن تكون صادقًا ومباشرًا. يمكننا أن نفخر بقدراتنا، ولكن يجب علينا أن نُقرّ بنقصنا وجهلنا وأخطائنا.

ثقة الفريق

إن ثقة الفريق فوق كل اعتبار، يحتاج فريقك لأن يكون قادرا على الثقة بك والاعتماد عليك - ويجب أن تكون مرتاحًا للاعتماد على أيّ منهم أيضًا. إن الثقة في الفريق ضرورية للغاية للإبداع والتعاون وفقًا لأدبيات البحث[6]. حيث يمكنك التعبير عن أفكارك بأمانٍ في بيئةٍ صحيّة قائمةٍ على الثقة.

قدّم أفكارك واعتمد على أعضاء فريقك الذين يمكنهم بدورهم الاعتماد عليك. أما دون ثقة، حسنًا.

تخيّل أن فِرقَة تسلل النينجا المتقدّمة تتسلل إلى مخبأ الشرير الوغد. وقد وصلتَ للموقع بعد شهور من التخطيط والتنفيذ الدقيق، وحان دورك الآن لإعداد شبكة توجيه الليزر: "عذرًا، أيها الرفاق، الليزر ليس معي كانت القطّة تلعب بالنقطة الحمراء وتركته في المنزل".

قد يصعب إصلاح هذا النوع من خيانة الثقة.

تحمّل المسؤوليّة

إن المسؤوليّة هي شيء توافق عليه بهمّة دائمة. تلتزم بضمان عمل شيءٍ ما بشكلٍ صحيح، ولكن ليس لديك بالضرورة تحكم مباشر في كل جوانبه. يجب عليك إضافةً إلى بذل قُصَارَى جهدك الشخصيّ، تحليل الموقف بحثًا عن أي مخاطر خارجة عن إرادتك. لك الحق في عدم تحمّل المسؤوليّة عن موقفٍ مستحيل، أو موقفٍ مخاطرته كبيرة جدًا، أو آثاره الأخلاقية غير واضحة. يجب عليك اتخاذ القرار بناء على قيمك وحكمك.

وفي حال ارتكبت خطأ ما (كما نفعل جميعًا) أو خطأ في الحكم، اعترف بذلك بأمانة وحاول تقديم الخيارات

لا تلم شخصًا أو أي شيء آخر، أو تختلق عذرًا. لا تلقي باللوم في كل المشكلات على المورد أو لغة البرمجة أو الإدارة أو زملائك في العمل. قد يلعب أي من هؤلاء أو كلّهم دورًا ما في ذلك، ولكن الأمر متروك لك لتقديم الحلول، وليس الأعذار.

6 انظر، على سبيل المثال، تحليل ميتا (meta) جيّد في *الثقة وأداء الفريق: التحليل المرافق للتأثيرات رئيسة، والمشرفين، والمتغيرات المشتركة*، http://dx.doi.org/10.1037/apl0000110

إذا كان هناك خطر من أن المورد لن يأتي من أجلك، فيجب أن يكون لديك خُطّة طوارئ بديلة. إذا تعطل جهاز التخزين لديك– آخذًا كل شِفرة المصدر معه - وليس لديك نسخة احتياطيّة أخرى، فهذا خطأك. إن إخبار رئيسك بأن "القطة أكلت شفرتي المصدرية، في الصفحة29" لن يحلّها.

نصيحة ٤ قدّم خيارات، لا تقدّم أعذارًا واهيّة

قبل أن تفاتح شخصًا ما لإخباره عن سبب عدم إمكانية فعل شيء ما، أو تأخره، أو تعطّله، توقّف واستمع إلى نفسك وأنت تتحدث إلى البطّة المطاطيّة على الشاشة أو إلى القطّة. هل يبدو عذرك معقولًا أم غبيًا؟ كيف سيبدو الأمر لرئيسك في العمل؟

مرّر المحادثة في ذهنك مالذي قد يقوله الأشخاص الآخرون؟ هل يسألون "هل جربت هذا..." أو "ألم تفكر في ذلك؟" كيف سترد؟ قبل أن تذهب وتنقل لهم الأخبار السيئة، هل ثمة هناك أي شيء آخر يمكنك تجربته؟ قد تعرف بالضبط في بعض الأحيان ما الذي سيقولونه، لذا وفّر عليهم المشكلات.

قدّم لهم الخيارات بدلاً من الأعذار. لا تقل أنّه لا يمكن فعل ذلك، اشرح ما يمكن فعله لإنقاذ الموقف. هل يجب حذف الشِفرة؟ أخبرهم بذلك، واشرح لهم أهميّة إعادة البناء (انظر الموضوع 40، إعادة البناء، في الصفحة 235).

هل تحتاج إلى قضاء بعض الوقت في النماذج الأوليّة لتحديد أفضل طريقة للمتابعة (راجع الموضوع،13 النماذج الأوليّة والملاحظات المُلحقة، في الصفحة 78)؟ هل تحتاج إلى تقديم اختبار أفضل (راجع الموضوع، 41 اختبار لتكتب الشِفرة، في الصفحة، 240 واختبار قاسٍ ومستمر، في الصفحة 301) أو الأتمتة لمنع حدوثه مجدّدا؟

ربما ستحتاج إلى موارد إضافيّة لإكمال هذه المهمّة أو ربما تحتاج إلى قضاء المزيد من الوقت مع المستخدمين؟ أو ربما أنت لوحدك: هل تحتاج إلى تعلّم بعض الأساليب أو التقنيات بعمقٍ أكبر؟ هل يكون وجود كتاب أو دورة مساعداً لك؟ لا تخف من السؤال أو الاعتراف بأنك بحاجة للمساعدة.

حاول أن ترمي الأعذار الواهية قبل التعبير عنها بصوتٍ عال. إذا كان ولا بد أخبرها للقطّة. بعد ذلك كلّه سيتحمل الصغير تيدي اللوم.

الأقسام ذات الصلة

- الموضوع 49، الفرق العملية، صفحة 289

تحدّيات

- كيف تستجيب عندما يأتيك شخص ما - كالمحاسب المصرفي، أو ميكانيكي السيارات، أو الكاتب - بحجّة واهية؟ ما رأيك فيهم وفي شركتهم تبعًا ذلك؟
- تأكّد عندما تجد نفسك تقول، "لا أعرف"، من إلحاقها بعبارة "ولكن سأكتشف ذلك. إنها طريقة رائعة للاعتراف بما لا تعرفه، ولتحمل المسؤوليّة بعد ذلك كالمحترفين.

الموضوع 3. اعتلاج البرمجيات (Software Entropy)

مع أنّ تطوير البرمجيات محصّن ضد جميع القوانين الفيزيائية تقريبًا، إلا أن الزيادة الحتميّة في الاعتلاج[7] "الإنتروبيـا" (*entropy*) تواجهنا بشدة. يشـير مصـطلح الاعتلاج في الفيزيـاء إلى مقـدار "الاضـطراب" في النظـام. ولسـوء الحـظ، تؤكّـد قـوانين الـديناميكا الحراريّة أن الاعتلاج في الكون يميل نحو الحد الأقصى. عندما يزيد الاضطراب في البرمجيات، ندعوه "تدهور البرمجيات". قد يصفه البعض بالمصطلح الأكثر تفاؤلاً، "الدَين التقني"، مع الفكرة الضمنيّة التي مفادها أنهم سيُعيدونه يومًا ما. ربما لن يفعلوا.

وبغض النظر عن الاسم، لكن، يمكن أن ينتشر كل من الدين والتدهور بشكل لا يمكن السيطرة عليه.

هناك العديد من العوامل التي يمكن أن تساهم في تدهور البرمجيات يبدو أن أهمّها هو علم النفس، أو الثقافة، في أثناء العمل على المشروع حتى لو كنتم فريقًا من شخصٍ واحد، فإن سيكولوجيا مشـروعكم يمكن أن يكـون أمـرًا حسّاسًـا. مـع أنّ وضـع أفضـل الخطـط وأفضل الأشخاص، فإنه لا يزال هناك إمكانية لأن يختبر المشروع التلف والتدهور خلال مدّة حياته.

هناك مشـروعات أخـرى، نجحت في محاربـة ميـل الطبيعـة للفوضـى وتمكنت من الخـروج بشـكل جميـل، رغم الصـعوبات الهائلـة والنكسات المستمرة.

ما الذي يصنع الفرق؟

إن بعض المباني في المدن الداخليّة جميلة ونظيفة، في حين أن بعضها الآخر متعفن الهيكـل لمـاذا؟ اكتشـف البـاحثون في مجـال الجريمة والانحلال المدني آلية فتيلٍ رائعة، إنها آلية تحوّل بسرعة مبنى نظيف وسليم إلى مبنى مهمل ومشوّه ومهجور[8].

نافذة مكسورة.

إن ترك نافذة مكسورة واحدة، دون إصلاح لأية مدّة زمنية طويلـة، تغـرس في سـكان المبـنى شـعورًا بـالتخلي عنهم - شـعورًا بـأن الجهات المسؤولة لا تهتم بالمبنى. ولهذا تكسر نافذة أخـرى فيبـدأ النّـاس في رمي النفايـات. تظهـر الكتابـة على الجـدران. ويبـدأ الضـرر الهيكلي شديد الْخَطَر، يتضرر المبنى خلال مدّة زمنية قصيرة نسبيًا بشكلٍ يتجاوز رغبة المالك في إصـلاحه، ويصـبح الشـعور بـالتخلي أمرًا حقيقيًا.

لماذا سيحدث ذلك فرقًا؟ أجرى علماء النفس الدراسة[9] التي تظهر أن اليأس يمكن أن يكون معديًا. فكّـر في فـيروس الإنفلـونزا في أوساطٍ قريبة. إن تجاهل حالة خاطئة بشكل واضحٍ يعزّز الأفكـار من قبيـل ربما لا يوجـد شـيء يمكن إصـلاحه، وأن لا أحـد يهتم، وكـل شيء محكوم عليه بالفشل؛ جميع الأفكار السلبية التي يمكن أن تنتشر بين أعضاء الفريق، مما يخلق دوامة مفرغة.

---

7 **[المترجم]** يشير اعتلاج البرمجيات Software Entropy إلى ميل البرمجيات، مع مرور الوقت، إلى أن تصبح مكلفة وصعبة الصيانة. إن النظام البرمجي الذي يخضع للتغيير المستمر، مثل إضافة وظائف جديدة إلى تصميمه الأصلي، سيصبح في النهاية أكثر تعقيدًا ويمكن أن يصبح غير منظم مع نموه، مع فقدان بنية التصميم الأصلية.

8 انظر الشرطة وسلامة الجوار [WH82]

9 انظر الاكتئاب المعدي: الوجود، وخصوصيّة الأعراض المكتئبة، ودور البحث عن الاطمئنان [Joi94]

نصيحة ٥ لا تعش مع نوافذ مكسورة

لا تترك "نوافذ مكسورة" (تصميمات سيئة أو قرارات خاطئة أو شفرات ضعيفة) دون إصلاح أصلح كل شيء بمجرد اكتشافه إذا لم يكن هناك وقت كاف لإصلاحه بشكل صحيح، فقم *بسدّه*. ربما يمكنك إخراج الشِفرة السيئة، أو عرض رسالة "لم ينفّذ"، أو استبدال بيانات وهميّة بدلاً من ذلك. اتخذ بعض الإجراءات لمنع الحاق المزيد من الضرر ولإظهار أنك مشرف على الوضع.

لقد رأينا أن الأنظمة الوظيفيّة النظيفة تتدهور بسرعةٍ كبيرة بمجرّد أن تبدأ النوافذ في التحطم. هناك أيضاً عوامل أخرى يمكن أن تساهم في تدهور البرامج، سنتطرّق إلى بعضها في مكانٍ آخر، ولكن الإهمال يسرّع التدهور أكثر من أي عامل آخر.

قد تعتقد بأنه ليس لدى أحد الوقت الكافي لتنظيف جميع الزجاج المكسور في المشروع. إذا كان الأمر كذلك فمن الأفضل، التخطيط للحصول على سلة قمامة، أو الانتقال إلى حيّ آخر. لا تدع الاعتلاج ينتصر.

أولا، لا تؤذ

كان لدى *آندي* ذات مرّة أحد المعارف، وكان فاحش الغنى. كان منزله نظيفًا جدًا، مليئًا بالتحف التي لا تقدّر بثمن، والقطع الأثرية، وما إلى ذلك. في أحد الأيام، كان هناك قطعةَ قماشٍ معلّقةً بالقرب من موقدةٍ مشتعلة بالنيران. اندفع قسم الإطفاء لإنقاذ الموقف- ومنزله. ولكن قبل أن يسحبوا خراطيمهم الكبيرة والمتسخة إلى المنزل، توقفوا - بالرغم اشتعال النار – لبَسط حصيرةٍ بين الباب الأمامي ومصدر الحريق.

فهم لم يرغبوا في تلويث السّجادة.

يبدو هذا متطرفًا جدا الآن.لكنهم قاموا بتقييم الوضع بوضوح، وكانوا واثقين من قدرتهم على احتواء الحريق، وكانوا حريصين على عدم إلحاق الضرر غير الضروري بالممتلكات. تلك هي الطريقة التي يجب التعامل بها مع البرمجيات: لا تتسبب في أضرار جانبيّة لمجرّد وجود أزمةٍ من نوعٍ ما. نافذة واحدة مكسورة..واحدة كثيرة جدًا.

إنّ وجود نافذة واحدة مكسورة - قطعة من الشِفرة البرمجيّة سيئة التصميم، وقرار إداريّ سيء يجب على الفريق التعايش معه طوال مدّة المشروع - هو كل ما يتطلبه الأمر لبَدء الانحدار. إذا وجدت نفسك تعمل في مشروع يحتوي على عدد قليل جدًا من النوافذ المكسورة، فمن السهل جدًا الانزلاق إلى ذهنيّة "كل ما تبقى من هذه الشِفرة لا معنى له، سوف أتبعهم فحسب. لا يهم إذا كان المشروع على ما يرام حتى هذه اللحظة. في التجربة الأصليّة التي أدت إلى "نظرية النافذة المكسورة"، بقيت سيارة مهجورة لمدّة أسبوع دون أن يمسّها أحد. ولكن بمجرد كسر نافذة واحدة، تم نهب السيارة وقلبها رأساً على عقب في غضون ساعات.

وعلى نفس المنوال، إذا وجدت نفسك تعمل في مشروع حيث الشِفرة جميلة جمالًا مذهلًا - مكتوبة كتابةً نظيفة، جيدة التصميم وأنيقة - فمن المرجّح أن توليها عناية خاصّة إضافيّة بعدم العبث بها، تمامًا مثل رجال الإطفاء حتى وإن كان هناك حريق مشتعل (الموعد النهائي، تاريخ الإصدار، عرض تجاري تجريبي، وما إلى ذلك)، فأنت لا تريد أن تكون أول من يُحدث فوضى ويتسبب في ضررٍ إضافي.

فقط أخبر نفسك أنه، "لا نوافذ مكسورة"

الأقسام ذات الصلة

- الموضوع 10، التعامدية، في الصفحة 62
- الموضوع 40، إعادة البناء، في الصفحة 235
- الموضوع 44، تسمية الأشياء، في الصفحة 262

تحدّيات

- ساعد في تقوية فريقك بواسطة مسح جوار المشروع الخاص بك، اختر نافذتين أو ثلاث نوافذ مكسورة وناقش مع زملائك ما هي المشكلات وما الذي يمكن فعله لإصلاحها.
- هل يمكنك معرفة متى كسرت نافذة أول مرة؟ ما رد فعلك؟ إذا كان نتيجة قرار شخص آخر، أو مرسوم إداري، فما الذي يمكنك فعله حيال ذلك؟

الموضوع 4. حَساء الحصى والضفادع المغليّة

"إنّ الجنود الثلاثة العائدون إلى ديارهم من الحرب جائعين. عندما رأوا القرية أمامهم تحسّن مزاجهم، كانوا على يقينٍ بأن القرويين سيقدمون لهم وجبةً من الطعام، ولكنهم وجدوا الأبواب مقفلةً والنوافذ مغلقة عندما وصلوا إلى هناك. إذ أن القرويون وبعد سنواتٍ عدّة من الحرب، كانوا يفتقرون إلى الطعام ويدّخرون ما لديهم.

قام الجنود، دون تردّد، بغليّ قدرٍ من الماء ووضعوا فيه ثلاثة حصىً بعناية. فخرج القرويّون المندهشون لمشاهدة ما يحدث.

أوضح الجنود قائلين "هذا حَساء الحصى. سأل القرويين "هل هذا كل ما تضعونه فيه؟". نعم بالتأكيد - مع أنّ البعض يقول أن مذاقه سيكون أفضل مع بعض الجزر ..." هرع قروي خارجًا، وعاد بعد وقتٍ قصير حاملًا سلّة من الجزر من مؤونته.

مضت بضع دقائق، وعاد القرويون للسؤال مجدّدا "هل هذا كل شيء؟"

قال الجنود: "حسناً، بضع حبات من البطاطا ستعطيها شكلا".

أدرج الجنود خلال الساعة التالية مزيدًا من المكونات التي من شأنها تحسين الحساء: لحم البقر والكراث والملح والأعشاب. وفي كلّ مرّة كان يهرع قرويّ مختلف للإغارة على مخزنه الشخصي.

حصلوا بالنهاية على قدرٍ كبير من الحَساء ينبعث منه البخار.أزال الجنود الحجارة، وجلسوا مع كامل سكان القرية للاستمتاع بأول وجبة كما يجب، لم يكونوا قد تذوقوها لشهور."

هناك زوج من الأخلاقيات في قصّة حَساء الحصى. حيث يخدع الجنودُ القرويين، مستغلين فضولهم للحصول على الطعام منهم. لكن الأهمّ من ذلك، أن الجنود تصرفوا كعامل تحفيز، جامعين سكان القرية معًا حتى يتمكنوا من إنتاج شيء لم يكن بإمكانهم القيام به من تلقاء أنفسهم - نتيجة تعاونيّة. في النهاية الجميع يربح.

قد ترغب بين الحين والآخر بمحاكاة قصة الجنود.

ربما تكون في وضع تعرف فيه بالضبط ما يجب فعله وكيفيّة تنفيذه.

يظهر النظام بِرُمَّته أمام عينيك - تعلم أنّه صحيح. ولكن اطلب الإذن للتعامل مع الأمر برمته وستقابلك تأخيرات وتحديقات فارغة. سيكوّن النّاس لجانًا، ستتطلب الميزانيات الحصول على الموافقة، وستتعقّد الأمور. سيَحرس الجميع مواردهم الخاصّة. يطلق على ذلك في بعض الأحيان "جهد بَدْء التشغيل"

إنه وقت رمي الحجارة اكتشف ما يمكنك طلبه بشكلٍ معقول طوّره جيّدًا. بمجرّد أن تحصل عليه، أظهره للنّاس، ودعهم يتعجبون ثم قل "بالطبع، سيكون من الأفضل إذا أضفنا..." تظاهر بأن الأمر ليس مهمًا. اجلس وانتظرهم حتى يبدؤوا مطالبتك بإضافة الوظيفة التي تريدها أنت في الأساس. يجد النّاس أنه من الأسهل الانضمام إلى نجاحٍ جار. أظهر لهم لمحة من المستقبل وستجعلهم يتجمّعون حولك[10].

نصيحة ٦	كن مُحفِّزًا للتغيير

10 في أثناء القيام بذلك، قد تشعر بالراحة من الجملة المنسوبة إلى الأدميرال د. غريس هوبر: "إن طلب المغفرة أسهل من الحصول على الإذن".

من جانب القرويين

من ناحيةٍ أخرى، تدور قصّة حَساء الحصى أيضًا حول الخداع اللطيف والتدريجي. إن الأمر يتعلق بشدّة بالتركيز. يفكّر القرويـون في الأحجار وينسون بقيّة العالم. نقع جميعنا في ذلك يوميًا. الأشياء فقط تقحم نفسها علينا.

نرى جميعنا الإشارات. تخرج المشروعات عن السيطرة ببطء وبلا هوادة. تبدأ معظم كوارث البرمجيات صغيرة جدًا بحيث لا يمكن ملاحظتها، وتحدث معظم تجاوزات المشروع يوميًا. تنحرف الأنظمة عن مواصفاتها مِيزة تلوى الأخرى، بينما يستمر إضافة التصحيح بعد التصحيح إلى جزء من الشِفرة حتى لا يتبقى هناك أي شيء من الشِفرة الأصليّة. غالبًا ما يكون تراكم الأشياء الصغيرة هو ما يكسر الروح المعنويّة والفِرق.

نصيحة ٧ تذّكر الصورة الكبيرة

بصدق، لم نجرّب هذا أبدًا لكن "هم" يقولون أنه إذا أخذت ضفدعًا ورميتهُ في الماء المُغلّى، فسوف يقفز مباشرةً خارجًا منه. ومع ذلك، إذا وضعت الضفدع في وعاء من الماء البارد، ثم سخنته تدريجيًا، فلن يلاحظ الضفدع الزيادة البطيئة في درجة الحرارة، وسيبقى في مكانـه حـتى ينضـج. لاحـظ أن مشـكلة الضـفدع مختلفـة عن مشـكلة النوافـذ المكسـورة الـتي نوقشـت في الموضـوع 3، اعتلاج البرمجيات، في الصفحة 31. يفقد النّاس في نظرية النافذة المكسورة الرغبة في محاربة الإنتروبيا، لأنهم يدركون أنه لا أحـد يهتم. في حين أن الضفدع حتى لا يلاحظ التغيير.

لا تكن مثل الضفدع المزعوم، احفظ الصورة الكبيرة في ذهنك. وراجع باستمرار ما يحدث حولك، وليس فقط ما تفعله شخصيًا.

الأقسام ذات الصلة

- الموضوع 1، إنها حياتك، في الصفحة 27
- الموضوع 38، البرمجة بالمصادفة، في الصفحة 223

تحدّيات

- أثار جون لاكوس في أثناء مراجعة مسودة الطبعة الأولى المسألة التالية: إن الجنود يخدعون القرويين تدريجيًا، لكن التغيير الذي يحفزهم يفيدهم جميعًا. إنما، بواسطة الخداع التدريجي للضفدع فأنت تلحق الضرر بـه. هـل يمكنـك تميـيز مـا إذا كنت تصنع حَساء الحصى أم حَساء الضفادع عندما تحاول تحفيز التغيير؟ هل القرار غير موضوعي أم موضوعي؟
- بسرعة، ودون الالتفات، كم عدد المصابيح الموجودة في السقف فوقك؟ كم عـدد مخـارج الغرفـة؟ كم عـدد الأشـخاص؟ هـل هناك أي شيء خارج السياق، أي شيء يبدو أنه لا ينتمي؟ هذا تمرين في الوعي الظرفي، وهي تقنية يمارسـها النّاس تـتراوح من الكشافة الصبيان والبنات الصغار إلى قوات النخبة البحريين. اعتد التمعّن وملاحظة محيطك. ثم افعل الشـيء نفسـه في مشروعك.

الموضوع 5. برمجيات جيّدة كفايةً

غالبًا ما نشوه ما جيّد بالسعي نحو الأفضل

← شكسبير، الملك المستوى 1.4

هناك نكتة قديمة إلى حدٍّ ما حول شركة قدّمت طلبًا للحصول على 100000 شريحة متكاملة ICs من شركة مصنعة يابانية. جزء من المواصفات كان معدّل العيب: شريحة واحدة في 10000. بعد بضعة أسابيع وصل الطلب: صندوق واحد كبير يحتوي على آلاف الدوائر المتكاملة وصندوق صغير يحتوي على عشرة فقط. وكان هناك ملصقاً أُرفق بالصندوق الصغير كُتب عليه "هذه هي الأجهزة المعيبة".

لو كان لدينا فقط هذا النوع من السيطرة على الجودة. لكن العالم الحقيقي لن يسمح لنا بإنتاج الكثير مما هو مثالي حقًا، على وجه الخصوص، البرمجيات الخالية من الأخطاء. الوقت والتِقَانَة والمزاجية يتآمرون علينا جميعا.

ومع ذلك، يَجِبُ ألاّ يكون هذا محبطا كما وصف إد يوردون في مقالة في *IEEE Software* عندما *البرمجيات الجيّدة بما فيه الكفاية هي الأفضل* *[ADSS18] When good-enough software is best* يمكنك تدريب نفسك لكتابة برنامَج جيّد بما فيه الكفاية لمستخدميك، لمشرفي المستقبل، لراحة بالك. ستجد أنك أكثر إنتاجيّةً وأن مستخدميك أكثر سعادة. وقد تجد أن برامجك هي في الواقع أفضل لزمن حضانةٍ أقصر.

قبل المضي قدما، نحتاج إلى التحفّظ فيما نحن على وشك قوله. لا تعني عبارة "جيّد بما فيه الكفاية" ضمنًا شفرةً خاطئةً أو هزيلةً. يجب أن تفي جميع الأنظمة بمتطلبات المستخدمين لتكون ناجحة، وتلبية معايير الأداء والخصوصيّة والأمان الأساسيّة. نحن ندعو ببساطة إلى منح المستخدمين فرصةً للمشاركة في عملية تحديد متى يكون ما أنتجته جيّدًا بما يكفي لاحتياجاتهم.

أشرك المستخدمين في المقايضة

عادةً ما تكتب برامج للأشخاص الآخرين، ستذكر غالبا ما يريدون[11]. ولكن هل سبق وسألتهم إلى أي حد يريدون لبرامجهم أن تكون جيّدة؟ في بعض الأحيان لن يكون هناك خِيار. إذا كنت تعمل على أجهزة تنظيم ضربات القلب أو الطيّار الآلي أو مكتبة منخفضة المستوى ستُنشر على نطاق واسع، فستكون المتطلبات أكثر صرامة وستكون خياراتك محدودة.

ومع ذلك، إذا كنت تعمل على منتج جديد تمامًا، فستكون لديك قيود مختلفة. سيكون لدى موظفي التسويق وعود لرعايتها، وقد يكون المستخدمون النهائيون القادمون قد وضعوا خططا استنادًا إلى جدول التسليم، وستتعرض شركتك بالتأكيد لقيود نقدية. سيكون من غير المهني تجاهل متطلبات هؤلاء المستخدمين ببساطة لإضافة ميزات جديدة إلى البرنامَج. أو لتنقيح الشِفرة مرّة أخرى فقط. نحن لا ندعو للذعر: من غير المهني أيضًا أن نعد بمقاييس زمنيّة مستحيلة وتقلص الزوايا الهندسية الأساسية لنفي بالموعد النهائي.

يجب مناقشة مجال وجودة النظام الذي تنتجه كجزء من متطلبات هذا النظام.

11 كان من المفترض أن تكون هذه مزحة!

نصيحة ٨ اجعل الجودة قضيّة متطلبات

غالبًا ما تكون في مواقف تنطوي على مقايضات. والمثير للدهشة، أن العديد من المستخدمين يفضلون استخدام البرامج مع بعض الحواف الخشنة اليوم بدلاً من الانتظار لمدّة عام للحصول على نسخة برّاقة، ذات مزايا مساعدة غير أساسيّة (وفي الواقع ما قد يحتاجون إليه بعد عام من الآنَ قد يكون مختلفًا تمامًا). يوافق على ذلك العديد من أقسام تِقَانَة المعلومات أصحاب الميزانيات الضيّقة. غالبًا ما يُفضّل البرنامَج الرائع اليوم على وهم البرنامَج المثالي غدًا. إذا أعطيتَ المستخدمين شيئًا لتجربه مبكرًا، فغالبًا ما ستقودك ملاحظاتهم إلى حلّ نهائي أفضل (راجع الموضوع 12، الرصاصات الخطّاطة، في الصفحة 73).

اعرف متى تتوقف

في بعض النواحي، تكون البرمجة مثل الرسم. تبدأ بقطعة قماش فارغة وبعض المواد الخام الأساسيّة. تستخدم مزيجًا من العلوم والفنون والحرفيّة لتحديد ما يجب القيام به معهم. ترسم شكلاً عاماً، ترسم البيئة الأساسيّة، ثم تملأ التفاصيل. تتراجع باستمرار مع عينٍ ناقدة لعرض ما قمت به. بين الحين والآخر سوف ترمي قطعة قماش وتبدأ مرّة أخرى.

لكن الفنانين سيقولون لك أن كل العمل الشاق يتهدّم إذا كنت لا تعرف متى تتوقف. إذا أضفت طبقة فوق طبقة، التفاصيل على التفاصيل، *فقد تضيع اللوحة في الطلاء*.

لا تفسد برنامجًا جيّدًا تمامًا بسبب الإفراط في الرقي والإفراط في التحسين.استمر، ودع الشِفرة الخاصّة بك تقف في حد ذاتها لمدّة من الوقت. قد لا تكون مثالية. لا تقلق: لن تكون مثالية أبدًا. (في الفصل 7، حين تبرمج، في الصفحة 216، سنناقش فلسفات تطوير الشِفرة في عالم غير كامل.)

الأقسام ذات الصلة

- الموضوع 45، هوة المتطلبات، في الصفحة 268
- الموضوع 46، حل الألغاز المستحيلة، في الصفحة 277

التحدّيات

- انظر إلى أدوات البرامج وأنظمة التشغيل التي تستخدمها بانتظام. هل يمكنك العثور على أي دليل على أن هذه المنظمات و / أو المطوّرين مرتاحون لشحن برامج يعرفون أنها ليست مثاليّة؟ بصفتك مستخدمًا، هل تفضل (1) الانتظار حتى يتم إزالة جميع الأخطاء، (2) امتلاك برمجيّة معقّدة مع قَبُول بعض الأخطاء، أو (3) اختيار برمجية أبسط مع عيوب أقل؟
- النظر في تأثير تطبيق نظام الوحدات على تسليم البرمجيات. هل يستغرق الأمر وقتًا أطول أو أقل للحصول على مجموعة متجانسة من البرمجيات المترابطة مع الجودة المطلوبة مقارنةً بالنظام المصمم كوحدات أو خِدْمَات صغيرة المرتبطة ارتباطا غير محكم على الإطلاق؟ ما هي مزايا أو عيوب كل نهج؟

- هل يمكنك التفكير في البرامج الشعبية التي تعاني تضّخم الوظائف؟ أي إن البرنامَج يحتوي على ميزات أكثر بكثير مما قد تستخدمه على الإطلاق، توفّر كل مِيزة فرصة أكبر للأخطاء ولِنِقَاط الضعف الأمنية، وتجعل من الميزات التي تستخدمها أصعب في العثور عليها وإدارتها. هل أنت في خطر الوقوع في هذا الفخ بنفسك؟

الموضوع 6. محفظتك المعرفيّة

> يقدم الاستثمار في المعرفة أفضل ربح دائمًا.
> ← بنيامين فرانكلين

آه، يا بنيامين فرانكلين الطيب – أبدًا لا توجد عظة بليغة، لماذا، إذا استطعنا النوم باكرًا والنهوض باكرًا، سنكون مبرمجين رائعين، أليس كذلك؟ قد يحظى الطائر المبكر بالدودة، لكن ماذا يحدث بالنسبة للدودة المبكرة؟

في هذه الحالة، أصاب بنيامين كبد الحقيقة. معرفتك وخبرتك هي أهم أصولك المهنيّة اليوميّة.

لكنها لسوء الحظ، أصولٌ منتهية الصلاحية[12]. تُصبح معرفتك متقادمة مع تطوير تقنياتٍ ولغاتٍ وبيئاتٍ جديدة. قد يجعل تغيير قِوَى السوق تجربتك مُهملة أو غير ذات صلة. يمكن أن يحدث هذا بسرعةٍ كبيرةٍ نظرًا لوتيرة التغيير المتزايدة باستمرار في مجتمعنا التكنولوجي.

مع انخفاض قيمة معرفتك تنخفض قيمتك بالنسبة لشركتك أو لعميلك أيضًا. نريد منع حدوث ذلك.

إنّ قدرتك على تعلم أشياء جديدة هي أهم أصولك الاستراتيجية. ولكن كيف تتعلّم كيف تتعلّم، وكيف تعرف مالذي يجب تعلّمه؟

محفظتك المعرفيّة

نودّ أن نفكّر في جميع الحقائق التي يعرفها المبرمجون عن الحوسبة، ومجالات التطبيق التي يعملون فيها، وكل خبرتهم كمحافظ للمعرفة خاصّة بهم. تشبه إدارة محفظة المعرفة إلى حد بعيد إدارة محفظةٍ ماليّة:

1. يستثمر المستثمرون الجادون بانتظام - كعادة.
2. التنويع هو مفتاح النجاح على المدى الطويل.
3. يوازن المستثمرون الأذكياء محافظهم الاستثمارية بين الاستثمارات المتحفّظة والاستثمارات عالية المخاطر والأرباح.
4. يحاول المستثمرون الشراء بسعرٍ منخفض والبيع بسعر مرتفع لتحقيق أقصى عائد.
5. من الواجب مراجعة المحافظ وإعادة موازنتها بشكلٍ دوريّ.

لكي تكون ناجحًا في حياتك المهنيّة، يجب أن تستثمر في محفظة معارفك باستخدام هذه الإرشادات نفسها.

12 الأصل منتهي الصّلاحِيَة هو شيء تقل قيمته بمرور الوقت. تشمل الأمثلة مستودعًا مليئًا بالموز أو تَذْكِرَة لحضور لُعبَة كرة.

الخبر السّار هو أنّ إدارة هذا النوع من الاستثمار مهارة مثل أي مهارةٍ أخرى - يمكن تعلّمها. الحيلـةُ هي في أن تجعـل نفسـك تفعـل ذلك في البداية وتكوّن عادة. طوّر روتينًا تتبعهُ حتى يستوعبه دماغك. عند هذه النقطة، ستجد نفسك تتقبّل المعرفة الجديدة تلقائيًا.

بناء محفظتك

استثمر بانتظام

عليك الاستثمار في محفظتك المعرفيّة بانتظام تمامًا كما هو الحال في الاستثمار الماليّ، حتى لو كان مبلغًا صغيرًا. هـذه العـادة لا تقل أهميّة عن المبالغ، لذا خطط لاستخدام وقتٍ ومكانٍ متسقين بعيدًا عن الانقطاعات. سنسرد بعض عيّنات الأهداف في القسم التالي.

التنوع

كلما ازدادت معرفتك بأشياء مختلفة، ازدادت قيمتـك. كقاعـدة أساسيّة، تحتـاج إلى معرفـة التفاصـيل لـتقنية معيّنـة تسـتخدمها حاليًا. لكن لا تتوقف عند هذا الحد. يتغير وجه الحوسبة بسرعة — قد تكون التقنية الساخنة اليوم قريبةً جدًا من عدم الجدوى (أو في الأقل ليست مطلوبةً) غدًا. كلما زاد عدد التقنيات التي اطلعت عليها، كلما أمكنك التكيّـف مـع التغيـير. ولا تنس جميـع المهـارات الأخـرى التي تحتاجها، بما في ذلك المهارات في المجالات غير التقنية.

إدارة المخاطر

توجد التِقَانَة على طول نطاق محتمل من المعايير الخطرة، والمحتمل عاليةِ المـردود، إلى المعـايير منخفضـةِ المخـاطر ومنخفضـة المردود.

ليست فكرةً جيّدةً أن تستثمر كل أموالك في الأسهم عالية المخاطر التي قـد تنهـار فجـأة، ولا يجب عليـك اسـتثمارها كلهـا بشـكلٍ متحفظٍ وتفويت الفرص المحتملة. لا تضع كل بيضك التقني في سلّةٍ واحدة.

اشتر برخص، بع بغلاء

يمكن أن يكون تعلّم تقنية ناشئة قبـل أن تصبح شـعبية بنفس صـعوبة العثـور على مخـزون مُقيّم بأقـل من قيمتـه، ولكن المـردود يمكن أن يكون مُجزياً بنفس القدر. ربما كان تعلُّم *جافا* سابقًا عندما قُدّمت لأول مـرةٍ وكـانت غـير معروفـةٍ محفوفًـا بالمخـاطر في ذلـك الوقت، لكنها أثمرت بشكل رائع للمتبنين في وقتٍ مبكر عندما أصبحت دعامةً أساسيّةً للصناعة في وقتٍ لاحق.

المراجعة وإعادة التوازن

هذه صناعة ديناميكية للغاية. قد تكون هذه التِقَانَة الساخنة التي بدأتم بحثها الشهر الماضي ببرودة الحجـر الآن. ربمـا تحتـاج إلى تحسين تِقَانَـة قاعـدة البيانـات الـتي لم تسـتخدمها منـذ مـدّة. أو ربمـا تكـون في وضـعٍ أفضـل للوظيفـة الجديـدة إذا جـربت تلـك اللغـة الأخرى....

من بين جميع هذه المبادئ الإرشادية، فإن أهمّها أكثرها بساطةً بالتنفيذ.

نصيحة ٩	استثمر بانتظام في محفظتك المعرفيّة

الأهداف

الآن بعد أن أصبحت لديك بعض الإرشادات تتعلق بماذا ومتى تضيف إلى محفظتك المعرفيّة، ما هي أفضل طريقة اكتساب رأس المال الفكري لتمويل محفظتك؟ فيما يلي بعض الاقتراحات:

تعلّم لغةً جديدةً واحدة في الأقل كل عام

تحل لغات مختلفة نفس المشكلات بطرقٍ مختلفة. يمكنك بسبب تعلّم العديد من المناهج المختلفة، المساعدة على توسيع تفكيرك وتجنّب شرك الرتابة. إضافة إلى ذلك يُعدّ تعلّم العديد من اللغات أمرًا سهلاً بفضل وفرة البرمجيات المتاحة مجانًا.

اقرأ كتابًا تقنيًا كل شهر

بالرغم وجود وفرة من المقالات القصيرة والإجابات الموثوقة في بعض الأحيان على الويب، إلا أنك بحاجةٍ إلى كتبٍ طويلةٍ من أجل الفهم العميق. تصفّح مواقع بائعي الكتب بحثًا عن كتب تقنية حول مواضيع مثيرةٍ للاهتمام تتعلق بمشروعك الحالي[13]. وبمجرّد أن تعتاد ذلك، اقرأ كتابًا في الشهر. تفرّغ لدراسة بعض الأشياء التي لا تتعلق بمشروعك بعد أن تتقن التقنيات التي تستخدمها حاليًا.

اقرأ كتبًا غير تقنية أيضًا

من المهم أن نتذكّر أن أجهزة الحاسوب تستخدم من قبل أناس - النّاس الذين تحاول تلبية احتياجاتهم. أنت تعمل مع النّاس، وتوظّف من قبل النّاس، وتُخترق من قبل النّاس. لا تنس الجانب الإنساني من المعادلة، لأن ذلك يتطلب مجموعةَ مهاراتٍ مختلفةٍ تمامًا (للسخرية أننا نسميها مهارات ناعمة، ولكنها في واقع الأمر مهارات يصعب إتقانها).

خذ دروسًا

ابحث عن دورات مثيرة للاهتمام في كليّة أو جامعة محليّة أو عبر الإنترنت، أو ربما في المَعْرِض أو المؤتمر التجاري القريب التالي.

اشترك في مجموعات المستخدمين المحلية واللقاءات

يمكن أن تكون العزلة مميتة في حياتك المهنيّة. تعرّف إلى ما يفعله النّاس خارج شركتك. لا تذهب وتستمع فقط: شارك بنشاط.

جرّب بيئات مختلفة

إذا كنت تعمل فقط في نظام ويندوز Windows، اقضِ بعض الوقت مع نظام لينكس Linux. إذا كنت قد استخدمت أداة makefiles ومحرّرًا فقط، فجرّب بيئة تطويرٍ متكاملة IDE معقّدة مع ميزات متطوّرة، والعكس صحيح.

ابق حاضرًا

اقرأ الأخبار والمشاركات عبر الإنترنت حول تقنية مختلفة عن تلك الموجودة في مشروعك الحالي. إنها طريقة رائعة لاكتشاف التجارِب التي يمرّ بها الأشخاص الآخرون، والمصطلحات الخاصّة التي يستخدمونها، وما إلى ذلك.

من المهم الاستمرار في الاستثمار. بمجرّد أن تعتاد لغةً جديدةً نوعا ما أو تقنيّةً ما، تابع قدمًا. تعلّم واحدةً أخرى.

13 قد نكون منحازين، ولكن هناك مجموعة مُختارة جيّدة متاحة على https://pragprog.com.

لا يهم إذا لم تستخدم أيًا من هذه التِقَانَات في أي مشروع، أو حتى لم تضعها في سيرتك الذاتية. ستوسع عملية التعلّم تفكيرك، وتفتح لك إمكانات جديدة وطرقًا جديدةً للقيام بالأشياء. إن تلاقح الأفكار أمرٌ مهمّ؛ حاول تطبيق الدروس التي تعلمتها على مشروعك الحالي. حتى إذا كان مشروعك لا يستخدم هذه التِقَانَة، فربما يمكنك استعارة بعض الأفكار. تعرّف إلى كائنيّة التوجّه، على سبيل المثال، وستكتب برمجيات إجرائيّة بشكل مختلف. افهم نموذج البرمجة الوظيفيّة وستكتب شِفرة كائنيّة التوجه بشكل مختلف، وهكذا.

فرص للتعلّم

بينما تقرأ بنهم، وتواكب آخر التطورات العاجلة في مجالك (هذا ليس بالأمر السهل)، قام أحدهم بطرح سؤال عليك. ليس لديك أدنى فكرة عن جوابه، وتعترف بنفس القدر من الصراحة.

لا تدع الأمر يتوقف عند هذا الحد. عدّه تحديًا شخصيًا للعثور على الإجابة. اسأل من حولك. ابحث في الويب ليس فقط في الجوانب الاستهلاكيّة. ولكن العلميّة أيضًا.

إذا لم تتمكن من العثور على الإجابة بنفسك، فاكتشف من يستطيع. لا تدع الأمر يرقد. سيساعدك التحدث إلى أشخاص آخرين في بناء شبكتك الشخصيّة، وقد تفاجئ نفسك في أثناء وِجدان حلول لمشكلات أخرى غير ذات صلة على طول الطريق. وتلك المحفظة القديمة تواصل الاتساع....

تستغرق كل هذه القراءة والبحث وقتًا، والوقت قليلٌ فعلًا. لذلك تحتاج إلى التخطيط للمستقبل. دائما لديك شيء لقراءته في لحظةٍ ضائعةٍ بشكلٍ أو بآخر. يمكن أن يكون الوقت الذي تقضيه في انتظار الأطباء وأطباء الأسنان فرصةً رائعةً لمتابعة قراءتك - ولكن تأكّد إحضار قارئك الإلكتروني معك، أو قد تجد نفسك تقلّب صفحات مقالةٍ من عام 1973 عن بابوا غينيا الجديدة.

التفكير النقدي

تتمثّل النقطة المهمّة الأخيرة في أن تفكّر تفكيرًا ناقدًا فيما تقرأه وتسمعه. تحتاج إلى التأكّد أنّ المعرفة في محفظتك دقيقة وغير متأثرة بأيّ من الموردين أو الضجيج الإعلامي.. احذر من المتعصبين الذين يصرّون على أن معتقدهم يقدّم الجواب الوحيد - قد ينطبق أو لا ينطبق عليك وعلى مشروعك.

لا نقلّل من قوة النزعة التجاريّة. فمجرّد وجود النتيجة في أعلى نتائج محرك البحث لا يعني أنها أفضل المطابقات؛ يمكن لموفر المحتوى الدفع للحصول على أعلى ظهور. فقط لأن مخزن كتب يميّز كتابًا بشكلٍ بارز لا يعني أنّه كتابٌ جيّد، أو حتى شائع؛ ربما تم الدفع لوضعه في ذاك المكان.

نصيحة ١٠ | **حلّل بشكل ناقد ما تقرأه وتسمعه**

التفكير الناقد هو نظام كامل في حد ذاته، ونحن نشجعك على قراءة ودراسة كلّما تستطيع عنه. في هذه الأثناء، إليك البداية ببعضِ الأسئلة التي يمكنك طرحها والتفكير فيها.

اسأل "لماذا خمس مرات"

إنها خدعة استشاريّة مفضّلة: اسأل "لماذا؟" خمس مرات في الأقل. اطرح سؤالاً واحصل على إجابة. تعمّق أكثر بالسؤال "لماذا؟" كرّر الأمر كما لو كنت طفلاً بعمر الأربع سنين (ولكن مهذب). قد تكون قادرًا على الاقتراب من سبب جذري بهذه الطريقة.

من المستفيد من هذا؟

قد يبدو الأمر هزليًا، لكن تتبع المال يمكن أن يكون مسارًا مفيدًا جدًا للتحليل. قد تتماهى الفوائد التي تعود شخص آخر أو منظمة أخرى مع مصالحك الخاصّة أو قد لا تتماهى.

ما هو السياق؟

إن كل شيء يحدث في سياقه الخاص، وهذا هو السبب في أن "مقاس واحد يناسب الجميع" لا تعمل غالبًا. فكّر في مقال أو كتاب يروّج لـ "أفضل الممارسات". الأسئلة الجيّدة التي يجب مراعاتها هي "الأفضل لمن؟" ما هي الشروط المسبقة، وما هي العواقب، على المدى القصير والطويل؟

متى أو أين سيعمل هذا؟

تحت أية ظروف؟ هل هو متأخر كثيرًا؟ أم سابق لأوانه؟ لا تتوقف عن التفكير من الدرجة الأولى (ما الذي سيحدث بعد ذلك)، ولكن استخدم التفكير من الدرجة الثانية: ماذا سيحدث بعد ذلك؟

لماذا تُعدّ هذه مشكلة؟

هل هناك نموذج ضمنيّ؟ كيف يعمل النموذج الضمني؟

لسوء الحظ، لم يعد هناك سوى عدد قليل من الإجابات البسيطة. ولكن مع محفظتك واسعة النطاق، وتطبيق بعض التحليل النقدي على سيلٍ من المواد التقنية التي ستقرأها، يمكنك فهم الإجابات المعقّدة.

الأقسام ذات الصلة

- الموضوع 1، إنها حياتك، في الصفحة 27
- الموضوع 22، دفاتر يوميات الهندسة، في الصفحة 121

تحدّيات

- ابدأ في تعلم لغة جديدة هذا الأسبوع. تُبرمج دائما بنفس اللغة القديمة؟ جرّب كلوجر Clojure، إكسير Elixir، إي إل إم Elm، إف شارب F ، جو Go، هاسكل Haskell، بايثون Python، أرِ R، ريزون إم إل ReasonML، روبي Ruby، رَست Rust، سكالا Scala، سويفت Swift، تايب سكريبت TypeScript أو أي شيء آخر يثير إعجابك و / أو يبدو أنه سيعجبك[14].

14 لم تسمع بأي من هذه اللغات؟ تذكّر أنّ المعرفة أصل منتهي الصّلاحِيَة، وكذلك التِقَانَة الشائعة.
كانت قائمة اللغات التجريبيّة والجديدة مختلفة تمامًا في الإصدار الأول، وربما تختلف مرّة أخرى بحلول الوقت الذي تقرأ فيه هذا.كل هذا سبب لمواصلة التعلم.

- ابدأ بقراءة كتاب جديد (ولكن أنهي هذا الكتاب أولاً!). إذا كنت تقوم بتنفيذ وكتابة شِفرة مفصّلين للغاية، فاقرأ كتابًا عن التصميم والهيكلة. إذا كنت تنفذ تصميمًا عالي المستوى، فاقرأ كتابًا عن تقنيات كتابة الشِفرة.
- اخرج وتحدّث عن التِقَانَة مع أشخاص غير مشاركين في مشروعك الحالي، أو الذين لا يعملون في نفس الشركة. كوّن علاقات ضمن كافتيريا شركتك، أو ابحث عن زملاء متحمسين في لقاءٍ محليّ.

الموضوع 7. تواصَل!

> أعتقد أنّه من الأفضل أن يُلقى عليك نظرة سريعة من أن يتم تجاهلك
> ← ماي ويست، حسناء التسعينيات، 1934

ربما يمكننا تعلّم درس من السيدة ويست. لا يقتصر الأمر على ما لديك فحسب، بل أيضًا كيف تحزمه. إن الحصول على أفضل الأفكار، أو أفضل الشفرات، أو التفكير الأكثر عمليةً هو في النهاية عقيمٌ ما لم تتمكن من التواصل مع الآخرين. تبقى الفكرة الجيّدة يتيمةً دون تواصل فعّال.

يجب علينا كمطوّرين التواصل على مستويات عدّة. نقضي ساعات في الاجتماعات والاستماع والتحدث. نعمل مع المستخدمين النهائيين، محاولين فهم احتياجاتهم. نكتب الشِفرة، التي تنقل نوايانا إلى الآلة وتوثّق تفكيرنا للأجيال القادمة من المطوّرين. نكتب مقترحات ومذكرات تطلب وتبرير الموارد، وتبلغ عن وضعنا، وتقترح نُهجًا جديدة. ونعمل يوميًا ضمن فرقنا للدفاع عن أفكارنا وتعديل الممارسات الحاليّة واقتراح ممارسات جديدة. نمضي جزءًا كبيرًا من يومنا على التواصل، لذا نحتاج إلى فعل ذلك بشكلٍ جيّد.

تعامل مع اللغة الإنجليزية (أو أياً كانت لغتك الأم) كلغة برمجة أخرى. اكتب اللغة الطبيعية كما تكتب الشِفرة: احترام مبدأ DRY وETC والأتمتة وما إلى ذلك. (نناقش مبادئ تصميم DRY وETC في الفصل التالي.)

نصيحة ١١	**اللغة الإنجليزية هي مجرّد لغة برمجة أخرى**

لقد جمّعنا قائمة بأفكارٍ إضافيّةٍ نجدها مفيدة.

اعرف جمهورك

أنت تتواصل فقط إذا كنت توصل ما تقصد أن توصله - مجرّد الكلام لا يكفي. تحتاج للقيام بذلك، إلى فهم احتياجات ومصالح وقدرات جمهورك. لقد جلسنا جميعًا في اجتماعات حيث يتأمل مهووس بالتنمية أعين نائب رئيس التسويق مع مونولوجٍ طويلٍ حول مزايا بعض التقنيات الغامضة. هذا ليس تواصلا: إنّه مجرّد كلام، ومزعج[15].

لنفترض أنك تريد تغيير نظام المراقبة عن بُعد لاستخدام وسيط رسائل تابع لجهة خارجية لنشر إشعارات الحالة. يمكنك تقديم هذا التحديث اعتمادًا على جمهورك بعدّة طرق مختلفة. سيقدّر المستخدمون النهائيون أن أنظمتهم يمكن أن تتفاعل الآن مع الخِدمات الأخرى التي تستخدم الوسيط. سيتمكن قسم التسويق من استخدام هذه الحقيقة لزيادة المبيعات. سيكون مديرو التطوير والعمليات سعداء لأن رعاية هذا الجزء من النظام وصيانته هي الآن مشكلة شخص آخر. أخيرًا، قد يستمتع المطوّرون بالحصول على الخبرة مع واجهات برمجة التطبيقات الجديدة، وقد يتمكنون حتى من العثور على استخدامات جديدة لوسيط الرسائل. ستجعلهم بواسطة تقديم عرض تقديمي ملائم لكل مجموعة، متحمسين بشأن مشروعك.

15 كلمة "إزعاج" تأتي من اللغة الفرنسية القديمة "enui"، والتي تعني أيضًا "أن تكون مملا".

إن الحيلة هنا كما هو الحال مع جميع أشكال التواصل، هي في جمع الملاحظات. لا تنتظر فقط الأسئلة: اسألها. انظر إلى لغة الجسد وتعابير الوجه. إحدى المسلمات للبرمجة اللغوية العصبية هي "المعنى من اتصالك هو مقدار الاستجابة التي تَحَصّلها." حسّن معرفتك بجمهورك باستمرار في أثناء التواصل.

اعرف ما تريد قوله

ربما يكون الجزء الأكثر صعوبةً من أساليب الاتصال الأكثر رسميّة المستخدمة في الأعمال هو تحديد ما تريد قوله بالضبط. غالبًا ما يرسم كتّاب الخيال كتبهم بالتفصيل قبل أن يبدؤوا، لكن الأشخاص الذين يكتبون مستندات تقنية يسعدهم غالبًا الجلوس على لوحة المفاتيح، ثم الضغط على زر الإدخال Enter:

1. المقدمة

والبَدْء في كتابة ما يدور في رؤوسهم بعد ذلك.

خطط لَمَا ترد أن تقوله. اكتب مخططًا تفصيليًا. ثم اسأل نفسك، "هل ينقل هذا ما أريد أن أعبّر عنه لجمهوري بطريقة تناسبهم؟" حسّنه ليصبح كذلك.

يصلح هذا النهج لأكثر من مجرّد التعامل مع الوثائق فقط. فعندما تواجه اجتماعًا مهمًا أو محادثةً مع عميلٍ رئيس، دوّن الأفكار التي ترغب في توصيلها، وخطط لبضع استراتيجيات من أجل توصيلها.

الآن بعد أن عرفت ما يريده جمهورك، دعنا نقدّمه.

اختر لحظتك

إنها الساعة السادسة من مساء يوم الجمعة، بعد أسبوع من دخول مدققي الحسابات. الابن الأصغر سنًا لرئيستك في المستشفى، والمطر يهطل في الخارج، ومن المؤكّد أن يكون التنقل كابوسًا. ربما هذا ليس وقتًا جيّدًا لمطالبتها بترقية ذاكرة حاسوبك المحمول.

تحتاج إلى تحديد أولويات جمهورك كجزء من فهم ما يحتاجون لسماعه. جِد مديرةً كانت قد تعرّضت للتو لمضايقة من رئيسها بسبب فقدان بعضا من شِفرة المصدر، وسيكون لديك مستمع أكثر تقبّلاً لأفكارك حول مستودعات الشِفرة المصدريّة. اجعل ما تقوله مناسبًا في الوقت كما في المحتوى. كل ما يتطلّبه الأمر في بعض الأحيان هو السؤال البسيط، "هل هذا هو الوقت المناسب للحديث عن...؟"

اختر نمطاً

اضبط نمط تقديمك ليناسب جمهورك. يريد بعض الأشخاص عرضًا موجزًا رسميًا "مجرّد حقائق". والبعض الآخر يحب الدردشة الطويلة والواسعة النطاق قبل الشروع في العمل. ما مستوى مهاراتهم وخبراتهم في هذا المجال؟ هل هم خبراء؟ مبتدئين؟ هل يحتاجون إلى أن تأخذ بيدهم خطوةً بخطوة أم لمجرد رحلةٍ سريعة؟ اسأل إذا كان هناك شك في ذلك.

تذكّر، مع ذلك، أنك نصف عملية الاتصال. إذا قال شخص ما أنه يحتاج إلى فَقَرة تصف شيئًا ولا يمكنك رؤية أي طريقة للقيام بذلك في أقل من عدّة صفحات، فأخبره بذلك. تذكّر أن هذا النوع من الملاحظات هو صورة من صور التواصل أيضًا.

اجعلها تبدو جيدة

إن أفكارك مهمّة. وتستحق وسيلة جيدة لنقلها إلى جمهورك.

يركّز الكثير من المطوّرين (ومديريهم) فقط على المحتوى عند إنتاج مستندات مكتوبة. نعتقد أن هذا خطأ. سيخبرك أي طاهي (أو مراقب شبكة الغذاء) أنه بإمكانك أن تخدم في المطبخ لساعات لتدمر جهودك فقط بسبب ضعف طريقة التقديم.

لا يوجد عذر اليوم لإنتاج وثائقَ مطبوعةٍ سيئةَ المظهر. يمكن أن تُنتج البرامج الحديثة مخرجات مذهلة، بغض النظر عما إذا كنت تكتب باستخدام تنسيق مارك داون Markdown أو تستخدم معالج النصوص. تحتاج إلى تعلّم بعض الأوامر الأساسيّة فقط. إذا كنت تستخدم معالج نصوص، فاستخدم أوراق الأنماط الخاصّة به من أجل الاتساق. (قد تكون شركتك لديها فعلًا أوراق أنماط محدّدة يمكنك استخدامها.) تعرّف إلى كيفية تعيين رؤوس الصفحات وتذييلاتها. انظر إلى نماذج المستندات المضمّنة في الحُزمَة الخاصّة بك للحصول على أفكار حول الأسلوب والتخطيط. تحقّق الإملاء تلقائيًا أولاً ثم يدويًا. بعد كل ذلك، هناك أخطاء تهجئة يمكن للمدقّق الإملائي أن يحلّها.

أشرك جمهورك

غالبًا ما نجد أن المستندات التي ننتجها في نهاية المطاف أقل أهميّة من العملية التي نمرّ بها لإنتاجها. أشرك القراء في المسودات الأولى للمستند إذا كان ذلك ممكنًا. احصل على ملاحظاتهم، واستشرهم. ستبني عَلاقة عمل جيّدة، وربما ستنتج وثيقةً أفضل في هذه العمليّة.

كن مستمعًا

هناك أسلوب واحد يجب عليك استخدامه إذا كنت تريد أن يستمع إليك النّاس: استمع إليهم. حتى إذا كان موقفًا لديك حوله جميع المعلومات، حتى لو كان اجتماعًا رسميًا وأنت تقف أمام طاقم من 20 شخصًا - إذا لم تستمع إليهم، فلن يستمعوا إليك.

شجّع الأشخاص على التحدث عن طريق طرح الأسئلة، أو اطلب منهم إعادة صياغة المناقشة بكلماتهم الخاصّة. حوّل الاجتماع إلى جلسة حوار، وستجعل وجهة نظرك فعّالة أكثر. من يدري، حتى أنّك قد تتعلّم شيئا ما.

عُد إلى النّاس

إذا سألت شخصًا ما سؤالا، ستشعر بأنه غير مهذب إذا لم يستجب لك. ولكن كم مرة أخفقت في العودة إلى النّاس عندما يرسلون إليك بريدًا إلكترونيًا أو مذكّرة تطلب معلومات أو تطلب عملاً ما؟ إنّه من السهل أن تنساها في اندفاع الحياة اليوميّة. أجب دائماً عن رسائل البريد الإلكتروني والرسائل الصوتيّة، حتى إذا كان الرد ببساطة "سأعود إليك لاحقًا". إن إبقاء الأشخاص على اطلاع يجعلهم أكثر تسامحًا للمرور العرضيّ، ويجعلهم يشعرون أنك لم تنساهم.

نصيحة ١٢ — إنها على حد سواء ما تقوله والطريقة التي تقوله بها

يجب أن تكون قادرًا على التواصل، ما لم تكن تعمل في الفراغ. وكلما زادت فعاليّة هذا التواصل، أصبحت أكثر تأثيرًا.

التوثيق

أخيرًا، هناك مسألة التواصل عبر التوثيق. عادةً لا يولي المطوّرون التوثيق كثيرًا من التفكير. إنها في أفضل أحوالها ضرورة غير سارة. في أسوأ الأحوال يتم التعامل معها على أنها مهمّة ذات أولويّة منخفضة على أمل أن تنسى الإدارة ذلك في نهاية المشروع.

يتبنى المبرمجون العمليون التوثيق كجزءٍ لا يتجزأ من عمليـة التطـوير الشـاملة. يمكن تسـهيل كتابـة الوثـائق بفضـل عـدم تكـرار الجهود أو إضاعة الوقت، وبإبقاء الوثائق في متناول اليد - في الشِفرة نفسها. نريد في الواقع تطبيق جميع مبادئنا العملية على التوثيق وكذلك على الشِفرة.

نصيحة ١٣ أنشئ الوثائق، لا تضف عليها

من السهل إنتاج وثائق جيّدة المظهر من التعليقات في الشِفرة البرمجيّة المصدريّة، ونوصي بإضافة التعليقات إلى الوحدات النمطيّة modules والدّالاّت functions المُصدَرة للأخذ بيد المطوّرين الآخرين عندما يستخدمونها.

ومع ذلك، هذا لا يعني أننا نتفق مع الأشخاص الذين يقولون أنّ كل دالّة، وهيكل بيانات، وتصريح عن النوع، وما إلى ذلك، تحتاج إلى تعليقها الخاص. في الواقع هذا النوع من الكتابة الميكانيكية للتعليقات يُصعّب صيانة الشِفرة: فالآن هناك شيئان يجب تحديثهما عند إجراء تغيير ما. لذا قيّد تعليقاتك غير المتعلقة بواجهة برمجة التطبيقات (API) لمناقشة لماذا قمنا بشيء ما والغرض منه وهدفه. تُظهر الشِفرة فعلًا كيفيّة فعل ذلك، لذا فإن التعليق على هذا أمرًا فائضًا - وانتهاكا لمبدأ DRY.

يمنحك التعليق على شِفرة المصدر الفرصة المثاليّة لتوثيق تلك الأجزاء المربكة من مشروع ما حيث لا يمكن توثيقها في أي مكان آخر: المقايضات الهندسيّة، وسبب اتخاذ القرارات، وما هي الأبدال الأخرى التي تم تجاهلها، وما إلى ذلك.

ملخص

- اعرف ما تريد قوله.
- اعرف جمهورك.
- اختر لحظتك.
- اختر النمط.
- اجعلها تبدو جيدة.
- أشرك جمهورك.
- كن مستمعا.
- عد إلى النّاس.
- احتفظ بالشِفرة والوثائق مع بعضهم البعض.

الأقسام ذات الصلة

- الموضوع 15، التقدير، في الصفحة 88
- الموضوع 18، قوّة التعديل، صفحة 102
- الموضوع 45، هوة المتطلبات، في الصفحة 268
- الموضوع 49، الفِرق العملية، صفحة 289

التحديات

- يوجد العديد من الكتب الجيّدة التي تحتوي على أقسام حول الاتصالات داخل الفِرق، بما في ذلك *شهر الرجل الأسطوري:* *مقالات حول هندسة البرمجيات* The Mythical Man-Month: Essays on Software Engineering [Bro96] *والبرمجيات الشعبية: المشروعات والفرق المنتجة* Peopleware: Productive Projects and Teams [DL13]. اجعلها وجهة لمحاولة قراءتها خلال الـ 18 شهرًا القادمة. إضافة إلى ذلك، *أدمغة الديناصورات: التعامل مع كل أولئك الذين لا يطاقون في العمل* Dinosaur Brains: Dealing with All Those Impossible People at Work [BR89] يناقش العبء العاطفي الذي نحمله جميعًا إلى بيئة العمل.
- في المرّة القادمة التي يتعيّن عليك فيها تقديم عرض تقديميّ، أو كتابة مذكرةٍ تدافع عن موقفٍ معين، حاول العمل انطلاقًا من النصائح الواردة في هذا القسم قبل البَدء. طابق بكل وضوحٍ بين حاجتك للتواصل وحاجة جمهورك. إذا لزم الأمر، تحدّث إلى جمهورك بعد ذلك وشاهد مدى دقّة تقييمك لاحتياجاتهم.

التواصل عبر الإنترنت

كل ما قلناه عن التواصل كتابةً ينطبق على حد سواء على البريد الإلكتروني ومنشورات وسائل التواصل الاجتماعي والمدونات وما إلى ذلك. قد تطوّر البريد الإلكتروني على وجه الخصوص إلى درجة كونه الدعامة الأساسيّة للاتصالات المؤسسيّة؛ يُستخدم لمناقشة العقود وتسوية النزاعات وكدليل في المحكمة أيضًا. ولكن ولسبب ما، فإن الأشخاص الذين لن يرسلوا أبدًا وثيقةً ورقيةً رثّة سعداء بإطلاق رسائل بريد إلكتروني سيئة المظهر وغير مترابطة في أنحاء العالم.

نصائحنا بسيطة:

- راجع قبل أن تضغط على زر إرسال.
- تحقّق الإملاء الخاص بك وابحث عن أيّة مشكلات عرضيّة بالتصحيح الإملائي.
- حافظ على تنسيق بسيط وواضح.
- ابقِ على الاقتباس في الحد الأدنى. لا أحد يحبّ أن يعود ويتلقى بريده الإلكتروني المؤلف من 100سطر مرفق به عبارة "أنا أوافق".
- إذا كنت تقتبس بريدًا إلكترونيًا للآخرين، فتأكّد إسناده إليهم، واقتباسه مُضمنًا (بدلاً من إرفاقه). وبالمثل عند الاقتباس في منصّات وسائل التواصل الاجتماعي.

- لا تهاجم وتتصرف كمتصيّد إنترنت "ترول" إلا إذا أردت له أن يؤرقك لاحقاً. إذا لم تقلها بوجه شخصٍ ما، لا تقلها على الإنترنت.
- تحقّق قائمة المستلمين قبل الإرسال. لقد أصبح من الأمثلة الشائعة أن تنتقد المدير على البريد الإلكتروني للإدارة دون أن تدرك أن بريد المدير مُدرج في قائمة نسخة كربونية CC. والأفضل من ذلك، لا تنتقد المدير عبر البريد الإلكتروني.

إن رسائل البريد الإلكتروني ووسائل التواصل الاجتماعي تبقى إلى الأبد كما اكتشف عدد لا يحصى من الشركات والسياسيين. حاول إيلاء البريد الإلكتروني نفس الاهتمام والعناية التي توليها لأيّة مذكّرةٍ أو تقريرٍ مكتوب.

الفصل الثاني:

نهج عملي

هناك بعض النصائح والحيل التي تنطبق على جميع مستويات تطوير البرمجيات، والعمليات العالميّة فعليًا، والأفكار البديهيّة بالغالب. ومع ذلك، نادرًا ما تُوثّق هذه النُهج على هذا النحو؛ ستجدها في الغالب مكتوبةً كجملٍ غريبة في مناقشات التصميم أو إدارة المشروع أو كتابة الشِفرة. ولكن لراحتك، سنجمع هنا هذه الأفكار والعمليات معًا.

الموضوع الأول وربما الأكثر أهميّة يصبّ في صُلب تطوير البرمجيات: جوهر التصميم الجيّد. كل شيء يستتبع هذا.

القسمان التاليان، DRY — شرور التكرار و التعامد، مرتبطان ارتباطًا وثيقًا. يحذّرك الأول من تكرار المعرفة ضمن أنظمتك، والثاني من تقسيم أيّة معلومة عبر مكونات النظام المتعدّدة.

ومع زيادة وتيرة التغيير، يصبح إبقاء تطبيقاتنا ساريةً لليوم صعبًا أكثر فأكثر. في قابلية العكس، سنلقي نظرة على بعض التقنيات التي تساعد في عزل مشروعاتك عن بيئتها المتغيّرة.

القسمان التاليان مرتبطان أيضًا. نتحدث في الرصاصة الخطّاطة عن أسلوب تطوير يسمح لك بجمع المتطلبات وتصاميم الاختبار وتنفيذ التعليمات البرمجيّة في نفس الوقت. إنها الطريقة الوحيدة لمواكبة وتيرة الحياة الحديثة.

توضح لك النماذج الأوليّة والملاحظات الملحقة كيفيّة استخدام النماذج الأوليّة لاختبار الهيكلية والخوارزميات والواجهات والأفكار. من المهم في العالم الحديث اختبار الأفكار والحصول على التعليقات قبل أن نلتزم بها بشكلٍ كامل.

مع النُضج البطيء لعلوم الحاسوب، ينتج المصممون لغاتٍ ذات مستوى أعلى بشكل متزايد. مع أنّ المترجم الذي يقبل "اجعله كذلك" لم يُخترع بعد، إلا أنّنا نقدّم في لغات النطاق بعض الاقتراحات الأكثر تواضعًا التي يمكنك تنفيذها بنفسك.

أخيرًا، نعمل جميعًا في عالم محدود الوقت والموارد. يمكنك النجاة من هذه الندرة بشكل أفضل (والحفاظ على مديرك أو عملائك سعداء أكثر) إذا كنت جيّدًا في معرفة الوقت الذي ستستغرقه الأمور، والتي نغطيها في التقدير.

ضع هذه المبادئ الأساسيّة في الاعتبار أثناء التطوير، وستكتب شِفرةً أفضل وأسرع وأقوى. يمكنك حتى جعلها تبدو سهلة.

الموضوع 8. جوهر التصميم الجيّد

العالم مليء بالمعلمين والنقّاد، وجميعهم حريصون على تمرير حكمتهم المكتسبة بشق الأنفس عندما يتعلق الأمر بكيفيّة تصميم البرمجيات. هناك اختصارات وقوائم (التي يبدو أنها تفضّل خمس إدخالات) وأنماط ورسوم بيانيّة ومقاطع فيديو ومحادثات و (الإنترنت هو الإنترنت) ربما تكون سلسلة رائعة من قانون ديميتر[16] موضّحة باستخدام الرقص التفسيري.

ونحن، مؤلفوكم اللطيفون، مذنبون أيضًا. لكننا نودّ أن نصحّح الأمور بواسطة توضيح شيء لم يظهر لنا إلا مؤخرًا إلى حدّ ما. أولًا: التعليمة العامّة:

نصيحة ١٤	التصميم الجيّد يسهل تغييره أكثر من التصميم الرديء

16 **[المترجم]** قانون ديميتر Law of Demeter (LoD) أو مبدأ المعرفة الأقل هو دليل تصميم لتطوير البرمجيات، وخاصّة البرمجيات كائنية التوجه. يُعرف قانون ديميتر باسم "لا تتحدث إلى الغرباء" لأنه: يجب أن يكون لكل وحدة معرفة محدودة فقط بالوحدات الأخرى – فقط الوحدات "وثيقة الصلة" بالوحدة الحاليّة.

يكون الشيء مُصممًا جيدًا إذا كان يتكيّف مع الأشخاص الذين يستخدمونه. وبالنسبة للشِفرة، هذا يعني أن عليها أن تتكيّف مع التغيير. لذلك نحن نؤمن بمبدأ ETC: **أسهل في التغيير**. (**E**asier **t**o **C**hange (ETC)) هو كذلك.

بقدرٍ ما يمكننا أن نقول، في كل مبدأ من مبادئ التصميم هناك حالة خاصّة من ETC.

لماذا مبدأ الفصل جيّد؟ لأنه بواسطة عزل المخاوف، نجعل كل منها أسهل في التغيير. *ETC*.

لماذا يعدّ مبدأ المسؤوليّة الفرديّة مفيدًا؟ لأن التغيير في المتطلبات ينعكس بتغيير في وحدة نمطيّة واحدة فقط. *ETC*.

لماذا تعدّ التسمية مهمّة؟ لأن الأسماء الجيّدة تجعل قراءة التعليمات البرمجيّة أسهل، ويجب عليك قراءتها لتغييرها. *ETC*!

ETC قيمة وليس قاعدة

القيم هي الأشياء التي تساعدك في اتخاذ القرارات: هل يجب أن أفعل هذا أم ذاك؟ عندما يتعلق الأمر بالتفكير في البرمجيات، فإن ETC هو دليل يساعدك على الاختيار بين المسارات. يجب أن يكون عائمًا خلف تفكيرك الواعي، ويدفعك بمهارة في الاتجاه الصحيح، تمامًا مثل جميع قيمك الأخرى.

لكن كيف يمكنك تحقيق ذلك؟ من تجربتنا الأمر يتطلب بعض التعزيز الواعي الأولي. قد تحتاج إلى قضاء أسبوع أو نحو ذلك تسأل نفسك عامدًا "هل الشيء الذي أقوم به من شأنه جعل النظام الكلي أسهل أم أصعب للتغيير؟" افعل ذلك عندما تكتب اختبارًا. تحفظ ملفًا، تُصحّح خطأ برمجيًا.

هناك فرضيّة ضمنيّة في ETC. تفترض أنه يمكن للشخص تقرير أي من مسارات متعددة سيكون من الأسهل تغييره في المستقبل. في معظم الأوقات، يكون الحس العام صحيح، ويمكنك أن تخمّن.

بعض الأوقات حين لا يمكنك استشعار دلالات، لا مشكلة، نعتقد في حالات كهذه أنه يمكنك القيام بأمرين:

الأول، نظرًا لأنك لست متأكّدًا من الشكل الذي سيتخذه التغيير، يمكنك دائمًا التراجع إلى المسار الأقصى ل "سهولة التغيير": حاول جعل ما تكتبه قابلًا للاستبدال. بهذه الطريقة، مهما حدث في المستقبل، لن يكون هذا الجزء من الشِفرة عثرةً على الطريق.

إنه فعلًا مجرّد التفكير بالحفاظ على الشِفرة مفصولةً ومتسقةً.

الثاني، تعامل مع هذا كطريقة لتنمية الحدس. دوّن الحالة في دفترك الهندسي اليومي: الخيارات المتاحة لك، وبعض التخمينات حول التغيير. اترك إشارة في المصدر ثم، لاحقًا، عندما يتعيّن عليك تغيير الشِفرة، يمكنك الرجوع وتقييم نفسك. هذا قد يساعد في المرّة القادمة التي تصل فيها إلى تفرعات مماثلة في الطريق.

لدى بقيّة الأقسام في الفصل أفكارًا محدّدة في التصميم، لكنها مستمدّة جميعها من هذه القاعدة الوحيدة.

الأقسام ذات الصلة

- الموضوع 9، DRY - شرور التكرار، في الصفحة 54
- الموضوع 10، التعامدية، في الصفحة 62
- الموضوع 11، قابلية العكس، في الصفحة 69

- الموضوع 14، لغات النطاق، في الصفحة 82
- الموضوع 28، الفصل، في الصفحة 151
- الموضوع 30، برمجة التحويل، في الصفحة 169
- الموضوع 31، ضريبة الوراثة، في الصفحة 180

تحدّيات

- فكّر في مبدأ التصميم الذي تستخدمه بانتظام. هل المقصود هو جعل الأمور سهلة التغيير؟
- فكّر أيضًا في اللغات وأنماط البرمجة (كائنية التوجه، الوظيفيّة، التفاعليّة (Reactive)، وغيرها). هل لدى أي منها إيجابيات كبيرة أو سلبيات كبيرة عندما يتعلق الأمر بمساعدتك بكتابة شِفرة ETC؟ أو كليهما؟

  عند كتابة الشِفرة، مالذي يمكنك فعله للقضاء على السلبيات، وإبراز الإيجابيات؟[17]
- لدى العديد من أدوات التحرير دعم (إما مضمّن أو عبر ملحقات) لتشغيل الأوامر عندما تحفظ ملفًا. اجعل محرّرك يُظهر لك رسالة ETC؟ في كل مرّة تقوم فيها بعملية حفظ واستخدمها[18] كإشارة للتفكير في التعليمات البرمجيّة التي كتبتها للتو. هل هي سهلة التغيير.

17 لإعادة صياغة أغنية آرلين/ ميرسر القديمة...

18 أو، ربما للبقاء متيقظًا، مرّة كل عشر مرّات...

الموضوع 9. DRY – شرور التكرار

كان إعطاء معلومتين متناقضين للحاسوب طريقة الكابتن جيمس ت. كيرك المفضّلة لتعطيل غزو الذكاء الاصطناعي. لسوء الحظ، يمكن أن يكون المبدأ نفسه فعالًا بالإطاحة بـشِفرتك البرمجيّة.

نقوم كمبرمجين بجمع وتنظيم وصيانة وتسخير المعرفة. نوثّق المعرفة في المواصفات، ونجعلها تنبض بالحياة في الشِفرة العاملة، ونستخدمها لتوفير الفحوصات اللازمة في أثناء الاختبار.

لسوء الحظ، إن المعرفة ليست ثابتة - غالبًا ما تتغير بسرعة. قد يتغير فهمك للمطلب بعد اجتماعٍ مع العميل. تغيّر الحكومة الضوابط ويصبح بعض منطق الأعمال قديمًا. قد تُظهر الاختبارات أن الخوارزمية المُختارة لن تعمل. يعني عدم الاستقرار هذا كلّه أننا نقضي جزءًا كبيرًا من وقتنا في وضع الصيانة، وفي إعادة تنظيم المعرفة في أنظمتنا وإعادة التعبير عنها.

يفترض معظم الناس أن الصيانة تبدأ عند إصدار التطبيق، الصيانة التي تعني إصلاح الأخطاء وتحسين الميزات. نحن نعتقد أن هؤلاء الناس مخطئين. فالمبرمجون في وضع الصيانة بشكل مستمر. يتغير فهمنا يومًا بعد يوم. تصل متطلبات جديدة وتتطور المتطلبات الحاليّة ونحن منكبين على عملنا في المشروع. ربما تتغير البيئة. مهما كان السبب، فإن الصيانة ليست نشاطاً منفصلًا، بل جزءا روتينيًا من عملية التطوير بِرُمَّتها.

عندما نقوم بالصيانة ينبغي لنا العثور وتغيير تصورات الأشياء — تلك المعارف الصغيرة المضمّنة في التطبيق. المشكلة أنه من السهل تكرار المعرفة في المواصفات والعمليات والبرامج التي نطوّرها، وعندما نفعل ذلك فنحن نستدعي كابوس الصيانة- الكابوس الذي يبدأ قبل وقتٍ طويل من شحن التطبيق.

ندرك أن الطريقة الوحيدة لتطوير البرمجيات بشكلٍ موثوق، وتسهيل فهم تطوراتنا وصيانتها، هي اتباع ما نسميه مبدأ DRY:

يجب أن يكون لكل معلومة تمثيل واحد واضح لا لبس فيه داخل النظام.

لماذا نسميها DRY؟

نصيحة ١٥	لا تكرّر نفسك Don’t Repeat Yourself

إن البديل عنها هو أن يُعبّر عن الشيء نفسه في مكانين أو أكثر. إذا غيّرت واحدًا، عليك تذكّر تغيير البقيّة، أو سينهار برنامَجك بالكامل بسبب التضارب مثل أجهزة حواسيب ألاين (alien). إنها ليست مسألة هل تذكر: إنها مسألة متى ستنسى.

ستجد مبدأ DRY يظهر مرارًا وتكرارًا خلال هذا الكتاب، غالبًا في سياقات لا عَلاقة لها بكتابة الشِفرة. نحن ندرك أنه أحد أهم الأدوات في صندوق أدوات المبرمج العملي.

سنشرح في هذا القسم مشكلات التكرار ونقترح استراتيجيات عامّة للتعامل معها.

DRY أكثر من مجرّد تعليمات برمجيّة

دعنا نوضح شيئًا من البداية. في الإصدار الأول من هذا الكتاب، قمنا بعمل ضعيف في توضيح ما قصدناه فقط بعدم تكرار نفسك. نسبها العديد من النّاس للشِفرة فقط: اعتقدوا أن DRY تعني "لا تنسخ وتلصق أسطر شِفرة المصدر".

إن هذا جزء من DRY، ولكنه جزءٌ صغير وتافه إلى حدٍ ما.

تعني DRY تكرار المعرفة، والقصد. يتعلق الأمر بالتعبير عن الشيء نفسه في مكانين مختلفين، ربما بطريقتين مختلفتين تمامًا.

إليك الاختبار الفصل: عندما يتعين عليك تغيير وجه واحد من الشِفرة البرمجيّة، هل تجد نفسك تقوم بهذا التغيير في أماكن متعددة، وبصيغ مختلفة متعددة؟ هل يجب عليك تغيير الشِفرة والتوثيق، أو مخطط قاعدة البيانات والبنية التي تربطها، أو ...؟ إذا كان الأمر كذلك، فإن شِفرتك لا تحقق مبدأ DRY

لذلك دعونا نلقي نظرة على بعض الأمثلة النموذجيّة للتكرار.

التكرار في الشِفرة البرمجيّة

قد يكون الأمر سخيفًا، ولكن تكرار الشِفرة أمر شائع جدًا. إليك مثال:

```
def print_balance(account)
    printf "Debits: %10.2f\n", account.debits
    printf "Credits: %10.2f\n", account.credits
  if account.fees < 0
      printf "Fees: %10.2f-\n", -account.fees
  else
      printf "Fees: %10.2f\n", account.fees
  end
      printf " ———-\n"
  if account.balance < 0
      printf "Balance: %10.2f-\n", -account.balance
  else
      printf "Balance: %10.2f\n", account.balance
  end
end
```

مبدئيا أغفل أننا نرتكب خطأ المبتدئين بتخزين العملات بصيغة أعدادٍ حقيقيّة floats. وعوضًا عن ذلك، تحقّق مما إذا كان بإمكانك اكتشاف التكرارات في هذه الشِفرة. (يمكننا أن نرى ثلاثة أشياءٍ في الأقل، ولكن قد ترى المزيد.)

ماذا وجدت؟ إليك قائمتنا.

أولًا، من الواضح أن هناك تكرارا للنسخ واللصق للتعامل مع الأرقام السالبة. يمكننا إصلاح ذلك بإضافة دالّة أخرى:

```
def format_amount(value)
    result = sprintf("%10.2f", value.abs)
  if value < 0
      result + "-"
  else
      result + " "
  end
end
def print_balance(account)
  printf "Debits: %10.2f\n", account.debits
  printf "Credits: %10.2f\n", account.credits
  printf "Fees: %s\n", format_amount(account.fees)
  printf " ———-\n"
```

```
    printf "Balance: %s\n", format_amount(account.balance)
end
```

يوجد تكرار آخر وهو إعادة عرض الحقل في جميع استدعاءات printf. ربما يمكننا إصلاح ذلك بواسطة إدراج ثابت وتمريره إلى كل استدعاء، ولكن لماذا لا نستخدم الدالّة الموجودة فقط؟

```
def format_amount(value)
    result = sprintf("%10.2f", value.abs)
    if value < 0
        result + "-"
    else
        result + " "
    end
end
def print_balance(account)
    printf "Debits: %s\n", format_amount(account.debits)
    printf "Credits: %s\n", format_amount(account.credits)
    printf "Fees: %s\n", format_amount(account.fees)
    printf " ————-\n"
    printf "Balance: %s\n", format_amount(account.balance)
end
```

هل هناك شيء آخر؟ حسنًا، ماذا لو طلب العميل مسافة إضافيّة بين التسميات (labels) والأرقام؟ سيتعيّن علينا تغيير خمسة أسطر. دعنا نزيل هذا التكرار:

```
def format_amount(value)
    result = sprintf("%10.2f", value.abs)
    if value < 0
        result + "-"
    else
        result + " "
    end
end
def print_line(label, value)
    printf "%-9s%s\n", label, value
end
def report_line(label, amount)
    print_line(label + ":", format_amount(amount))
end
def print_balance(account)
    report_line("Debits", account.debits)
    report_line("Credits", account.credits)
    report_line("Fees", account.fees)
    print_line("", "————-")
    report_line("Balance", account.balance)
end
```

إذا كان علينا تغيير تنسيق المبالغ، فإننا نغيّر format_amount. وإذا أردنا تغيير تنسيق التسمية، فإننا نغيّر report_line.

لا يزال هناك انتهاك ضمنيّ لـ DRY: يرتبط عدد علامات الوصل "الشرطات" (-) في الخط الفاصل بعرض حقل المبلغ. لكنها ليست متطابقة تمامًا: فهي حاليًا أقصر بمحرفٍ واحد، لذا فإن أيّة علامات ناقص زائدة ستمتد إلى ما وراء العمود. هذا هو قصد الزَّبُون، وهو قصد مختلف عن التنسيق الفعلي للمبالغ.

ليس كل تكرار للشِفرة تكرار للمعرفة

كجزء من تطبيق طلب النبيذ عبر الإنترنت، فإنك تلتقط عمر المستخدم وتتحقّقه، إلى جانب الكميّة التي يطلبها. يجب أن يكون كلاهما أرقامًا، وكلاهما أكبر من الصفر وفقًا لمالك الموقع. بذلك يمكنك كتابة شِفرة عمليات التحقّق:

```
def validate_age(value):
```

```
    validate_type(value, :integer)
    validate_min_integer(value, 0)
def validate_quantity(value):
   validate_type(value, :integer)
   validate_min_integer(value, 0)
```

في أثناء مراجعة الشِفرة البرمجيّة، يُرجع المقيمون الضليعون هذه الشِفرة، مدّعين أنها تنتهك مبدأ DRY: كلا الدالتين متماثلتين.

إنهم مخطئون. الشِفرات هي نفسها، ولكن المعرفة التي تمثلها كل منهما مختلفة. تتحقّق الدالتان من شيئين منفصلين يحدثان بنفس القواعد. هذه مصادفة وليست تكرارًا.

التكرار في التوثيق

ولدت بطريقة ما الأسطورة التي تقول إنه يجب عليك التعليق على جميع دوالك.

يُنتج أولئك الذين يؤمنون بهذا الجنون شيئًا كهذا:

```
#  حساب رسوم هذا الحساب
# * يكلف كل شيك مرتجع 20 دولارًا
# * إذا كان الحساب على المكشوف لأكثر من 3 أيام،
#  الرسوم 10 دولارات لكل يوم
# *إذا كان متوسط رصيد الحساب أكبر من 2000 دولار أمريكي
# خفّض الرسوم بنسبة 50٪
def fees(a)
   f = 0
   if a.returned_check_count > 0
      f += 20 * a.returned_check_count
   end
   if a.overdraft_days > 3
      f += 10*a.overdraft_days
   end
   if a.average_balance > 2_000
      f /= 2
   end
      f
end
```

يُعطى القصد من هذه الدالّة مرّتين: مرّة في التعليق ومرّة أخرى في الشِفرة. يُغيّر العميل الرسوم، وعلينا عندها تحديث كل من التعليق والشِفرة. وبمرور الوقت، يمكننا أن نضمن إلى حدٍّ بعيد أن التعليق والشِفرة سوف يخرجان عن التماهي.

اسأل نفسك ما الذي يضيفه التعليق إلى الشِفرة. من وجهة نظرنا، فإنّه ببساطة يعوّض عن بعض التسميات السيئة والتخطيط السيئ. ماذا لو كتبنا هذا فقط:

```
def calculate_account_fees(account)
    fees = 20 * account.returned_check_count
    fees += 10 * account.overdraft_days   if account.overdraft_days > 3
    fees /= 2                                  if account.average_balance > 2_000
    fees
end
```

الاسم هنا يوضّح ما يقوم به، وإذا احتاج شخص ما إلى تفاصيل، فهي موجودة في شِفرة المصدر. هذا هو مبدأ DRY!

انتهاكات DRY في البيانات

تمثل بُنى بياناتنا معرفة، ويمكن أن تتعارض مع مبدأ DRY. دعونا نلقي نظرة على صنف (class) يمثل خطًا:

```
class Line {
    Point start;
    Point end;
   double length;
};
```

قد يبدو هذا الصنف للوهلة الأولى منطقيًا. فمن الواضح أن الخط له بداية ونهاية، وسيكون له طول دائمًا (حتى لو كان صفرًا).

لكن هنا لدينا تكرار. يُعرّف الطول بنقَاط البداية والنهاية: غيّر إحدى النُّقَاط ويتغير الطول. من الأفضل جعل الطول حقلًا محسوبًا:

```
class Line {
     Point start;
     Point end;
     double length() { return start.distanceTo(end); }
};
```

قد تختار انتهاك مبدأ DRY في وقتٍ لاحق في عملية التطوير لأسباب تتعلق بالأداء. يحدث هذا بشكل متكرر عندما تحتاج إلى تخزين البيانات مؤقتًا لتجنّب تكرار العمليات باهظة التكلفة. تكمن الحيلة في تحديد مكان التأثير. الانتهاك غير مكشوف للعالم الخارجي: فقط الطرائق (methods) داخل الصنف يجب أن تقلق بشأن الحفاظ على الأمور سويّة.

```
class Line {
   private double length;
   private Point start;
   private Point end;
   public Line(Point start, Point end) {
      this.start = start;
      this.end = end;
       calculateLength();
}
// public
   void setStart(Point p)   {  this.start = p; calculateLength();  }
   void setEnd(Point p)     {  this.end = p; calculateLength();  }
   Point getStart()           {  return start;  }
   Point getEnd()             {  return end;  }
  double getLength()      {  return length;  }
  private void calculateLength() {
    this.length = start.distanceTo(end);
  }
};
```

يوضح هذا المثال قضيّة مهمّة أيضًا: عندما تكشف الوحدة النمطيّة عن بنية بيانات، فإنك تقرن كل الشِفرة التي تستخدم هذه البنية بتنفيذ تلك الوحدة. استخدم دائمًا دوال الوصول (accessor functions)حيثما أمكنك لقراءة وكتابة سمات (attributes) الكائنات. سيُسهّل ذلك إضافة الوظائف في المستقبل.

يرتبط هذا الاستخدام لدوال الوصول *بمبدأ مايير للوصول الموحد*، الموصوف في *بناء البرمجيات كائنية التوجه* Object-Oriented Software Construction [Mey97]، والذي ينصّ على أن جميع الخِدْمَات التي تقدمها الوحدة يجب أن تكون متاحة بواسطة تدوينٍ موحّد، والذي لا يُضلل سواء كانت تُنفّذ بواسطة التخزين أو بواسطة العمليات الحسابيّة.

التكرار التمثيلي Representational Duplication

واجهات شِفرتك البرمجيّة للعالم الخارجي: مكتبات أخرى عبر واجهات برمجة التطبيقات APIs، وخِدمَات أخرى عبر الاستدعاءات البعيدة، والبيانات في مصادر خارجية، وما إلى ذلك.

وغالبًا في كل مرّة تتعامل معها، فأنت تقدم على نوع من انتهاك DRY: يجب أن تحتوي شفرتك على معرفة ممثلة أيضًا في الشيء الخارجي. تحتاج إلى معرفة واجهات برمجة التطبيقات API، أو المخطط، أو معنى رموز الخطأ، أو أيًا كان. التكرار هنا هو أن شيئين (الشِفرة الخاصّة بك والكيان الخارجي) يجب أن يكون لديهما معرفة بتمثيل الواجهة الخاصّة بهما. غيّرها في أحد الأطراف، ويتعطّل الطرف الآخر.

هذا التكرار أمر لا مفر منه، ولكن يمكن تخفيفه. فيما يلي بعض الاستراتيجيات.

التكرار خلال واجهات برمجة التطبيقات الداخليّة

بالنسبة إلى واجهات برمجة التطبيقات API الداخليّة، ابحث عن الأدوات التي تتيح لك تحديد واجهة برمجة التطبيقات في نوع من التنسيق المحايد. ستُنشئ هذه الأدوات عادةً وثائق، وواجهات برمجة تطبيقات وهميّة، واختبارات وظيفيّة، وعملاء API، وهذا الأخير في عدد من اللغات المختلفة.

ستُخزن الأداة في الحالة المثاليّة جميع واجهات برمجة التطبيقات الخاصّة بك في مستودع مركزي، مما يتيح مشاركتها عبر الفِرق.

التكرار خلال واجهات برمجة التطبيقات الخارجيّة

ستجد وعلى نحوٍ متزايد أن واجهات برمجة التطبيقات العامّة موثّقة رسميًا باستخدام شيء مثل OpenAPI[19]. وهذا يسمح لك باستيراد مواصفات واجهة برمجة التطبيقات إلى أدوات واجهة برمجة التطبيقات المحليّة والاندماج بشكل أكثر موثوقيّة مع الخدمة. إذا لم تتمكن من العثور على مثل هذه المواصفة، ففكّر في إنشاء واحدة ونشرها. لن يجدها الآخرون مفيدةً فحسب؛ بل قد تجد مساعدة في صيانتها.

التكرار ومصادر البيانات

تسمح لك العديد من مصادر البيانات بتأمل مخطط بياناتها. يمكن استخدام هذا الأمر لإزالة الكثير من التكرار بينهم وبين شِفرتك البرمجيّة. حيث يمكنك توليد الحاويات مباشرةً من المخطط بدلًا من إنشاء الشِفرة يدويًا لاحتواء هذه البيانات المخزّنة. ستقوم العديد من أطر العمل المملّة بهذا العمل الشاق من أجلك.

هناك خيار آخر، وكثيرًا ما نفضّله. بدلًا من كتابة الشِفرات التي تمثّل البيانات الخارجية في بنية ثابتة (على سبيل المثال، حالة بُنية(instance of a structure)، صنف class)، فقط ضعها في بنية بيانات "مفتاح/قيمة" (قد تطلق عليها لغتك خريطة map أو هاش hash أو قاموس dictionary أو حتى كائن object).

19 https://github.com/OAI/OpenAPI-Specifcation

هذا الأمر محفوف بالمخاطر بحد ذاته: معرفتك للبيانات التي تعمل بها فقط تفقدك كثيرًا من الأمان. لذا نوصي بإضافة طبقة ثانية إلى هذا الحل: مجموعة تحقّق بسيطة معتمدة على الجدول تتحقق أن الخريطة التي أنشأتها تحوي في الأقل البيانات التي تحتاجها وبالتنسيق الذي تحتاجه. قد تتمكن أداة توثيق API الخاصّة بك من إنشاء ذلك.

التكرار بين المطورين

ربما يكون أصعب نوع من التكرار الذي يتم اكتشافه والتعامل معه يحدث بين مطورين مختلفين في المشروع. قد تُكرّر مجموعات كاملة من الوظائف عن غير قصد، وربما لا يكتشف هذا التكرار لسنوات، مما يؤدي إلى مشكلات في الصيانة. سمعنا عن ولايةٍ أمريكية تم مسح أنظمة حواسبها الحكوميّة للتوافق لعام 2000. حيث كشف التدقيق عن أكثر من 10000 برنامَج يحتوي كل منها على نسخة مختلفة من رمز التحقّق رَقَم الضمان الاجتماعي.

على مستوى عال، تعامل مع المشكلة بواسطة بناء فريق قوي ومتماسك مع اتصالات جيّدة.

ومع ذلك، على مستوى الوحدة module، فإن المشكلة أكثر دهاء. فقد تُنفّذ الوظائف أو البيانات المطلوبة عادةً والتي لا تقع في منطقة مسؤوليّة واضحة عدّة مرات.

نرى أن أفضل طريقة للتعامل مع هذا هو تشجيع التواصل ناشط والمتكرر بين المطورين. ربما بإجراء اجتماع سكروم[20](scrum) يوميّ. قم بإعداد منتديات (مثل قنوات Slack) لمناقشة المشكلات الشائعة. يوفّر ذلك طريقة غير مُتطفلة للتواصل - حتى عبر مواقع إلكترونيه متعددة - مع الاحتفاظ بسجل دائم لكل ما قيل.

عيّن عضو فريق أمينًا لمكتبة للمشروع، مهمته تسهيل تبادل المعرفة. احصل على مكانٍ مركزيّ في شجرة المصدر حيث يمكن إيداع إجرائيات المساعدة والبرامج النصيّة (scripts). تحقّق قراءة شِفرة المصدر والوثائق للأشخاص الآخرين، إما بشكل غير رسمي أو في أثناء مراجعات التعليمات البرمجيّة. أنت لا تتطفل - أنت تتعلم منهم. وتذكّر، إن الوصول متبادل.

نصيحة ١٦	اجعل إعادة استخدامها سهلًا

إنّما تحاول القيام به هو تعزيز بيئة يسهل فيها العثور على الأشياء الموجودة وإعادة استخدامها بدلًا من كتابتها بنفسك. إذا لم يكن الأمر سهلًا فلن يفعله الناس. وإذا أخفق إعادة الاستخدام، فإنك تخاطر بتكرار المعرفة.

الأقسام ذات الصلة

- الموضوع 8، جوهر التصميم الجيّد، صفحة 51
- الموضوع 28، الفصل، في الصفحة 151

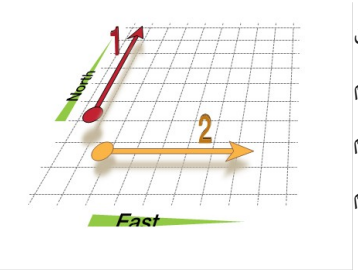
20 **[المترجم]** سْكَرَمْ (Scrum) هو أحد إطارات العمل وفقاً لمقاييس منهجية تطوير البرمجيات أجايل لإدارة تطوير المنتجات. يتميز بأنه ذو نمط تكراري وتزايدي. تمتاز استراتيجية تطوير المنتجات هذه بكونها طريقه مرنة وشمولية، حيث يعمل فريق المطورين جميعاً كوحدة واحدة من أجل تحقيق هدف محدّد مسبقاً. من أهم ميزات هذه الطريقة أنها تعطي إمكانات كبيرة للفريق لإدارة نفسه بنفسه، وتشجع على تواجد الفريق بشكل جماعي في نفس المكان أو عن طريق التواصل الحثيث عن طريق الاتصال عن بعد (الإنترنت، الهاتف). فهناك تركيز واضح على التواصل بين أعضاء الفريق الواحد بواسطة اللقاءات اليوميّة وجها لوجه وبواسطة المحافظة على الانضباط في جميع جوانب المشروع.

- الموضوع 32، الضبط، في الصفحة 188
- الموضوع 38، البرمجة بالمصادفة، في الصفحة 223
- الموضوع 40، إعادة البناء، في الصفحة 235

الموضوع 10. التعامدية

تُعدّ التعامدية مفهومًا بالغ الأهميّة إذا كنت ترغب في إنتاج أنظمة سهلة التصميم والبناء والاختبار والتوسّع. ومع ذلك، نادرًا ما يُدرّس مفهوم التعامد بشكل مباشر. غالبًا ما يكون مِيزة ضمنيّة للعديد من الطرائق والتقنيات الأخرى التي تتعلمها. وهذا خاطئ. فبمجرّد أن تتعلم تطبيق مبدأ التعامدية، ستلاحظ مباشرةً تحسنًا فوريًا في جودة الأنظمة التي تنتجها.

ما هي التعامدية؟



"التعامديـة" هي مصـطلح مسـتعار من الهندسـة. يكـون الخطّ ان متعامـدان إذا التقيـا معًـا بزاويـة قائمـة، كمـا في المحـاور على الرسـم البياني. بمصطلحات المتّجه، فإن الخطين مستقلان. فبينما يتحرك الرَّقَم 1 في الرسم البياني شمالًا، فإنه لا يغير بعده شرقًا أو غربًا. يتحرك الرَّقَم 2 شرقًا، لكن بعده لا يتغير شمالًا أو جنوبًا.

في الحوسبة، أصبح هذا المصطلح يشير إلى نوع من الاستقلال أو الانفصال. يكون شيئان أو أكثر متعامدان إذا لم تؤثر التغييرات في أحدهم على أي من الأشياء الأخرى. ففي نظام مُصمَّم جيدًا، ستكون شِفرة قاعدة البيانات متعامدة مع واجهة المستخدم: يمكنك تغيير الواجهة دون التأثير على قاعدة البيانات، وتبديل قواعد البيانات دون تغيير الواجهة.

قبل أن نلقي نظرة على فوائد الأنظمة المتعامدة، دعنا نلقي أولاً نظرة على نظام غير متعامد.

نظام غير متعامد

أنت في جولة بطائرة مروحيّة (هليكوبتر) في جراند كانيون حين بدأ الطيّار، الذي ارتكب خطأ فادحًا بتناوله الأسماك على الغداء، يئن فجأة وفقد الوعي. لحسن الحظ، تركك تحوم على ارتفاع 100 قدم فوق سطح الأرض.

للمصادفة، كنت قد قرأت صفحة ويكيبيديا حول الطائرات المروحيّة في الليلة السابقة. أنت تعلم أن الطائرات المروحيّة لديها أربع أدوات تحكم أساسية. العصا التدويرية cyclic وهي عصا القيادة التي تمسكها بيدك اليمنى. حرّكها، وستتحرك المروحيّة في الاتجاه الموافق. تضبط يدك اليسرى مِقبَض مجمع التأرجح collective pitch lever. اسحبه وزد زاوية التأرجح على جميع شفرات المروحيّة، لتوليد الرفع. في نهاية مجمع التأرجح توجد دواسة الوقود throttle. وأخيرًا، لديك دوّاستان للقدم[21] foot pedals، واللتان تُغيّران من مقدار دفع الذيل الدوار، وبالتالي تساعدان في تحويل اتجاه المروحيّة.

"أمر سهل!" تعتقد ذلك. "اخفض برفق مجمع التأرجح وستهبط بخفّة على الأرض كبطل". ومع ذلك عند تجربتها، تكتشف أن الحياة ليست بهذه البساطة. تنخفض مقدمة المروحيّة وتبدأ هبوطاً لولبيًا نحو اليسار.

تكتشف فجأة أنك تحلّق بنظام كل مدخل تحكّم فيه لديه تأثيرات ثانوية. اخفض رافعة اليد اليسرى وستحتاج إلى إضافة حركة عكسيّة تعويضيّة لمقبض اليد اليمنى ودفع الدواسة اليمنى. ولكن بعد ذلك ستؤثّر كل من هذه التغييرات على جميع عناصر التحكم

21 **[المترجم]** الدواسات المضادة لعزم الدوران.

الأخرى مرّة ثانية. فجأة كنت تتلاعب مع نظام معقّد بشكل لا يصدق، حيث يؤثّر أي تغيير على كل المُدخَلات الأخرى. عليك عبء عمل هائل: يداك وقدماك تتحركان باستمرار، محاولًا موازنة كل القِوَى المتفاعلة.

لا شكّ أن أدوات التحكم بالمروحيّة ليست متعامدة.

فوائد التعامدية

كما يوضح مثال الطائرة المروحيّة، فإن الأنظمة غير المتعامدة بطبيعتها أكثر تعقيدًا في التغيير والتحكم. عندما تكون مكونات أي نظام مترابطة للغاية، لا يوجد شيء اسمه إصلاح موضعي.

نصيحة ١٧ ألغِ التأثير بين الأشياء غير المترابطة

نريد تصميم مكونات مكتفية ذاتياً: مستقلّة، ولها غرض واحد ومحدّد جيّدًا (ما يسميه يوردون وقسطنتين **التماسك**[22] في أساسيات التصميم البنيوي: *أساسيات التدريب في برنامج الحاسوب وتصميم النظم* Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design [YC79]). عندما تَعزل المكونات عن بعضها البعض، فأنت تعلم أنه بإمكانك تغيير أحد المكونات دون القلق بشأن البقيّة. ومادام أنك لا تغير واجهات هذا المكون الخارجية، يمكنك أن تكون واثقًا من أنك لن تتسبب في مشكلات تنتقل للنظام بِرُمَّته.

يمكنك الحصول على فائدتين رئيستين إذا كتبت أنظمة متعامدة: إنتاجية مرتفعة ومخاطر منخفضة.

ربح الإنتاجية

- التغييرات موضعيّة، لذا تقلّل من وقت التطوير ووقت الاختبار. من الأسهل كتابة مكونات صغيرة نسبيًا قائمة بذاتها بدلًا من كتابة كتلة كبيرة واحدة من الشِفرة. يمكن تصميم المكونات البسيطة وكتابة شِفرتها واختبارها ثم نسيانها - لا حاجة لمواصلة تغيير الشِفرة الحاليّة في أثناء إضافة شِفرة جديدة.
- يدعم النهج المتعامد إعادة الاستخدام أيضًا. إذا كانت للمكونات مسؤوليات واضحة المعالم، فيمكن دمجها مع مكونات جديدة بطرق لم يتصوّرها منفذوها الأصليّون. وكلما كانت أنظمتك مقترنةً بشكل غير مُحكم، كلما كان من الأسهل إعادة ضبطها وإعادة هندستها.
- هناك مكاسب دقيقة إلى حدّ ما في الإنتاجيّة عندما تجمع بين المكونات المتعامدة. افترض أن أحد المكونات يقوم بأشياء مميّزة M وآخر يقوم بأشياءN. إذا كان المكونان متعامدين ودمجتهما، فإن النتيجة تقوم ب M × N شيئًا. ومع ذلك إذا لم يكن المكونان متعامدين، فسيكون هناك تداخل، وستكون النتيجة أقل. يمكنك الحصول على المزيد من الوظائفية لكل جهد وحدة بواسطة دمج المكونات المتعامدة.

خفض المخاطر

يقلّل النهج المتعامد من المخاطر الكامنة في أي تطوّر.

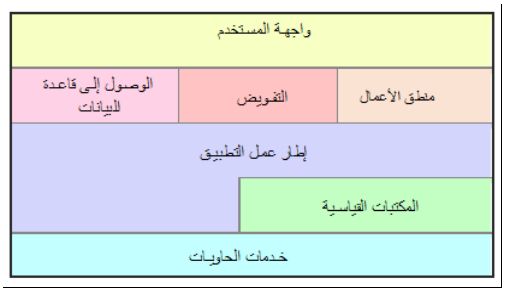
22 **[المترجم]** يشير التماسك إلى درجة انتماء العناصر داخل الوحدة النمطيّة إلى بعضها البعض.

- تعزل الأجزاء المريضة من الشِفرة. إذا كانت الوحدة النمطيّة module مريضة، فمن غير المرجح أن تنشر الأعراض في بقية النظام. وهو إلى ذلك من الأسهل عزلها ووضع شيء جديد وصحيّ مكانها.
- النظام الناتج أقل هشاشة. قم بتغييرات وإصلاحات صغيرة على منطقة معيّنة، وسيتم تقييد أي مشكلات تنشأ لتلك المنطقة.
- من المحتمل أن يُختبر النظام المتعامد بشكل أفضل، لأنه سيكون من الأسهل تصميم وتشغيل الاختبارات على مكوناته.
- لن تكون مرتبطًا ارتباطًا وثيقًا بمورّد (بائع) أو منتج أو منصّة، لأن الواجهات إلى مكونات الجهات الخارجية هذه ستُعزل إلى أجزاء أصغر من التطوير العام.

دعونا نلقي نظرة على بعض الطرق التي يمكنك بواسطتها تطبيق مبدأ التعامدية على عملك.

التصميم

معظم المطوّرين على دراية بضرورة تصميم أنظمة متعامدة، بالرغم أنهم قد يستخدمون كلمات مثل معياري modular، ومستند إلى المكونات component-based، وطبقيّ layered لوصف العمليات. ينبغي أن تتألف الأنظمة من مجموعة من الوحدات النمطيّة المتعاونة، التي تنفذ كل منها وظائف مستقلة عن الوحدات الأخرى. في بعض الأحيان تُنظّم هذه المكونات في طبقات، كل منها يوفّر مستوى من التجريد. إن هذا النهج متعدد الطبقات طريقة قويّة لتصميم أنظمة متعامدة. نظراً لأن كل طبقة تستخدم فقط التجريدات التي توفرها الطبقات أدناها، فلديك مرونة كبيرة في تغيير عمليات التنفيذ الضمنيّة دون التأثير على الشِفرة. كما يقلّل التقسيم الطبقيّ من مخاطر التبعيات المنفلتة بين الوحدات النمطية. سترى غالبًا الطبقات المعبّر عنها في الرسوم البيانيّة:



هناك اختبار سهل للتصميم المتعامد. بمجرّد تعيين المكونات الخاصّة بك، اسأل نفسك: إذا غيّرت المتطلبات وراء وظيفةٍ معيّنة بشكل كبير، فكم عدد الوحدات المتأثّرة؟ في نظام متعامد، يجب أن يكون الجواب "واحد"[23]. يجب ألا يتطلب نقل زر على لوحة واجهة المستخدم الرسوميّة تغييراً في مخطط قاعدة البيانات. يجب ألا تؤدي إضافة المساعدة الحسّاسة للسياق إلى تغيير النظام الفرعيّ للفوترة.

دعونا ننظر في نظام معقّد للمراقبة والتحكم على محطة تدفئة. دعا المتطلب الأصليّ إلى واجهة مستخدم رسوميّة، ولكن تم تغيير المتطلبات لإضافة واجهة جوال (mobile interface) تتيح للمهندسين مراقبة القيم الأساسيّة. ستحتاج في نظام مصمم بشكل متعامد، إلى تغيير فقط تلك الوحدات المرتبطة بواجهة المستخدم للتعامل مع هذا: سيظل المنطق الأساسي للتحكم في المصنع

23 في الواقع، هذا ساذج. ما لم تكن محظوظًا بشكل ملحوظ ستؤثر معظم التغييرات في متطلبات العالم الحقيقي على وظائف متعددة في النظام. ومع ذلك، إذا حللت التغيير من حيث الدلالات (الوظائف)، فيجب أن يؤثر كل تغيير وظيفي بشكل مثالي على وحدة نمطيّة واحدة فقط.

دون تغيير. في الواقع إذا هيكلت نظامك بعناية، يجب أن تكون قادرًا على دعم كلتا الواجهتين مع نفس قاعدة الشِفرة البرمجيّة الأساسيّة.

اسأل نفسك أيضًا عن كيفيّة فصل تصميمك عن التغييرات في العالم الحقيقي. هل تستخدم رَقَم الهاتف كمعرّف (identifier) للزَّبُون ماذا سيحدث عندما تعيد شركة الهاتف تعيين رموز المنطقة؟ تعدّ الرموز البريدية وأرقام الضمان الاجتماعي أو المعرّفات الحكوميّة وعناوين البريد الإلكتروني والنطاقات جميعها معرّفات خارجية لا يمكنك التحكم فيها ويمكن أن تتغير في أي وقت ولأي سبب. لا تعتمد على خصائص الأشياء التي لا يمكنك التحكم فيها.

مجموعات الأدوات والمكتبات

احرص على الحفاظ على تعامدية نظامك عندما تعتمد مجموعات أدوات ومكتبات خارجيّة. اختر التِقَانَات الخاصّة بك بحكمة.

عندما تحضر مجموعة أدوات (أو حتى مكتبة من أعضاء آخرين في فريقك)، اسأل نفسك عما إذا كانت تفرض تغييرات على شِفرتك لا ينبغي أن تكون موجودة. إذا كان مخطط استمرارية الكائن شفافًا، فهو متعامد. أما إذا كان يتطلب منك إنشاء كائنات أو الوصول إليها بطريقة خاصّة، فلن يكون متعامداً. إن الاحتفاظ بهذه التفاصيل بمعزل عن الشِفرة له فائدة إضافيّة تتمثّل في تسهيل تغيير الموردين في المستقبل.

نظام Enterprise Java Beans (EJB)[24] هو مثال تعامد مثير للاهتمام. في معظم الأنظمة الموجهة بالمناقلات transaction-oriented systems، يجب أن تحدّد شِفرة التطبيق بداية ونهاية كل مناقلة. مع EJB، يُعبّر عن هذه المعلومات بشكل توضيحي كملاحظات، خارج الطرائق methods التي تؤدي العمل. يمكن تشغيل شِفرة التطبيق نفسها في بيئات معاملات EJB مختلفة دون تغيير.

بطريقة ما، EJB هو مثال على (الزخرفة): الذي يضيف وظائف للأشياء دون تغييرها. يمكن استخدام هذا النمط من البرمجة في كل لغة برمجة تقريبًا، ولا يتطلب بالضرورة إطار عمل أو مكتبة. لا يتطلب الأمر سوى القليل من الانضباط عند البرمجة.

كتابة الشِفرة

في كل مرّة تكتب فيها شِفرة، فإنك تتعرض لخطر تقليل تعامدية تطبيقك. ما لم تراقب باستمرار ليس فقط ما تفعله ولكن أيضًا السياق الأوسع للتطبيق، فقد تُكرّر الوظائف عن غير قصد في وحدة أخرى، أو تعبّر عن المعرفة الموجودة مرتين.

هناك العديد من التقنيات التي يمكنك استخدامها للحفاظ على التعامدية:

حافظ على شِفرتك منفصلة

اكتب شِفرة خجولة - وحدات لا تكشف عن أي شيء غير ضروري للوحدات الأخرى ولا تعتمد على تنفيذ الوحدات الأخرى. جرّب قانون ديميتر، الذي نناقشه في الموضوع 28، الفصل، في الصفحة 151.

24 **[المترجم]** EJB: هي واحدة من عدّة واجهات برمجة تطبيقات Java لبناء معياريّ لبرامج المؤسسة. EJB هو مكوّن برمجي من جانب الخادم يغلّف منطق الأعمال الخاص بالتطبيق. توفر حاوية ويب EJB بيئة وقت تشغيل لمكونات البرامج ذات الصلة بالويب، بما في ذلك أمان الحاسوب وإدارة دورة حياة Java servlet ومعالجة المعاملات وخِدْمَات الويب الأخرى. ومواصفات EJB هي مجموعة فرعية من مواصفات Java EE.

إذا كنت بحاجة إلى تغيير حالة الكائن، دعه يفعل ذلك لك. وبهذه الطريقة تظل الشِفرة معزولة عن تنفيذ الشفرات الأخرى وتزيد من فرص أن تظلّ متعامدًا.

تجنّب البيانات العموميّة

في كل مرة تشير فيها الشِفرة البرمجيّة إلى البيانات العموميّة (global data)، فإنها تربط نفسها بالمكونات الأخرى التي تتشارك هذه البيانات. حتى العموميات التي تنوي فقط قراءتها يمكن أن تؤدي إلى مشكلات (على سبيل المثال، إذا احتجتَ فجأة إلى تغيير شِفرتك لتصبح متعددة المسارات multithreaded). عمومًا تصبح شفرتك أسهل للفهم والصيانة إذا مرّرتَ أي سياق مطلوب بشكل صريح إلى الوحدات الخاصّة بك. في التطبيقات كائنية التوجه، غالبًا ما يمرّر السياق كمعاملات لباني الكائنات. في الشفرات الأخرى، يمكنك إنشاء بنىً تحتوي على السياق وتمرير المراجع إليها.

يعدّ نمط المفردة[25] Singleton pattern في *أنماط التصميم: عناصر البرمجيات كائنية التوجه القابلة لإعادة الاستخدام* [GHJV95] Design Patterns: Elements of Reusable Object-Oriented Software طريقةً لضمان وجود حالة (instance) واحدة فقط لكائن من صنف معين. يستخدم العديد من الأشخاص هذه الكائنات المفردة كنوع من المتغيرات العموميّة globals (خاصة في اللغات، مثل جافا، التي لا تدعم مفهوم العموميات). كن حذرًا عند استخدام نمط المفردة، فقد تؤدي أيضًا إلى ربط غير ضروري.

تجنّب الدوال المتشابهة

غالبًا ما تصادف مجموعة من الدوال التي تبدو جميعها متشابهة - ربما تشترك في شِفرة مشتركة في البداية والنهاية، ولكن لكل منها خوارزمية مركزية مختلفة. وجود شِفرة مكرّرة هو أحد أعراض المشكلات البنيويّة. ألق نظرة على نمط الاستراتيجية في أنماط التصميم لتحسين التنفيذ.

اعتد أن تنتقد شِفرتك باستمرار. ابحث عن أي فرص لإعادة تنظيمها ولتحسين بنيتها وتعامدها. تُسمى هذه العملية إعادة البناء، وهي مهمّة جدًا لدرجة أننا خصّصنا قسمًا لها (انظر الموضوع 40، إعادة البناء، في الصفحة 235).

الاختبارات

إن النظام المُصمّم والمنفّذ بطريقة متعامدة أسهل في الاختبار. نظرًا لأن التفاعلات بين مكونات النظام رسميّة ومحدودة، يمكن إجراء المزيد من اختبارات النظام على مستوى الوحدة النمطيّة الفردية. هذه أخبار جيّدة، لأن اختبار مستوى الوحدة النمطيّة module (أو الوحدة unit) أسهل بكثير في تحديده وأداءه من اختبار التكامل. في الواقع، نقترح إجراء هذه الاختبارات تلقائيًا كجزء من عملية البناء العادية (انظر الموضوع 41، اختبر لتكتب الشِفرة، في الصفحة 240).

تُعد كتابة اختبارات الوحدة بحدّ ذاتها اختبار تعامدية مثير للاهتمام. ما المطلوب لبناء وتشغيل اختبار الوحدة؟ هل يجب عليك استيراد نسبة كبيرة من بقيّة شِفرة النظام؟ إذا كان الأمر كذلك، فتكون بذلك قد وجدت وحدة غير مفصولة جيدًا عن بقيّة النظام.

25 **[المترجم]** نمط المفردة (Singleton pattern) هو نمط تصميم إنشائي يضمن وجود نسخة واحدة فقط من فئة ما في نفس الوقت الذي يوفّر فيه نقطة وصول عامة لهذه النسخة.
للاستفاضة: https://wiki.hsoub.com/Design_Patterns/singleton

يعدّ إصلاح الأخطاء أيضًا وقتًا مناسبًا لتقييم تعامدية النظام كليًا. عندما تواجه مشكلة ما، قيّم مدى موضعيّة الإصلاح. هل تُغيّر وحدة نمطية واحدة فقط، أم أن التغييرات منتشرة في النظام بِرُمَّته؟ عندما تُجري تغييرًا، هل يصلح كل شيء، أم أن مشكلات أخرى تنشأ بشكل غامض؟ هذه فرصة جيدة لاستقدام الأتمتة. إذا كنت تستخدم نظام التحكم في الإصدار (وسوف تستخدمه بعد قراءة الموضوع 19، التحكم في الإصدار، في الصفحة 105)، فضع علامة على إصلاحات الأخطاء عند التحقق من الشِفرة مرّة أخرى بعد الاختبار. يمكنك بعد ذلك تشغيل تقارير شهريّة تحلل التوجهات بعدد المِلَفّات المصدريّة المتأثرة بكل إصلاح للأخطاء.

التوثيق

ربما من المدهش أن التعامدية تنطبق أيضًا على التوثيق. المحاور هنا هي المحتوى والعرض. مع التوثيق المتعامد حقًا، يجب أن تكون قادرًا على تغيير المظهر بشكل كبير دون تغيير المحتوى. توفر معالجات برنامج معالجة النصوص Word أوراق أنماط ووحدات ماكرو[26] تساعد في ذلك. نحن نفضل شخصيًا استخدام نظام تأشير مثل مارك داون Markdown: عند الكتابة، نركّز فقط على المحتوى، ونترك مهمّة العرض التقديمي لأي أداة نستخدمها لتقديمه[27].

التعايش مع التعامد

إن التعامد يرتبط ارتباطًا وثيقًا بمبدأ DRY في الصفحة 54. في DRY، أنت تتطلع إلى تقليل التكرار داخل النظام، بينما في التعامد تقلل من الترابط بين مكونات النظام. قد تكون كلمة موضوعة جزافًا، ولكن إذا كنت تستخدم مبدأ التعامد، مقترنًا اقترانًا وثيقًا بمبدأ DRY، فستجد أن الأنظمة التي تطوّرها أكثر مرونة، وأكثر قابلية للفهم، وأسهل في التصحيح والاختبار والصيانة.

إذا دخلت في مشروع حيث يكافح الناس فيه بشدة لإجراء تغييرات، وحيث يبدو أن كل تغيير يتسبب في حدوث أربعة أشياء أخرى خاطئة فتذكّر كابوس الطائرة المروحيّة. ربما لم يتم تصميم وكتابة شِفرة المشروع بشكل متعامد. حان الوقت لإعادة البناء.

وإذا كنت طيّارًا لطائرة مروحيّة فلا تأكل السمك...

الأقسام ذات الصلة

- الموضوع 3، اعتلاج البرمجيات، في الصفحة 31
- الموضوع 8، جوهر التصميم الجيّد، صفحة 51
- الموضوع 11، قابلية العكس، في الصفحة 69
- الموضوع 28، الفصل، في الصفحة 151
- الموضوع 31، ضريبة الوراثة، في الصفحة 180
- الموضوع 33، كسر الاقتران الزمني، صفحة 193

26 [المترجم] لفظة ماكرو (بالإنجليزية: Macro) تستخدم للتعبير عن دمج عدة أوامر نمطيّة وكثيرة التكرار في أمر واحد بسيط يمكن استخدامه بسهولة.

27 في الواقع، هذا الكتاب مكتوب بتنسيق مارك داون Markdown، ومنضّد مباشرة من مصدر مارك داون.

- الموضوع 34، الحالة المشتركة حالة غير صحيحة، في الصفحة 198
- الموضوع 36، السبّورات، صفحة 211

تحدّيات

- ضع في اعتبارك الفرق بين الأدوات التي تحتوي على واجهة مستخدم رسوميّة وبين أدوات سطر أوامر صغيرة ولكنها قابلة للدمج تستخدم في مِحَث شِل shell. أي مجموعة أكثر تعامدًا ولماذا؟ أيّهما أسهل في الاستخدام للغرض المحدّد من أجله بالضبط؟ ما المجموعة التي يسهل دمجها مع أدوات أخرى لمواجهة التحدّيات الجديدة؟ أي مجموعة أسهل في التعلم؟
- تدعم سي++ الوراثة المتعددة، وتسمح جافا للصنف بتنفيذ واجهات متعدّدة. لدى روبي مزيج من ذلك. ما تأثير استخدام هذه التسهيلات على التعامد؟ هل هناك فرق في التأثير بين استخدام الوراثة المتعددة والواجهات المتعددة؟ هل هناك فرق بين استخدام التفويض واستخدام الوراثة؟

تمارين

تمرين 1 (إجابة محتملة في الصفحة 319)

يطلب منك قراءة مِلَفّ سطرًا في كل مرة. عليك تقسيم كل سطر إلى حقول. أي من المجموعات التالية من تعريفات مسودة الأصناف مرجّحة لأن تكون أكثر تعامدًا؟

```
class Split1 {
  constructor(fileName)    # opens the file for reading
  def readNextLine()       # moves to the next line
  def getField(n)          # returns nth field in current line
}
```

أو

```
class Split2 {
  constructor(line)        # splits a line
  def getField(n)          # returns nth field in current line
}
```

تمرين 2 (إجابة محتملة في الصفحة 319)

ما هي الاختلافات في التعامد بين اللغات كائنية التوجه واللغات الوظيفيّة؟ هل هذه الاختلافات متأصّلة في اللغات نفسها، أم فقط في الطريقة التي يستخدمها فيها النّاس؟

الموضوع 11. قابلية العكس (Reversibility)

لا شيء أخطر من فكرة إذا كانت هي الفكرة الوحيدة التي لديك.
← إميل أوغست شارتييه (آلان)، اقتراح سور الدين 1938

يفضّل المهندسون الحلول البسيطة والمفردة للمشكلات. إن اختبارات الرياضيات التي تسمح لك الإعلان بثقةٍ كبيرة أن 2= x أكثر راحةً من المقالات المبهمة والمحتّدة عن الأسباب التي لا تعد ولا تحصى للثورة الفرنسية.

تميل الإدارة إلى الاتفاق مع المهندسين: حيث تناسب الإجابات السهلة والمفردة جداول البيانات وخطط المشروع بشكل جيّد.

ليت العالم الحقيقي يتعاون فقط! لسوء الحظ، في حين أن x قيمتها 2 اليوم، فقد تحتاج إلى أن تكون 5 غدًا و3 الأسبوع المقبل. لا شيء ثابت إلى الأبد - وإذا كنت تعتمد بشكل كبيرٍ على حقيقةٍ ما، فيمكنك أن تضمن أنّها ستتغير.

يوجد دائمًا أكثر من طريقة واحدة لتنفيذ شيء ما، وعادةً ما يتوفر أكثر من مورّد واحد لتوفير منتج خارجي. إذا دخلت في مشروع تعوقه فكرةُ قصر النظر القائلة بأن هناك طريقة واحدة فقط للقيام بذلك، فقد تواجه مفاجأة غير سارة. تُبقي العديد من فرق المشروع أعينهم مفتوحة بنشاط على تكشُفات المستقبل.

"لكنك قلت أننا سنستخدم قاعدة بيانات XYZ! لقد انتهينا من كتابة شِفرة المشروع بنسبة 85٪، ولا يمكننا التغيير الآن! " قالها المبرمج معترضًا. "للأسف، قرّرت شركتنا توحيد قاعدة بيانات إلى PDQ بدلًا من ذلك – لجميع المشروعات. إن الأمر خارج عن إرادتي. علينا فقط إعادة البرمجة. ستعملون جميعًا في عُطل نهاية الأسبوع حتى إشعار آخر."

يَجِبُ ألاّ تكون التغييرات شديدة الصعوبة، أو حتى فوريّة. ولكن قد تجد نفسك مع مرور الوقت ومع تقدم مشروعك، عالقًا في وضع لا يمكن تحمله. يلتزم فريق المشروع مع كل قرار حاسم بتحقيق هدف أصغر - نسخة أضيق من الواقع لديها خيارات أقل.

مع اتخاذ العديد من القرارات الحاسمة، يصبح الهدف صغيرًا جدًا بحيث أنه إذا تزحزح، أو غيّرت الريح اتجاهها، أو رفرفت بجناحيها فراشة في طوكيو، تُخطِئُه[28]. وقد تخطئ بقدرٍ كبيرٍ جدا.

تكمن المشكلة في أن القرارات الحاسمة لا يمكن عكسها بسهولة.

بمجرّد أن تقرّر استخدام قاعدة بيانات هذا المورد، أو نمط الهيكلية ذاك، أو نموذج نشر معين، فإنك تصبح ملتزمًا بمسار عمل لا يمكن التراجع عنه، إلا بتكلفةٍ كبيرة.

قابلية العكس (التراجع)

إن العديد من المواضيع في هذا الكتاب موجّهة لإنتاج برمجيات مرنة وقابلة للتكيّف. فبواسطة الالتزام بتوصيات هذه المواضيع - خاصةً مبدأ DRY في الصفحة 54، والفصل في الصفحة 151، واستخدام الضبط الخارجي في الصفحة 188 – فلن يتعين علينا

28 خذ نظامًا غير خطي، أو فوضويًا، وأجري تغييرًا صغيرًا على أحد مدخلاته. قد تحصل نتيجة كبيرة وغير متوقعة في كثير من الأحيان. يمكن أن تكون الرفرفة الاعتيادية للفراشة التي تسببها أجنحتها في طوكيو بداية لسلسلة من الأحداث التي تنتهي بإحداث إعصار في تكساس. هل هذا يبدو مشابها لأي مشروعات تعرفها؟

اتخاذ العديد من القرارات الحاسمة التي لا رجعة فيها. هذا شيء جيّد، لأننا لا نتخذ دائمًا أفضل القرارات من المرّة الأولى. نلتزم بتكنولوجيا معيّنة فقط لنكتشف أننا لا نستطيع توظيف عدد كافي من الأشخاص بالمهارات اللازمة. نلتزم بمورّد خارجي معيّن قبل أن يتم شراء أسهمهم من قبل منافسيهم. تتغير المتطلبات والمستخدمون والأجهزة بشكل أسرع مما يمكننا تطوير البرمجيات.

لنفترض أنك قرّرت، في وقتٍ مبكر من المشروع، استخدام قاعدة بيانات علائقيّة من المورد A. وبعد ذلك بكثير، في أثناء اختبار الأداء، اكتشفت أن قاعدة البيانات ببساطة بطيئة جدًا، في حين أن قاعدة بيانات المستندات من المورّد B أسرع. لن تكون محظوظًا في معظم المشروعات التقليدية. تتشابك استدعاءات منتجات الجهات الخارجيّة في جميع أنحاء الشِفرة معظم الوقت. ولكن إذا كنت قد قمت فعلًا بتجريد فكرة قاعدة البيانات - إلى الحدّ الذي توفّر فيه ببساطة الاستدامة كخدمة - عندئذٍ يكون لديك إمكانية تغيير الخيول في منتصف الطريق.

وبالمثل، لنفترض أن المشروع يبدأ كتطبيق معتمد على المتصفح، ولكن بعد ذلك وفي وقت متأخر من الخطّة، يقرّر قسم التسويق أن ما يريدونه فعلًا هو تطبيق جوال. ما مدى صعوبة ذلك بالنسبة لك؟ في الحالة المثاليّة، يَجِبُ ألاّ يؤثر عليك كثيرًا، في الأقل من جهة الخادم. ستزيل بعض تقديمات HTML وتستبدلها بواجهة برمجة تطبيقات.

يكمن الخطأ في افتراض أن أي قرار يُتّخذ منقوش على الحجر – وفي عدم الاستعداد للطوارئ التي قد تنشأ. بدلًا من نحت القرارات في الحجر، فكّر فيها أكثر على أنها مكتوبة في الرمال على الشاطئ. ويمكن أن تأتي موجة كبيرة وتمحوها في أي وقت.

نصيحة ١٨	لا توجد قرارات نهائية

هيكلية مرنة

بينما يحاول الكثير من النّاس الحفاظ على شِفرتهم البرمجيّة مرنة، تحتاج أيضًا إلى التفكير في الحفاظ على المرونة في مجالات الهيكلة والنشر وتكامل الموردين.

نكتب هذا في عام 2019. منذ مطلع القرن، رأينا الهيكليات التالية "لأفضل الممارسات" من جانب الخادم:

- كتل كبيرة من الحديد[29].
- اتحادات كتل الحديد الكبير.
- عناقيد متوازنة الحمل (Load-balanced clusters) من العتاد الشعبي[30].
- أجهزة افتراضيّة مُستندة إلى السحابة تُشّغل التطبيقات.
- أجهزة افتراضيّة مُستندة إلى السحابة تُشّغل الخِدمَات.
- الإصدارات الواردة أعلاه في حاويات.

29 **[المترجم]** حاسوب كبير ومكلف جدًا وقادر على دعم المئات، أو حتى الآلاف، من المستخدمين في وقت واحد. يُعتقد على نطاق واسع أن مشتق من أجهزة الحواسيب المركزية المبكرة، والتي كانت محاطة بإطارات معدنية بحجم الغرفة.

30 **[المترجم]** أجهزة حواسيب بأسعار معقولة ويسهل الحصول عليها. عادةً ما يكون نظامًا منخفض الأداء متوافقًا مع IBM للحاسوب الشخصي وقادرًا على تشغيل Microsoft Windows أو Linux أو MS-DOS دون الحاجة إلى أي أجهزة أو مُعِدّات خاصّة.

- تطبيقات دون خادم مستندة إلى السحابة.
- وحتمًا، عودة واضحة إلى كتل كبيرة من الحديد لبعض المهام.

انطلق وأضف أحدث وأروع البدع إلى هذه القائمة، ثم عدّها مدهشة: إنها أعجوبة أن أيّ منها اشتغل فيما مضى.

كيف يمكنك التخطيط لهذا النوع من التقلبات الهيكلية؟ لا يمكنك.

ما يمكنك فعله هو تسهيل التغيير. أخفِ واجهات برمجة التطبيقات الخارجية خلف طبقات التجريد الخاصّة بك. قسّم الشِفرة إلى مكونات: حتى إذا انتهى بك الأمر إلى نشرها على خادمٍ ضخم واحد، فإن هذا النهج أسهل بكثير من أخذ تطبيق مترابط وتقسيمه. (شَقينا لإثبات ذلك).

ومع أنّ هذه ليست مشكلة قابلية عكس بشكل خاص، إلا أنها نصيحة أخيرة.

نصيحة ١٩ تخلّ عن البدع التالية

لا أحد يعرف ما يخبئه المستقبل، لا سيما نحن! لذا مكّن الشِفرة الخاصة بك على نمط موسيقى الروك آند رول rock-n-roll: تَثبُت عندما يمكنها، وتلتف مع الضربات عندما يجب عليها.

الأقسام ذات الصلة

- الموضوع 8، جوهر التصميم الجيّد، صفحة 51
- الموضوع 10، التعامدية، في الصفحة 62
- الموضوع 19، التحكم في الإصدار، في الصفحة 105
- الموضوع 28، الفصل، في الصفحة 151
- الموضوع 45، هوّة المتطلبات، في الصفحة 268
- الموضوع 51، مجموعة البادئ العملية، في الصفحة 300

تحدّيات

- حان الوقت لقليل من ميكانيك الكم مع قطة شرودنغر[31].

افترض أن لديك قطة في صندوق مغلق، إلى جانب جسيم مشع. لدى الجسيم فرصة 50 ٪ بالضبط للانشطار إلى قسمين. إذا حدث ذلك، ستُقتل القطة. وإذا لم يحدث، فستكون القطة على ما يرام. إذا، هل القطة ميتة أم حيّة؟ وفقًا لشرودنغر، إن

31 **[المترجم]** قطة شرودنغر: هي تجربة ذهنيّة قدّمها إرفين شرودنغر ليبيّن فيها المشكلات التي رآها بتفسير كوبنهاغن وتأثير الوعي الإنساني في عملية الرصد والقياس الفيزيائي خصوصًا في الحالات الكموميّة. العلبة تحتوي ذرة متفككة، في حال أصدرت الذرة جسيمًا باتجاه عداد غايغر ينطلق سم سيانيد قاتلا القطة، الاتجاه المعاكس لا يقتل القطة. دون الاستعانة برصد بشري مباشر تكون حالة الذرة المتفككة دالة موجيّة باحتمال 50 % إطلاق جسيم بالاتجاه القاتل و 50 % بالاتجاه غير القاتل أي إن حالة القطة هي حالة مركبة من الموت والحياة.

الإجابة الصحيحة هي كليهما (في الأقل مادام بقي الصندوق مغلقًا). في كل مرّة يحدث تفاعل شبه نووي له نتيجتان محتملتان، الكون مُستنسَخ. في إحدى النتائج، وقع الحدث، في الأخرى لم يقع. القطة حيّة في كون، وميّتة في آخر. فقط عندما تفتح الصندوق، تعرف أي كونٍ أنت فيه.

لا عجب في أن كتابة الشِفرة للمستقبل أمر صعب.

ولكن فكّر في تطوّر الشِفرة على نفس المنوال مثل الصندوق المملوء بقطط شرودنغر: كل قرار ينتج عنه نسخة مختلفة من المستقبل. كم عدد الإمكانات المستقبلية التي يمكن أن تدعمها شفرتك؟ أيها أكثر احتمالًا؟ ما مدى صعوبة دعمهم عندما يحين الوقت؟ هل تجرؤ على فتح الصندوق؟

الموضوع 12. الرصاصات الخطّاطة

استعد، نار، صوّب...

← أنون

غالبًا ما نتحدث عن ضرب الأهداف عند تطوير البرمجيات. نحن في الواقع لا نطلق أي شيء في ميدان الرماية، لكنها لا تزال استعارةً مجازيّة مفيدة وتصويريّة جدًا. من المثير للاهتمام على وجه الخصوص النظر في كيفيّة إصابة هدف في عالم معقّد ومتغير.

الجواب بالطبع يعتمد على طبيعة الأداة التي تسدّد بواسطتها. بالعديد منها لديك فرصة واحدة فقط للتسديد، ومن ثم تذهب لتتبين ما إذا كنت قد أصبت الهدف أم لا. لكن هناك طريقة أفضل.

أنت تعرف كل تلك الأفلام والبرامج التلفزيونية وألعاب الفيديو حيث يرمي النّاس بالأسلحة الرشاشة؟ غالبًا ما سترى في هذه المشاهد مسار الرّصاص كخطوط مضيئة في الهواء. تأتي هذه الخطوط من الطلقات الخطّاطة.

تُعبئ الطلقات الخطّاطة على فترات جنبًا إلى جنب مع الذخيرة العادية. عند إطلاقها يشتعل الفسفور ويترك أثرًا ناريًا يمتد من البندقية إلى أي شيء تصيبه. إذا أصابت الطلقات الخطّاطة الهدف، فإن الرّصاص العادي سيصيبه كذلك. يستخدم الجنود القذائف الخطّاطة هذه لتحسين إصابة أهدافهم: إنها تمثّل تغذية راجعة آنيّة وعمليّة في ظل الظروف الفعليّة.

ينطبق نفس المبدأ على المشروعات، خاصّة عندما تبني شيء لم يُبن من قبل. نستخدم مصطلح تطوير الرصاصة الخطّاطة للتعبير بصريًا عن الحاجة إلى تغذية راجعة فوريّة في ظلّ الظروف الفعليّة عند التعامل مع هدف متحرك.

مثل الرماة، أنت تحاول إصابة هدف في الظلام. قد تكون متطلبات المستخدمين غامضة لأنهم لم يروا نظامًا كهذا من قبل. وقد تواجه عددًا كبيرًا من الأمور المجهولة نظرًا لأنك قد تستخدم خوارزميات أو تقنيات أو لغات أو مكتبات لست معتاداً عليها. ولأن المشروعات تستغرق وقتًا حتى تكتمل، يمكنك أن تضمن إلى حدٍّ بعيد أن البيئة التي تعمل فيها ستتغير قبل أن تُنهي عملك.

تتمثل الاستجابة التقليدية بتحديد النظام تحديدًا مفصّلًا حتى النهاية. أعدّ رزمًا من الورق مُفصّلاً كل متطلب، مُحدّدًا كل شيء غير معروف، مُقيّدًا البيئة. ارمِ مستخدمًا نظام تقدير الموقع. أمامك حساب واحد ضخم، ثم أطلق وانتظر بأمل.

يميل المبرمجون العمليون، مع ذلك، إلى تفضيل استخدام البرمجيات المكافئة للطلقات الخطّاطة.

الشِفرة التي تضيء في الظلام

تعمل الرصاصات الخطّاطة لأنها تشتغل في نفس البيئة وتحت نفس القيود مثل الرّصاص الحقيقي. تصل إلى الهدف بسرعة، لذلك يحصل الرامي على معلومات فوريّة. إنها تمثل حلًّا رخيصًا نسبيًا من وجهة نظر عمليّة.

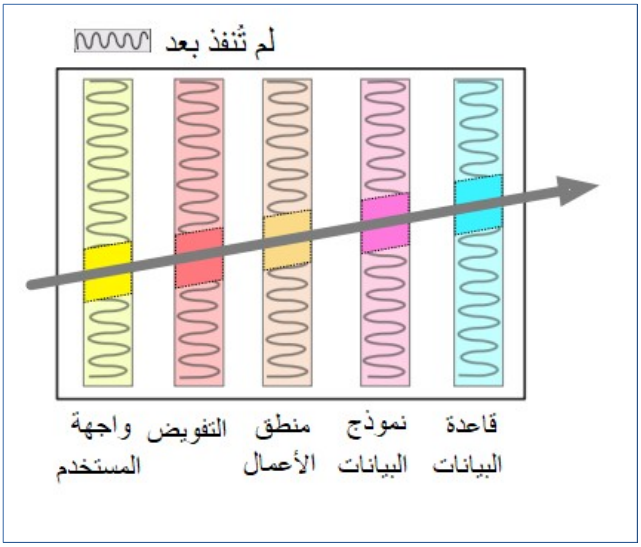
للحصول على نفس التأثير في الشِفرة البرمجيّة، نبحث عن شيء ينقلنا من متطلب إلى جانب ما من جوانب النظام النهائي بسرعة وبشكل مرئي وقابل للتكرار.

ابحث عن المتطلبات الهامّة، تلك التي تحدّد النظام. ابحث عن المناطق التي لديك شكوك فيها، وحيث ترى أكبر المخاطر. ثم أعطها الأولوية بحيث تكون هذه المناطق هي المناطق الأولى التي تكتب شفرتها.

نصيحة ٢٠ استخدم الرَّصاص الخطاط للعثور على الهدف

في الواقع، نظرًا لتعقيد إعداد مشروعات اليوم، مع وجود حشود من الاعتمادات والأدوات الخارجيّة، تصبح الرصاصات الخطّاطة أكثر أهميّة. بالنسبة لنا، فإن أول رصاصة خطاطة هي ببساطة إنشاء المشروع، وإضافة "مرحبًا بالعالم!"[32] والتأكّد ترجمته وتشغيله بشكل صحيح. ثم البحث عن مناطق عدم اليقين في التطبيق الإجمالي وإضافة الهيكل اللازم لجعلها تعمل.

ألقِ نظرة على الرسم البياني التالي. لهذا النظام خمس طبقات هيكليّة. لدينا بعض المخاوف بشأن كيفيّة دمجهم، لذلك نبحث عن مِيزة بسيطة تسمح لنا باستخدامهم معًا. يُظهر الخط القطري المسار الذي تسلكه المِيزة خلال الشِفرة. لتفعيل المِيزة، علينا فقط تنفيذ المناطق المظلّلة تظليلًا مُحكمًا في كل طبقة: ستُنفذ المناطق ذات التظليل المتمايل لاحقًا.



لقد تعهدنا ذات مرّة بمشروع تسويق قاعدة بيانات خادم-عميل معقّد. كان أحد المتطلبات يتمثل في القدرة على تحديد وتنفيذ الاستعلامات المؤقتة. كانت الخوادم مجموعة من قواعد البيانات العلائقيّة والمتخصّصة. استخدمت واجهة المستخدم العميل، المكتوبة بلغة عشوائية A، مجموعة من المكتبات المكتوبة بلغة مختلفة لتوفير واجهة للخوادم. خُزّن استعلام المستخدم على الخادم بصيغة تشبه Lisp[33] قبل تحويله إلى SQL محسّن وقبيل التنفيذ مباشرة. كان هناك الكثير من الأشياء المجهولة والعديد من البيئات المختلفة، ولم يكن أحد متيقّنًا من كيفيّة تعامل واجهة المستخدم.

كانت هذه فرصة رائعة لاستخدام شِفرة خطاطة. لقد طوّرنا إطار عمل للواجهة الأماميّة، ومكتبات لتمثيل الاستعلامات، وبنية لتحويل استعلام مخزّن إلى استعلام خاص بقاعدة البيانات. ثم جمعناهم معًا وتحققنا عملهم. كل ما يمكننا فعله لهذا البناء الأولي هو إرسال استعلام يعرض جميع الصفوف في جدول، ولكنه أثبت أن واجهة المستخدم يمكن أن تتخاطب مع المكتبات، ويمكن للمكتبات إجراء تسلسل للاستعلام وإلغاء تسلسله، ويمكن للخادم إنشاء SQL من النتائج. وقمنا خلال الأشهر التالية بالتوسيع في هذه البنية

32 **[المترجم]** برنامَج بسيط للغاية، غالبًا ما يُنفذ كاختبار للتأكّد أن لغة البرمجة مثبّتة بشكل صحيح.
33 **[المترجم]** https://ar.wikipedia.org/wiki/%D9%84%D9%8A%D8%B3%D8%A8

الأساسيّة تدريجيًا، مُضيفين وظائف جديدة عن طريق زيادة كل مكوّن من الشِفرة الخطّاطة على التوازي. عندما أضافت واجهة المستخدم نوع استعلام جديد، ازدهرت المكتبة وصنعت جيل SQL أكثر تطوّرًا.

الشِفرة الخطّاطة ليست للاستخدام مرّة واحدة: تكتبها للاحتفاظ بها. تحتوي على جميع تدقيقات الأخطاء، والبنية، والتوثيق، والتحقّق الذاتيّ التي تمتلكها أي قطعة من شِفرة الإنتاج. إنها ببساطة لا تعمل بكامل طاقتها. ومع ذلك، يمكنك التحقّق مدى قربك من الهدف، وتعديله إذا لزم الأمر بمجرّد أن تحقق اتصال نهاية إلى نهاية بين مكونات نظامك. وبمجرّد الوصول إلى الهدف، تصبح إضافة الوظائف أمرًا سهلاً.

يتوافق التطوير الخطاط مع فكرة أن المشروع لا ينتهي أبدًا: ستكون هناك دائمًا تغييرات مطلوبة ووظائف لإضافتها. إنه نهج تزايد تدريجي.

إن البديل المتعارف هو نوع من نهج الهندسة الثقيلة: تُقسم الشِفرة إلى وحدات، تُكتب شفرتها في فراغ. تُدمج الوحدات النمطية في مجمعات فرعية، والتي تُدمج بدورها بعد ذلك، حتى يصبح لديك تطبيق كامل ذات يوم. عندها فقط يمكن تقديم التطبيق كليًا للمستخدم واختباره.

يتميز نهج الشِفرة الخطّاطة بالعديد من المزايا:

يتيح للمستخدمين رؤية شيء ما يعمل في مرحلةٍ مبكرة

إذا نجحت في إيصال ما تفعله (راجع الموضوع 52، أبهج مستخدميك، في الصفحة 307)، فسيعرف المستخدمون أنهم يرون شيئًا غير ناضج. لن يشعروا بخيبة أملٍ بسبب نقص الوظائف؛ وسيكونون متحمسين جدًا لرؤية بعض التقدّم الواضح تجاه نظامهم. كما يمكنهم المساهمة مع تقدم المشروع، مما يزيد من فرصة شرائهم. من المرجّح أن يكون هؤلاء المستخدمون هم الأشخاص الذين سيخبرونك بمدى قرب كل تكرار من الهدف.

يبني المطوّرون هيكلية للعمل فيها

إن أكثر ورقة شاقّة هي تلك التي لم يكتب عليها أي شيء. إذا كنت قد وضعت جميع التعاملات من نهاية إلى نهاية لتطبيقك، وجسّدتها في الشِفرة البرمجيّة، فلن يحتاج فريقك إلى سحب أكبر قدر ممكن من الهواء. هذا يجعل الجميع أكثر إنتاجيّة، ويشجّع على الانسجام.

لديك منصّة تكامل

مادام أن النظام متصل بالكامل (من نهاية إلى نهاية)، فإن لديك بيئة يمكنك بواسطتها إضافة أجزاء جديدة من الشِفرة بمجرد اجتيازها اختبار الوحدة. بدلاً من محاولة الدمج الشامل (دفعة واحدة)، ستدمج كل يوم (غالبًا عدّة مرّات في اليوم). يكون تأثير كل تغيير جديد أكثر وضوحًا، وتكون التفاعلات أكثر محدوديّة، لذلك يكون التنقيح (debugging) والاختبار أسرع وأكثر دقّة.

لديك شيء للعرض

يميل رعاة المشروع وكبار الشخصيات إلى مشاهدة العروض التجريبيّة في أكثر الأوقات إزعاجًا. وباستخدامك للشِفرة الخطّاطة، سيكون لديك دائمًا شيء لعرضه.

لديك شعور أفضل للتقدم

يعالج المطوّرون في تطوير الشِفرة الخطّاطة، حالات الاستخدام (use cases) واحدة تلو الأخرى. عند الانتهاء من واحدة، ينتقلون إلى الأخرى. فإنه من الأسهل بكثير قياس الأداء وإظهار التقدّم للمستخدم. ونظرًا لأن كل تطوير فردي يكون أصغر، فإنك تتجنب إنشاء كتل الشِفرة الموحّدة التي يتم الإبلاغ عنها على أنها 95٪ كاملة أسبوعًا بعد أسبوع.

الرَّصاص الخطّاط لا يصيب هدفه دائمًا

تُظهر الرصاصات الخطّاطة كلما تضربه. قد لا تصيب الهدف دائمًا. ثم تُعدل تصويبك حتى يصبح على الهدف. هنا تكمن الفكرة.

الأمر نفسه مع الشِفرة الخطّاطة. فأنت تستخدم التقنية في الحالات التي لا تكون متيقِّنًا فيها تمامًا من المكان الذي ستصل إليه. يجب ألا تتفاجأ إذا فشلت محاولاتك الأولى: يقول المستخدم "ليس هذا ما قصدته" أو أن البيانات التي تريدها غير متاحة عندما تحتاج إليها، أو يبدو أن مشكلات الأداء محتملة. لذا غيّر ما لديك لتقريبه من الهدف، وكن ممتنًا لاستخدامك منهجية تطوير مرنة؛ قطعة صغيرة من الشِفرة بها قصور ذاتي منخفض – هذا سهل وسريع التغيير. ستتمكن من جمع الانطباعات حول تطبيقك وإنشاء إصدار جديد وأكثر دقّة بسرعة وبتكلفة منخفضة. ونظرًا لأن كل مكوّن رئيس للتطبيق ممثل في شِفرتك الخطّاطة، يمكن للمستخدمين أن يكونوا على ثقة بأن ما يشاهدون يعتمد على الواقع، وليس مجرّد مواصفات ورقيّة.

الشِفرة الخطّاطة مقابل النماذج الأوليّة

قد تعتقد أن مفهوم الشِفرة الخطّاطة هذا ليس أكثر من إنشاء نماذج أولية تحت اسم عدائي. يوجد اختلاف. فأنت تهدف بواسطة النموذج الأولي إلى استكشاف جوانب محدّدة من النظام النهائي. ستتخلص عند استخدامك لنموذج أولي حقيقي من كلما جمعته معًا عند تجربتك للمفهوم، وستعيد كتابة شِفرته بشكل صحيح باستخدام الدروس التي تعلمتها.

لنفترض على سبيل المثال أنك تُنتج تطبيقًا يساعد المتسوقين في تحديد كيفيّة حزم الصناديق ذات المقاسات الغريبة في حاويات. من بين المشكلات الأخرى، يجب أن تكون واجهة المستخدم سهلة وأن تكون الخوارزميات التي تستخدمها لتحديد الحزم الأمثل معقّدة للغاية.

يمكنك إنشاء نموذج أولي لواجهة الاستخدام للمستخدمين النهائيين عن طريق أداة واجهة المستخدم UI. أنت تكتب فقط ما يكفي من الشِفرة لجعل الواجهة تستجيب لإجراءات المستخدم. وبمجرّد الموافقة على التخطيط، يمكنك التخلص منها وإعادة كتابة شِفرتها، هذه المرّة من منطلق منطق الأعمال، باستخدام اللغة الهدف. وبالمثل، قد ترغب في إنشاء نموذج أولي لعدد من الخوارزميات التي تؤدي الحزم الفعلي. قد تكتب شِفرة الاختبارات الوظيفيّة بلغة عالية المستوى ومتسامحة مثل بايثون Python، وتكتب شِفرة واختبارات الأداء منخفضة المستوى بلغة أقرب إلى لغة الآلة. على أي حال ستبدأ مرّة أخرى، بمجرّد اتخاذك القرار، وتكتب شِفرة الخوارزميات في بيئتها النهائية، وتتفاعل مع العالم الحقيقي. هذه نمذجة أوليّة وهي مفيدة جدًا.

يعالج نهج الشِفرة الخطّاطة مشكلة مختلفة. تحتاج إلى معرفة كيف التطبيق كليًا معلّق معا. تريد أن تظهر للمستخدمين كيف ستعمل التفاعلات في الممارسة العملية، وتريد أن تعطي المطوّرين هيكل لتعليق التعليمات البرمجيّة. في هذه الحالة، يمكنك بناء متتبع يتكون من تنفيذ بسيط لخوارزمية تعبئة الحاوية (ربما شيء من قبيل من يأتي أولاً، يُخدّم أولاً) وواجهة مستخدم بسيطة ولكن فعّالة. بمجرّد أن تُجمّع كل المكونات في التطبيق معًا، يكون قد أصبح لديك إطار عمل لعرضه على المستخدمين والمطوّرين لديك.

تضيف إلى هذا الإطار بمرور الوقت وظائف جديدة، وتستكمل الإجرائيات المتعثرة. لكن تبقى المنصّة سليمة، وتعلم أن النظام سيستمر في التصرف بالطريقة التي كان عليها عند إكمالك أول شِفرة خطاطة.

التمييز مهم بما يكفي لتبرير التكرار. يولّد النموذج الأولي شِفرةً يمكن التخلص منها. الشِفرة الخطّاطة هزيلة ولكنها مكتملة، وتشكل جزءًا من الهيكل الأساسي للنظام النهائي.

فكّر في النماذج الأولية على أنها تجمع الاستطلاع والذكاء الذي يحدث قبل إطلاق رصاصة خطّاطة واحدة.

الأقسام ذات الصلة

- الموضوع 13، النماذج الأولية والملاحظات الملحقة، في الصفحة 78
- الموضوع 27، لا تتجاوز مصابيحك الأمامية، في الصفحة 146
- الموضوع 40، إعادة البناء، في الصفحة 235
- الموضوع 49، الفرق العملية، صفحة 289
- الموضوع 50، جوز الهند لا يفي بالغرض، في الصفحة 296
- الموضوع 51، مجموعة البادئ العملية، في الصفحة 300
- الموضوع 52، أبهج مستخدميك، في الصفحة 307

الموضوع 13. النماذج الأوليّة والملاحظات المُلحقة

تَستخدم العديد من الصناعات نماذجَ أوليّة لتجربة أفكارٍ محدّدة؛ فالنماذج الأوليّة أرخص بكثير من الإنتاج واسع النطاق. قد يَبني صانعو السيارات، على سبيل المثال، العديد من النماذج الأوليّة المختلفة لتصميمِ سيارةٍ جديدة. يُصمَّم كل نموذج لاختبار جانبٍ محدّد من السيارة - الديناميكية الهوائيّة، والتصميم، والخصائص الهيكليّة، وما إلى ذلك. قد يستخدم أصحاب المدرسة القديمة نموذجًا طينيًا لاختبار نفق الرياح، ربما نموذجَ خشبِ البلسا وشريطًا لاصقًا سيعمل في قسم الفن، وهكذا. سينفّذ النّاس الأقل رومنسيّة نماذجهم على شاشة الحاسوب أو في الواقع الافتراضي، مُقللين أكثر من التكاليف. يمكن تجرِبة العناصر الخطرة أو غير المؤكّدة بهذه الطريقة دون الحاجة إلى الشراء لبناء العنصر الحقيقي.

نحن نبني نماذج برمجيّة أوليّة بنفس الطريقة، وللأسباب نفسها – أي لتحليل المخاطر وكشفها، ولعرض فرص التصحيح بتكلفةٍ مخفضةٍ كثيرًا. يمكننا العمل على بناء نموذج أوليّ لاختبار جانب محدّد أو أكثر من جوانب المشروع مثل مُصنّعي السيارات.

إننا نميل إلى التفكير في النماذج الأوليّة على أنها مُستندة إلى الشِفرة، ولكنها يجب ألا تكون دائمًا كذلك. يمكننا بناء نماذج أوليّة من مواد مختلفة كما هو الحال مع مُصنعي السيارات. الملاحظات المُلحقة رائعة للنموذج الأوليّ بالنسبة للأشياء الديناميكية مثل تدفق العمل ومنطق التطبيق. يمكن تصميم نموذج أوليّ لواجهة مستخدم بالرسم على لوح أبيض، كتصميم غير وظيفي يُرسم باستخدام برنامَج رسم أو باستخدام إحدى أدوات بناء الواجهة.

صُممت النماذج الأوليّة للإجابة على بعض الأسئلة فقط، لذا فهي أرخص بكثير وأسرع في التطوير من التطبيقات التي تدخل في الإنتاج. يمكن أن تتجاهل الشِفرة التفاصيل غير المهمّة– غير المهمّةِ بالنسبة لك في الوقت الحالي، ولكن ربما تكون لاحقًا مهمّةً جدًا للمستخدم. يمكنك إذا كنت تبني نموذجًا أوليًا لواجهة مستخدم التخلص من النتائج أو البيانات غير الصحيحة. ومن ناحيةٍ أخرى يمكنك البَدْء من واجهة مستخدم ضعيفة جدًا، أو ربما بلا واجهة مستخدم على الإطلاق في حال كنت تبحث فقط في الجوانب الحسابيّة أو الأداء.

ولكن إذا وجدت نفسك في بيئة حيث لا يمكنك التخلي عن التفاصيل، فأنت بحاجة إلى أن تسأل نفسك عما إذا كان يجب عليك إنشاء نموج أوليّ أصلًا. ربما يكون نمط تطوير الرصاصة الخطّاطة أكثر ملاءمة في هذه الحالة (انظر الموضوع 12، الرصاصات الخطّاطة، في الصفحة 73).

أشياء إلى نموذج أوليّ

ما أنواع الأشياء التي قد تختار تحقيقها باستخدام نموذج أوليّ؟ أي شيء ينطوي على مخاطر. أي شيء لم يُختبر من قبل، أو أنه مهم للغاية للنظام النهائي. أي شيء غير مُثبت أو تجريبي أو مشكوك فيه. أي شيء لا تشعر معه بالراحة. يمكنك إعداد النموذج الأوليّ لـ:

- الهيكليّة
- وظائف جديدة في نظام قائم
- بنية أو محتوى بيانات خارجية

- أدوات أو مكونات خارجية
- قضايا الأداء
- تصميم واجهة المستخدم

النماذج الأوليّة هي خبرة تعليميّة. ولا تكمن قيمتها في الشِفرة المُنتجة، وإنما في الدروس المستفادة. وهنا تكمن الأهميّة الحقيقيّة للنماذج الأوليّة.

نصيحة ٢١ تعلم النموذج الأوليّ

كيفيّة استخدام النماذج الأوليّة

ما هي التفاصيل التي يمكنك تجاهلها عند إنشاء نموذج أوليّ؟

الصّحة

قد تتمكن من استخدام بيانات وهميّة حيثما كان ذلك مناسبًا.

الاكتمال

قد يعمل النموذج الأوليّ فقط بمعناه المحدود للغاية، ربما مع جزء واحد فقط من بيانات الإدخال المحدّدة مسبقًا وعنصر قائمة (menu) واحد.

المتانة

من المرجح أن يكون تدقيق الأخطاء غير كامل أو مفقود تمامًا. إذا ابتعدت عن المسار المحدّد مسبقًا، فقد يتعطّل النموذج الأوليّ ويحترق في عرض رائع للألعاب الناريّة. لا بأس بذلك.

الأسلوب

لا ينبغي أن تحتوي شِفرة النموذج الأوليّ على الكثير من التعليقات أو التوثيق (مع أنّك قد تُنتج كمية كبيرة (ماعون) من الوثائق كنتيجة لتجربتك مع النموذج الأوليّ).

تُلغي النماذج الأوليّة التفاصيل، وتركّز على جوانب محدّدة من النظام المأخوذ بعين الاعتبار، لذلك قد ترغب في تنفيذها باستخدام لغة برمجة نصيّة عالية المستوى- أعلى من بقيّة المشروع (ربما لغة مثل **بايثون** أو **روبي**)، في حال كانت هذه اللغات بعيدة عن وجهتك. يمكنك اختيار الاستمرار في التطوير باللغة المُستخدمة في بناء النموذج الأوليّ، أو يمكنك التبديل؛ في النهاية، ستتخلص من النموذج الأوليّ على أي حال.

استخدم لعمل نموذج أوليّ لواجهات المستخدم أداةً تتيح لك التركيز على المظهر و/ أو التفاعلات دون القلق بشأن الشِفرة أو التأشير.

تعمل لغات البرمجة النصيّة أيضًا بشكل جيّد مثل "الغراء" لدمج القطع منخفضة المستوى في مجموعات جديدة. يمكنك باستخدام هذا النهج تجميع المكونات الموجودة بسرعة في تكوينات (configurations) جديدة لمعرفة كيفيّة عمل الأشياء.

النماذج الأوليّة للهيكليّة

تُبنى العديد من النماذج الأوليّة لنمذجة كامل النظام قيد الدراسة، لا تحتاج أي من الوحدات الفردية في نظام النموذج الأوليّ إلى أن تكون فعّالة بشكل خاص، على عكس نهج الرصاصات الخطّاطة. في الواقع، قد لا تحتاج حتى إلى كتابة شِفرة من أجل إنشاء نموذج أوليّ للهيكلية- يمكنك إنشاء نموذج أوليّ على لوح أبيض، مع ملاحظات مُلصقة أو بطاقات فهرسة. ما تبحث عنه هو كيفيّة ربط النظام معًا كليًا، مؤجلًا التفاصيل مرّة أخرى. فيما يلي بعض المجالات المحدّدة التي قد ترغب في البحث عنها في النموذج الأوليّ الهيكلي:

- هل مسؤوليات المناطق الرئيسة محدّدة تحديدًا جيّدًا وملائمًا؟
- هل التعاون بين المكونات الرئيسة محدّد جيّدًا؟
- هل الاقتران بحدوده الدنيا؟
- هل يمكنك تحديد المصادر المُحتمَلة للتكرار؟
- هل تعاريف الواجهة وقيودها مقبولة؟
- هل لدى كل وحدة مسار وصول إلى البيانات التي تحتاجها في أثناء التنفيذ؟ هل لديها إمكانية الوصول هذه عندما تحتاج إليها؟

يميل هذا العنصر الأخير إلى توليد أكثر المفاجآت والنتائج القيّمة من تجربة النماذج الأوليّة.

كيفيّة عدم استخدام النماذج الأوليّة

تأكّد قبل الشروع في أي نماذج أوليّة مستندة إلى الشِفرة، من أن الجميع يفهم أنك تكتب شِفرة يمكن التخلص منها. يمكن أن تكون النماذج الأوليّة جذابةً بشكلٍ مضلّل للأشخاص الذين لا يعرفون أنها مجرّد نماذج أوليّة. يجب أن توضح تمامًا أن هذه الشِفرة يمكن التخلص منها وهي غير مكتملة ويتعذر إكمالها.

من السهل أن تُضَلّل بواسطة الاكتمال الظاهر لنموذج أوليّ معروض، وقد يُصّر رعاة المشروع أو إدارته على نشر النموذج الأوليّ (أو شيء منحدر منه) إذا لم تحدّد التوقعات الصحيحة. ذكّرهم أنه يمكنك بناء نموذج أوليّ رائع لسيارة جديدة من خشب البلسا وشريط لاصق، لكنك بالتأكيد لن تحاول قيادتها في ساعات ذروة حركة المرور!

إذا كنت تشعر أن هناك احتمالًا قويًا في بيئتك أو ثقافتك بأن الغرض من شِفرة النموذج الأوليّ قد يساء تفسيره، فقد تكون أفضل حالًا مع نهج الرصاصات الخطّاطة. سينتهي بك المطاف بإطار عمل متين تبني عليه التطوير المستقبلي.

يمكن للنماذج الأوليّة المستخدمة استخدامًا صحيحًا أن توفّر لك قدرًا كبيرًا من الوقت والمال والمعاناة بواسطة تحديد نِقَاط المشكلات المحتملة وتصحيحها في وقتٍ مبكرٍ من دورة التطوير— الوقت الذي يكون فيه إصلاح الأخطاء رخيصًا وسهلًا.

الأقسام ذات الصلة

- الموضوع 12، الرصاصات الخطّاطة، في الصفحة 73
- الموضوع 14، لغات النطاق، في الصفحة 82
- الموضوع 17، ألعاب شِل، الصفحة 99
- الموضوع 27، لا تتجاوز مصابيحك الأمامية، في الصفحة 146
- الموضوع 37، استمع إلى حدسك، في الصفحة 218
- الموضوع 45، هوّة المتطلبات، في الصفحة 268
- الموضوع 52، أبهج مستخدميك، في الصفحة 307

تمارين

تمرين 3 (إجابة محتملة في الصفحة 320)

يرغب قسم التسويق في الجلوس معك وتبادل الأفكار "عصف ذهني" حول بعض تصميمات صفحات الويب. إنهم يفكرون في خرائط صور قابلة للنقر لنقلك إلى صفحات أخرى، وهكذا. لكن لا يمكنهم اتخاذ قرار بشأن نموذج للصورة- ربما تكون سيارة أو هاتف أو منزل. لديك قائمة بالصفحات المستهدفة والمحتوى؛ ويرغبون في رؤية بعض النماذج الأوليّة. بالمناسبة، لديك 15 دقيقة. ما هي الأدوات التي قد تستخدمها؟

الموضوع 14. لغات النطاق[34]

> إنّ حدود لغة المرء هي حدود عالمه.
>
> ← لودفيج فيتجنشتاين

تؤثر لغات الحاسوب في كيفيّة تفكيرك في المشكلة، وفي كيفيّة تفكيرك في التواصل. تأتي كل لغة مع قائمة من الميزات: الكلمات الطنانة مثل الكتابة الساكنة static مقابل الكتابة الديناميكية، الربط المبكر مقابل الربط المتأخر، الوظائفية مقابل كائنية التوجه، نماذج الوراثة، وأصناف mixin[35]، وحدات الماكرو - كلها قد تقترح أو تحجب حلولًا معينة. سينتج تصميم حل ما عند وضع لغة **سي+** + في الاعتبار نتائج مختلفة عن تلك التي قد نحصل عليها عند التفكير بتصميمه اعتمادا على **هاسكل** (Haskell)، والعكس صحيح. على العكس من ذلك، وأكثر أهميّة باعتقادنا، قد تقترح لغة نطاق المشكلة حلًا برمجيًا.

نحن نحاول دائمًا كتابة الشِفرة باستخدام مفردات نطاق التطبيق (انظر **صيانة المعجم**، في الصفحة 251). يمكن للمبرمجين العمليين في بعض الحالات الانتقال إلى المستوى التالي والبرمجة فعليًا باستخدام المفردات وبناء الجمل والدلالات- أي اللغة – الخاصة بالنطاق.

نصيحة ٢٢	برمج ضمن نطاق المشكلة

بعض لغات نطاق العالم الحقيقي

دعونا نلقي نظرة على بعض الأمثلة حين قام الناس بذلك تماماً.

RSpec

آرسبيك[36] هي مكتبة اختبار لـلغة روبي Ruby. ألهمت هذه المكتبة إصدارات لمعظم اللغات الحديثة الأخرى. الغرض من الاختبار في آرسبيك هو إظهار السلوك الذي تتوقعه من شفرتك البرمجيّة.

```
describe BowlingScore do
    it "totals 12 if you score 3 four times" do
        score = BowlingScore.new
        4.times { score.add_pins(3) }
        expect(score.total).to eq(12)
    end
end
```

34 https://en.wikipedia.org/wiki/Domain-specific_language [المترجم]
35 https://ar.wikipedia.org/wiki/%D8%AE%D9%84%D8%B7_Mixin [المترجم]
36 https://rspec.info

Cucumber

كيوكَمبر[37] هو طريقة محايدة للغة البرمجة لتحديد الاختبارات. عليك تشغيل الاختبارات باستخدام إصدار كيوكَمبر مناسب للغة التي تستخدمها. يجب عليك أيضًا من أجل دعم اللغة الطبيعية كبناء الجمل، وكتابة تطابقات محدّدة تتعرف على العبارات وتستخرج المُعاملات من أجل الاختبارات.

```
Feature: Scoring
Background:
  Given an empty scorecard
Scenario: bowling a lot of 3s
  Given I throw a 3
  And I throw a 3
  And I throw a 3
  And I throw a 3
  Then the score should be 12
```

كان القصد من اختبارات كيوكَمبر أن يقرأها عملاء البرمجيات (مع أنّ ذلك نادر الحدوث عمليًا إلى حد ما؛ يراعي ما يلي سبب حدوث ذلك).

لماذا لا يقرأ العديد من مستخدمي الأعمال ميزات كيوكَمبر؟

إن أحد أسباب عدم نجاح الجمع التقليدي للمتطلبات، والتصميم، والشِفرة، ونهج الشحن. هو أنها ترتكز على فكرة أننا نعرف ما هي المتطلبات. ولكن نادرًا ما يكون الوضع كذلك. سيكون لدى مستخدمي أعمالك فكرة ضبابيّة عما يريدون إنجازه، لكنهم لا يعرفون ولا يهتمون بالتفاصيل. وهذا جزء من قيمتنا: فنحن نستشعر القصد ونحوله إلى شِفرة.

لذلك عندما تجبر شخصًا تجاريًا على توقيع مستند المتطلبات، أو تجعله يوافق على مجموعة من ميزات كيوكَمبر، فأنت تفعل ما يعادل حملهم على التحقّق الإملاء في مقال مكتوب باللغة السومرية.

سيقومون بإجراء بعض التغييرات العشوائية في المستند لحفظ ماء الوجه وتوقيعه لإخراجك من مكتبهم.

أعطهم شِفرة تعمل، على أي حال، يمكنهم التعامل معها. هذا هو المكان حيث تظهر احتياجاتهم الحقيقية.

موجهات فونيكس Phoenix Routes

تحتوي العديد من أطر عمل الويب على وسيط توجيه، حيث تعيين طلبات HTTP الواردة على وظائف المُداول (handler) في الشِفرة. إليك مثال من فونيكس[38].

```
scope "/", HelloPhoenix do
  pipe_through :browser # Use the default browser stack
  get "/", PageController, :index
  resources "/users", UserController
end
```

هذا يعني أن الطلبات التي تبدأ بـ "/" ستُشغّل بواسطة سلسلة من المرشحات المناسبة للمتصفحات. سيُعالج طلب "/" نفسه بواسطة دالة index في وحدة PageController. تنفذ وحدة UsersController الدوال اللازمة لإدارة مصدر يمكن الوصول إليه عبر عنوان (url): /users.

37 https://cucumber.io
38 https://phoenixframework.org

Ansible

أنسبل[39] هي أداة لإعداد البرمجيّة، عادة على مجموعة من الخوادم البعيدة. تقوم أنسبل بذلك بواسطة قراءة المواصفات التي تقدمها أنت لها، ثم تنفذ كل ما هو مطلوب على الخوادم لجعلها تظهر تلك المواصفات. يمكن كتابة المواصفات باستخدام لغة تسلسل البيانات YAML[40]، وهي لغة تبني هياكل البيانات من أوصاف النص:

```
---
- name: install nginx
  apt: name=nginx state=latest
- name: ensure nginx is running (and enable it at boot)
  service: name=nginx state=started enabled=yes
- name: write the nginx config file
  template: src=templates/nginx.conf.j2 dest=/etc/nginx/nginx.conf
  notify:
   - restart nginx
```

يضمن هذا المثال تثبيت أحدث إصدار من Nginx[41] على خوادمي، وبَدْء تشغيله افتراضيًا، واستخدام مِلَفّ الأعداد الذي زوّدته به.

خصائص لغات النطاق

دعونا نلقي نظرة من كثب على هذه الأمثلة.

يُكتب كل من آرسبيك وموجّه فونيكس باللغات المضيفة (روبي Ruby وإكسير Elixir). يستخدمان بعض الشِفرة الموازية إلى حدّ ما، بما في ذلك البرمجة الوصفية[42] metaprogramming ووحدات الماكرو، ولكن في النهاية يُجمّعون ويُشغّلون كشِفرة عادية.

تُكتب اختبارات كيوكَمبر وضبط أنسبل بلغاتهم الخاص~ة. يُحوَّل اختبار كيوكَمبر إلى شِفرة ليتم تشغيلها أو إلى بنية بيانات، في حين تُحوّل مواصفات أنسبل دائمًا إلى بنية بيانات تُشغّل بواسطة أنسبل نفسها.

ونتيجة لذلك، تُضمن آرسبيك وشِفرة الموجّه في الشِفرة التي تُشغّلها أنت: فهي ملحقات حقيقية لمفردات شِفرتك. يُقرأ كل من كيوكَمبر و أنسبل بواسطة الشِفرة ويُحولان إلى صيغة يمكن للشِفرة التعامل معها.

ندعو كل من آرسبيك والموجّه أمثلة على لغات النطاق الداخلي، بينما يستخدم كيوكَمبر وأنسبل لغات خارجية.

39 https://www.ansible.com

40 https://yaml.org

41 **[المترجم]** نجن إكس (ويلفظ Engine-X) وهو خادم ويب مفتوح المصدر وخادم بروكسي للبروتوكولات: مراسم نقل النص الفائق، مراسم إرسال البريد البسيط، مراسم مكتب البريد وبروتوكول الوصول إلى رسائل الإنترنت مع التركيز العالي على التوافقية والأداء وقلة استهلاك الذاكرة. مرخص تحت رخصة BSD ويعمل على منصات يونكس، لينكس، توزيعة برمجيات بيركلي، ماك، سولاريس، آي بي إم إيه آي إكس، إتش بي – يو إكس ووينـدوز.

42 **[المترجم]** Metaprogramming هي تقنية برمجة تمتلك فيها برامج الحاسوب القدرة على التعامل مع البرامج الأخرى على أنها بياناتها. هذا يعني أنه يمكن تصميم البرنامَج لقراءة أو إنشاء أو تحليل أو تحويل برامج أخرى، وحتى تعديل نفسه في أثناء التشغيل. في بعض الحالات، يسمح هذا للمبرمجين بتقليل عدد أسطر الشِفرة البرمجيّة للتعبير عن الحل، وبذلك يقل وقت التطوير. كما يتيح للبرامج مرونة أكبر للتعامل مع المواقف الجديدة بكفاءة دون إعادة تجميعها.

المفاضلات بين اللغات الداخلية والخارجية

يمكن عمومًا أن تستفيد لغة النطاق الداخلية من ميزات اللغة المُضيفة: لغة النطاق التي تُنشئها أكثر قوة، وتأتي هذه القوة مجانًا. يمكنك على سبيل المثال استخدام بعض شِفرات روبي لإنشاء مجموعة من اختبارات آرسبيك تلقائيًا. نستطيع في هذه الحالة اختبار الدرجات scores حيث لا توجد أبدال أو أخطاء:

```
describe BowlingScore do
  (0..4).each do |pins|
    (1..20).each do |throws|
      target = pins * throws
      it "totals #{target} if you score #{pins} #{throws} times" do
        score = BowlingScore.new
        throws.times { score.add_pins(pins) }
        expect(score.total).to eq(target)
      end
    end
  end
end
```

ها قد كتبت 100 اختبار للتو. خذ بقية اليوم عطلة.

إن الجانب السلبي للغات النطاق الداخلي هو أنه عليك الالتزام بصياغة هذه اللغة النَحوِيّة ودلالاتها. مع أنّ بعض اللغات مرنة بشكل لافت في هذا الصدد، إلا أنك لا تزال مجبرًا على التوفيق بين اللغة التي تريدها واللغة التي يمكنك تنفيذها.

في النهاية، كل ما توصلت إليه لا يزال يتطلب صياغة نَحوِيّة صالحة في لغتك المستهدفة. تمنحك اللغات التي تحتوي على وحدات ماكرو (مثل إكسيرElixir و كلوجور Clojure و كريستال Crystal) مزيدًا من المرونة، ولكن في نهاية المطاف الصياغة النَحوِيّة هي الصياغة النَحوِيّة.

ليس لدى اللغات الخارجية مثل هذه القيود. مادام يمكنك كتابة محلل لَغَوي parser للغة، فأنت على ما يرام. في بعض الأحيان يمكنك استخدام المحلل اللغوي لشخص آخر (كما فعلت أنسيبل باستخدام YAML)، ولكن بعد ذلك تعود إلى تحقيق توافق.

من المحتمل أن تعني كتابة محلّل لَغَوي إضافة مكتباتٍ جديدة وربما أدوات لتطبيقك. وكتابة محلّل جيّد ليست مهمّة سهلة. ولكن، إذا كنت تشعر برغبة داخليّة نابضة، فيمكنك النظر إلى مولدات المحللات مثل البيسون bison أو ANTLR، ومنصّات التحليل مثل العديد من محلّلي PEG[43] في الخارج.

اقتراحنا بسيط إلى حدّ ما: لا تبذل مجهودًا أكبر مما توفره. تضيف كتابة لغة النطاق بعض التكاليف إلى مشروعك، وعليك أن تكون مقتنعًا بأن هناك مدخرات تعويضيّة (ربما على المدى الطويل).

عمومًا، استخدم اللغات الخارجيّة الجاهزة (مثل YAML أو JSON أو CSV) إذا استطعت. إذا لم تستطع، انظر إلى اللغات الداخلية. نوصي باستخدام اللغات الخارجية فقط في الحالات التي ستُكتب فيها لغتك من قبل مستخدمي تطبيقك.

لغة النطاق الداخلي بتكلفة منخفضة

أخيرًا، هناك خدعة لإنشاء لغات النطاق الداخلي إذا كنت لا تمانع في تسرب الصياغة اللغوية للمضيف بواسطتها. لا تُكثر من البرمجة الوصفيّة. وبدلاً من ذلك، اكتب دالات للقيام بهذا العمل. في الواقع هذا ما يفعله آرسبيك إلى حدٍ بعيد:

43 **[المترجم]** في علوم الحاسوب، إن قواعد التعبير اللغوي (Parsing Expression Grammar) أو PEG، هي نوع من القواعد التحليلية الرسميّة، أي أنها تصف لغة رسميّة من حيث مجموعة من القواعد للتعرف على السلاسل في اللغة.

```
  it "totals 12 if you score 3 four times" do
      score = BowlingScore.new
      4.times { score.add_pins(3) }
      expect(score.total).to eq(12)
    end
end
```

إن describe، وit، وexpect، وto، و eq في هذه الشِفرة هي مجرّد طرائق روبي فقط. هناك القليل من السباكة خلف الكواليس من حيث كيفية تمرير الأشياء، ولكنها كلها مجرد شِفرة. سنستعرض ذلك قليلاً في التمارين.

الأقسام ذات الصلة

- الموضوع 8، جوهر التصميم الجيّد، صفحة 51
- الموضوع 13، النماذج الأولية والملاحظات الملحقة، في الصفحة 78
- الموضوع 32، الضبط، في الصفحة 188

تحدّيات

- هل يمكن التعبير عن بعض متطلبات مشروعك الحالي بلغة خاصة بالنطاق؟ هل من الممكن كتابة مترجم برمجي compiler أو مترجم لَغَوي translator يمكنه توليد معظم الشِفرة المطلوبة؟
- إذا قرّرت اعتماد اللغات المصغّرة كطريقة للبرمجة أقرب إلى نطاق المشكلة، فأنت تقبل أنه سيلزم بذل بعض الجهد لتنفيذها. هل يمكنك رؤية الطرق التي يمكن بها إعادة استخدام إطار العمل الذي تُطوره لمشروع واحد في مشروعات أخرى؟

تمارين

تمرين 4 (إجابة محتملة في الصفحة 320)

نريد تنفيذ لغة صغيرة للتحكم في نظام رسوميات سلحفاة[44] (turtle-graphics) بسيط. تتكون اللغة من أوامر مؤلفة من محرف واحد، يتبع بعضها رَقَم واحد. سيؤدي الإدخال التالي على سبيل المثال إلى رسم مستطيل:

```
P 2 # select pen 2
D # pen down
W 2 # draw west 2cm
N 1 # then north 1
E 2 # then east 2
S 1 # then back south
U # pen up
```

نفّذ الشِفرة التي تحلل هذه اللغة. يجب تصميمها بحيث يكون من السهل إضافة أوامر جديدة.

تمرين 5 (إجابة محتملة في الصفحة 321)

44 **[المترجم]** هي رسومات متجهة باستخدام مؤشر نسبي (السلحفاة) على مستوى ديكارتي. رسوميات السلاحف هي سمة أساسيّة في لغة برمجة الشعارات.

في التمرين السابق نفذنا محللا للغة الرسم - كانت لغة نطاق خارجية. الآن نفّذه مرّة أخرى كلغة داخلية. لا تفعل أي شيء ذكي: ما عليك سوى كتابة دالّة لكل أمر من الأوامر. قد تضطر إلى تغيير أسماء الأوامر إلى أحرف صغيرة، وربما تغليفها داخل شيء ما لتوفير بعض السياق.

تمرين 6 (إجابة محتملة في الصفحة 321)

صمم قواعد BNF[45] لتحليل مواصفات الوقت. يجب أن تكون جميع الأمثلة التالية مقبولة:

4 مساءً، 7:38 مساءً، 23:42، 3:16، 3:16 صباحًا

تمرين 7 (إجابة محتملة في الصفحة 321)

نفّذ محلل لقواعد BNF في التمرين السابق باستخدام منشئ محلل قواعد التعبير اللغوي PEG باللغة التي تختارها. يجب أن يكون الناتج عددًا صحيحًا يحتوي على عدد الدقائق بعد منتصف الليل.

تمرين 8 (إجابة محتملة في الصفحة 322)

نفّذ محلل الوقت باستخدام لغة نصيّة scripting language وتعبيرات عادية.

45 **[المترجم]** في علوم الحاسب الآلي، (BNF) صيغة باكوس العادية أو صيغة باكوس نور(هو أسلوب تدوين لقواعد السياق الحر، وكثيرا ما تستخدم لوصف تكوين الجملة من اللغات المستخدمة في الحاسوب، مثل لغات البرمجة، وأشكال الوثيقة، ومجموعات التعليمات وبروتوكولات الاتصال. ويطبّق في الحالات التي تحتاج إلى وصف دقيق للغة، على سبيل المثال، في مواصفات اللغة الرسمية، في الكتيبات، وفي كتب دراسة نظرية لغة البرمجة. وتستخدم العديد من الملحقات والمتغيرات من التدوين الأصلي؛ البعض يتم تعريفها حرفيا، بما فيها صيغة باكوس نور المطولة (EBNF) صيغة باكوس نور المضافة (ABNF).

الموضوع 15. التقدير (Estimating)

تحتوي مكتبة الكونغرس في العاصمة واشنطن حاليًا على حوالي 75 تيرا بايت من المعلومات الرَقَميّة عبر الإنترنت. أجب بسرعة! كم من الوقت يستغرق إرسال كل تلك المعلومات عبر شبكة 1Gbps؟ ما مقدار التخزين الذي ستحتاجه لمليون اسم وعنوان؟ كم من الوقت يستغرق ضغط 100 ميجا بايت من النص؟ كم عدد الشهور التي يستغرقها تسليم مشروعك؟

في مستوى معين، هذه كلها أسئلة لا معنى لها — كلها معلومات مفقودة. ومع ذلك، يمكنك الإجابة عليها جميعًا، مادام أنك مرتاح للتقدير. وستفهم في عملية إنتاج تقدير ما، المزيد عن العالم الذي تسكن فيه برامجك.

ستتمكن بواسطة تعلم التقدير، وبواسطة تطوير هذه المهارة إلى الحدّ الذي تشعر فيه بالحدس لحجم الأشياء، من إظهار قدرة سحريّة واضحة على تحديد جدواها. فعندما يقول أحدهم "سنرسل النسخ الاحتياطي عبر اتصال الشبكة إلى S3"، ستتمكن من معرفة ما إذا كان هذا عمليًا أم لا. وستكون قادرًا في أثناء كتابتك للشِفرة، من معرفة أي من الأنظمة الفرعيّة تحتاج إلى تحسين وأيّها يمكن تركها كما هي.

نصيحة ٢٣	قدّر لتجنب المفاجآت

كمكافأة، سنكشف في نهاية هذا القسم عن الجواب الصحيح الوحيد الذي ستقدمه كلما طُلب منك تقديرًا.

ما مدى الدقّة لتكون دقّة كافية؟

إن جميع الإجابات إلى حدّ ما هي تقديرات. تختلف فقط بأن بعضها أكثر دقّة من البعض الآخر. لذا فإن السؤال الأول الذي يجب أن تطرحه على نفسك عندما يسألك أحد الأشخاص عن تقدير ما هو السياق الذي ستؤخذ فيه إجابتك. هل يحتاجون إلى دقّة عالية، أم أنهم يبحثون عن رَقَم تقريبيّ؟

إن أحد الأشياء المثيرة للاهتمام حول التقدير هو أن الوحدات (units) التي تستخدمها تُحدث فرقًا في تفسير النتيجة. إذا قلت أن شيئًا ما سيستغرق حوالي 130 يوم عمل، فإن النّاس يتوقعون أن تقترب النتيجة كثيرًا من هذا التقدير. ومع ذلك، إذا قلت "آه، حوالي ستة أشهر"، فإنّهم يعرفون عندها أن ذلك يعني أن يتوقعوا النتيجة في وقت ما بين خمسة وسبعة أشهر من الآنَ. يمثّل كلا الرَقَمين نفس المدّة، ولكن "130 يومًا" ربما تعني ضمنًا درجة أعلى من الدقّة مما تشعر به. نوصي بقياس تقديرات الوقت على النحو التالي:

المدّة	تقدّر المدّة بـ
1-15 يوم	يوم
3-6 أسابيع	أسابيع
8-20 أسبوع	أشهر

أكثر من 20 أسبوع	فكّر مليًا قبل إعطاء تقدير

لذلك، إذا قرّرت بعد القيام بجميع الأعمال الضرورية، أن المشروع سيستغرق 125 يوم عمل (25 أسبوعًا)، فقد ترغب في تقديم تقدير بـ "حوالي ستة أشهر".

تنطبق نفس المفاهيم على تقديرات أي كميّة: اختر وحدات إجابتك لتظهر الدقّة التي تنوي إيصالها.

من أين تأتي التقديرات؟

تستند جميع التقديرات إلى نماذج المشكلة. ولكن قبل أن نتعمّق في تقنيات نماذج البناء، علينا أن نذكر خدعة تقدير أساسيّة والتي تقدّم دائمًا إجابات جيّدة: اسأل شخصًا فعل ذلك سابقًا فعلًا. ألقِ نظرة قبل أن تلتزم كثيرًا ببناء النماذج على شخص كان في موقف مشابه سابقًا. وانظر كيف حُلّت مشكلتهم. من غير المحتمل أن تجد تطابقًا تامًا، لكنك ستفاجأ بعدد المرّات التي يمكنك فيها الاعتماد على تجارِب الآخرين بنجاح.

افهم ما يُطلب منك

إن الجزء الأول من أي عملية تقدير هو بناء فهم لما يتم طلبه. فبالإضافة إلى مشكلات الدقّة التي نُوقشت أعلاه، تحتاج إلى فهم مجال النطاق. غالبًا ما يكون هذا ضمنيًا، ولكن عليك أن تعتاد التفكير في المجال قبل البَدْء في التخمين. غالبًا ما يمثل المجال الذي تختاره جزءًا من الإجابة التي تقدمها: "بافتراض عدم وجود حوادث مرورية ووجود غاز في السيارة، يجب أن أكون هناك في غضون 20 دقيقة".

ابنِ نموذجًا للنظام

هذا هو الجزء الممتع من التقدير. ابنِ نموذجا عقليًا أساسيًا وغير معقّد بناءً على فهمك للسؤال المطروح. إذا كنت تقدّر أوقات الاستجابة، فقد يتضمن نموذجك خادمًا ونوعًا من الحِمل (traffic) الواصل. وبالنسبة للمشروع، قد يكون النموذج هو الخطوات التي تستخدمها مؤسستك في أثناء التطوير، إلى جانب صورة تقريبيّة جدًا لكيفيّة تنفيذ النظام.

يمكن أن يكون بناء النموذج خلّاقًا ومفيدًا على المدى الطويل. غالبًا ما تؤدي عملية بناء النموذج إلى اكتشاف الأنماط والعمليات الأساسيّة التي لم تكن ظاهرة على السطح. قد ترغب أيضًا في إعادة فحص المطلب الأصلي: "لقد طُلب منك تقديرًا لإجراء X. على كل حال، يبدو أن Y، أحد متغيرات X، يمكن إجراؤه في نصف الوقت تقريبًا، وتبقى فقط مِيزة واحدة."

يقدّم بناء النموذج عدم دقّة في عملية التقدير. هذا أمر لا مفر منه، ولكنه مفيد أيضًا. أنت تتاجر في بساطة النموذج من أجل الدقّة. قد يمنحك مضاعفة الجهد على النموذج زيادةً طفيفةً في الدقّة. سوف تخبرك تجربتك متى تتوقف عن إعادة التنقيح.

قسّم النموذج إلى مكونات

بمجرّد أن يكون لديك نموذج، يمكنك تحليله إلى مكونات. ستحتاج إلى اكتشاف القواعد الرياضيّة التي تصف كيفيّة تفاعل هذه المكونات. يساهم المكوّن في بعض الأحيان بقيمة واحدة تُضاف إلى النتيجة. قد توفّر بعض المكونات عوامل مضاعفة (multiplying factors)، في حين أن البعض الآخر قد يكون أكثر تعقيدًا (مثل تلك التي تحاكي وصول الحِمل إلى العقدة).

ستجد أن كل مكوّن سيحتوي عادةً على معاملات تؤثر على كيفيّة مساهمته في النموذج العام. عرّف ببساطة كل مُعامل في هذه المرحلة.

أعطِ قيمة لكل معامل

بمجرّد عزل المعاملات، يمكنك تعيين قيمة لكل منها. تتوقع إدخال بعض الأخطاء في هذه الخطوة. تكمن الحيلة في تحديد أي المعاملات لها أكبر تأثير على النتيجة والتركيز على فهمها فهمًا صحيحًا. تكون المعاملات التي تُضاف قيمها إلى نتيجة ما عادةً أقل أهميّة من تلك التي تُضرب أو تُقسّم. فقد يؤدي مضاعفة سرعة الخط إلى مضاعفة كَمّيّة البيانات المستلمة في الساعة، بينما لن يكون لإضافة تأخير عبور لمدّة 5 ملي ثانية تأثيرًا ملحوظًا.

يجب أن يكون لديك طريقة مبرّرة لحساب هذه المعامِلات الهامّة. فبالنسبة لمثال الرتل، قد ترغب في قياس المعدل الفعليّ لوصول المناقلات (transaction) للنظام الحالي، أو إيجاد نظام مماثل للقياس. وبشكل مشابه، يمكنك قياس الوقت الحالي الذي يستغرقه تخديم الطلب، أو التوصل إلى تقدير باستخدام التقنيات الموضّحة في هذا القسم. في الواقع، غالبًا ما ستجد نفسك تستند في تقديرك على التقديرات الفرعيّة الأخرى. وهو المكان الذي ستتسلل فيه أكبر أخطائك.

احسب الإجابات

سيكون للتقدير إجابة واحدة في أبسط الحالات فقط. قد تكون سعيدًا لقول "يمكنني المشي خمس كتل عبر المدينة في غضون 15 دقيقة". ومع ذلك، عندما تصبح الأنظمة أكثر تعقيدًا، ستحتاج إلى تحديد إجاباتك. أجري حسابات متعددة، مع تغيير قيم المعاملات الحرجة، حتى تستطيع تحديد أي منها هو من يقود النموذج حقًا. يمكن أن يكون جدول البيانات مساعدة كبيرة. ثم صغ إجابتك بدلالة هذه المعاملات. "وقت الاستجابة هو ما يقرب من ثلاثة أرباع الثانية إذا كان النظام يحتوي على أقراص SSD وذاكرة 32 جيجا بايت، وثانية واحدة بذاكرة 16 جيجا بايت". (لاحظ كيف أن "ثلاثة أرباع الثانية" تنقل شعورًا مختلفًا بالدقّة عن 750 ملي ثانية.)

تحصل خلال مرحلة الحساب على إجابات تبدو غريبة. لا تتسرع في استبعادها. إذا كانت حساباتك صحيحة، فربما يكون فهمك للمشكلة خاطئا أو نموذجك خاطئًا. هذه معلومة قيّمة.

تتبع براعتك في التقدير

نعتقد أن تسجيل تقديراتك هي فكرة رائعة حتى تتمكن من معرفة مدى قربك من الواقع. إذا كان التقدير العام يتضمن حساب التقديرات الفرعيّة، فتتبعها أيضًا. غالبًا ما ستجد أن تقديراتك جيّدة جدًا - في الواقع، ستتوقع ذلك. عندما يتبيّن لك أن التقدير خاطئ، لا تتغاض عنه فقط وتذهب بعيدًا - اكتشف السبب. ربما اخترت بعض المُعامِلات التي لا تتناسب مع حقيقة المشكلة. أو ربما كان نموذجك خاطئا. مهما كان السبب، خذ بعض الوقت للكشف عمّا حدث. وسيكون تقديرك التالي أفضل إذا فعلت ذلك.

تقدير جدولة المشروع.

عادةً ما يُطلب منك تقدير المدّة التي سيستغرقها شيء ما. إذا كان هذا "الشيء" معقدًا، فقد يكون من الصعب جدًا إعطاء التقدير، سنلقي نظرة في هذا القسم على طريقتين لتقليل عدم اليقين هذا.

طلاء القذيفة

"كم من الوقت سيستغرق طلاء المنزل؟"

"حسنًا، إذا سارت الأمور على ما يرام، وكان لهذا الطلاء التغطية التي يزعمون فإنّها قد تصل إلى 10 ساعات. لكن هذا غير مرجّح: أعتقد أن الرَّقَم الأكثر واقعيّة أقرب إلى 18 ساعة. وبالطبع، إذا أصبح الطقس سيئًا، يمكن أن يمتد إلى 30 ساعة أو أكثر".

هكـذا يُقـدّر النّـاس في العـالم الحقيقي. ليس بـرَقَم واحـد (مـا لم تجـبرهم على إعطائـك رقمًـا واحـدًا) ولكن بمجموعـة من السيناريوهات.

اعتمدت البحريّة الأمريكيّة هذا الأسلوب في التقدير عنـدما احتـاجت إلى تخطيـط مشـروع غواصـة بـولاريس، بمنهجيّـة أطلقـوا عليها تقنيّة مراجعة تقييم البرامج Program Evaluation and Review Technique أو PERT.

يوجـد لكـل مهمّـة PERT تقـدير متفائـل، وتقـدير مـرجّح، وتقـدير متشـائم. تُرتّب المهـام في شـبكة اعتماديّـة، ثم تسـتخدم بعض الإحصاءات البسيطة لتحديد أفضل وأسوأ الأوقات المحتملة للمشروع كليًا.

يعدّ استخدام مجال من القيم كهذا طريقة رائعة لتجنّب أحد الأسباب الأكثر شيوعًا لخطأ التقدير: وهو حشو رَقَم لأنك غير متأكّد. بدلاً من ذلك، فإن الإحصائيات الكامنة وراء PERT تنشر عدم اليقين بالنسبة لك، مما يمنحك تقديرات أفضل للمشروع بِرُمَّته.

ومع ذلك، لسنا من كبار المعجبين بهذا. يميل النّـاس إلى إنتـاج مخططـات بحجم الجـدار لجميـع المهـام في المشـروع، ويعتقـدون ضمنيًا أنّه لمجرّد أنهم استخدموا صيغةً رياضيّة، فإن لديهم تقديرًا دقيقًا من المحتمـل أن الوضـع ليس كـذلك، لأنهم لم يفعلـوا ذلـك من قبل.

تناول الفيل

نجد أن الطريقة الوحيدة لتحديد الجدولة الزمنيّة للمشروع غالبًا ما تكون بواسـطة اكتسـاب الخـبرة في نفس المشـروع. لا ينبغي أن يشكل هذا مفارقةً إذا كنت تمارس تطويرًا تدريجيًا، مُكرّرا الخطوات التالية بشرائح رقيقة جدًا من الوظائف:

- التحقّق المتطلبات
- تحليل المخاطر (وتحديد أولويات العناصر الأكثر خطورة بشكل مبكر)
- التصميم والتنفيذ والتكامل
- التحقّق الصّحة بالنسبة للمستخدمين

في البداية، قد تكون لديك فقط فكرة غامضة عن عدد التكرارات المطلوبة أو المدّة التي قد تستغرقها. تتطلب بعض الطرق تثبيت ذلك كجزء من الخِطَّة الأوليّة ؛ ومع ذلك، فإن هذا خطأ بالنسبة للجميع باستثناء أكثر المشروعات بساطة. ما لم تكن تنفذ تطـبيق مشـابه لتطبيق سابق، مع نفس الفريق وبنفس التقنية، فأنت تخمّن فقط.

لذلك تكمل كتابة الشِـفرة واختبـار الوظيفـة الأوليّـة وتضـع علامـة على أنهـا نهايـة التكـرار الأول. يمكنـك بنـاءً على هـذه التجربـة تحسين تخمينك الأولي حول عدد التكرارات وما يمكن تضمينه في كل منها. تتطور التحسينات مرّة تلو الأخـرى، وتنمـو معهـا الثقـة في الجدول الزمني. غالبًا ما يتم هذا النوع من التقدير خلال مراجعة الفريق في نهاية كل دورة تكرارية.

هكذا أيضًا تقول المزحة القديمة من أجل أن تأكل فيلًا: عليك أن تأكل قضمة واحدة كل مرّة.

نصيحة ٢٤ كرّر الجدولة الزمنيّة باستخدام الشِفرة

قد لا يكون هذا شائعًا لدى الإدارة، والتي تريد عادةً رقمًا واحدًا صعبًا وسريعًا حتى قبل بَدْء المشروع. سيكون عليك مساعدتهم على فهم أن الفريق وإنتاجيتهم والبيئة ستحدّد الجدولة الزمنية. ستقدّم لهم تقديرات الجدولة الأكثر دقّة قدر المستطاع بواسطة صياغة، وتنقيح الجدولة كجزء من كل تكرار.

ماذا أقول عندما أسئل عن تقدير

تقول "سأعود إليك".

دائمًا ما تحصل أفضل نتائج إذا أبطأت العملية وأمضيت بعض الوقت في اتباع الخطوات الموضحة في هذا القسم. ستعود التقديرات المقدّمة في آلة القهوة (مثل القهوة) لتراودك.

الأقسام ذات الصلة

- الموضوع 7، تواصل!، في الصفحة 44
- الموضوع 39، سرعة الخوارزمية، في الصفحة 229

تحدّيات

- ابدأ في الاحتفاظ بسجل لتقديراتك. تتبع مدى دقّة كل منها. إذا كان الخطأ أكبر من 50٪، فحاول العثور على الخطأ الذي حدث في التقدير.

تمارين

التمرين 9 (إجابة محتملة في الصفحة 323)

إذا سُئلت "أيّهما لديه عرض حُزْمَة أعلى: اتصال شبكة بسرعة 1 جيجابت في الثانية أو شخص يسير بين جهازي حاسوب حاملًا في جيبه جهاز تخزين ممتلئ بسعة 1 تيرابايت؟" ما القيود التي ستضعها على إجابتك لضمان أن نطاق إجابتك صحيح؟ (على سبيل المثال، قد تقول أنّ الوقت المستغرق للوصول إلى جهاز التخزين مُهمل.)

تمرين 10 (إجابة محتملة في الصفحة 323)

إذاً، أيّ عرض حُزْمَة هو الأعلى؟

الفصل الثالث:

الأدوات الأساسيّة

3

يبدأ كل صانع رحلته بمجموعة أساسيّة من الأدوات ذات الجودة الجيّدة. قد يحتاج النجار إلى مساطر، ومقاييس، ومجموعة من المناشير، وبعض المساحج[46] الجيّدة، وأزاميل دقيقة، ومثاقب ومشابك، ومطارق، وملازم. هذه الأدوات سيتم اختيارها بعناية، وبناءها لتدوم، ولتؤدي أعمالاً محدّدة مع القليل من التداخل مع الأدوات الأخرى، وربما الأهم من ذلك، أنها ستفي بالغرض بين يدي نجارٍ ناشئ.

ثم تبدأ عملية التعلّم والتكيّف. سيكون لكل أداة سماتها الفريدة وخصوصيتها، وستحتاج إلى تعاملها الخاص. لا بد من شحذ كل منها بطريقة خاصّة، أو الحافظ عليها. بمرور الوقت، سيُستهلك كل منها حسب الاستخدام، حتى تبدو القبضة كأنها قالب ليد النجار ويتوافق سطح القطع تمامًا مع الزاوية التي تُحمل بها الأداة. تصير الأدوات عند هذه النقطة، قنوات من دماغ المُصنع إلى المنتج النهائي- لقد أصبحت امتداداً ليديه. مع الزمن، سيضيف النجار أدوات جديدة، مثل أداة قطع الألواح ومنشار متري الموجهة بالليزر، ومعدات ربط مفاصل الأخشاب (dovetail jigs)- وكلها تقنيات رائعة. ولكن يمكنك أن تراهن على أن النجارين سيكونون أكثر سعادة مع إحدى هذه الأدوات الأصلية في اليد، وهم يشعرون بالمسحج يغني وهو ينزلق فوق الخشب.

تُضخم الأدوات موهبتك. فكلما كانت أدواتك أفضل، وعرفت كيفيّة استخدامها بشكل أفضل، كلما أصبحت أكثر إنتاجيّة. ابدأ بمجموعة أساسيّة من الأدوات القابلة للتطبيق للأغراض العامّة. وأثناء اكتسابك الخبرة، ومواجهتك لمتطلبات خاصّة، ستضيف إلى هذه المجموعة الأساسيّة. تطلّع مثل مبتكر، بأن تضيف إلى صندوق أدواتك بانتظام. ابحث دائمًا عن طرق أفضل للقيام بالأشياء. إذا واجهت موقفًا تشعر فيه أن أدواتك الحاليّة لا تستطيع إنجازه، دوّن ملاحظة للبحث عن شيء مختلف أو أكثر فعاليّة والذي قد يساعد. دع الحاجة تقود مكتسباتك.

يرتكب العديد من المبرمجين الجدد خطأ اعتماد أداة قوة واحدة، مثل بيئة التطوير المتكاملة (IDE)مثلًا، فلا يغادرون أبدًا واجهتها المريحة. يُعدّ هذا أمراً خاطئاً فعلًا. يجب أن تكون مرتاحاً فيما هو أبعد من الحدود التي تفرضها بيئة التطوير هذه. الطريقة الوحيدة للقيام بذلك هي الحفاظ على مجموعة الأدوات الأساسية مشحوذة وجاهزة للاستخدام.

سنتحدث في هذا الفصل عن الاستثمار في صندوق أدواتك الأساسي. كما هو الحال مع أي مناقشة جيدة حول الأدوات، سنبدأ في قوة النص الواضح بالنظر إلى موادك الخام أي الأشياء التي ستشكلها. من هناك سننتقل إلى طاولة العمل، أو في حالتنا هي الحاسوب. كيف يمكنك استخدام جهاز الحاسوب لتحقيق أقصى استفادة من الأدوات التي تستخدمها؟ سنناقش هذا في لعاب شل. الآن بعد أن أصبح لدينا مادة خام ومنصّة للعمل عليها، سننتقل إلى الأداة التي قد تستخدمها أكثر من أي شيء آخر، محرّرك. سنقترح في قوّة التعديل، طرقًا تجعلك أكثر كفاءة.

لضمان ألا نفقد أبدًا أيًا من عملنا الثمين، يجب أن نستخدم دائماً نظام التحكم في الإصدار — حتى بالنسبة للأشياء الشخصيّة مثل الوصفات أو الملاحظات.

46 **[المترجم]** المسحج: آلةٌ يدوية يُبْرَي بها الخشَب (الجمع: مَسَاحج)

وبما أن مورفي كان متفائلًا حقًا، فلا يمكنك أن تكون مبرمجاً عظيماً حتى تصبح ماهراً للغاية في تنقيح الأخطاء.

ستحتاج إلى بعض الغراء لربط الكثير من الروعة معًا. سنناقش بعض الاحتمالات في معالجة النص.

أخيراً، لا يزال الحبر الجاف أفضل من أفضل ذاكرة. ابق متتبعاً لأفكارك ولسجلك، كما وصفنا في دفتر يوميات الهندسة.

اقضِ بعض الوقت في تعلم استخدام هذه الأدوات، وستفاجأ في مرحلة ما لاكتشاف أصابعك وهي تتحرك فوق لوحة المفاتيح، تعالج النص دون تفكير واعي. تصبح الأدوات حينها امتداداً ليديك.

الموضوع 16. قوّة النص الواضح (Plain Text)

كمبرمجين عمليين، فإن موادنا الأساسيّة ليست من الخشب أو الحديد، إنها معرفة. نجمع المتطلبات كمعرفة، ثم نعبّر عن تلك المعرفة في التصميمات والتطبيقات والاختبارات والمستندات الخاصة بنا. ونحن نعتقد أن أفضل صيغة لتخزين المعرفة باستمرار هي النص الواضح " البسيط"[47]. مع نص بسيط، نعطي أنفسنا القدرة على التعامل مع المعرفة، سواء يدويًا أم برمجيًا، باستخدام كل الأدوات المتاحة لنا تقريبًا.

المشكلة مع معظم الصيغ الثنائية هي أن السياق ضروري لفهم البيانات بمعزل عن البيانات نفسها. فأنت تفصل بشكل مصطنع البيانات عن معناها. قد يتم تشفير البيانات أيضًا؛ فتصبح بلا معنى إطلاقاً دون منطق التطبيق لتحليلها. بكافة الأحوال، باستخدام نص واضح، يمكنك تحقيق تدفق بيانات ذاتي التوصيف مستقل عن التطبيق الذي أنشأه.

ما النص الواضح؟

يتكون *النص الواضح* من محارف قابلة للطباعة في شكل ينقل المعلومات. يمكنها أن تكون بسيطة كقائمة تسوّق:

```
* milk
* lettuce
* coffee
```

أو معقّدة مثل مصدر هذا الكتاب (نعم، إنه مكتوب بنص بسيط، بكثير من الاستياء من قبل الناشر، الذي أرادنا أن نستخدم معالج النصوص) القسم الخاص بالمعلومات مهم. ما يلي ليس نصًا عاديًا مفيدًا:

```
hlj;uijn bfjxrrctvh jkni'pio6p7gu;vh bjxrdi5rgvhj
```

ولا هذا:

```
Field19=467abe
```

ليس لدى القارئ فكرة عمّا هي الأهمية الممكنة لـ 467abe .

نحن نريد من نصنا الواضح أن يكون قابلاً للفهم من قبل البشر.

47 **[المترجم]** في الحوسبة، النص البسيط أو العادي هو البيانات (على سبيل المثال، محتويات المِلَفّ) التي لا تمثل سوى محارف من المواد التي يمكن قراءتها، وليس تمثيلها الرسومي ولا الكائنات الأخرى (الصور، وما إلى ذلك). وقد يتضمن أيضًا عددًا محدودًا من المحارف التي تتحكم في الترتيب البسيط للنص، مثل فواصل الأسطر أو محارف الجدولة.
https://ar.wikipedia.org/wiki/%D9%86%D8%B5_%D8%A8%D8%B3%D9%8A%D8%B7

نصيحة ٢٥ ابق المعرفة مخزنة في شكل نص واضح

قوة النص

لا يعـني النص البسـيط أن النص غـير مُهيكـل؛ إن HTML وJSON وYAML ومـا إلى ذلـك كلهـا نص واضح. وكـذلك غالبيـة البروتوكولات الأساسيّة على الإنترنت، مثل HTTP وSMTP وIMAP وغيرها. وذلك لبعض الأسباب الوجيهة:

- التأمين من التقادم
- الاستفادة من الأدوات الموجودة
- اختبار أسهل

التأمين من التقادم

إن أشكال البيانات القابلة للقراءة من قبل الإنسان، والبيانات ذاتية التوصيف، سوف تدوم أكثر من جميع أشـكال البيانـات الأخـرى ومن التطبيقات التي أنشأتها تمامًا. مادام أن البيانات باقية، سـيكون لـديك فرصـة لتكـون قـادراً على اسـتخدامها – من المحتمـل لـزمن طويل بعد فَنَاء التطبيق الأصلي الذي كتبها-.

بإمكانك تحليل مِلَفّ مثل هذا بمعرفة جزئية بتنسيقه؛ لكن مع معظم المِلَفّات ذات الصيغ الثنائيـة، يجب أن تعـرف جميـع تفاصـيل التنسيق بِرُمَّته من أجل تحليله بنجاح.

افترض أن تم إعطاؤك مِلَفّ بيانات من بعض الأنظمة القديمة "المتوارثة"[48]. أنت تعرف القليل عن التطبيق الأصلي؛ كـل مـا يهمـك هو أنه يحفظ قائمة بأرقام الضمان الاجتماعي للعملاء، والتي تحتاج إلى العثور عليها واستخراجها من بين البيانات، تجد

```
<FIELD10>123-45-6789</FIELD10>...
<FIELD10>567-89-0123</FIELD10>...
<FIELD10>901-23-4567</FIELD10>
```

يمكنك بواسطة التعرّف على تنسيق رَقَم الضمان الاجتماعي، كتابة برنامَج صغير سريعًا لاستخراج تلك البيانات - حتى لـو لم يكن لديك معلومات حول أي شيء آخر في المِلَفّ.

ولكن تخيّل لو كان المِلَفّ قد نُسق بهذه الطريقة بدلاً مما سبق:

```
AC27123456789B11P
...
XY43567890123QTYL
...
6T2190123456788AM
```

قد لا تتعرف إلى أهمية الأرقام بنفس السهولة.

48 تصبح جميع البرمجيات برمجيات قديمة "موروثة" بمجرّد كتابتها.

هذا هو الفرق بين *ما يمكن قراءته وما يمكن فهمه من قبل الإنسان*.

بينما نحن في هذا الأمر، فإن FIELD10 لا يساعد كثيرًا أيضا. شيء من قبيل

<SOCIAL-SECURITY-NO>123-45-6789</SOCIAL-SECURITY-NO>

يجعل العملية شديدة الوضوح - ويضمن أن البيانات سيتجاوز أمدها أي مشروع يُنشئها.

الفاعليّة

فعليًا إن كل أداة في عالم الحوسبة، من أنظمة التحكم في الإصدار إلى المحرّرات وإلى أدوات سطر الأوامر، يمكن أن تعمل على نص واضح.

على سبيل المثال، افترض أن لديك نشر إنتاج نهائي لتطبيق كبير مع مِلَفّ ضبط معقّد خاص بالموقع. إذا كان هذا المِلَفّ بصيغة نص واضح، يمكنك وضعه تحت نظام التحكم في الإصدار (انظر الموضوع 19، التحكم في الإصدار، في الصفحة 105)، بحيث تحتفظ تلقائيًا بسجل لجميع التغييرات. تتيح لك أدوات مقارنة المِلَفّات مثل diff وfc رؤية التغييرات التي تم إجراؤها في لمحة بصر، بينما يسمح لك "sum" بإنشاء تدقيق المجموع checksum[49] لمراقبة المِلَفّ من أجل التعديل العرضي (أو المقصود"الخبيث").

فلسفة اليونكس

تشتهر يونكس بكونها مصممة بناءً على فلسفة الأدوات الصغيرة الحادة، كل منها مُعدّ للقيام بشيء واحد بشكل جيد. يتم تمكين هذه الفلسفة باستخدام تنسيق أساسي مشترك - مِلَفّ نص واضح سطري التوجه. line-oriented قواعد بيانات مُستخدَمة لإدارة النظام (المستخدمون وكلمات المرور وضبط الشبكات وما إلى ذلك) يُحتفظ بها جميعاً كمِلَفّات نص عادي.

(تحتفظ بعض الأنظمة أيضًا بصيغة ثنائية لقواعد بيانات معيّنة من أجل تحسين الأداء. يتم الاحتفاظ بنسخة النص الواضح كواجهة إلى الإصدار الثنائي).

عند تعطل نظام، قد لا تتعامل سوى مع أدنى بيئة لاستعادته (ربما لا تتمكن من الوصول إلى برامج تشغيل الرسوميّات، على سبيل المثال). مثل هذه الحالات يمكن أن تجعلك تقدّر حقًا بساطة النص الواضح.

النص الواضح أيضا أسهل للبحث. إذا كنت لا تستطيع تذكر أي مِلَفّ ضبط يدير النسخ الاحتياطية للنظام الخاص بك، ستخبرك تعليمة grep -r backup /etc. السريعة بذلك.

49 **[المترجم]** تدقيق المجموع (بالإنجليزية: Checksum) هو شكل من أشكال فحص صحّة البيانات، وهو أحد الإجراءات المبسّطة للتحقق من سلامة البيانات المرسلة عبر شبكة (كالإنترنت مثلاً) أو المخزّنة في وسيطة ما (قرص مدمج مثلاً)، وهي تسمح باكتشاف وجود الأخطاء في هذه البيانات. يعمل تدقيق المجموع بواسطة خوارزمية تقرأ البيانات وتوليد عدد ثابت من البتات (تبعاً للخوارزمية)، وهذه البتات تُستخدم للمقارنة مع ناتج تدقيق المجموع التالي بحيث يجب أن يتطابق الناتجان إن بقيت البيانات سليمة دون أي تغيير.

اختبار أسهل

إذا كنت تستخدم نصًا واضحًا لإنشاء بيانات تجريبيّة لقيادة اختبارات النظام، فمن السهل إضافة بيانات الاختبار أو تحديثها أو تعديلها *دون الحاجة إلى إنشاء أي أدوات خاصة للقيام بذلك*. وبالمثل، يمكن تحليل خرج النص الواضح الناتج من اختبارات الانحدار (regression tests) بشكل بسيط باستخدام أوامر *شِل* أو برنامَج نصيّ بسيط.

القاسم المشترك الأصغر

حتى في مستقبل الوكلاء الأذكياء المعتمدين على سلسلة الكتل "بلوك تشاين" (blockchain)[50] المسافرين بمفردهم في عالم الإنترنت الموحش شديد الْخَطَر، والمتفاوضين على تبادل البيانات بينهم، سيظل المِلَفّ النصي الشائع موجودًا. في الواقع، في بيئات مختلطة يمكن أن تفوق مزايا النص الواضح جميع العوائق.

تحتاج إلى التأكّد أن جميع الأطراف يمكنها التواصل باستخدام معيار مشترك. النص الواضح هو ذلك المعيار.

الأقسام ذات الصلة

- الموضوع 17، ألعاب شِل، الصفحة 99
- الموضوع 21، معالجة النص، في الصفحة 118
- الموضوع 32، الضبط، في الصفحة 188

تحديات

- صمّم قاعدة بيانات صغيرة لدفتر العناوين (الاسم ورَقَم الهاتف وما إلى ذلك) باستخدام تمثيل ثنائي مباشر في لغتك المفضلة. أجري ذلك قبل قراءة بقيّة هذا التحدي.

  - ترجم ذاك التنسيق إلى تنسيق نص واضح باستخدام XML أو JSON.

  - أضف من أجل كل إصدار، حقلًا جديدًا متغير الطول يسمى *الاتجاهات (directions)* يمكنك أن تُدخِل فيه الاتجاهات إلى منزل كل شخص.

  ما هي المشكلات التي تظهر فيما يتعلق بالإصدارات وقابلية التوسع؟ أي تنسيق أسهل في التعديل؟ ماذا عن تحويل البيانات الموجودة؟

---

50 **[المترجم]** blockchain هو دفتر أستاذ رَقَمي لامركزي، وموزع، وغالبًا ما يكون عامًا يتكون من سجلات تسمى الكتل تُستخدم لتسجيل المعاملات عبر العديد من أجهزة الحاسوب بحيث لا يمكن تغيير أي كتلة مشاركة بأثر رجعي، دون تغيير جميع الكتل اللاحقة.

الموضوع 17. ألعاب شِل Shell

يحتاج كل نجّار إلى منضدة عمل جيّدة وصلبة وموثوقة في مكان ما لتثبيت قطع العمل على ارتفاع مناسب في أثناء تشكيلها. تصبح منضدة العمل هذه مركزًا لورشة الخشب، ويعود المصنّع إليها مرارًا وتكرارًا حتى تتشكل القطعة.

بالنسبة لمبرمج يعالج مِلَفّات نصيّة، فإن منضدة العمل له هي سطر أوامر الشِل shell[51]. يمكنك استدعاء مخزن كامل من الأدوات من محث شِل (shell prompt) باستخدام مسارات التنفيذ (pipes) للجمع بينها بطرق لم يحلم بها مطوروها الأصليّون حتى. من *شِل* يمكنك تشغيل التطبيقات، والمنقّحات، والمتصفحات، والمحرّرات والأدوات الخدميّة. يمكنك البحث عن المِلَفّات، والاستعلام عن حالة النظام، وفلترة الخرج. كما يمكنك بواسطة برمجة *الشل*، بناء أوامر *ماكرو* معقّدة للأنشطة التي تقوم بها بشكل متكرّر.

بالنسبة للمبرمجين الذين نشؤوا على واجهات واجهة المستخدم الرسوميّة GUI وبيئات التطوير المتكاملة (IDEs)، قد يبدو هذا موقفًا متطرفًا. في النهاية، هل بإمكانك فعل كل شيء بصورة متكافئة بواسطة التأشير والنقر بالفأرة؟

الجواب البسيط هو "لا". واجهات GUI رائعة، ويمكن أن تكون أسرع وأكثر ملاءمة لبعض العمليات البسيطة. إن نقل المِلَفّات وقراءة وكتابة البريد الإلكتروني، وبناء مشروعك ونشره كلها أمور قد ترغب في تنفيذها في بيئة رسوميّة. ولكن إذا قمت بكل عملك باستخدام واجهات المستخدم الرسوميّة، فإنك تخسر الإمكانات الكاملة لبيئتك. لن تتمكن من أتمتة المهام الشائعة، أو استخدام القوة الكاملة للأدوات المتاحة لك. ولن تتمكن من دمج أدواتك لإنشاء *أدوات الماكرو* المخصّصة. إن مِيزة واجهة المستخدم الرسوميّة هي في أن- ما تراه هو ما تحصله (WYSIWY)[52]. وسلبيتها هي في أن- ما تراه هو كل ما تحصله (WYSIAYG)[53].

تقتصر بيئات واجهة المستخدم الرسوميّة عادةً على القدرات التي أرادها مصمموها. إذا كنت بحاجة إلى تجاوز حدود النموذج الذي قدّمه المصمم، فلن تكون محظوظا غالبًا - وفي أغلب الأحيان تحتاج لتجاوز النموذج. لا يقتصر المبرمجون العمليون على اقتطاع الشِفرة أو تطوير نماذج الكائنات، أو كتابة التوثيق، أو أتمتة عملية البناء - نحن نقوم *بكل* من هذه الأشياء.

يقتصر نطاق أي أداة معينة عادةً على المهام التي من المتوقع أن تقوم بها هذه الأداة. على سبيل المثال، افترض أنك بحاجة إلى دمج المعالجة المُسبقة للشِفرة (لتنفيذ *التصميم حسب العقد* أو موجهات المعالجة المتعددة، أو بعضًا من هذا القبيل) في بيئة التطوير المتكاملة الخاصة بك لا يمكنك القيام بذلك. ما لم يقدّم مصمم IDE صراحة خطافات (Hooks) لهذه الإمكانية.

نصيحة ٢٦	**استخدم قوّة أوامر شِل.**

51 **[المترجم]** في الحوسبة، تُعد Shell "الصدفة" (ويطلق عليها أحيانا القشرة أو الغلاف) واجهة مستخدم للوصول إلى خِدمَات نظام التشغيل. عمومًا، تستخدم أصداف نظام التشغيل إما واجهة سطر الأوامر (CLI) أو واجهة المستخدم الرسوميّة (GUI). وتدعى بالغلاف لأنها تشكل الطبقة الخارجية حول نظام التشغيل.
تتطلب الأصداف الرسومية عبئًا منخفضًا من مستخدمي الحاسوب المبتدئين، وتتميز بأنها سهلة الاستخدام. ونظرًا لأنها تأتي أيضًا مع عيوب معينة، فإن معظم أنظمة التشغيل التي تدعم واجهة المستخدم الرسومية توفر أيضًا أصداف CLI.

52 اختصار لـ (what you see is what you get)

53 اختصار لـ (what you see is *all* what you get)

اعتد استخدام الشِل وستحلّق إنتاجيتك. أأنت بحاجة إلى إنشاء قائمة بجميع أسماء الحزم الفريدة التي تم استيرادها صراحة بواسطة شِفرة جافا؟ ما يلي يخزنها في مِلَفّ يسمى "list":

sh/packages.sh

```
grep '^import ' *.java |
   sed -e's/^import *//' -e's/;.*$//' |
   sort -u >list
```

إذا لم تكن قد قضيت الكثير من الوقت في استكشاف إمكانات أوامر الشِل على الأنظمة التي تستخدمها، فقد يبدو هذا الأمر شاقاً. ومع ذلك، ابذل بعض الجهد في التعرّف على أداة الشِل الخاص بك وستبدأ الأمور بالتوضّح سريعًا.

جرّب أوامر شِل، وسُفاجأ إلى أي مدى ستجعلك أكثر إنتاجية.

الشِل الخاص بك

بنفس الطريقة التي سيُخصّص بها عامل النجارة مساحة العمل الخاصة به، يجب على المطوّر تخصيص الشِل الخاص به. وهذا يشمل عادة تغيير ضبط برنامَج الطرفيّة الذي تستخدمه.

تتضمن التغييرات الشائعة ما يلي:

- *تحديد سمات الألوان* يمكن قضاء العديد والعديد من الساعات في تجربة *كل* سمة (theme) متاحة عبر الإنترنت لشِل معين *على حِدَةٍ*.
- *ضبط المحث (prompt)*. المحث الذي يخبرك أن الشِل أصبح جاهزًا لتلقي الأوامر منك يمكن ضبطه ليعرض مباشرة أية معلومات ترغب بها (ومجموعة أشياء لا تريدها مطلقاً). التفضيلات الشخصيّة هي كل شيء هنا: نحن نميل إلى تفضيل محثات بسيطة، مع اختصار اسم الدليل الحالي وحالة التحكم في الإصدار مع الوقت.
- *الأسماء المستعارة ودوال شل*. بسّط سير عملك بواسطة تحويل الأوامر التي تستخدمها بكثرة إلى أسماء مستعارة بسيطة. ربما تحدث صندوق لينكس بانتظام، ولكن لا يمكنك تذكّر ما إذا كنت تقوم بالتحديث والترقية أم الترقية والتحديث. أنشئ اسم مستعار:

```
alias apt-up='sudo apt-get update && sudo apt-get upgrade'
```

- ربما قمت بحذف المِلَفّات عن طريق الخطأ باستخدام الأمر rm مرّة واحدة فقط في كثير من الأحيان؛ اكتب اسمًا مستعارًا حتى يتم عرضه دائمًا في المستقبل:

```
alias rm ='rm -iv'
```

- *إكمال الأوامر*. ستكمل معظم أنواع الشل أسماء الأوامر والمِلَفّات: اكتب الأحرف القليلة الأولى، وانقر tab، وسوف تملأ ما أمكنها. ولكن يمكنك أن تأخذه إلى أبعد من هذا الحدّ بكثير، وأن تضبط الشِل للتعرف على ما تدخله وتقديم الإكمال المحدّدة للسياق؛ حتى إن البعض يخصّصون الإكمال تبعًا للدليل الحالي.

ستقضي الكثير من الوقت في العيش في واحدة من هذه المحثات. كن مثل السلطعون الناسك واجعلها منزلك.

الأقسام ذات الصلة

- الموضوع 13، النماذج الأولية والملاحظات الملحقة، في الصفحة 78
- الموضوع 16، قوة النص الواضح، في الصفحة 95
- الموضوع 21، معالجة النص، في الصفحة 118
- الموضوع 30، برمجة التحويل، في الصفحة 169
- الموضوع 51، مجموعة البادئ العملية، في الصفحة 300

تحدّيات

- هل توجد أشياء تقوم بها حاليًا بشكل يدوي في واجهة المستخدم الرسوميّة؟ هل سبق ومرّرتَ تعليمات إلى الزملاء تنطوي على عدد من الخطوات الفرديّة "انقر فوق هذا الزر"، "حدّد هذا العنصر"؟ هل يمكن أتمتة ذلك؟
- كلما انتقلت إلى بيئة جديدة، احرص على معرفة ما هي *الشِل* المتاحة. تحقّق مما إذا كان بإمكانك إحضار الشِل الحالي معك.
- التحقيق في أبدال الشِل الحالية الخاصة بك. إذا صادفت مشكلة لا يمكن للشِل خاصتك أن تعالجها، انظر ما إذا كانت *شِل* بديلة تتعامل بشكل أفضل.

الموضوع 18. قوّة التعديل

لقد تحدثنا من قبل عن الأدوات وكونها امتدادًا ليدك. حسناً هذا ينطبق على المحرّرات أكثر من أي أداة برمجيّة أخرى. يجب أن تكون قادرًا على معالجة بالنص بسلاسة ما أمكن، لأن النص هو مادة البرمجة الخام.

في الإصدار الأول من هذا الكتاب، أوصينا باستخدام محرّر واحد لكل شيء: التعليمات البرمجية والوثائق والمذكرات (memos) وإدارة النظام وما إلى ذلك. لقد خفّفنا هذا الموقف قليلاً. يسعدنا أن تستخدم أكبر عدد ممكن من المحرّرات قدر ما تريد. نودّ منك فقط أن تعمل على اكتساب الطلاقة في كلّ منها.

نصيحة ٢٧	حقّق طلاقة في المحرّر.

لماذا هذا الأمر مهم؟ هل نقول أنك ستوفّر الكثير من الوقت؟ في الواقع نعم: على مدار عام، قد تكسب فعلًا أسبوعًا إضافيًا إذا جعلت عملية التحرير الخاصة بك أكثر كفاءة بنسبة 4% فقط وحررت لمدة 20 ساعة في الأسبوع.

لكن هذه ليست الفائدة الحقيقية. لا، المكسب رئيس هو أنه باكتسابك الطلاقة، بحيث لا يعود التفكير في آليات التحرير. ستقِل المدّة الزمنية بين التفكير في شيء ما وظهوره في المخزن المؤقت (buffer) للمحرّر. سوف تتدفق أفكارك، وستصبح برمجتك أكثر جدوى. إذا سبق لك أن علّمت شخصًا ما القيادة، فسوف تفهم الفرق بين شخص عليه أن يفكر في كل عمل يقوم به و سائق متمرس يتحكم في السيارة تلقائيًا.

ماذا تعني "طلاقة"؟

ما الذي يعد طلاقة؟ إليك قائمة التحدي:

- عند تحرير النص، قم بالتحريك والاختيار حسب المحرف أو الكلمة أو السطر والفقرة.
- عند تحرير الشِفرة، انتقل بواسطة وحدات نَحوِيّة مختلفة (محددات مطابقة، دوال، وحدات،...).
- أعد إزاحة(Reindent) الشِفرة بعد التغييرات.
- علق وألغ تعليق كتل من التعليمات البرمجية بأمر واحد.
- تراجع عن التغييرات ثم أعدها.
- قسّم نافذة المحرّر إلى لوحات متعددة، والتنقل بينها.
- انتقل إلى رَقّم سطر معين.
- افرز السطور المختارة.

- ابحث عن كل من السلاسل النصيّة والتعبيرات النظامية، وكرّر عمليات البحث السابقة.
- أنشئ مؤشرات متعددة مؤقتًا بناءً على تحديد أو على مطابقة نمط وعدّل النص في كل منهما بالتوازي.
- اعرض أخطاء التَّرْجَمَةً في المشروع الحالي.
- شغّل اختبارات المشروع الحالي.

هل يمكنك القيام بكل هذا دون استخدام الفأرة / لوحة التتبع (trackpad)؟

قد تقول إن المحرّر الحالي لا يمكنه القيام ببعض هذه الأشياء. ربما حان الوقت لتبديله؟

التحرك نحو الطلاقة

نشك في أن هناك أكثر من حفنة من النّاس يعرفون جميع الأوامر في أي محرّر قوي معين. نحن لا نتوقع منك ذلك أيضًا. بدلاً من ذلك، نحن نقترح عليك نهجاً أكثر واقعية: تعلّم الأوامر التي تجعل حياتك أسهل.

إن وصفة هذا الأمر بسيطة إلى حدّ ما.

أولاً، راقب نفسك في أثناء التحرير. في كل مرّة تجد نفسك تفعل شيئاً بشكل متكرر، تعوّد التفكير بـ "يجب أن يكون هناك طريقة أفضل. " ثم ابحث عنها.

بمجرّد اكتشاف مِيزة جديدة ومفيدة، تحتاج الآن إلى تثبيتها في ذاكرة عضلاتك، لتتمكن من استخدامها دون تفكير. الطريقة الوحيدة التي نعرفها للقيام بذلك هي بواسطة التكرار. ابحث بوعي عن فرص لاستخدام قوتك العظمى الجديدة، من الناحية المثالية عدّة مرات في اليوم. ستجد بعد أسبوع أو أكثر، أنك تستخدمها دون تفكير.

تطوير محررك

تُبنى معظم محرّرات الشِفرة القوية حول جوهر أساسي والذي يتم زيادته بعد ذلك بواسطة الملحقات " الامتدادات " (extensions). يتم توفير العديد منها مع المحرّر، والباقي يمكن إضافته لاحقًا.

عندما تصطدم ببعض القيود الواضحة في المحرر الذي تستخدمه، ابحث عن امتداد يقوم بهذه المهمة. من المحتمل أنك لست وحدك من يحتاج هذه الإمكانية، وإذا كنت محظوظًا فقد يكون شخص آخر قد نشر حلّها.

خذ هذا خطوة أبعد للأمام. ابحث في لغة امتداد المحرّر. استنبط كيفيّة استخدامها من أجل أتمتة بعض الأشياء المتكررة التي تقوم بها. في كثير من الأحيان ستحتاج إلى سطر أو اثنين من التعليمات البرمجية فقط.

في بعض الأحيان قد تأخذ الأمر أبعد أيضاً، وتجد نفسك تكتب امتداداً كاملاً. إذا كان الأمر كذلك فأنشره؛ إذا كنت أنت بحاجة إليه، فإن أشخاص آخرون سيحتاجونه أيضًا.

الأقسام ذات الصلة

- الموضوع 7، تواصل!، في الصفحة 44

تحديات

- لا مزيد من التكرار التلقائي.

الجميع يفعلها: تحتاج إلى حذف آخر كلمة كتبتها، لذلك تضغط على مفتاح الإعادة backspace وتنتظر الإعادة التلقائية. في الواقع، نراهن على ذلك لقد فعل عقلك هذا كثيرًا بحيث يمكنك التقدير بالضبط تماماً متى تُحرّر المفتاح. لذا قم بإيقاف الإعادة التلقائية، وبدلاً من ذلك تعرف على تسلسل المفاتيح للتحرك، والتحديد والحذف بالأحرف والكلمات والخطوط والكتل.

- هذا الأمر سيكون شاقاً.

استغني عن الفأرة / لوحة التتبع. لمدة أسبوع كامل، حرّر باستخدام لوحة المفتاح فقط. ستكتشف مجموعة من الأشياء التي لا يمكنك القيام بها دون الإشارة والنقر، فالآن هو الوقت المناسب للتعلم. احتفظ بالملاحظات (نوصي باتباع الطريقة القديمة واستخدام القلم والورق) للتسلسلات الرئيسة التي تتعلمها.

ستعاني مع انخفاض في الإنتاجية لبضعة أيام. ولكن، بينما تتعلم القيام بالأشياء دون تحريك يديك بعيدًا عن الموضع الأصلي، ستجد أن تحريرك يصبح أسرع وأكثر طلاقة من أي وقت مضى.

- ابحث عن عمليات الدمج. في أثناء كتابة هذا الفصل، تساءل ديف عمّا إذا كان يمكن معاينة التخطيط النهائي (مِلَفّ PDf) في المخزن المؤقت للمحرّر. بتنزيل واحد لاحقا، أصبح التصميم يتوضع بجانب النص الأصلي، كل ذلك في المحرّر. احتفظ بقائمة بالأشياء التي ترغب في جلبها إلى محرّرك، ثم ابحث عنها.

- إذا طمحنا إلى أكثر من هذا قليلاً، إذا لم تتمكن من العثور على مكون إضافي (plugin) أو امتداد (extension) يفعل ما تريد، اكتب واحداً. أندي مغرم بعمل إضافات ويكي محليّة مخصّصة مستندة إلى المِلَفّات للمحرّرات المفضلة لديه . إذا لم تجدها، قم ببنائها!

الموضوع 19. التحكم في الإصدار

> إن التقدم، بعيدًا عن كونه تغييرًا، فهو يعتمد على القدرة على الحفاظ عليه.
> أولئك الذين لا يستطيعون تذكّر الماضي محكومٌ عليهم بتكراره.
> ← جورج *سانتايانا، حياة العقل*

أحد الأشياء الهامّة التي نبحث عنها في واجهة المستخدم هو مفتاح التراجع undo - مفتاح واحد يغفر لنا أخطائنا. من الأفضل حتى لو كانت البيئة تدعم مستويات متعددة من التراجع والإعادة، وبذلك تتمكن من العودة واسترداد شيء حدث قبل دقيقتين.

ولكن ماذا لو حدث الخطأ الأسبوع الماضي، وقمت بتشغيل وإيقاف حاسوبك عشر مرات منذ ذلك الحين؟ حسنًا، هذه إحدى الفوائد العدّة لاستخدام نظام التحكم في الإصدار (VCS): إنه مفتاح undo العملاق – آلة الزمن على مستوى المشروع والتي يمكنها إعادتك إلى تلك الأيام الذهبية من الأسبوع الماضي، عندما تم تَزْجَمَة شفرتك وتشغيلها فعلًا.

بالنسبة للعديد من النّاس، هذا هو الحد الأقصى لاستخدام VCS. هؤلاء الناس يُضيّعون عالما شاملًا أكبر من التعاون، ومسارات تنفيذ عمليات النشر، ومتابعة القضايا، والتفاعل مع الفريق العام.

لذلك دعونا نلقي نظرة على VCS، أولاً كمستودع للتغييرات، ثم كمكان التقاء مركزي لفريقك وشِفراتهم.

الأدلة المشتركة Shared Directories *ليست* تحكم في الإصدار

ما زلنا نصادف (من وقت لآخر) الفريق العابر الذي يشارك مِلَفّات مصدر المشروع عبر الشبكة: إما داخليًا أو باستخدام نوع من التخزين السحابي.

هذا أمر غير مجدٍ.

تُفسد الفرق التي تفعل ذلك باستمرار عمل بعضها البعض، وتفقد التغيرات، وتكسر عمليات البناء، وتدخل عراك بالأيدي في موقف سيارات. إنها مثل كتابة متزامنة للشِفرة ببيانات مشتركة مع عدم وجود آلية للتزامن. استخدم التحكم في الإصدار.

ولكن هناك ما هو أفظع! يستخدم بعض الناس التحكم في الإصدار، ويبقون مستودعهم الرئيس على الشبكة أو على محرك أقراص سحابي. إنهم يعتقدون أن هذا هو أفضل ما في الوضعين: المِلَفّات يمكن الوصول إليها في أي مكان و(في حالة التخزين السحابي) تُنسخ احتياطيًا خارج الموقع.

اتضح أن هذا أسوأ، وأنت تخاطر بفقدان كل شيء. تستخدم برمجيّة التحكم في الإصدار مجموعة من المِلَفّات والأدلة المتفاعلة. إذا قام مثيلان بإجراء تغييرات في وقت واحد، فيمكن أن تتلف الحالة الإجمالية، ولا يمكن معرفة مقدار الضرر الحاصل. ولا أحد يحب رؤية المطورين يبكون.

إنه يبدأ من المصدر

تتبّع أنظمة التحكم في الإصدار كل تغيير نقوم فيه في الشِفرة المصدرية وفي التوثيق. بوجود نظام تحكم في شِفرة المصدر مضبوط بشكل صحيح، *يمكنك دائمًا الرجوع إلى إصدار سابق من برمجيتك.*

لكن نظام التحكم في الإصدار يفعل أكثر بكثير من التراجع عن الأخطاء. سيتيح لك النظام الجيّد للتحكم في الإصدار تتبع التغييرات والإجابة على أسئلة مثل: من أجرى التغييرات في هذا السطر من الشِفرة؟ ما الفرق بين النسخة الحاليّة ونسخة الأسابيع الماضية؟ كم عدد أسطر الشِفرة التي غيّرناها في هذا الإصدار؟ ماهي المِلَفّات التي يتم تغييرها في معظم الأحيان؟ هذا النوع من المعلومات لا يقدر بثمن لأغراض تتبع الخطأ والتدقيق والأداء والجودة.

كما سيتيح لك نظام التحكم بالإصدار أيضًا تحديد إصدارات برامجك. وبمجرّد تحديدها ستتمكن من العودة وإعادة توليد الإصدار في أي وقت، بغض النظر عن التغييرات التي قد تكون حدثت لاحقًا.

قد تحتفظ أنظمة التحكم في الإصدار بالمِلَفّات التي تحتفظ بها في مستودع مركزي - مرشح رائع للأرشفة.

وأخيرًا، تسمح أنظمة التحكم في الإصدار لمستخدمين اثنين أو أكثر بالعمل بشكل متزامن على نفس مجموعة المِلَفّات، وحتى بإجراء تغييرات متزامنة في نفس المِلَف. ثم يقوم النظام بعد ذلك بإدارة دمج هذه التغييرات عند إعادة إرسال المِلَفّات إلى المستودع. مع أنّها تبدو محفوفة بالمخاطر، إلاّ أن هذه الأنظمة تعمل بشكل جيّد عمليًا على المشروعات من جميع الأحجام.

نصيحة ٢٨ **استخدم التحكم في الإصدار دائمًا.**

دائما. حتى لو كنت فريقًا من شخص واحد في مشروع لمدة أسبوع. حتى لو أنه نموذج أولي "ستتخلص منه". حتى ولو لم تكن الأشياء التي تعمل عليها شِفرة مصدرية . تأكّد أن *كل شيء* يعمل بظل التحكم بالإصدار: التوثيق، قوائم أرقام الهواتف ومذكرات البائعين، ومِلَفّات makefile وإنشاء وتحرير الإجراءات (procedures) خلال ذلك البرنامَج النصي الصغير *شِل* الذي يرتّب مِلَفّات السجلات - كل شيء. نحن نستخدم التحكم في الإصدار بشكل روتيني في كل ما نكتبه تقريبًا (بما في ذلك نص هذا الكتاب). حتى لو كنا لا نعمل في مشروع، فإن عملنا اليومي محميّ في مستودع.

التفرّع

لا تحتفظ أنظمة التحكم في الإصدار بسجل "تاريخ" واحد فقط لمشروعك.

أحد أكثر ميزاتها قوة وإفادة هي الطريقة التي تسمح لك بعزل جزر التطوير إلى أشياء تسمى *فروع (branches)*. يمكنك إنشاء فرع في أي وقت من تاريخ مشروعك، وسيُعزل أي عمل تقوم به في هذا الفرع عن جميع الفروع الأخرى. وفي وقت ما في المستقبل يمكنك دمج الفرع الذي تعمل عليه مرّة أخرى في فرع آخر، وبذلك يحتوي ذلك الفرع الجديد الآن على التغييرات التي أجريتها في فرعك. حتى أنه يمكن للعديد من الأشخاص أن يعملوا على فرع ما: بطريقة ما، تكون الفروع مثل مشروعات مستنسخة صغيرة.

إحدى فوائد الفروع هي العزل التي تمنحك إياه. إذا طورت المِيزة (أ) في أحد الفروع، وعمل زميل في الفريق على المِيزة (ب) في فرع آخر، فلن تتداخل الميزات مع بعضها البعض.

فائدة ثانية، والتي قد تكون مفاجئة، هي أن تلك الفروع في كثير من الأحيان تكون في صميم سير عمل مشروع الفريق.

وهنا تصبح الأمور مربكة بعض الشيء. فلدى فروع التحكم بالإصدار ومنظمة الاختبار شيء ما مشترك: كلاهما لديه آلاف الناس هناك يقولون لك كيف يجب أن تفعل ذلك. وهذه النصيحة لا معنى لها إلى حدٍ بعيد، لأن ما يقولونه فعلًا هو "هذا ما نجح معي"

لذا استخدم التحكم في الإصدار في مشروعك، وإذا اصطدمت بمشكلات في سير العمل، فابحث عن الحلول الممكنة. وتذكّر أن تستعرض وتضبط ما تقوم به في أثناء اكتسابك للخبرة.

تجربة فكرية

انسكب كوب كامل من الشاي (إفطار إنجليزي، مع القليل من الحليب) على لوحة مفاتيح حاسوبك المحمول. خذ الجهاز إلى حانة الشخص الذكي، وتعرّض لاستهجانهم وعبوسهم. اشتر جهاز حاسوب جديد. خذه إلى المنزل.

كم من الوقت سيستغرق الأمر لتُعيد الجهاز مرّة أخرى إلى نفس الحالة التي كان عليها (مع جميع مفاتيح SSH، وضبط المحرّر، وإعداد الشِل، والتطبيقات المثبّتة، وغيرها) عند اللحظة التي رفعت فيها أولا هذا الكأس المشؤوم؟ كانت هذه مسألة واجهها أحدنا مؤخرًا.

تم تخزين كل شيء تقريبًا يحدّد ضبط واستخدام الجهاز الأصلي في التحكم في الإصدار، بما في ذلك:

- جميع تفضيلات المستخدم والمِلَفّات
- ضبط المحرّر
- قائمة البرامج المثبتة باستخدام Homebrew
- استخدام البرنامَج أنسبل (Ansible) لضبط التطبيقات
- جميع المشروعات الحاليّة

تم استعادة الجهاز بحلول نهاية مدّة ما بعد الظهر.

التحكم في الإصدار كمركز مشروع

مع أنّ التحكم في الإصدار مفيد بشكل لا يصدق في المشروعات الشخصيّة، إلا أنه حقًا يلقى حقه عند العمل مع فريق. ويأتي الكثير من هذه القيمة من كيفيّة استضافته مستودعك الخاص.

لا تحتاج العديد من أنظمة التحكم في الإصدار الآن إلى أي استضافة. هي لا مركزية تمامًا، حيث يتعاون كل مطوّر على أساس الند للند. لكن حتى مع هذه الأنظمة، يجدر النظر في وجود مستودع مركزي، لأنه بمجرّد القيام بذلك، يمكنك الاستفادة من الكثير من عمليات الدمج لجعل تدفق المشروع أسهل.

العديد من أنظمة المستودعات مفتوحة المصدر، لذا يمكنك تثبيتها وتشغيلها في شركتك. لكن هذا ليس حقًا مجال عملك، لذلك فإننا نوصي معظم النّاس باستضافة طرف ثالث. ابحث عن ميزات مثل:

- أمان جيد ومراقبة وصول

- واجهة مستخدم بديهية
- القدرة على القيام بكل شيء من سطر الأوامر أيضًا (لأنه قد تحتاج لأتمتة ذلك)
- البناءات والاختبارات المؤتمتة
- الدعم الجيّد لدمج الفروع (تسمى أحيانًا طلبات السحب pull requests)
- إدارة المشكلات (مدمجة بشكل مثالي في عمليات التثبيت والدمج، حتى تتمكن الاحتفاظ بالمقاييس)
- تقارير جيدة (يمكن أن يكون عرض يشبه لوح كانبانKanba[54] [55] للقضايا والمهام المعلّقة يمكن أن تكون مفيدة للغاية)
- اتصالات جيدة للفريق : رسائل البريد الإلكتروني أو الإخطارات الأخرى بشأن التغييرات، ويكي، وهلم جراً.

تم ضبط VCS الخاص بالعديد من الفرق بحيث يعمل الدفع نحو فرع معين على إنشاء النظام تلقائيًا، وإجراء الاختبارات، وإذا نجحت هذه الاختبارات ينشر الشِفرة الجديدة إلى الإنتاج.

يبدو مخيفًا؟ لكن ليس عندما تدرك أنك تستخدم التحكم في الإصدار. يمكنك دائما التراجع.

الأقسام ذات الصلة

- الموضوع 11، قابلية العكس، في الصفحة 69
- الموضوع 49، الفِرق العملية، صفحة 289
- الموضوع 51، مجموعة البادئ العملية، في الصفحة 300

تحديات

- معرفة أنه بإمكانك التراجع إلى أي حالة سابقة باستخدام VCS هي أحد الميزات، ولكن هل يمكنك في الواقع فعل ذلك ؟ هل تعرف الأوامر للقيام بذلك بصورةٍ صحيحة؟ تعلّمهم الآن، ليس عندما تقع الكوارث وأنت تحت الضغط.
- اقضِ بعض الوقت في التفكير في استعادة بيئة حاسوبك المحمول في حالة وقوع كارثة. ما الذي تحتاج إلى استرداده؟ العديد من الأشياء التي تحتاجها هي مجرّد مِلَفّات نصيّة. إذا لم يكونوا في VCS (مستضافًا على حاسوبك المحمول) اعثر على طريقة لإضافتهم. ثم فكّر في الأشياء الأخرى: التطبيقات المثبتة، ضبط النظام، وما إلى ذلك. كيف يمكنك التعبير عن كل هذه الأشياء في المِلَفّات النصية بحيث يمكن أيضًا أن يتم حفظها؟

54 **[المترجم]** ألواح Kanban هي تقنية لإظهار تدفق العمل وتنظيم المشروعات. وخاصّةً في مجال تطوير البرمجيات، فهي توفّر نظامًا مرئيًا لإدارة العمليّات للمساعدة في تحديد كيفيّة تنظيم الإنتاج.

55 **[المترجم]** ألواح Kanban هي تقنية لإظهار تدفق العمل وتنظيم المشروعات. وخاصّةً في مجال تطوير البرمجيات، فهي توفّر نظامًا مرئيًا لإدارة العمليّات للمساعدة في تحديد كيفيّة تنظيم الإنتاج.

هناك تجربة مثيرة للاهتمام، بمجرد إحراز لبعض التقدم، وهي العثـور على حاسـوب قـديم لم تعـد تسـتخدمه ومعرفـة مـا إذا كان بإمكانك استخدام نظامك الجديد لإعداده.

- استكشف بوعي ميزات مزود خدمة VCS الحالي والمُستضيف الذي لا تستخدمه. إذا لم يكن فريقك يستخدم مِـيزة الفـروع، جـرّب تقـديمها. الشـيء نفسـه مـع طلبـات السـحب / الـدمج (pull/merge). التكامـل المسـتمر. بنـاء مسـارات التنفيـذ (pipelines). حتى النشر المستمر. ابحث في أدوات التواصل الجماعي أيضًا: الويكي، ألواح كانبان، وما شابه.

  ليس عليك أن تستخدم أي من ذلك. ولكن عليك أن تعرف ما تفعله حتى تتمكن من اتخاذ هذا القرار.

- استخدام التحكم بالإصدار للأشياء خارج المشروع أيضاً.

الموضوع 20. تنقيح الأخطاء (Debugging)

> من المؤلم أن تنظر إلى مشكلاتك الخاصّة وتعلم
> أنك أنت ولا أحد غيرك قد فعلها
> ← سوفوكليس، أجاكس

استُخدمت كلمة (bug)[56] حشرة لوصف "كائن الرعب" منذ القرن الرابع عشر. يعود الفضل للأدميرال "العميد البحري" د. جريس هوبر، مخترع كوبول (COBOL)، في مراقبة أول "حشرة" حاسوب - حرفيًا، بعوضة عالقة في مُرحِّل (relay) في نظام حاسوب مبكّر. عندما طُلب منه شرح سبب عدم عمل الجهاز بالشكل المطلوب، أفاد أحد التقنيين بوجود "حشرة في النظام"، وسجلها - بجناحيها وكل شيء - في دفتر السجل.

للأسف، لا يزال لدينا أخطاء في النظام، وإن لم يكن من النوع الطائر. لكن معنى– كائن رعب- القرن الرابع عشر ربما يكون أكثر قابلية للتطبيق الآن مما كان عليه في ذلك الوقت. تتجلى عيوب البرمجيات في مجموعة متنوعة من الطرق، من متطلبات يساء فهمها إلى أخطاء التشفير. للأسف، لا تزال أنظمة الحاسوب الحديثة مقيّدة بفعل ما تطلب منها القيام به، وليس بالضرورة ما تريد منها أن تفعله.

لا أحد يكتب برمجيات مثالية، لذلك من المفترض أن يستغرق تصحيح الأخطاء جزءًا كبيرًا من يومك. دعونا نلقي نظرة على بعض القضايا التي ينطوي عليها تصحيح الأخطاء وبعض الاستراتيجيات العامّة لجمع الأخطاء المستعصية على الفهم.

سيكولوجية "علم نفس" التنقيح

التصحيح هو موضوع حسّاس وعاطفي للعديد من المطورين. بدلًا من التعامل معه على أنه لغز يجب حله، فقد تواجه الإنكار أو توجيه إصبع الاتهام أو الأعذار الواهية أو مجرد اللامبالاة الصريحة.

اعتنق حقيقة أن التصحيح هو مجرد حل للمشكلات، وتعامل معه على هذا الأساس.

بعد العثور على خطأ شخص آخر، يمكنك إهدار الوقت والطاقة في إلقاء اللوم على الجاني السفيه الذي تسبب به. في بعض أماكن العمل، يعدّ هذا جزءًا من الثقافة، وقد يكون حلًا مريحًا. ومع ذلك، في المجال التقني، تريد التركيز على إصلاح المشكلة، وليس على اللوم.

نصيحة ٢٩	عالج المشكلة وليس اللوم

لا يهم حقًا ما إذا كان الخطأ هو خطأك أو خطأ شخص آخر. لا تزال مشكلتك.

56 **[المترجم]** Bug – حشرة أو بعوضة "بقة" وتعني في مجال الحاسوب: علّة أو خطأ أو خلل.

ذهنيّة التصحيح

> أسهل شخص يمكن للمرء خداعه هو الشخص نفسه.
>
> ← إدوارد بولوير ليتون، *المنفي*

قبل البَدء في تصحيح الأخطاء، من المهم تبني طريقة التفكير الصحيحة. أنت بحاجة إلى إيقاف العديد من الدفاعات التي تستخدمها كل يوم لحماية نفسك "الأنا"، وعدم الاكتراث بأي ضغوط لمشروع قد ترزح تحتها، وإراحة نفسك. قبل كل شيء، تذكّر القاعدة الأولى لتصحيح الأخطاء:

نصيحة ٣٠ — لا داعي للذعر

من السهل أن تصاب بالذعر، خاصة إذا كنت تواجه موعدًا نهائيًا، أو لديك رئيس عصبي أو عميل عصبي يتنفس أسفل رقبتك في أثناء محاولتك العثور على سبب الخلل. ولكن من المهم للغاية أن تتراجع خطوة إلى الوراء، وأن تفكر فعليًا فيما قد يسبب الأعراض التي تعتقد أنها تشير إلى خلل.

إذا كان رد فعلك الأول عند مشاهدة خلل أو رؤية تقرير الخلل "هذا مستحيل"، فأنت مخطئ تمامًا. لا تضيّع عصبونًا واحدًا في دماغك على سلسلة الأفكار التي تبدأ بـ "ولكن لا يمكن أن يحدث" لأنه من الواضح أنه يمكن، وقد حدث.

احذر من قصر النظر عند التصحيح. قاوم الرغبة في إصلاح فقط ما تراه من أعراض: من المرجح أن يكون الخطأ الفعلي قد تم إخفاؤه بعدّة خطوات مما تلاحظه من أعراض، وقد يتضمن عددًا من الأشياء الأخرى ذات الصلة. حاول دائمًا اكتشاف السبب الجذري للمشكلة، وليس فقط هذا المظهر الخاص بها.

من أين أبدا

قبل أن تبدأ في النظر إلى الخلل، تأكّد أنك تعمل على شِفرة برمجيّة أُنشئت بطريقة نظيفة - دون تحذيرات. نقوم عادةً بتعيين مستويات تحذير المترجم (compiler) على أعلى مستوى ممكن. ليس من المنطقي إضاعة الوقت في محاولة العثور على مشكلة قد يجدها الحاسوب لك! نحن بحاجة إلى التركيز على المشكلات الأصعب التي تواجهنا.

تحتاج عند محاولة حل أي مشكلة، إلى جمع كافة البيانات ذات الصلة. للأسف، تقرير الأخطاء ليس علمًا دقيقًا. من السهل أن تُخدع بالمصادفة، ولا يمكنك أن تضيع الوقت في تصحيح أخطاء المصادفات. تحتاج أولاً إلى أن تكون دقيقًا في رصدك.

تنخفض الدقة في تقارير الأخطاء بشكل أكبر عندما تأتي بواسطة طرف ثالث - قد تحتاج فعلًا إلى مشاهدة المستخدم الذي أبلغ عن الخلل في أثناء العمل للحصول على مستوى كافٍ من التفاصيل.

عمل *آندي* ذات مرّة على تطبيق رسوميّات كبير. قبيل الإصدار، أفاد المختبرون أن التطبيق يتعطّل في كل مرّة ينفذون فيها ضربة بفرشاة معينة. جادل المبرمج المسؤول بأنه لا توجد مشكلة كهذه؛ فقد حاول الرسم بها، وعملت على ما يرام.

استمر هذا الحوار بين أخذٍ وردّ لعدة أيام، مع زيادة سريعة في حدة التوتر.

أخيرًا، جمعناهما معا في نفس الغرفة. حدّد المختبر أداة الفرشاة وقام بضربة فرشاة من الزاوية اليمنى العليا إلى الزاوية اليسرى السفلية.

فانفجر التطبيق. "آه" قالها المبرمج، بصوت ضعيف، ثم اعترف بخجل أنه أجرى ضربات اختبار الفرشاة فقط من أسفل اليسار إلى أعلى اليمين، والتي لم تكشف عن الخطأ.

هناك نقطتان لهذه القصة:

- قد تحتاج إلى مقابلة المستخدم الذي أبلغ عن الخلل من أجل جمع بيانات أكثر مما أعطاك في البداية.
- لا تستعمل الاختبارات المُصطنعة (مثل ضربة فرشاة المبرمج من الأسفل إلى الأعلى) قدرًا كافيًا من التطبيق. يجب عليك اختبار كل من الشروط الحدّية وأنماط استخدام المستخدم النهائي الواقعية بشدّة. تحتاج إلى القيام بذلك بشكل منهجي (انظر اختبار قاسٍ ومستمر، في الصفحة 301).

استراتيجيات التصحيح

بمجرّد أن تعتقد أنك تعرف ما يحدث، فقد حان الوقت لمعرفة ما يعتقد البرنامَج أنه يحدث.

تكاثر الخطأ

لا، أخطاءنا حقًا لا تتكاثر (مع أنّ بعضها ربما يكون ناضجًا بما يكفي للقيام بذلك بشكل شرعي). نحن نتحدث عن نوع مختلف من التكاثر.

أفضل طريقة لبَدْء إصلاح الأخطاء هي جعلها قابلة للإنتاج من جديد. في النهاية، إذا لم تتمكن من إعادة إنتاجها، فكيف ستعرف ما إذا كان قد تم إصلاحها؟

لكننا نريد أكثر من الخطأ الذي يمكن إعادة إنتاجه باتباع سلسلة طويلة من الخطوات؛ نريد خطأ يمكن إعادة إنتاجه باستخدام *أمر واحد*. من الصعب جدًا إصلاح خطأ إذا كان عليك اتباع 15 خطوة للوصول إلى النقطة التي يظهر فيها هذا الخطأ.

إذن إليك أهم قاعدة تنقيح الأخطاء:

نصيحة ٣١ **أجري اختبار الفشل قبل إصلاح الشِفرة**

في بعض الأحيان بواسطة إجبار نفسك على عزل الظروف التي تعرض الخطأ، ستحصل على رؤية متبصّرة حول كيفيّة إصلاحه. فعل كتابة الاختبار ينبه إلى الحل.

المبرمج في أرض غريبة

كل هذا الحديث عن عزل الخطأ لا بأس به، لكن ما الذي يفعله المبرمج المسكين عندما يواجه 50000 سطر من الشِفرة وساعة موقوتة؟

أولاً، انظر إلى المشكلة. هل هي تعطّل؟ من المفاجئ لنا دائمًا - عندما ندرس الدورات التي تنطوي على البرمجة – كم هو عدد المطوّرين الذين يرون استثناءً يظهر باللون الأحمر وينتقلون فورًا إلى الشِفرة.

نصيحة ٣٢ اقرأ رسالة الخطأ اللعينة

قالها نوف.

نتائج سيئة

ماذا لو لم يكن تعطّلا؟ ماذا لو كانت نتيجة سيئة فحسب؟

ادخل إلى منقّح الأخطاء واستخدم اختبار الفشل لإثارة المشكلة.

تأكّد قبل أي شيء آخر، من رؤيتك أيضًا للقيمة غير الصحيحة في المنقّح. لقد أهدرنا كلانا ساعات في محاولة تعقب خطأ فقط لاكتشاف أن هذا التشغيل المحدّد للشِفرة يعمل بشكل جيد.

في بعض الأحيان تكون المشكلة واضحة: قيمة interest_rate هي 4.5 ويجب أن تكون 0.045. في كثير من الأحيان عليك أن تبحث بشكل أعمق لتعرف لماذا القيمة خاطئة في المكان الأول. تأكّد أنك تعرف كيفيّة تحريك مكدّس الاستدعاء (call stack) للأعلى وللأسفل وفحص بيئة المكدّس المحلي.

نجد أنّه غالبًا ما يكون إبقاء القلم والورق بالقرب منّا مساعدًا لنا لنتمكن من تدوين الملاحظات. على وجه الخصوص، غالبًا ما نكتشف دليلًا ونتعقبه، فقط لنكتشف أنه غير ناجح. إذا لم نسجل المكان الذي كنّا فيه عندما بدأنا التعقب، فقد نهدر الكثير من الوقت للعودة إليه.

في بعض الأحيان، تنظر إلى سجل تتبع المكدّس يبدو كما لو أنه غير منتهي. في هذه الحالة، غالبًا ما تكون هناك طريقة أسرع للعثور على المشكلة من فحص كل إطار للمكدس: استخدم الفرز الثنائي. ولكن قبل أن نناقش ذلك، دعونا نلقي نَظْرة على سيناريوهين آخرين شائعين للأخطاء.

الحساسيّة لقيم المدخلات

كنتَ هناك. حيث يعمل برنامَجك جيّدًا مع جميع بيانات الاختبار، وينجو من أسبوعه الأول في الإنتاج بشرف. ثم يتعطل فجأة عند تغذيته بمجموعة بيانات معينة.

يمكنك تجربة النظر إلى المكان الذي تعطّل فيه والعمل للخلف. ولكن في بعض الأحيان يكون من الأسهل البَدْء باختبار بالبيانات. خذ نسخة من مجموعة البيانات وغذّها في نسخة محليّة شغّالة للتطبيق، مع التأكد استمرار تعطّله.

ثم قسّم البيانات ثنائيا إلى أن تعزل قيم الإدخال التي تؤدي إلى التعطّل.

الانحدارات عبر الإصدارات

أنت في فريق جيد، وتطلق برمجيتك إلى الإنتاج. في مرحلة ما، يظهر خطأ في الشِفرة التي عملت جيدًا قبل أسبوع. ألن يكون لطيفًا إذا استطعت معرفة التغيير المحدّد الذي أدخلته؟ خمّن ما هو؟ حان وقت الفرز الثنائي.

الفرز الثنائي

اضطر كل طالب جامعي في علوم الحاسوب إلى كتابة شِفرة فرز ثنائي (يطلق عليه أحيانًا بحث ثنائي). الفكرة بسيطة. أنت تبحث عن قيمة معينة في مصفوفة مرتبة. يمكنك فقط إلقاء نظرة على كل قيمة على حدة، ولكن سينتهي بك المطاف بالنظر وسطيًا إلى ما يقرب من نصف الإدخالات حتى تستطيع أيجاد القيمة التي تريدها، أو تجد قيمة أكبر منها، مما يعني أن القيمة ليست في المصفوفة.

ولكن من الأسرع استخدام نهج فرّق تسد. اختر قيمة في منتصف المصفوفة. إذا كانت هي القيمة التي تبحث عنها، فتوقف. وإلا يمكنك تجزئة المصفوفة إلى قسمين. إذا كانت القيمة التي تعثر عليها أكبر من الهدف، فستعلم أنها يجب أن تكون في النصف الأول من المصفوفة، وإلاّ فهي في النصف الثاني. كرّر الإجراء في المصفوفة الفرعيّة المناسبة، وستحصل على نتيجة بلمح البصر. (كما سنرى عندما نتحدث عن تدوين Big-O، في الصفحة 229، البحث الخطي هو O(n)، والبحث الثنائي هو O(logn)).

لذا، فإن الفرز الثنائي هو طريقة أسرع في أي مسألة بحجم محترم. دعونا نرى كيفيّة تطبيقه على التصحيح.

عندما تواجه سجل تتبع ضخم للمكدس وتحاول أن تكتشف بالضبط الدالة التي أتلفت القيمة عن طريق الخطأ، فإنك تفرز عن طريق اختيار إطار مكدّس موجود في مكان ما في الوَسَط ومعرفة ما إذا كان الخطأ ظاهرًا هناك أم لا. إذا كان ظاهرًا، فستعرف انّه عليك التركيز على الإطارات السابقة، وإلا فستكون المشكلة في الإطارات التالية. افرز مرّة أخرى. حتى لو كان لديك 64 إطارًا في المكدّس، فإن هذا النهج سيعطيك إجابة بعد ست محاولات على الأكثر.

إذا وجدت أخطاءً تظهر في مجموعات بيانات معينة، فقد تتمكن من فعل الشيء نفسه. قسّم مجموعة البيانات إلى قسمين، وتحقّق مما إذا كانت المشكلة تحدث إذا أدخلت القسم الأول أم الثاني عبر التطبيق. استمر في تقسيم البيانات حتى تحصل الحد الأدنى من القيم التي تُظهر المشكلة.

إذا أدخل فريقك خللًا خلال مجموعة من الإصدارات، يمكنك استخدام نوع هذه التقنية نفسه. أنشئ اختبار يؤدي إلى فشل الإصدار الحالي. ثم اختر إصدارًا في منتصف الطريق بين الإصدار الحالي وآخر إصدار شغّال معروف. شغّل الاختبار مرّة أخرى، وحدّد كيفيّة تضييق نطاق البحث. إن إمكانية القيام بذلك هي مجرّد واحدة من الفوائد العدّة لامتلاك تحكم جيّد في الإصدار في مشروعاتك. في الواقع، ستأخذ العديد من أنظمة التحكم في الإصدار هذا الأمر إلى أبعد من ذلك وستعمل على أتمتة العملية، فتختار لك الإصدارات اعتمادًا على نتيجة الاختبار.

التسجيل و / أو التتبع

يركز المصححون عمومًا على حالة البرنامَج في اللحظة الحالية. في بعض الأحيان تحتاج إلى معرفة المزيد — تحتاج إلى مشاهدة حالة البرنامَج أو بنية البيانات على امتداد الوقت. يمكن أن تخبرك رؤية تتبع المكدس فقط كيف وصلت إلى هنا مباشرة. لا يمكنها عادةً إخبارك بما كنت تفعله قبل سلسلة الاستدعاء هذه، خاصةً في الأنظمة المستندة إلى الأحداث[57] .

تعليمات التتبع هي تلك الرسائل التشخيصيّة الصغيرة التي تطبعها على الشاشة أو إلى مِلَفّ يقول أشياء مثل "وصلت هنا" و "قيمة x = 2." إنها تقنية بدائية مقارنة بمنقّحات بيئات التطوير المتكاملة (IDE-style debuggers)، لكنها فعّالة بشكل خاص في

57 مع أنّ لغة Elm لديها منقّح أخطاء زمني.

تشخيص عدّة أصناف من الأخطاء والتي لا تستطيع المنقّحات تشخيصها. التتبع لا يقدر بثمن في أي نظام يكون فيه الوقت بحد ذاته عاملًا (factor): العمليات المتزامنة، وأنظمة الوقت الحقيقي والتطبيقات القائمة على الأحداث.

يمكنك استخدام تعليمات التتبع للتوغل في الشِفرة. بمعنى، يمكنك إضافة تعليمات التتبع بينما تنزل في شجرة الاستدعاء.

يجب أن تنسّق رسائل التتبع بشكل منتظم ومتسق حيث قد ترغب في تحليلها تلقائيًا. على سبيل المثال، إذا كنت بحاجة إلى تعقب تسرب الموارد (مثل فتح / إغلاق غير متوازن لِملَفّ) ، يمكنك تتبع كل فتح وكل إغلاق في مِلَفّ السجل. وبواسطة معالجة مِلَفّ السجل باستخدام أدوات معالجة النصوص أو أوامر شِل (shell)، يمكنك بسهولة تحديد مكان حدوث عملية الفتح المخالفة.

الانحناء المطاطي Rubber Ducking

يوجد أسلوب بسيط للغاية ولكنه مفيد بشكل خاص لإيجاد سبب المشكلة وهو ببساطة شرحه لشخص آخر. يجب أن ينظر الشخص الآخر من فوق كتفك على الشاشة، ويومئ برأسه باستمرار (مثل بطة مطاطية تتمايل لأعلى ولأسفل في حوض الاستحمام). لا يحتاجون لقول أي كلمة؛ الفعل البسيط في شرح ما تفترضه الشِفرة، خطوة بخطوة، غالبًا ما يؤدي بالمشكلة إلى القفز من الشاشة والإعلان عن نفسها [58].

يبدو الأمر بسيطًا، ولكن في شرح المشكلة لشخص آخر، يجب أن تذكر بوضوح الأشياء التي قد عدّها أمرًا مسلمًا به عند مراجعة الشِفرة البرمجيّة بنفسك. قد تكتسب فجأة بواسطة الاضطرار إلى صياغة بعض هذه الافتراضات، رؤية جديدة للمشكلة. وإذا لم يكن لديك شخص، فستفي بالغرض بطة مطاطية، أو دمية دب، أو نبات بوعاء[59].

عملية العزل Elimination

في معظم المشروعات، قد تكون الشِفرة التي تصححها مزيج من شِفرة التطبيق الذي كتبته أنت والآخرون في فريق المشروع الخاص بك، ومنتجات الطرف الثالث (قاعدة البيانات، والاتصال، وإطار الويب، والاتصالات المتخصصة أو الخوارزميات، وما إلى ذلك) وبيئة المنصّة (نظام التشغيل ومكتبات النظام والمترجمات).

من الممكن أن يكون هناك خلل في نظام التشغيل، أو المترجم البرمجي، أو منتج تابع لجهة خارجية، ولكن لا ينبغي أن يكون هذا أول فكرة تراودك. من المرجح أن يكون الخطأ موجودًا في شِفرة التطبيق قيد التطوير. عمومًا أن تفترض أن شِفرة التطبيق تستدعي مكتبة بشكل غير صحيح أكثر ربحيّة من افتراض أن المكتبة نفسها معطلّة. حتى ولو كانت المشكلة تكمن في وجود جهة خارجية، فسيتعين عليك عزل الشِفرة الخاص بك قبل إرسال تقرير الخطأ.

لقد عملنا في مشروع كان فيه أحد كبار المهندسين مقتنعًا بأنه تم تعطيل استدعاء نظام الـ select على نظام يونكس (Unix). لا يمكن لأي قدر من المنطق أو القدرة على الإقناع أن يغيّر رأيه (حقيقة أن كل تطبيق شبكة آخر في النظام يعمل بشكل جيّد كان غير ذي صلة). أمضى أسابيع في كتابة الحلول، والتي، ولسبب غريب، لا يبدو أنها تحلّ المشكلة. وحين اضطر في النهاية إلى الجلوس

58 لماذا "البطة المطاطية"؟ قام ديف في أثناء دراسته الجامعية في إمبريال كوليدج في لندن، بالكثير من العمل مع مساعد بحوث يدعى جريج بوج، أحد أفضل المطوّرين الذين عرفهم ديف. لعدة أشهر، حمل جريج بطة مطاطية صفراء صغيرة، وضعها على طرفيّة في أثناء كتابة الشِفرة. لقد مرّ وقت طويل قبل قيام ديف بذلك.

59 تحدثت إصدارات سابقة من الكتاب عن التحدث إلى نبات القِدر "الأصيص" الخاص بك. لقد كان خطأ مطبعي بصدق.

وقراءة الوثائق عن select، اكتشف المشكلة وصحّحها في غضون دقائق. نستخدم الآن عبارة " select مُعطّل" كتذكير لطيف كلما بدأ أحدنا بإلقاء اللوم على النظام لخطأ من المحتمل أن يكون خطأنا.

نصيحة ٣٣	إن "select" ليس معطّلًا

تذكّر، إذا رأيت طبعات حافر، ففكّر في الخيول، وليس بالحمار الوحشي. ربما نظام التشغيل ليس مُعطّلًا. وربما select على ما يرام.

إذا "غيرت شيئًا واحدًا فقط" وتوقف النظام عن العمل، فمن المحتمل أن يكون هذا الشيء الواحد مسؤول، بشكل مباشر أو غير مباشر، بغض النظر عن مدى بعد الاحتمال الذي يبدو عليه. في بعض الأحيان يكون الشيء الذي تغير خارج نطاق سيطرتك: الإصدارات الجديدة من نظام التشغيل، أو المترجم، أو قاعدة البيانات، أو غيرها من برمجيات الطرف الثالث يمكن أن تسبب الخراب في شِفرة صحيحة سابقًا. قد تظهر أخطاء جديدة. الأخطاء التي كان لديك حل بديل لها، وتعطّل هذا الحل. تغيير واجهات برمجة التطبيقات، تغييرات الوظائف؛ باختصار، إنها لعبة كرة جديدة بالكامل، ويجب عليك إعادة اختبار النظام في ظل هذه الظروف الجديدة. لذا راقب الجدول الزمني من كثب عند التفكير في الترقية؛ قد ترغب في الانتظار إلى ما بعد الإصدار التالي.

عنصر المفاجأة

عندما تجد نفسك مندهشًا من خلل (ربما تتمتم في سَرك حيث لا نستطيع سماعك "هذا مستحيل")، يجب عليك إعادة تقييم الحقائق التي تثق بها. في خوارزمية حساب الحسم (discount)- تلك التي تعرف أنها مقاومة للرصاص ولا يمكن أن تكون سبب هذا الخطأ - هل اختبرت جميع شروط الحديّة؟ ذاك الجزء من الشِفرة الآخر الذي تستخدمه منذ سنوات - ربما لا يزال بالإمكان أن يحتوي على خطأ فيه. هل يمكن ذلك؟

بالطبع يمكن. مقدار المفاجأة التي تشعر بها عند حدوث خطأ ما يتناسب مع مقدار الثقة والإيمان الذي لديك في الشِفرة التي تُنفذها. لهذا السبب، عندما تواجه إخفاقًا "مفاجئًا"، يجب عليك قَبُول أن واحد أو أكثر من افتراضاتك خاطئ. لا تحجب روتينًا أو جزءًا من التعليمات البرمجية المتضمنة في الخطأ لأنك "تعرف" أنه يعمل. اثبت ذلك. أثبت ذلك في هذا السياق، مع هذه البيانات، ضمن هذه الشروط الحدّية.

نصيحة ٣٤	لا تفترض- أثبت ذلك

عندما تصادف خطأ مفاجئًا، عليك تجاوز مجرّد إصلاحه فقط، تحتاج إلى تحديد سبب عدم اكتشاف هذا الفشل سابقًا.

ضع في اعتبارك ما إذا كنت بحاجة إلى إصلاح الوحدة أو الاختبارات الأخرى حتى يتمكنوا من اكتشافه.

أيضًا، إذا كان الخطأ ناتجًا عن بيانات خاطئة تم نشرها عبر عدّة مستويات قبل التسبب في الانفجار، فانظر ما إذا كان التحقق من المعامِلات بشكل أفضل في هذه الإجراءات من شأنه أن يعزلها في وقت أبكر (راجع المناقشات حول التعطل المبكر والتأكيدات في الصفحة 134 وفي الصفحة 133 على التوالي).

في أثناء وجودك في الخلل، هل هناك أي أماكن أخرى في الشِفرة قد تكون عرضة لهذا الخطأ نفسه؟ الآن هو الوقت المناسب للعثور عليها وإصلاحها. تأكّد أنّه مهما حدث، فإنك ستعرف ما إذا كان سيحدث مرّة أخرى.

إذا استغرق إصلاح هذا الخطأ وقتًا طويلاً، اسأل نفسك عن السبب. هل هناك أي شيء يمكنك القيام به لتسهيل إصلاح هذا الخطأ في المرّة القادمة؟ ربما يمكنك بناء خطافات اختبار أفضل، أو كتابة محلل مِلَفّات السجل (log file analyzer).

أخيرًا، إذا كان الخطأ ناتجًا عن افتراض خاطئ لشخص ما، فناقش المشكلة مع الفريق بِرُمَّته: إذا أخطأ شخص واحد في فهمه، فمن المحتمل أن يخطئ الكثير من الناس.

افعل كل هذا، ونأمل ألا تفاجأ في المرّة القادمة.

قائمة مراجعة التنقيح

- هل يتم الإبلاغ عن المشكلة كنتيجة مباشرة للخطأ الأساسي أم أنها مجرّد عرض؟
- هل الخطأ حقًا في إطار العمل الذي تستخدمه؟ هل هو في نظام التشغيل؟ أم هو في الشِفرة الخاصة بك؟
- إذا شرحت هذه المشكلة بالتفصيل لزميل ما في العمل، فماذا سيقول؟
- إذا اجتازت الشِفرة المشتبه فيها اختبارات الوحدة الخاصة بها، فهل تكون الاختبارات مكتملة بما فيه الكفاية؟ ماذا يحدث إذا أجريت الاختبارات بهذه البيانات؟
- هل الظروف التي تسببت في هذا الخطأ موجودة في أي مكان آخر في النظام؟ هل لا تزال هناك "حشرات" أخرى في مرحلة اليرقات، فقط تنتظر الفقس؟

الأقسام ذات الصلة

- الموضوع 24، البرامج الميتة لا تكذب، في الصفحة 133

تحديات

- إن تنقيح الأخطاء هو تحدي كافٍ.

الموضوع 21. معالجة النص

يعالج المبرمجون العمليون النص بنفس الطريقة التي يشكل بها النجارون الأخشاب. ناقشنا في الأقسام السابقة بعض الأدوات المحددة– الصدفات (Sells) والمحرّرات والمصحّحات - التي نستخدمها. وهي تشبه الأزاميل والمناشير والسطوح المستوية لعامل الأخشاب- وهي أدوات مُخصصة للقيام بوظيفة واحدة أو وظيفتين بشكل جيّد. ومع ذلك، نحتاج بين الحين والآخر إلى إجراء بعض عمليات التحويل والتي لا يمكن التعامل معها بسهولة بواسطة مجموعة الأدوات الأساسيّة. نحن بحاجة إلى أداة معالجة النص لأغراض عامة.

إن لغات معالجة النص بالنسبة للبرمجة بمنزلة أجهزة (التوجيه)[60] بالنسبة لأعمال النجارة. إنها صاخبة وفوضوية ومتجبّرة بعض الشيء. ارتكب خطأ في استعمالهم، ويمكن أن تتلف القطع بِرُمَّتها. يُقسم بعض الناس أنه ليس لديهم مكان في صندوق الأدوات. ولكن عند الأشخاص المناسبين، يمكن لكل من أجهزة التوجيه ولغات معالجة النصوص أن تكون قوية ومتعددة الاستخدامات بشكل لا يصدق. يمكنك نحت شيء ما سريعًا إلى شكل ما، وإنشاء المفاصل، والنقش. تمتلك هذه الأدوات براعة ودقة مدهشة عند استخدامها بشكل صحيح. لكنها تحتاج بعض الوقت لإتقانها.

لحسن الحظ، هناك عدد من لغات معالجة النصوص الرائعة. غالبًا ما يرغب مطورو يونيكس (ونحن ندرج مستخدمي ماك او إس macOS هنا) في استخدام قوة أوامر **شِل** الخاص بهم، بالإضافة إلى أدوات مثل awk و sed. قد يفضّل الأشخاص الذين يُحبّذون أداةً أكثر تنظيمًا لغاتٍ مثل بايثون أو روبي.

تُعدّ هذه اللغات تكنولوجيات تمكين مهمة. يمكنك باستخدامها تجسيد الأدوات المساعدة وأفكار النموذج الأولية بسرعة - الأعمال التي قد تستغرق وقتا أطول بخمس أو عشر مرات عند استخدام اللغات التقليدية. عامل المضاعفة هذا مهم للغاية لنوع التجارِب التي نقوم بها. إن قضاء 30 دقيقة في تجرِبة فكرة مجنونة أفضل بكثير من قضاء خمس ساعات.

فمن المقبول قضاء يوم في أتمتة المكونات المهمة للمشروع؛ لكن قضاء أسبوع قد لا يكون مقبولا. بنى *كيرنيغان* و *بايك* في كتابهما ***ممارسة البرمجة*** *The Practice of Programming [316]*، نفس البرنامَج بخمس لغات مختلفة. كانت إصدارة بيرل (Perl) هي الأقصر (17 سطرًا، مقارنة بـ 150 في لغة سي).

يمكنك باستخدام *بيرل*، معالجة النص والتفاعل مع البرامج والتحدث عبر الشبكات وقيادة صفحات الويب وإجراء عمليات حسابية دقيقة عشوائية وكتابة البرامج التي تبدو مثل قسَم سنوبي[61].

نصيحة ٣٥	تعلّم لغة معالجة نص

لإظهار مدى قابلية تطبيق لغات معالجة النصوص على نطاق واسع، إليك عيّنة من بعض الأشياء التي قمنا بها باستخدام روبي و بايثون تتعلق فقط بإنشاء هذا الكتاب:

60 هنا جهاز التوجيه يعني الأداة التي تدوّر شفرات القطع بسرعة كبيرة جدًا، وليس جهازًا لتوصيل الشبكات.

61 **[المترجم]** سنوبي هو شخصية خياليّة هزليّة مشهورة.

بناء الكتاب

كُتب نظام البناء لـ "رف كتاب المبرمج العملي" باستخدام لغة البرمجة روبي. يستخدم المؤلفون والمحررون ومسؤولي التخطيط و موظفي الدعم مهام Rake[62] لتنسيق بناء مِلَفّات PDF والكتب الإلكترونية.

تضمين الشِفرة وتمييزها

نعتقد أنه من المهم أن تُختبر أي شِفرة مقدّمة في كتاب أولاً. اختُبرت معظم الشفرات في هذا الكتاب. ومع ذلك، استخدامًا لمبدأ عدم التكرار (راجع الموضوع 9، DRY - شرور التكرار، في الصفحة 54)، لم نرغب في نسخ ولصق أسطر التعليمات البرمجية من البرامج المُختبَرة في الكتاب. وهذا يعني أننا بذلك سنُكرّر الشِفرة، مما يضمن فعليًا أننا سوف ننسى تحديث مثال عندما يتم تغيير البرنامَج المقابل. بالنسبة لبعض الأمثلة، لم نرغب أيضًا في أن نضجرك بكامل شِفرة إطار العمل المطلوبة لجعل مثالنا جاهزًا للتَزجَمَة والتشغيل مباشرة. لجأنا إلى روبي. حيث نستدعي نصًا برمجيًا بسيطًا نسبيًا عندما ننسق الكتاب - فهو يستخرج جزءًا محدّدًا من مِلَفّ المصدر، ويميز بناء الجملة، ويحول النتيجة إلى لغة الطباعة التي نستخدمها.

تحديث الموقع الإلكتروني

لدينا نص برمجي بسيط يبني كتاب جزئي، ويستخرج جدول المحتويات، ثم يحمّله إلى صفحة الكتاب على موقعنا. لدينا أيضًا نص برمجي يستخرج أقسام الكتاب ويحمّلها كعيّنات.

تضمين المعادلات

هناك برنامَج نصي "بايثون" يحول ترميز الرياضيات "لاتك"LaTe[63] إلى نص مُنسّق بشكل جيّد.

توليد الفِهْرِس

تُنشئ معظم الفهارس كمستندات منفصلة (مما يجعل الاحتفاظ بها صعبًا في حال تغيرت الوثيقة). يتم تمييز فهارسنا في النص نفسه، ويقوم برنامَج نصي لروبي بجمع الإدخالات وتنسيقها.

وهكذا. وبطريقة واقعية للغاية، فإن "رف الكتاب العملي" مبني على معالجة النص. وإذا اتبعتَ نصيحتنا لإبقاء الأشياء في نص واضح (plain)، فإن استخدام هذه اللغات لمعالجة هذا النص سيجلب مجموعة كاملة من الفوائد.

الأقسام ذات الصلة

- الموضوع 16، قوة النص الواضح، في الصفحة 95
- الموضوع 17، ألعاب شِل، الصفحة 99

62 **[المترجم]** Rake – أداة لإدارة مهام البرمجيات وبناء الأتمتة. يسمح للمستخدم بتحديد المهام ووصف التبعيات وكذلك تجميع المهام في فضاء الاسم (namespace).

63 **[المترجم]** https://ar.wikipedia.org/wiki/%D9%84%D8%A7%D8%AA%D8%AE

تمارين

تمرين 11

افترض أنك تعيد كتابة تطبيق كان يستخدم YAML كلغة ضبط. وتم توحيد العمل في شركتك الآن على JSON، لذا لديك مجموعة من مِلَفّات yaml. التي يجب تحويلها إلى مِلَفّات json. اكتب نصًا برمجيًا يأخذ دليلاً معينًا ويحوّل كل مِلَفّ yaml. فيه إلى مِلَفّ json. (بحيث أن database.yaml يصبح database.json، وتصبح المحتويات صالحة بالنسبة لـ JSON).

تمرين 12

اختار فريقك في البداية استخدام أسماءcamelCas[64] للمتغيرات، ولكن بعد ذلك غيّروا رأيهم الجماعي وتحولوا إلى snake_case[65]. اكتب برنامَج نصي يمسح جميع المِلَفّات المصدر لأسماء camelCase وينشئ تقارير عنها.

تمرين 13

متابعةً للتمرين السابق، أضف إمكانية تغيير أسماء المتغيرات هذه تلقائيًا في مِلَفّ واحد أو أكثر. تذكر أن تحتفظ بنسخة احتياطية من النسخ الأصليّة في حالة حدوث شيء ما بشكل سيء أو فظيع.

---

64 **[المترجم]** حالة الجمل (منسّقة على هيئة camelCase) هي ممارسة كتابة العبارات بحيث تبدأ كل كلمة أو اختصار في منتصف العبارة بحرف كبير، مع عدم وجود مسافات أو علامات ترقيم. تشمل الأمثلة الشائعة "iPhone" و "eBay". كما يتم استخدامه أحيانًا في أسماء المستخدمين عبر الإنترنت مثل "johnSmith"، ولجعل أسماء النطاقات المتعددة الكلمات أكثر وضوحًا، على سبيل المثال في الإعلانات.

65 **[المترجم]** حالة الثعبان (منسّقة على هيئة snake_case) هي ممارسة كتابة الكلمات أو العبارات المركبة التي يتم فيها فصل العناصر بتسطيرة سفلية واحدة (_) ودون مسافات، عادة ما يتم كتابة الحرف الأول من كل كلمة بأحرف صغيرة. على سبيل المثال fibonacci_sequence و code_breaker. وهي أكثر قواعد التسمية شيوعًا المستخدمة في الحوسبة كمعرفات لأسماء المتغيرات والوظائف والمِلَفّات.

وجدت دراسة واحدة في الأقل أن القراء يمكنهم التعرف على قيم snake_case بشكل أسرع من camelCase.

الموضوع 22. دفاتر يوميّات الهندسة

عمل ديف ذات مرّة مع شركة تصنيع أجهزة حاسوب صغيرة، مما يعني العمل جنبًا إلى جنب مع المهندسين الإلكترونيين وأحيانًا الميكانيكيين.

كان الكثير منهم يتجولون مع دفتر الملاحظات الورقي، وعادة مع قلم محشو في ظهر الدفتر. وبين الحين والآخر عندما كنا نتحدث، كانوا يخرجون دفتر الملاحظات ويخربشون شيئًا ما.

في النهاية سألهم ديف السؤال الواضح. اتضح أنه تم تدريبهم على الاحتفاظ بدفتر الهندسة اليومي، وهو نوع من الدفاتر التي يسجلون فيها ما فعلوه، والأشياء التي تعلموها، وملخصات الأفكار، وقراءات العدادات: إجمالًا أي شيء يتعلق بعملهم. عندما كان يمتلئ دفتر الملاحظات، كانوا يكتبون المجال الزمني على ظهره، ثم يضعونه على الرف بجانب الكتب اليوميّة السابقة. ربما كانت هناك منافسة لطيفة تدور حول مجموعة الكتب التي تحتل مساحة أكبر من الرف.

نستخدم دفاتر اليوميات لتدوين الملاحظات في الاجتماعات، ولتدوين ما نعمل عليه ولتدوين قيم المتغيرات عند تصحيح الأخطاء، ولحفظ التذكيرات للمكان الذي نضع فيه الأشياء، ولتسجيل الأفكار الجامحة، وأحيانًا لمجرد الخربشة[66].

لدفتر اليوميات ثلاث مزايا رئيسة:

- أنه أكثر موثوقية من الذاكرة. قد يسأل الناس "ما اسم تلك الشركة التي اتصلت بها الأسبوع الماضي بشأن مشكلة إمدادات الطاقة؟" ويمكنك تقليب صفحة أو نحو ذلك للخلف وإعطائهم الاسم والرَقَم.
- يمنحك مكانًا لتخزين الأفكار التي ليست ذات صلة بالمهمة قيد النظر على الفور. وبهذه الطريقة يمكنك الاستمرار في التركيز على ما تفعله، وانت تعرف أنه لن يتم نسيان تلك الفكرة الرائعة.
- يعمل كنوع من البط المطاطي (الموصوف في الصفحة 94). عندما تتوقف لكتابة شيء ما، قد يغير دماغك تردد أفكاره، تقريبًا كما لو كنت تتحدث إلى شخص آخر - فرصة عظيمة للتأثير. قد تبدأ في تدوين ملاحظة ثم تدرك فجأة أن ما قمت به للتو، موضوع الملاحظة، هو مجرّد خطأ واضح.

هناك فائدة إضافية أيضًا. يمكنك بين الحين والآخر إلقاء نظرة على ما كنت تفعله منذ سنوات عدّة والتفكير في الأشخاص والمشروعات والملابس المريعة وتسريحات الشعر.

لذا، حاول الاحتفاظ بدفتر يوميات الهندسة. استخدم الورق، وليس ملفًا أو ويكي: هناك شيء ما خاص حول فعل الكتابة مقارنة بالطباعة. امنحها شهرًا، وانظر إذا كنت ستحصل على مزايا أم لا.

إذا لم تتحصل شيء إضافي، فسيجعل ذلك كتابة مذكراتك أسهل عندما تصبح ثريًا ومشهورًا.

66 . هناك بعض الأدلة على أن الخربشة تساعد على التركيز وتحسن المهارات المعرفية، على سبيل المثال، انظر *ماذا تفعل الخربشة؟ What does doodling do? [And10]*

الأقسام ذات الصلة

- الموضوع 6، محفظتك المعرفية، في الصفحة 38
- الموضوع 37، استمع إلى حدسك، في الصفحة 218

الفصل الرابع:

جنون العظمة العملي

نصيحة 36 لا يمكنك كتابة برمجيّة مثاليّة

هل هذا مؤذٍ للمشاعر؟ لا ينبغي أن يكون كذلك. اقبله كحقيقة بديهيّة في الحياة. تقبّله. احتفل به. لأن البرمجيات المثاليّة غير موجودة. لم يكتب أحد في التاريخ القصير للحوسبة برمجيّة مثاليّة واحدة. من غير المرجّح أن تكون أول من يكتبها. وما لم تقبل هذا كحقيقة، فسوف ينتهي بك الأمر إلى إضاعة الوقت والطاقة في ملاحقة حلمٍ مستحيل.

لذا، وفي ضوء هذه الحقيقة المُحبطة، كيف يمكن للمبرمج الواقعي أن يحولها إلى مِيزة؟ هذا هو موضوع هذا الفصل.

يعتقد الجميع أنهم هم السائق الوحيد الجيّد على وجه الأرض. بقية العالم موجودون هنا لتجدهم متجاوزين لإشارات التوقف. متأرجحين في سيرهم، لا يشيرون عند المنعطفات، ويكتبون الرسائل النصيّة على هواتفهم، ببساطة لا يرتقون إلى مستوى معاييرنا. لذلك نحن نقود بشكل دفاعي. فنحن نترقب المشكلات قبل حدوثها، ونتوقع ما هو غير متوقع، ولا نضع أنفسنا أبدًا في موقف لا يمكننا تخليص أنفسنا منه.

إن تشابه ذلك مع كتابة الشِفرة واضح جدًا. فنحن نتواصل باستمرار مع شِفرة الناس الآخرين - شِفرة قد لا ترقى إلى مستوى معاييرنا العالية - ونتعامل مع المدخلات التي قد تكون صالحة أو قد لا تكون. لذلك نحن نتعلم البرمجة بطريقة دفاعيّة. إذا كان هناك أي شك، فإننا نتحقق صحة جميع المعلومات التي نقدّمها. نستخدم التأكيدات لكشف البيانات السيئة ونرتاب في البيانات القادمة من المهاجمين المحتملين أو المتصيدون. نتحقق الاتساق، ونضع قيودًا على أعمدة قاعدة البيانات، ونشعر عمومًا بالرضى عن أنفسنا.

لكن المبرمجين العمليين يأخذون هذا خطوةً إضافيةً إلى الأمام. *إنهم لا يثقون في أنفسهم أيضًا*. ولمعرفتهم بأنه لا أحد يكتب شِفرة مثالية، بما في ذلك هم أنفسهم، فإنهم يبنون دفاعات ضد أخطائهم. نشرح الإجراء الدفاعي الأول في **التصميم حسب العقد**: يجب على العملاء والموردين الاتفاق على الحقوق والمسؤوليات.

في **البرامج المينة لا تكذب**، نريد التأكّد عدم حدوث أي ضرر في أثناء عملنا على إصلاح الأخطاء. لذا نحاول التحقّق الأشياء كثيرًا وإنهاء البرنامَج إذا ساءت الأمور.

تصف **البرمجة التوكيديّة** طريقة سهلة للتحقّق على طول الطريق - كتابة الشفرات التي تتحقق افتراضاتك.

كلما أصبحت برامجك أكثر ديناميكية، ستجد نفسك تتلاعب بموارد النظام - الذاكرة والمِلَفّات والأجهزة وما شابه ذلك. سنقترح في **كيفيّة موازنة الموارد**، طرقًا لضمان عدم إسقاط أي من الكرات.

والأهم من ذلك، أننا نلتزم دائمًا بخطوات صغيرة، كما هو موضح في **لا تتجاوز مصابيحك الأماميّة**، حتى لا نقع من على حافة الجرف.

في عالم مليء بالأنظمة غير الكاملة، والمقاييس الزمنية السخيفة، والأدوات المضحكة، والمتطلبات المستحيلة، فلنلعبها بأمان. كما قال وودي آلن، "عندما يكون الجميع جاهزين حقًا للتباهي عليك، فإن جنون العظمة هو مجرّد تفكير جيد."

الموضوع 23. التصميم حسب العقد

> لا شيء يُذهل الرجال بقدر المنطق السليم والتعامل الواضح.
>
> ← رالف والدو إيمرسون، مقالات

من الصعب التعامل مع أنظمة الحاسوب. والتعامل مع الناس أصعب. لكن كبشر، كان لدينا وقت أطول لمعرفة قضايا التفاعلات البشرية. يمكن تطبيق بعض الحلول التي توصلنا إليها خلال آلاف السنين القليلة الماضية على كتابة البرمجيات أيضًا. العَقد contract هو أحد أفضل الحلول لضمان التعامل البسيط.

يحدّد العقد حقوق ومسؤوليات كلا الطرفين. إضافة إلى ذلك هناك اتفاق بشأن التبِعات في حال فشل أي منهما في الالتزام **بالعقد**.

ربما لديك عقد عمل يحدد ساعات العمل وقواعد السلوك التي يجب عليك اتباعها. في المقابل، تدفع لك الشركة راتبًا ومزايا أخرى. يفي كل طرف بالتزاماته ويستفيد الجميع.

إنها فكرة مستخدمة في جميع أنحاء العالم – بشكل رسمي وغير رسمي - لمساعدة البشر على التفاعل. هل يمكننا استخدام نفس المفهوم لمساعدة وحدات البرنامَج على التفاعل؟ الجواب" نعم".

التصميم حسب العقد DBC

طوّر برتراند ماير (*بناء البرمجيات كائنية التوجه [Mey97] Object-Oriented Software Construction*) مفهوم التصميم حسب العقد للغة إيفلEiffel.[67] إنها تقنية بسيطة لكنها قوية تركز على توثيق (والموافقة على) حقوق ومسؤوليات وحدات البرامج لضمان صحته. ما هو البرنامَج الصحيح؟ هو البرنامَج الذي يقوم بالمطلوب منه لا أكثر ولا أقل. إن توثيق والتحقّق هذا الادعاء هو جوهر التصميم حسب العقد Design by Contract (اختصارا DBC).

كل دالّة وطريقة في نظام البرمجيات تقوم بعمل ما. قبل أن تبدأ هذا العمل، قد يكون للدالة بعض التوقعات للحالة، وقد تكون قادرة على الإدلاء ببيان حول الحالة بعد تنفيذها. يصف *ماير* هذه التوقعات والمطالبات على النحو التالي:

الشروط المُسبقة

وهي ما يجب أن يكون صحيحًا من أجل استدعاء الروتين "الإجرائية"؛ متطلبات الروتين: يجب ألا يتم استدعاء الروتين أبدًا عندما تُنتهك شروطه المُسبقة. يتحمّل المُستدعي مسؤولية تمرير البيانات الجيّدة (انظر الصندوق المعنون "من المسؤول؟" في الصفحة 130).

67 استنادًا جزئيًا إلى عمل سابق قام به Dijkstra و Floyd و Hoare و Wirth وغيرهم.

الشروط اللاحقة

ماهي الأمور التي يضمن الروتين القيام بها؛ حالة العالم بعد تنفيذ الروتين. إن حقيقة أن الروتين يحتوي على شرط لاحق يعني أنه سينتهي: لا يُسمح بالحلقات اللانهائية.

ثوابت "لا متغيرات" الصنف (Class invariants)

يضمن الصنف أن هذا الشرط صحيح دائمًا من منظور المستدعي. في أثناء المعالجة الداخلية للروتين قد تتغيّر قيمة الثابت، ولكن بحلول الوقت الذي يخرج فيه التحكم من الروتين ويعود إلى المستدعي، يجب أن تكون قيمة الثابت صحيحة. (لاحظ أنه لا يمكن للصنف منح حق وصول غير مقيّد للكتابة إلى أي عضو بيانات يشارك في الثابت.)

وهكذا يمكن قراءة العَقد بين الروتين وأي مُستدعي محتمل على أنه

إذا استوفى المُستدعي جميع الشروط المُسبقة للروتين، فيجب أن يضمن الروتين أن جميع الشروط اللاحقة والثوابت ستكون صحيحة عند اكتمال تنفيذه.

إذا فشل أي من الطرفين في الالتزام بشروط العقد، فسيتم اللجوء إلى علاج "تسويّة" (تم الاتفاق عليها سابقًا) - ربما يُرفع استثناء، أو يُنهى البرنامَج. وأياً كان ما يحدث، فلا ينبغي لنا أن نخطئ في أن الفشل في الوفاء بالعقد يُعدّ خللًا. وهذا ليس بالأمر الذي يجب أن يحدث على الإطلاق، ولهذا السبب لا ينبغي استخدام الشروط المُسبقة لتنفيذ أمور مثل التحقّق صحة مدخلات المستخدم.

بعض اللغات لديها دعم أفضل لهذه المفاهيم من غيرها. على سبيل المثال، يدعم Clojure الشروط المسبقة واللاحقة فضلًا عن الأدوات الأكثر شمولاً التي توفرها المواصفات. فيما يلي مثال على دالّة مصرفيّة للقيام بالإيداع باستخدام شروط مسبقة ولاحقة:

```
(defn accept-deposit [account-id amount]
    { :pre [ (> amount 0.00)
             (account-open? account-id) ]
     :post [ (contains? (account-transactions account-id) %) ] }
    "Accept a deposit and return the new transaction id"
    ;; Some other processing goes here...
    ;; Return the newly created transaction:
    (create-transaction account-id :deposit amount))
```

هناك شرطان مسبقان للدالة accept-deposit. الأول هو أن المبلغ يجب أن يكون أكبر من الصفر، والثاني هو أن الحساب يجب أن يكون مفتوحًا وصالحًا، كما هو محدّد بالدالة المُسماة account-open?. هناك أيضًا شرط لاحق: تضمن الدالة إمكانية العثور على المناقلة (transaction) الجديدة (القيمة المُعادة لهذه الدالة، الممثلة هنا بـ "%") بين المناقلات لهذا الحساب.

إذا استدعيت الدالة " accept-deposit " مُستخدمًا مبلغًا موجبًا للإيداع وحسابًا صالحًا، فسيتم المضي في إنشاء مناقلة من النوع المناسب والقيام بأي معالجة أخرى مطلوبة. ومع ذلك، إذا كان هناك خطأ في البرنامَج وقمت بطريقة ما بتمرير مبلغ سالب للإيداع، فستحصل على استثناء وقت التشغيل:

```
Exception in thread "main"...
Caused by: java.lang.AssertionError: Assert failed: (> amount 0.0)
```

وبالمثل، تتطلب هذه الدالة أن يكون الحساب المحدّد مفتوحًا وصالحًا. إذا لم يكن كذلك، فسترى هذا الاستثناء بدلاً من السابق:

```
Exception in thread "main"...
Caused by: java.lang.AssertionError: Assert failed: (account-open? account-id)
```

تحتوي اللغات الأخرى على ميزات، والتي بالرغم من أنها ليست خاصة بالتصميم حسب العقد DBC، فإنه لا يزال من الممكن استخدامها لتحقيق تأثير جيد. على سبيل المثال، يستخدم *إكسير* عبارات حماية (guard clauses) لإرسال استدعاءات الدالة إلى العديد من الأجزاء المتاحة:

```
defmodule Deposits do
  def accept_deposit(account_id, amount) when (amount > 100000) do
    # Call the manager!
  end
  def accept_deposit(account_id, amount) when (amount > 10000) do
    # Extra Federal requirements for reporting
    # Some processing...
  end
  def accept_deposit(account_id, amount) when (amount > 0) do
    # Some processing...
  end
end
```

في هذه الحالة، قد يؤدي استدعاء "accept_deposit" باستخدام مبلغ كبير بما يكفي لقدح استدعاء خطوات ومعالجة إضافية. حاول استدعاءه باستخدام مبلغ أقل أو يساوي الصفر، ومع ذلك، ستحصل على استثناء لإبلاغك أنه لا يمكنك ذلك:

```
** (FunctionClauseError) no function clause matching in Deposits.accept_deposit/2
```

إن هذا النهج أفضل من مجرد التحقّق المدخلات الخاصة بك. ببساطة، في هذه الحالة لا يمكنك استدعاء هذه الدالة إذا كانت وسائطك (arguments) خارج المجال.

نصيحة ٣٧ | **صمّم باستخدام العقود**

في موضوع **التعامد**، أوصينا بكتابة شِفرة "خجولة". هنا، ينصبّ التركيز على الشِفرة "الكسولة": كن صارمًا فيما ستقبله قبل أن تبدأ العمل، وعِد بأقل قدر ممكن في المقابل. تذكر، إذا كان عقدك يشير إلى أنك ستقبل أي شيء وتعِد العالم بالمقابل، فلديك الكثير من الشِفرة لكتابتها!

في أي لغة برمجة، سواء كانت وظيفيّة، أو كائنية التوجه، أو إجرائيّة، يجبرك التصميم حسب العقد على التفكير.

ثوابت الصنف واللغات الوظيفيّة

هي تسمية لشيء. إن إيفل Eiffel هي لغة كائنية التوجه، لذلك أطلق *ماير* على هذه الفكرة اسم " ثابت الصنف". ولكن، في الحقيقة، إنها أكثر شمولية من ذلك. ما تشير إليه هذه الفكرة حقًا هي الحالة (state). ترتبط الحالة بمثيلات الصنف في اللغات كائنية التوجه. لكن اللغات الأخرى تمتلك حالة أيضًا.

في اللغات الوظيفيّة، عادة ما تمرر الحالة إلى الدوال وتتلقي حالة محدّثة نتيجة لذلك. إن مفاهيم الثوابت مفيدة بنفس القدر ضمن هذه الظروف.

التصميم حسب العقد والتطوير المعتمد على الاختبار

هل التصميم حسب العقد مطلوب في عالم يمارس فيه المطورون اختبار الوحدة، أو التطوير القائم على الاختبار Test-Driven Development اختصارا (TDD)، أو الاختبار المعتمد على الخاصيّة property-based testing، أو البرمجة الدفاعيّة؟

الإجابة القصيرة هي "نعم."

التصميم حسب العقد DBC والاختبار هما نهجان مختلفان للموضوع الأوسع لصحة البرنامَج. كلاهما له قيمة ولديه استخدامات في حالات مختلفة. تقدم DBC العديد من المزايا زيادة على نُهج الاختبار المحددة:

- لا يتطلب DBC أي إعداد أو إنشاء أغراض محاكية (mocking)
- يحدد DBC معامِلات النجاح أو الفشل في جميع الحالات، في حين أن الاختبار يمكن أن يستهدف حالة واحدة فقط في كل مرّة
- يحدث TDD والاختبارات الأخرى فقط في "وقت الاختبار" ضمن دورة البناء. لكن DBC والتأكيدات لا تنتهي: في أثناء التصميم والتطوير والنشر والصيانة
- لا يركز TDD على التحقق من الثوابت الداخلية داخل الشِفرة قيد الاختبار، إنه أسلوب الصندوق الأسود في التحقّق الواجهة العامة
- DBC أكثر كفاءة (ومراعاة لعدم التكرار) من البرمجة الدفاعية، حيث يتعين على الجميع التحقق من صحة البيانات في حال عدم قيام أي شخص آخر بذلك.

إن TDD هي تقنية رائعة، ولكن كما هو الحال مع العديد من التقنيات، قد تدعوك إلى التركيز على "المسار السعيد"، وليس العالم الحقيقي المملوء بالبيانات السيئة والممثلين السيئين والإصدارات السيئة والمواصفات السيئة.

تنفيذ التصميم حسب العقد

ببساطة إن تعداد ما هو نطاق مجال الإدخال، وما هي الشروط الحديّة، وما يعِد الروتين بتقديمه - أو الأهم، ما لا يعِد بتقديمه - قبل كتابة الشِفرة يمثّل قفزة نوعيّة إلى الأمام في كتابة برمجيات أفضل. وفي حال عدم ذكر هذه الأشياء، تكون قد عدت إلى *البرمجة بالمصادفة* (انظر المناقشة في الصفحة 223)، حيث تبدأ العديد من المشروعات ثم تنتهي وتفشل.

في اللغات التي لا تدعم التصميم حسب العقد في الشِفرة، قد يكون هذا أقصى ما يمكنك الذّهاب إليه - وهذا ليس سيئًا للغاية. في النهاية التصميم حسب العقد، هو تقنية تصميم - وحتى دون الفحص التلقائي- يمكنك وضع العقد في الشِفرة كتعليقات أو في اختبارات الوحدة مع الاستمرار في الحصول على فائدة حقيقية جدًا.

التأكيدات Assertions

بينما يعدُّ توثيق هذه الافتراضات بدايةً رائعة، فإنه بإمكانك الحصول على فائدة أكبر بكثير بواسطة جعل المترجم (compiler) يتحقق عقدك. يمكنك محاكاة ذلك جزئيًا في بعض اللغات باستخدام التأكيدات: فحوصات وقت التشغيل للشروط المنطقية (انظر الموضوع 25، البرمجة التوكيدية، في الصفحة 135). لماذا جزئياً فقط؟ ألا يمكنك استخدام التأكيدات للقيام بكل ما يمكن أن يفعله DBC؟

للأسف، فإن الجواب هو لا. قبل كل شيء، ربما لا يوجد في اللغات كائنية التوجه دعم لنشر التأكيدات أسفل التسلسل الهرمي للوراثة. هذا يعني أنه إذا تجاوزتَ (override)[68] طريقة صنف أساس "أب" (a base class method) لديها عقد، فلن يتم استدعاء التأكيدات التي تنقذ هذا العقد بشكل صحيح (إلا إذا كرّرتها يدويًا في الشِفرة الجديدة). يجب أن تتذكر استدعاء ثابت الصنف (وجميع ثوابت الصنف الأب) يدويًا قبل الخروج من كل طريقة. المشكلة الأساسيّة هي أن العقد لا يُنفّذ تلقائيًا.

في بيئات أخرى، قد يتم إيقاف تشغيل الاستثناءات الناتجة عن التأكيدات بنمط DBC عمومًا أو يتم تجاهلها في الشِفرة.

أيضًا، لا يوجد مفهوم مضمّن للقيم القديمة " old "؛ أي القيم كما كانت موجودة عند إدخالها إلى الطريقة method. إذا كنت تستخدم التأكيدات لفرض العقود، فيجب إليك إضافة تعليمات برمجية إلى الشروط المُسبقة لحفظ أي معلومات تريد استخدامها في الشروط اللاحقة، في حال كانت اللغة ستسمح بذلك. في لغة إيفل، حيث ولدت DBC، يمكنك فقط استخدام التعبير old.

أخيرًا، لم تُصمّم أنظمة ومكتبات وقت التشغيل التقليدية لدعم العقود، لذلك لا يتم فحص هذه الاستدعاءات. وهذه خسارة كبيرة، لأنها غالبًا ما تكون عند الحد الفاصل بين الشِفرة البرمجية الخاصة بك والمكتبات التي تستخدمها والتي يتم اكتشاف معظم المشكلات فيها (راجع الموضوع 24، البرامج الميتة لا تكذب، في الصفحة 133 للحصول على مناقشة أكثر تفصيلًا).

التصميم حسب العقد والتحطيم المبكر

يتناسب التصميم حسب العقد DBC جيّدًا مع مفهومنا الخاص بالتعطيل المبكر (انظر الموضوع 24، البرامج الميتة لا تكذب، في الصفحة 133). باستخدام آلية تأكيد أو DBC للتحقّق الشروط المسبقة، والشروط اللاحقة، والثوابت، يمكنك التعطيل مبكرًا والإبلاغ عن معلومات أكثر دقّة حول المشكلة.

على سبيل المثال، افترض أن لديك طريقة لحساب الجذور التربيعية. تحتاج إلى شرط مسبق DBC يقصر المجال إلى أرقام موجبة. إذا مرّرت قيمة سالبة لمعامل sqrt، في اللغات التي تدعم DBC، فسوف تحصل خطأ إعلامي مثل

```
sqrt_arg_must_be_positive, along with a stack trace.
```

68 **[المترجم]** تعني إعادة تعريفك لطريقة في الصنف الابن معرّفة أصلا في الصنف الأب، حيث يتم تنفيذها بدلًا من الأصلية.

هذا أفضل من البديل في لغات أخرى مثل جافا و سي و سي++ حيث يؤدي تمرير رَقَم سالب إلى sqrt إلى إرجاع القيمة الخاصة NaN (Not a Number) "ليس رقمًا". قد يستغرق الأمر بعض الوقت في البرنامَج الذي تحاول إجراء بعض الحسابات فيه على NaN، مع الحصول على نتائج عجيبة.

من المسؤول؟

من المسؤول عن التحقّق الشرط المُسبق المُستدعي أم الروتين؟ عند تنفيذه كجزء من اللغة، فإن الإجابة لا هذا ولا ذاك: يتم اختبار الشرط المسبق خلف الكواليس بعد استدعاء المُستدعي ولكن قبل الدخول إلى الروتين نفسه. لذا إذا كان هناك أي فحص صريح للمعامِلات المطلوب إجراؤها، فيجب أن يقوم بها المُستدعي، لأن الروتين نفسه لن يرى أبدًا معامِلات تنتهك شرطه المسبق. (بالنسبة للغات التي لا تحتوي على دعم مضمّن، ستحتاج إلى دعم الروتين المُستدعى بتمهيد سابق و/ أو لاحق لفحص هذه التأكيدات.)

افترض وجود برنامجًا يقرأ رقمًا من وحدة التحكم "كونسول" (console)، ويحسب جذره التربيعي (عن طريق استدعاء الدالة sqrt)، ويطبع النتيجة. للدالة sqrt شرط مسبق- يجب ألا يكون المحدّد (argument) سلبي. إذا قام المستخدم بإدخال رَقَم سالب في وحدة التحكم، فالأمر متروك لشِفرة الاستدعاء للتأكّد عدم تمريره إلى sqrt. تحتوي شِفرة الاستدعاء هذه على العديد من الخيارات: يمكن أن تُنهى العملية، أويمكنها إصدار تحذير وقراءة رَقَم آخر، أو يمكن أن تجعل الرَقَم موجبًا وتلحق i بالنتيجة المُرجعة بواسطة sqrt.

بغض النظر عن خيارها، هذه بالتأكيد ليست مشكلة الدالة sqrt.

بواسطة التعبير عن مجال دالة الجذر التربيعي في الشرط المسبق لروتين sqrt، تحول عبء التأكد الصّحة إلى المستدعي - حيث ينتمي. يمكنك بعد ذلك تصميم روتين sqrt بأمان مع العلم أن مدخلاته ستكون ضمن المجال المقبول.

من الأسهل إيجاد المشكلة وتشخيصها بواسطة التعطيل المبكر في موضع المشكلة.

الثوابت الدلالية

يمكنك استخدام الثوابت الدلاليّة للتعبير عن المتطلبات التي لا تُنتهك، وهو نوع من "العقد الفلسفي".

كتبنا ذات مرة مفتاح مناقلة بطاقة الخَصم "المدين". كان الشرط الرئيس هو أن مستخدم بطاقة الخَصم يجب ألا يكون له نفس المناقلة (transaction) مطبّقة على حسابه مرتين. بمعنى آخر، بغض النظر عن نوع وضع الفشل الذي قد يحدث، يجب أن يكون الخطأ في جانب عدم معالجة المناقلة وليس في جانب معالجة مناقلة مكرّرة.

أثبت هذا القانون البسيط، المُستند مباشرة على المتطلبات، أنه مفيد جدًا في فرز سيناريوهات استرداد الأخطاء المعقّدة، وتوجيه التصميم التفصيلي والتنفيذ في العديد من المجالات.

تأكّد عدم الخلط بين المتطلبات الثابتة، وغير المنتهكة للقوانين مع تلك التي هي مجرّد سياسات قد تتغير مع تغير نظام الإدارة. لهذا السبب نستخدم مصطلح الثوابت الدلاليّة- يجب أن يكون محوريًا لمعنى شيء ما، وليس خاضعًا لأهواء السياسة (وهذا هو الغرض من قواعد الأعمال الأكثر ديناميكية).

عندما تجد متطلبًا مؤهلاً، تأكّد أن يصبح جزءًا معروفًا جيّدًا من أي وثائق ننتجها - سواء كانت قائمة تعداد نقطية في وثيقة المتطلبات التي تُوقّع على ثلاث نسخ أو مجرّد ملاحظة كبيرة على السَّبُّورَة البيضاء المشتركة التي يمكن للجميع رؤيتها. حاول أن توضح ذلك بشكل لا لبس فيه. على سبيل المثال، في مثال بطاقة الخَصْم، قد نكتب

خطأ لمصلحة العميل.

هذه عبارة واضحة وموجزة لا لبس فيها قابلة للتطبيق في العديد من المجالات المختلفة للنظام. إنه عقدنا مع جميع مستخدمي النظام، ضماننا للسلوك.

العقود والوكلاء الديناميكيون

تحدثنا حتى الآن، عن العقود كمواصفات ثابتة وغير قابلة للتغيير. ولكن من منظور الوكلاء المستقلين، لا يلزم أن يكون الحال كذلك. بواسطة تعريف " الاستقلالية"، يحقّ للعملاء رفض الطلبات التي لا يرغبون في احترامها. إنهم أحرار في إعادة التفاوض على العقد- "لا يمكنني تقديم ذلك، ولكن إذا أعطيتني هذا، فقد أقدم شيئًا آخر."

من المؤكّد أن أي نظام يعتمد على تقنية الوكيل يعتمد بشكل كبير على الترتيبات التعاقدية - حتى لو تم إنشاؤها ديناميكيًا.

تخيّل: مع ما يكفي من المكونات والوكلاء الذين يمكنهم التفاوض بشأن عقودهم فيما بينهم لتحقيق هدف، قد نحل أزمة إنتاجية البرامج فقط عن طريق السماح للبرنامَج بحلها لنا.

ولكن إذا لم نتمكن من استخدام العقود يدويًا، فلن نتمكن من استخدامها تلقائيًا. لذا في المرّة القادمة التي تصمم فيها برنامجًا، صمّم عقده أيضًا.

الأقسام ذات الصلة

- الموضوع 24، البرامج الميتة لا تكذب، في الصفحة 133
- الموضوع 25، البرمجة التوكيدية، في الصفحة 135
- الموضوع 38، البرمجة بالمصادفة، في الصفحة 223
- الموضوع 42، الاختبار المعتمد على الخاصية، في الصفحة 249
- الموضوع 43، ابق آمنًا هناك، في الصفحة 255
- الموضوع 45، هوّة المتطلبات، في الصفحة 268

تحديات

- نِقَاط للتفكير: إذا كان التصميم حسب العقد قويًا للغاية، فلماذا لا يُستخدم على نطاق أوسع؟ هل من الصعب إنتاج العقد؟ هل يجعلك تفكّر في القضايا التي تفضل تجاهلها الآن؟ هل يجبرك على التفكير!؟ من الواضح أنه أداة شديد الْخَطَر!

تمارين

تمرين 14 (إجابة محتملة في الصفحة 324)

تصميم واجهة لخلاط المطبخ. سيكون في النهاية خلاطًا معتمدًا على الويب وممكنًا بـإنترنت الأشـياء[69]، ولكن في الـوقت الحـالي نحن بحاجة فقط إلى الواجهة للتحكم فيه. لديه عشـرة إعـدادات للسـرعة (0 يعـني إيقـاف). لا يمكنـك تشـغيله فارغًا، ويمكنـك تغيـير السرعة وحدة واحدة فقط في كل مرة (أي من 0 إلى 1 ومن 1 إلى 2، وليس من 0 إلى 2).

إليك الطرائق. أضف الشروط المسبقة واللاحقة والثابت.

```
int getSpeed()
void setSpeed(int x)
boolean isFull()
void fill()
void empty()
```

تمرين 15 (إجابة محتملة في الصفحة 324)

كم عدد الأرقام في السلسلة 0، 5، 10، 15، ...، 100؟

69 **[المترجم]** إنترنت الأشياء (بالإنجليزية: Internet of Things – IoT)، مصطلح يُقصد به الجيل الجديـد من الإنـترنت الـذي يتيح التفاهم بين الأجهزة المترابطة مع بعضها (عبر مراسم الإنترنت). وتشمل هذه الأجهـزة الأدوات والمستشـعرات والحساسـات وأدوات الذكاء الاصطناعي المختلفة وغيرها. ويتخطى هـذا التعريـف المفهـوم التقليـدي وهـو تواصـل الأشـخاص مـع الحواسـيب والهواتف الذكية عبر شبكة عالميـة واحـدة وبواسـطة مراسـم الإنـترنت التقليـدي المعـروف. ومـا يمـيز إنـترنت الأشـياء أنهـا تـتيح للإنسان التحرر من المكان، أي إن الشخص يستطيع التحكم في الأدوات من دون الحاجة إلى التواجد في مكان محدّد للتعامل مـع جهاز معين.

الموضوع 24. البرامج الميتة لا تكذب

هل لاحظت أنه في بعض الأحيان يمكن للأشخاص الآخرين اكتشاف أن أمورك ليست على ما يرام قبل أن تدرك أنت المشكلة بنفسك؟ ينسحب الأمر نفسه على شِفرة الآخرين. إذا بدأ شيء ما في الانحراف في أحد برامجنا، فأحيانًا تكون المكتبة أو روتين إطار العمل أول من يكتشفه.

ربما مرّرنا قيمة صفرية أو قائمة فارغة. ربما هناك مفتاح مفقود في هذه التجزئة الهاش (hash)، أو أن القيمة التي اعتقدنا أنها تحتوي على hash تحتوي حقيقةً على list بدلاً منها. ربما حدث خطأ في الشبكة أو خطأ في نظام المِلَفّات لم نكتشفه، ولدينا بيانات فارغة أو تالفة. وجود خطأ منطقي قبل بضعة ملايين من التعليمات يعني أن محدّد تعليمة case لم تعد قيمته كما المتوقع 1 أو 2 أو 3. سنصل إلى الحالة default بشكل غير متوقع. وهذا أيضًا أحد الأسباب التي تجعل كل تعليمة case/switch بحاجة إلى عبارة default: نريد أن نعرف متى حدث "المستحيل".

من السهل الوقوع في ذهنية "لا يمكن أن يحدث". معظمنا كتب شِفرة لم يتحقق فيها إغلاق المِلَفّ بنجاح، أو من أن تعليمة التتبع قد كُتبت كما توقعنا. وبالنظر إلى جميع الأمور بتساوٍ، فمن المحتمل أننا لم نكن بحاجة إلى ذلك- لن تفشل الشِفرة المعنية في ظل أي ظروف عادية. لكننا نكتب الشِفرة بشكل دفاعي. فنحن نتأكد أن البيانات هي ما نعتقد أنها عليه، وأن الشِفرة في الإنتاج هي الشِفرة التي نعتقد وجودها. ونتحقق أنه تم تحميل الإصدارات الصحيحة من الاعتماديات فعلًا.

تمنحك جميع الأخطاء معلومات. يمكنك إقناع نفسك بأن الخطأ لا يمكن أن يحدث، واختيار تجاهلها. بدلاً من ذلك، يقول المبرمجون الواقعيون لأنفسهم أنه إذا كان هناك خطأ، فقد حدث شيء سيئ جدًا. لا تنس قراءة رسالة الخطأ اللعينة (انظر المبرمج في أرض غريبة، في الصفحة 112).

الالتقاط والتحرير مخصص للأسماك[70]

يشعر بعض المطورين أنه من الجيّد أن تلتقط أو تنقذ جميع الاستثناءات، وتعيد رفعها بعد كتابة نوع من الرسائل التوضيحية. شفرتهم مليئة بأشياء من هذا القبيل (حيث إن تعليمة الرفع المجرّدة تعيد رفع الاستثناء الحالي):

```
try do
    add_score_to_board(score);
rescue InvalidScore
    Logger.error("لا يمكن إضافة درجة غير صالحة. جاري الخروج");
    raise
rescue BoardServerDown
    Logger.error("لا يمكن إضافة النتيجة: اللوحة معطلة. جاري الخروج.");
    raise
rescue StaleTransaction
    Logger.error("لا يمكن إضافة النتيجة: المناقلة تالفة. جاري الخروج.");
    raise
end
```

إليك كيفية كتابة المبرمجين الواقعيين لما سبق:

```
add_score_to_board(score);
```

70 **[المترجم]** الصيد والإفراج ممارسة في الصيد الترفيهي. فبعد التقاط الأسماك، تُنزع من الماء وتعاد إليه ثانيةً. في كثير من الأحيان، يعد القياس السريع ووزن الأسماك، متبوعًا بالتصوير موضع الاهتمام. وغالبًا ما يكون بالإمكان إطلاق السمكة دون إزالتها من الماء.

نحن نفضل هذا لسببين. أولاً، لا يتم تجاوز شِفرة التطبيق بواسطة معالجة الخطأ. ثانيًا، وربما الأكثر أهمية، أن الشِفرة أقل اقترانًا. في المثال المطوّل، يجب أن نذكر كل استثناء يمكن أن ترفعه الطريقة add_score_to_board. إذا أضاف كاتب هذه الطريقة استثناءًا آخر، فإن شفرتنا تصبح منتهية الصَّلاحِيَة. في الإصدار الثاني الأكثر واقعية، يُنشر الاستثناء الجديد تلقائيًا.

نصيحة ٣٨ عطّل في وقت مبكر

حطّم، لا تهمل

تتمثل إحدى مزايا اكتشاف المشكلات في أقرب وقت ممكن في أنه يمكنك تعطيلها مبكرًا، وكثيرًا ما يكون التعطيل أفضل شيء يمكنك القيام به. قد يكون البديل هو الاستمرار، وكتابة البيانات التالفة إلى بعض قواعد البيانات الحيويّة أو توجيه الغسالة إلى دورة التدوير العشرين التالية.

تتبنى لغتا إرلانج Erlang و إكسير Elixir هذه الفلسفة. غالبًا ما يُقتبس جو أرمسترونغ، مخترع إرلانج ومؤلف *برمجة إرلانج: برمجيات لعالم متزامن [Arm07] Programming Erlang: Software for a Concurrent World*، قائلاً: "البرمجة الدفاعية مضيعة للوقت. دعها تتحطم! " في هذه البيئات، صُممت البرامج للفشل، ولكن يتم إدارة هذا الفشل من قبل المشرفين. فالمشرف مسؤول عن تشغيل الشِفرة ويعرف ما يجب فعله في حالة فشلها، والذي قد يشمل التنظيف بعدها، وإعادة تشغيلها، وما إلى ذلك. ماذا يحدث عندما يفشل المشرف نفسه؟ يدير المشرف الخاص به ذلك الحدث، مما يؤدي إلى تصميم يتكون من أشجار(هرمية) المشرفين. هذه التقنية فعالة للغاية وتساعد على تفسير استخدام هذه اللغات في أنظمة عالية التوافرية ومقاومة للخطأ.

في بيئات أخرى، قد يكون من غير المناسب ببساطة إنهاء برنامَج قيد التشغيل. ربما تكون قد طالبت بموارد لم يتم إصدارها بعد، أو قد تحتاج إلى كتابة رسائل السجل أو ترتيب المعامَلات المفتوحة أو التفاعل مع العمليات الأخرى.

ومع ذلك، يظل المبدأ الأساسي كما هو - عندما تكتشف شفرتك أن شيئًا كان من المفترض أن يكون مستحيلاً حدث للتو، فإن برنامَجك لم يعد قابلاً للتطبيق. أي شيء يفعله من هذه النقطة يصبح مشبوهًا، لذا قم بإنهائه في أقرب وقت ممكن.

البرنامَج الميِّت عادةً ما يلحق ضررًا أقل بكثير من البرنامَج المعوّق.

الأقسام ذات الصلة

- الموضوع 20، تنقيح الأخطاء، في الصفحة 110
- الموضوع 23، التصميم حسب العقد، في الصفحة 125
- الموضوع 25، البرمجة التوكيدية، في الصفحة 135
- الموضوع 26، كيفية موازنة الموارد، في الصفحة 139
- الموضوع 43، ابق آمنًا هناك، في الصفحة 255

الموضوع 25. البرمجة التوكيديّة

> هناك ترف في لوم الذات. عندما نلوم أنفسنا فنحن لا تشعر بأن أحدًا آخر لديه الحق في إلقاء اللوم علينا.
>
> ← أوسكار وايلد، صورة دوريان جراي

يبدو أن هناك شعارًا يجب على كل مبرمج حفظه في وقت مبكر من حياته أو حياتها المهنية. إنه معتقد أساسي للحوسبة، وهو اعتقاد جوهري نتعلم تطبيقه على المتطلبات والتصميمات والشِفرة والتعليقات وكل ما نقوم به تقريبًا. تقول

هذا لا يمكن أن يحدث...

"لن يُستخدمَ هذا التطبيق أبدًا خارج البلاد، فما الحاجة لتدويله؟" "لا يمكن أن يكون count سالبًا". " لا يمكن أن يفشل التسجيل."

دعونا لا نمارس هذا النوع من خداع الذات، خاصّة عند كتابة الشِفرة.

نصيحة ٣٩ استخدم التأكيدات لمنع المستحيل

عندما تجد نفسك تفكّر "ولكن بالطبع هذا لا يمكن أن يحدث أبدًا"، فأضف شفرةً للتحقق منه. أسهل طريقة للقيام بذلك هي التأكيدات. ستجد في العديد من تطبيقات اللغة، بعض أشكال التأكيد التي تتحقق الشروط المنطقية Boolean[71]. يمكن أن تكون عمليات التحقق هذه لا تقدر بثمن. إذا كان معامل أو نتيجة ما يجب ألا تكون أبدًا قيمه فارغة null، فتحقق ذلك بشكل صريح:

```
assert (result != null);
```

في تنفيذ جافا، يمكنك (ويجب) إضافة سلسلة نصيّة للشرح:

```
assert result != null && result.size() > 0 : "Empty result from XYZ";
```

التأكيدات هي أيضًا عمليات تحقق مفيدة في عملية الخوارزمية. ربما تكون قد كتبت خوارزمية فرز ذكيّة تسمى my_sort. تحقق أنها تعمل:

```
books = my_sort(find("scifi"))
assert(is_sorted?(books))
```

لا تستخدم التأكيدات بدلاً من المعالجة الحقيقية للأخطاء. تتحقق التأكيدات الأشياء التي يجب ألا تحدث أبدًا: لا تريد أن تكتب شِفرة كالشِفرة التالية:

```
puts("Enter 'Y' or 'N': ")
ans = gets[0] # خذ المحرف الأول من الاستجابة
assert((ch == 'Y') || (ch == 'N')) # فكرة سيئة جدا
```

71 في سي و سي++ يتم تنفيذ هذه عادةً كوحدات ماكرو. وفي جافا، يتم تعطيل التأكيدات بشكل افتراضي.استدع آلة جافا الافتراضية Java VM باستخدام "تمكين التأكيدات" لتمكينها وتركها ممكّنة.

ولأن معظم عمليات تنفيذ التأكيد ستُنهي العملية عندما يفشل التأكيد، فليس هناك سبب يدعو إلى ضرورة استخدام الإصدارات التي تكتبها. إذا كنت بحاجة إلى تحرير الموارد، فالتقط استثناء التأكيد أو اعترض "تصّيد" الخروج، وشغل معالج الأخطاء الخاص بك. ما عليك سوى التأكّد أن الشِفرة التي تنفذها في تلك الجزء من الألف من الثانية تلك لا تعتمد على المعلومات التي تسببت في فشل التأكيد في المكان الأول.

التأكيدات والآثار الجانبية

إنه أمرٌ محرج في الواقع عندما ينتهي الأمر بالشِفرة التي نضيفها لاكتشاف الأخطاء إلى إنشاء أخطاء جديدة. يمكن أن يحدث هذا مع التأكيدات إذا كان لتقييم الحالة آثار جانبية. على سبيل المثال، قد تكون كتابة شِفرة شيء من هذا القبيل، فكرة سيئة

```
while (iter.hasMoreElements()) {
    assert(iter.nextElement() != null);
     Object obj = iter.nextElement();
    // ....
}
```

إن استدعاء nextElement(). في التأكيد له تأثير جانبي في نقل المكرّر إلى ما بعد العنصر الذي يتم جلبه، وبالتالي فإن الحلقة ستعالج نصف العناصر في المجموعة فقط. سيكون من الأفضل أن تكتب

```
while (iter.hasMoreElements()) {
     Object obj = iter.nextElement();
    assert(obj != null);
    // ....
}
```

هذه المشكلة هي نوع من هيسنبوغ Heisenbug[72] - التصحيح الذي يغير سلوك النظام الذي يتم تصحيحه.

(نعتقد أيضًا أنه في الوقت الحاضر، عندما تحظى معظم اللغات بدعم لائق لتكرار الدوال عبر المجموعات (collections)، فإن هذا النوع من الحلقات الصريحة غير ضروري وسيء التشكيل.)

اترك التأكيدات قيد التشغيل

هناك سوء فهم شائع حول التأكيدات. يبدو شيء مثل:

> التأكيدات تضيف بعض النفقات العامة إلى الشِفرة. نظرًا لأنها تبحث عن الأشياء التي يجب ألا تحدث أبدًا، فلن يتم تشغيلها إلا بواسطة خطأ في الشِفرة. بمجرّد اختبار الشِفرة وشحنها، لا تعود هناك حاجة إليها، ويجب إيقاف تشغيلها لجعل الشِفرة تعمل بشكل أسرع. التأكيدات هي وسيلة تصحيح.

هناك افتراضان خاطئان تمامًا هنا. أولاً، يفترضون أن الاختبار يجد جميع الأخطاء. في الواقع، بالنسبة لأي برنامَج معقّد، من غير المرجح أن تختبر حتى نسبة ضئيلة من التباديل (permutations) التي سيتم وضع شفرتك فيها. ثانيًا، ينسى المتفائلون أن برنامَجك يعمل في عالم شديد الخَطَر. في أثناء الاختبار، من المحتمل ألا تنخر الفئران في كابل اتصالات، وألا يستنفد الذاكرة شخصٌ

72 http://www.eps.mcgill.ca/jargon/jargon.html#heisenbug

يلعب لعبة ما، وألا تملأ مِلَفّات السجل قطاع التخزين. لكن قد تحدث هذه الأشياء عندما يعمل برنامَجك في بيئة إنتاج. خط الدفاع الأول هو التحقّق أي خطأ محتمل، والثاني هو استخدام التأكيدات في محاولة كشف تلك الأخطاء التي فاتتك.

يشبه إيقاف التأكيدات عند تسليم برنامَج للإنتاج مثل العبور على حبل رفيع مُعلّق دون شبكة أمان لأنك عبرته مرّة في مرحلة الاختبار. هناك قيمة كبيرة، ولكن من الصعب الحصول على تأمين على الحياة.

حتى إذا كانت لديك مشكلات في الأداء، فأوقف فقط التأكيدات التي أثرت عليك حقًا. قد يكون مثال الفرز أعلاه جزءًا مهمًا من تطبيقك، وقد يحتاج إلى أن يكون سريعًا. تعني إضافة الفحص مرورًا آخر عبر البيانات، ما قد يكون غير مقبول. اجعل هذا الفحص بالتحديد اختياريًا، لكن اترك الباقي.

استخدم التأكيدات في الإنتاج، واربح أموالاً طائلة

ترأس جار سابق لـ*أندي* على شركةً ناشئةً صغيرة صنعت أجهزة الشبكة. كان من أسرار نجاحهم قرار ترك التأكيدات في مكانها في إصدارات الإنتاج. صيغت هذه التأكيدات جيّدًا للإبلاغ عن جميع البيانات ذات الصلة التي تؤدي إلى الفشل، وقُدّمت عبر واجهة مستخدم جميلة المظهر للمستخدم النهائي. سمح هذا المستوى من ردود فعل المستخدمين الحقيقيين في ظل الظروف الفعليّة، سمح للمطورين بسد الثغرات وإصلاح هذه الأخطاء الغامضة التي يصعب إعادة إنتاجها، مما أدى إلى إنتاج برمجيّة مستقرة بشكل ملحوظ مقاومة للرصاص.

هذه الشركة الصغيرة غير المعروفة التي تملك هذا المنتج المتين، سرعان ما تم شراؤها بمئات الملايين من الدولارات.

مجرد قول.

تمرين 16 (إجابة محتملة في الصفحة 324)

فحص سريع للواقع. أي من هذه الأشياء "المستحيلة" يمكن أن تحدث؟

- شهر بأقل من 28 يومًا
- شِفرة خطأ من استدعاء النظام: تعذر الوصول إلى الدليل الحالي
- في سي++: a = 2; b =3; لكن (a + b) لا تساوي 5
- مثلث مجموع زواياه الداخلية ≠ 180 °
- دقيقة لا تحوي 60 ثانية
- a => (a + 1)

الأقسام ذات الصلة

- الموضوع 23، التصميم حسب العقد، في الصفحة 125
- الموضوع 24، البرامج الميتة لا تكذب، في الصفحة 133
- الموضوع 42، الاختبار المعتمد على الخاصية، في الصفحة 249
- الموضوع 43، ابق آمنًا هناك، في الصفحة 255

الموضوع 26. كيفيّة موازنة الموارد

> إشعالك لشمعة.. لابد من أن ينشر ظلالًا
>
> ← أورسولا ك. لو جوين، ساحر الأرض

ندير جميعًا الموارد كلما كتبنا الشِفرة: الذاكرة، والمناقلات، والتشعبات (threads)[73] واتصالات الشبكة والمِلَفّات والمؤقتات الزمنيّة- جميع أنواع الأشياء بتوافرية محدودة.

في معظم الأحيان، يتبع استخدام الموارد نمطًا يمكن التنبؤ به: تخصص المورد واستخدامه ثم إلغاء تخصيصه.

ومع ذلك، ليس لدى العديد من المطورين خِطّة متناسقة للتعامل مع تخصيص الموارد وإلغاء تخصيصها. لذلك دعونا نقترح نصيحة بسيطة:

نصيحة ٤٠	أنهِ ما تبدأه

هذه النصيحة سهلة التطبيق في معظم الظروف. هذا يعني ببساطة أن الدالة أو الكائن الذي يخصّص موردًا ما يجب أن يكون مسؤولاً عن إلغاء تخصيصه. دعنا نرى كيف يُطبّق ذلك بواسطة إلقاء نظرة على مثال لبعض الشفرات السيئة - جزء من برنامَج روبي الذي يفتح ملفًا، ويقرأ معلومات العملاء منه، ويحدّث حقلاً، ويكتب النتيجة مرّة أخرى. لقد أزلنا معالجة الأخطاء لجعل المثال أكثر وضوحًا:

```
def read_customer
   @customer_file = File.open(@name + ".rec", "r+")
   @balance = BigDecimal(@customer_file.gets)
end
def write_customer
   @customer_file.rewind
   @customer_file.puts @balance.to_s
   @customer_file.close
end
def update_customer(transaction_amount)
    read_customer
    @balance = @balance.add(transaction_amount,2)
    write_customer
end
```

للوهلة الأولى، يبدو الروتين "الإجرائية" update_customer معقولاً. يبدو أنه ينفّذ المنطق الذي نطلبه - قراءة السجل وتحديث الرصيد وكتابة السجل مرّة أخرى. ومع ذلك، فإن هذا الترتيب يخفي مشكلة كبيرة. إن الروتين "read_customer" والروتين "write_customer" مرتبطان بإحكام[74] - وهما يشتركان في متغير المثيل المسمى "customer_file". يفتح

73 **[المترجم]** تسمى أحيانًا المسارات أو الخيوط (https://ar.wikipedia.org/wiki/%D8%AE%D9%8A%D8%B7_(%D8%AD%D8%A7%D8%B3%D9%88%D8%A8)

74 لمناقشة مخاطر الشفرات المقترنة، راجع الموضوع 28، الفصل، في الصفحة 151.

read_customer المِلَفّ ويخزن مرجع المِلَفّ في "customer_file"، ثم يستخدم "write_customer" هـذا المرجـع المخـزّن لإغلاق المِلَفّ عند الانتهاء. هذا المتغير المشترك لا يظهر حتى في الروتين "update_customer".

لماذا يُعدّ هذا سيئًا؟ لنأخذ مبرمج الصيانة البـائس الـذي قيـل لـه أن المواصـفات تغـيرت - يجب تحـديث الرصـيد فقـط إذا لم تكن القيمة الجديدة سالبة. سيذهبون إلى المصدر ويغيرون "update_customer":

```
def update_customer(transaction_amount)
    read_customer
    if (transaction_amount >= 0.00)
       @balance = @balance.add(transaction_amount,2)
       write_customer
     end
end
```

يبدو كل شيء على ما يرام في أثناء الاختبار. ومع ذلك، عندما تدخل الشِفرة حيز الإنتاج، فإنها تنهـار بعـد عـدّة سـاعات، مُشـتكيةً من عدد كبير من المِلَفّات المفتوحة. اتضح أن "write_customer" لا يتم استدعاؤه في بعض الظروف. وعندما يحدث ذلـك، لا يتم إغلاق المِلَفّ.

أحد الحلول السيئة جدًا لهذه المشكلة التعامل مع الحالة الخاصة في "update_customer":

```
def update_customer(transaction_amount)
    read_customer
    if (transaction_amount >= 0.00)
      @balance += BigDecimal(transaction_amount, 2)
      write_customer
   else
      @customer_file.close # فكرة سيئة
   end
end
```

سيؤدي هذا إلى حل المشكلة - سيتم إغلاق المِلَفّ الآن بغض النظر عن الرصـيد الجديـد - ولكن هـذا الإصـلاح الآن يعـني أن هنـاك ثلاثة روتينات مقترنة مع بعضها بواسطة المتغير المشترك customer_file، وأن تتبع متى يكون المِلَفّ مفتوحًا أم لا يبدأ في إحداث فوضى. نحن نقع في فخ، وستبدأ الأمور في الانحدار بسرعة إذا واصلنا هذا المسار. فهو غير متوازن!

تخبرنا نصيحة هذا الموضوع **"أنهي ما بدأته"** أنه، من الناحية المثالية، يجب على الروتين الذي يخصّص مـوردًا أن يحـرّره أيضًـا. يمكننا تطبيقها هنا عن طريق إعادة بناء الشِفرة قليلاً:

```
def read_customer(file)
    @balance=BigDecimal(file.gets)
end
def write_customer(file)
    file.rewind
    file.puts @balance.to_s
end
def update_customer(transaction_amount)
    file=File.open(@name + ".rec", "r+")                    # >--
    read_customer(file)                                     #    |
   @balance = @balance.add(transaction_amount,2)  #    |
   write_customer(file)                                     #    |
   file.close                                                 # <--
end
```

بدلاً من التمسك بمرجع المِلَفّ، غيرنا الشِفرة لتمريره كمُعامل[75]. الآن كل مسؤولية المِلَفّ تقع على عاتق الروتين "update_customer". يفتح المِلَفّ و (إنهاء ما يبدأه) وإغلاقه قبل الإرجاع. يوازن الروتين بين استخدام المِلَفّ: فتح وإغلاق في نفس المكان، ومن الواضح أنه لكل فتح سيكون هناك إغلاق مقابل. تزيل إعادة البناء أيضًا متغيرًا مشتركًا قبيحًا.

هناك تحسين صغير آخر مهم يمكننا القيام به. في العديد من اللغات الحديثة، يمكنك تحديد مجال عمر المورد إلى كتلة مُغلّفة (enclosed block) من نوع ما. في *روبي*، هناك اختلاف في المِلَفّ open الذي يمر في مرجع المِلَفّ المفتوح إلى الكتلة، كما هو موضح هنا بين الـ do والـ end:

```
def update_customer(transaction_amount)
  File.open(@name + ".rec", "r+") do |file|                  # >--
    read_customer(file)                                      #    |
    @balance = @balance.add(transaction_amount,2)            #    |
    write_customer(file)                                     #    |
  end                                                        # <--
end
```

في هذه الحالة، في نهاية الكتلة، يخرج المتغير file عن المجال ويُغلق المِلَفّ الخارجي. نهائيًا. لا حاجة لتذكر إغلاق المِلَفّ وتحرير المصدر، فمن المضمون حدوث ذلك.

عندما تكون في شك، فإن ذلك يدفع دائمًا لتقليص المجال.

نصيحة ٤١ تصرّف موضعيًا

تداخل التخصيصات Nest Allocations

يمكن توسيع النمط الأساسي لتخصيص الموارد للروتينات التي تحتاج إلى أكثر من مورد واحد في كل مرّة. هناك اقتراحان إضافيان فقط:

- ألغِ تخصيص الموارد بالترتيب المعاكس لذلك الذي تخصصها به. وبهذه الطريقة لن تُيتَّم الموارد إذا كان أحدها يحتوي على مراجع لمورد آخر.
- عند تخصيص نفس مجموعة الموارد في أماكن مختلفة في الشِفرة الخاصة بك، خصّصها دائمًا بنفس الترتيب. سيقلل هذا من احتمال الجمود. (إذا كانت العملية A تطالب بـ resource1 وكانت على وشك المطالبة بـ resource2 ، في الوقت الذي طالبت العملية B بـ resource2 وتحاول الحصول على resource1 ، فستنتظر العمليتان إلى الأبد.)

لا يهم نوع الموارد التي نستخدمها - المناقلات واتصالات الشبكة والذاكرة والمِلَفّات والمسارات (threads) والنوافذ - يُطبق النمط الأساسي: أي شخص يخصص موردًا يجب أن يكون مسؤولاً عن إلغاء تخصيصه. ومع ذلك، يمكننا في بعض اللغات تطوير المفهوم بشكل أكبر.

75 . راجع النصيحة في الصفحة 315.

موازنة عبر الوقت

في هذا الموضوع، ننظر غالبًا إلى الموارد المؤقتة التي تستخدمها عملية التشغيل. ولكن قد ترغب في التفكير في الفوضى الأخرى التي قد تتركها خلفك.

على سبيل المثال، كيف يتم التعامل مع مِلَفّات التسجيل الخاصة بك؟ أنت تنشيء البيانات وتستهلك مساحة التخزين. هل هناك شيء موجود لتدوير السجلات وترتيبها؟ ماذا عن مِلَفّات التصحيح غير الرسمية التي تسقطها؟ إذا كنت تضيف سجلات تسجيل في قاعدة بيانات، فهل هناك عملية مماثلة لإنهاء صلاحيتها؟ لأي شيء تقوم بإنشائه يستهلك موردًا محدودًا، فكّر في كيفيّة موازنة ذلك.

ماذا ستترك وراءك؟

الكائنات والاستثناءات

إن التوازن بين التخصيصات وإلغاء التخصيصات يذكرنا بباني (constructor) وهادم (destructor) الصنف غرضي التوجه. يمثل الصنف (class) موردًا، يمنحك الباني كائنًا محدّدا من هذا النوع من الموارد، ويزيله الهادم من مجالك.

إذا كنت تبرمج بلغة كائنية التوجه، فقد تجد أنه من المفيد تغليف الموارد في أصناف. في كل مرّة تحتاج فيها إلى نوع مورد معين، تنشيء مثيل لكائن من هذا الصنف. عندما يخرج الكائن عن مجاله، أو يستعيده جامع البيانات المهملة (garbage collector)، يقوم هادم الكائن بعد ذلك بإلغاء تخصيص المورد المغلّف.

هذا النهج له فوائد خاصة عندما تعمل بلغات يمكن أن تتداخل فيها الاستثناءات مع تخصيص الموارد.

الموازنة والاستثناءات

يمكن للغات التي تدعم الاستثناءات أن تجعل تخصيص الموارد صعبًا. إذا رُمي استثناء، كيف تضمن أن كل شيء تم تخصيصه قبل الاستثناء مرتب؟ تعتمد الإجابة إلى حدّ ما على دعم اللغة. لديك خياران عمومًا:

1. استخدام مجال متغير (على سبيل المثال، متغيرات المكدس في سي++ أو رست Rust)
2. استخدم عبارة finally في كتلة "try... catch"

اعتمادًا على قواعد تحديد المجال المعتادة بلغات مثل سي++ أو رست، سيتم استعادة ذاكرة المتغير عندما يخرج المتغير عن النطاق عن طريق الإرجاع، أو انتهاء الكتلة، أو الاستثناء. ولكن يمكنك أيضًا ربط هادم المتغير لتنظيف أي موارد خارجية. في هذا المثال، سيغلق متغير رست المسمى accounts المِلَفّ المرتبط تلقائيًا عند خروجه عن المجال:

```
{
let mut accounts = File::open("mydata.txt")?; // >--
// 'accounts'استخدم  // |
... // |
} // <--
//'accounts' هو الآن خارج المجال   والمِلَفّ
// سيُغلق تِلْقائيًا
```

الخِيار الآخر، في حال كانت اللغة تدعمه، هو عبارة finally. ستضمن الفِقْرة finally تشغيل الشِفرة المحددة سواء تم رفع استثناء أم لا في كتلة "try... catch" :

```
try
// بعض الأشياء المراوغة
catch
// رُفع استثناء
finally
// التنظيف في كلا الحالتين
```

بكافة الأحوال catch موجودة.

استثناء النمط المضاد

نرى عادةً الأشخاص يكتبون شيئًا كهذا

```
begin
    thing = allocate_resource()
    process(thing)
finally
    deallocate(thing)
end
```

هل تستطيع رؤية ما الخطأ؟

ماذا يحدث إذا فشل تخصيص الموارد وأثار استثناء؟ سوف تلتقطه عبارة " finally"، وتحاول إلغاء تخصيص شيء لم يتم تخصيصه أصلًا.

النمط الصحيح للتعامل مع تخصيص الموارد في بيئة مع وجود استثناءات هو

```
thing = allocate_resource()
begin
    process(thing)
finally
    deallocate(thing)
end
```

عندما لا تستطيع موازنة الموارد

في بعض الأحيان يكون النمط الأساسي لتخصيص الموارد غير مناسب. عادة ما يوجد هذا في البرامج التي تستخدم بنى البيانات الديناميكية.

يقوم روتين واحد بتخصيص مساحة من الذاكرة وربطها ببنية أكبر، حيث قد تبقى لبعض الوقت.

الحيلة هنا تتمثل في إنشاء ثابت دلالي لتخصيص الذاكرة. تحتاج إلى تحديد من المسؤول عن البيانات في بنية بيانات مجمّعة.

ماذا يحدث عندما تلغي تخصيص بنية المستوى الأعلى؟ لديك ثلاثة خيارات رئيسة:

- بنية المستوى الأعلى مسؤولة أيضًا عن تحرير أي بنى فرعية تحتوي عليها. ثم تحذف هذه البنى البيانات التي تحتوي عليها بشكل متكرر، وهكذا.
- يُلغى تخصيص بنية المستوى الأعلى ببساطة. وأي بنى أشارت هذه البنية إليها (البنى التي لم تتم الإشارة إليها في أي مكان آخر) تُيتَم.
- ترفض بنية المستوى الأعلى إلغاء تخصيص نفسها إذا كانت تحتوي على أي بنى فرعية.

يعتمد الاختيار هنا على ظروف كل بنية بيانات فرديّة. ومع ذلك، تحتاج إلى توضيح ذلك لكل منها، وتنفيذ قرارك باستمرار. يمكن أن يمثّل تنفيذ أي من هذه الخيارات بلغة إجرائية مثل سي مشكلة: بنى البيانات نفسها غير نشطة. تفضيلنا في هذه الظروف هو كتابة وحدة لكل بنية رئيسة توفر تسهيلات قياسيّة للتخصيص وإلغاء التخصيص لهذه البنية. (يمكن أن توفر هذه الوحدة أيضًا تسهيلات مثل طباعة التصحيح، والتسلسل، وإلغاء التسلسل، وخطافات الاجتياز.)

التحقّق التوازن

لأن المبرمجين الواقعيين لا يثقون بأحد، بما في ذلك أنفسنا، فإننا نشعر أنه من الجيد دائمًا بناء شِفرة تتحقق أن الموارد تُحرَر على الوجه المناسب. بالنسبة لمعظم التطبيقات، يعني هذا عادةً إنتاج أغلفة لكل نوع من الموارد، واستخدام هذه الأغلفة لتتبع جميع عمليات التخصيص وإلغاء التخصيص. عند نِقَاط معينة في الشِفرة البرمجية الخاصة بك، سوف يملي منطق البرنامَج أن الموارد ستكون في حالة معينة: استخدم الأغلفة للتحقق من ذلك. على سبيل المثال، من المحتمل أن يكون للبرنامَج طويل الأمد الذي تطلبه الخِدمَات نقطة محددة في الجزء العلوي من حلقة المعالجة الرئيسة حيث ينتظر وصول الطلب التالي. هذا مكان جيد للتأكد أن استخدام الموارد لم يزدد منذ آخر تنفيذ للحلقة.

على مستوى أقل، ولكن ليس أقل فائدة، يمكنك الاستثمار في الأدوات التي (من بين أشياء أخرى) تتحقق برامج التشغيل الخاصة بك للتأكد عدم وجود نقص ذاكرة.

الأقسام ذات الصلة

- الموضوع 24، البرامج الميتة لا تكذب، في الصفحة 133
- الموضوع 30، برمجة التحويل، في الصفحة 169
- الموضوع 33، كسر الاقتران الزمني، صفحة 193

تحدّيات

- مع أنّ عدم وجود طرق مضمونة لضمان تحريرك للموارد دائمًا، ستساعدك بعض تقنيات التصميم في ذلك عند تطبيقها باستمرار. ناقشنا في النص كيف يمكن أن يؤدي إنشاء ثوابت دلاليّة لبنى البيانات الرئيسة إلى توجيه قرارات تخصيص الذاكرة. ضع في اعتبارك كيف يمكن أن يساعد الموضوع 23، التصميم حسب العقد، في الصفحة 125، في تحسين هذه الفكرة.

تمرين 17 (إجابة محتملة في الصفحة 325)

يضبط بعض مطوري سي و سي++ المؤشر على NULL بعد أن يعيدوا تخصيص الذاكرة التي يشير إليها. لماذا تعد هذه فكرة جيّدة؟

تمرين 18 (إجابة محتملة في الصفحة 102)

يضبط بعض مطوري جافا نقطة تعيين متغير كائن إلى NULL بعد الانتهاء من استخدام الكائن. لماذا تعدّ هذه فكرة جيدة؟

الموضوع 27. لا تتجاوز مصابيحك الأماميّة

من الصعب وضع تنبؤات، خاصةً حول المستقبل.

← لورانس "يوجي" *بيرا، بعد المثل الدنماركي*

في وقت متأخر من الليل، ظلام، وأمطار غزيرة. تلتف السيارة الفخمة ذات المقعدين حول المنحنيات الضيّقة للطرق الجبليّة الصغيرة الملتوية، بالكاد تمسك السيارة بحواف المنعطفات. يأتي منعطف حاد وتخطئه السيارة، حيث تنهار بالرغم الدرابزين الفخم وتطير لتتحطم مشتعلةً في أسفل الوادي. تصل شرطة الولاية إلى مكان الحادث، ويهز كبير الضباط رأسه بأسف قائلًا. "لا بد أنه تجاوز مصابيحه الأمامية."

هل كانت السيارة ذات المقعدين تسير أسرع من سرعة الضوء؟ لا، تم تثبيت حدّ السرعة هذا بإحكام. ما أشار إليه الضابط هو قدرة السائق على التوقف أو القيادة في الوقت المناسب استجابة لإضاءة المصباح.

المصابيح الأمامية لها نطاق محدود معين، يعرف باسم مسافة الرمي (*throw distance*). بعد هذه النقطة، يكون انتشار الضوء واسعا جدًا بحيث لا يكون فعالًا. إضافة إلى ذلك، تضيء المصابيح الأمامية خط مستقيم فقط، ولن تضيء أي شيء خارج المحور، كالمنحنيات أو التلال أو الانخفاضات في الطريق. وفقًا للإدارة الوطنية لسلامة المرور على الطرق السريعة، يبلغ متوسط المسافة المُضاءة بواسطة المصابيح الأماميّة منخفضة الشعاع حوالي 160 قدمًا. لسوء الحظ، فإن مسافة التوقف عند سرعة 40 ميل في الساعة هي 189 قدمًا، وفي سرعة 70 ميل في الساعة 464[76] قدمًا.

في تطوير البرمجيات، "المصابيح" لدينا محدودة بشكل مشابه.

لا يمكننا أن نرى زمنا بعيدًا جدًا في المستقبل، وكلما نظرت بعيدًا عن المحور، كلما أصبحت أعتم. لذلك المبرمجين الواقعيين لديهم قاعدة ثابتة:

نصيحة ٤٢	اتخذ خطوات صغيرة- دائما

احرص دائمًا على اتخاذ خطوات صغيرة ومدروسة، والتحقق من التغذية الراجعة والتعديل قبل المتابعة. ضع في اعتبارك أن معدل التغذية الراجعة هو الحدّ الأقصى لسرعتك. أنت لا تتخذ خطوة أو مهمة "كبيرة جدًا".

ماذا نعني بالضبط بالتغذية الراجعة؟ أي شيء يؤكد أو ينفي فعلك بشكل مستقل. فمثلًا:

- توفر النتائج في REPL[77] تغذية راجعة حول فهمك لواجهات برمجة التطبيقات والخوارزميات

76 وفقًا لـ NHTSA، مسافة التوقف = مسافة رد الفعل + مسافة الكبح، بافتراض متوسط وقت رد فعل يبلغ 1.5 ثانية وتباطؤ 17.02 قدم / ثانية^2.

77 **[المترجم]** إن حلقة القراءة – التقييم – الطباعة read–eval–print loop (REPL)، والتي تسمى أيضًا المستوى التفاعلي للمستوى أو اللغة، هي بيئة برمجة تفاعليّة بسيطة للحاسوب تأخذ مدخلات مستخدم واحد (أي التعبيرات الفردية)، وتقيمها (تنفّذ)، وترجع النتيجة للمستخدم ؛ يتم تنفيذ برنامَج مكتوب في بيئة REPL بشكل تدريجي. يستخدم المصطلح عادة للإشارة إلى واجهات برمجة مشابهة للبيئة التفاعلية لآلة Lisp الكلاسيكية. تشمل الأمثلة الشائعة الأصداف الموجودة في سطر الأوامر والبيئات المماثلة للغات البرمجة، ثم أن التقنية مميزة جدًا للغات

- تقدّم اختبارات الوحدة تغذية راجعة حول آخر تغيير لك في الشِفرة
- يوفر العرض التجريبي للمستخدم والمحادثة تغذية راجعة حول الميزات وسهولة الاستخدام

ما هي المهمّة التي تعدّ كبيرةً جدًا؟ أي مهمّة تتطلب "التكهن". فكما أن المصابيح الأماميّة للسيارات محدودة، فيمكننا أن نرى في المستقبل ربما خطوة أو خطوتين، ربما بضع ساعات أو أيام على الأكثر. أبعد من ذلك، يمكنك سريعا تجاوز التخمين المدروس وإلى التكهن غير المنطقي. قد تجد نفسك تنزلق إلى التكهن عندما يجب عليك:

- تقدير تواريخ الانتهاء لأشهر في المستقبل
- تخطيط تصميم للصيانة المستقبلية أو التمديد
- تخمين احتياجات المستخدم المستقبلية
- تخمين توفر التِقَانَة في المستقبل

ولكن، نسمعك تصيح، أليس من المفترض أن نصمم من أجل الصيانة المستقبلية؟ نعم، ولكن فقط إلى حد ما: فقط بمقدار ما يمكنك أن ترى أمامك. كلما كان عليك توقع ما سيبدو عليه المستقبل، زادت المخاطر التي قد تتعرض لها والتي ستكون مخطئًا فيها. بدلاً من إضاعة الجهد في التصميم لمستقبل مجهول، يمكنك دائمًا الرجوع إلى تصميم الشِفرة البرمجية الخاصة بك لتكون قابلة للاستبدال. اجعل من السهل التخلص من الشِفرة واستبدالها بشيء أكثر ملاءمة. سيساعد أيضًا جعل الشِفرة قابلة للاستبدال في التماسك والفصل وعدم التكرار، مما يؤدي إلى تصميم أفضل عمومًا.

مع أنّك قد تشعر بالثقة في المستقبل، هناك دائمًا فرصة لبجعة سوداء في الجوار.

البجعات السوداء

يفترض **نسيم نيكولاس طالب** في كتابه *"البجعة السوداء:"*، أن جميع الأحداث المهمّة في التاريخ أتت من أحداث بارزة يصعب التنبؤ بها ونادرة والتي تقع خارج نطاق عالم التوقعات العادية. هذه القيم المتطرفة، بالرغم أنها نادرة إحصائيًا، إلا أن لها تأثيرات غير متناسبة. إضافة إلى ذلك، تميل تحيزاتنا المعرفيّة الخاصة بنا إلى تعميتنا عن التغييرات التي تزحف إلى حواف عملنا (انظر حَساء الحصى والضفادع المغليّة).

قريبًا من وقت الإصدار الأول من **المبرمج العملي**، احتدم الجدل في مجلات الحاسوب والمنتديات عبر الإنترنت حول السؤال الملح: "من سيفوز في حروب واجهة المستخدم الرسوميّة المكتبيّة، Motif أو OpenLook؟"[78] كان السؤال خاطئًا. من المحتمل أنك لم تسمع أبدًا بهذه التقنيات حيث لم "تقز" أي منهما وسرعان ما سيطرت شبكة الويب التي تركّز على المتصفح على المشهد.

نصيحة ٤٣	تجنّب التكهن

البرمجة النصية.

https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop

78 كانت Motif و OpenLook معايير GUI لمحطات عمل يونكس التي تستند إلى X-Window.

في الغالب الغد يشبه كثيرًا اليوم. لكن لا تبني على ذلك.

الأقسام ذات الصلة

- الموضوع 12، الرصاصات الخطاطة، في الصفحة 73
- الموضوع 13، النماذج الأولية والملاحظات الملحقة، في الصفحة 78
- الموضوع 40، إعادة البناء، في الصفحة 235
- الموضوع 41، اختبر لتكتب الشِفرة، في الصفحة 240
- الموضوع 48، جوهر التطوير الرشيق، في الصفحة 284
- الموضوع 50، جوز الهند لا يفي بالغرض، في الصفحة 296

الفصل الخامس:

انحن، وإلاّ تُكسر

الحياة لا تبقى على حالها. ولا الشِفرة التي نكتبها تستطيع ذلك. نحتاج من أجل مواكبة وتيرة التغيير شبه المحمومة اليوم، إلى بذل كل جهد ممكن لكتابة شِفرة حرّة - ومرنة - قدر الإمكان. وإلا فقد نجد أن الشِفرة الخاصة بنا أصبحت قديمة أو هشّة للغاية بحيث لا يمكن إصلاحها، وربما يتم تركها في نهاية المطاف في الاندفاع الجنوني نحو المستقبل.

بالعودة إلى **قابلية العكس** (Reversibility) كنا قد تحدثنا عن مخاطر القرارات التي لا رجعة فيها. في هذا الفصل، سنخبرك بكيفيّة اتخاذ قرارات قابلة للعكس، حتى تظل شفرتك مرنة وقابلة للتكيف في مواجهة عالم غير مستقر.

أولاً، نلقي نظرة على إلى *الاقتران* — التبعيات بين أجزاء الشِفرة. يوضح **الفصل** كيفيّة الحفاظ على المفاهيم المنفصلة منفصلة، مما يقلل من الاقتران.

بعد ذلك، سنلقي نظرة على تقنيات مختلفة يمكنك استخدامها عند **تلاعب العالم الحقيقي**. سنفحص أربع استراتيجيات مختلفة للمساعدة في إدارة الأحداث والتفاعل معها - جانب مهم من تطبيقات البرمجيات الحديثة.

قد تكون الشِفرة الإجرائية التقليدية والشِفرة كائنية التوجه مرتبطّتان جدًا بأهدافك. في **برمجة التحويل** (Transforming Programming)، سنستفيد من النمط الأكثر وضوحًا والأكثر مرونة الذي توفره مسارات التنفيذ الوظيفيّة (function pipelines)[79]، حتى وإن كانت لغتك لا تدعمها بشكل مباشر.

يمكن أن يجذبك النمط كائني التوجه الشائع إلى فخ آخر. لا تقع فيه، وإلا سوف ينتهي بك الأمر إلى دفع **ضريبة وراثة** ضخمة. سنستكشف أبدال أفضل للحفاظ على شفرتك مرنة وسهلة التغيير.

وبالطبع فإن الطريقة الجيدة للبقاء مرنا- هي كتابة شِفرة *أقل*. يتركك تغيير الشِفرة مُعرضًا لإمكانية إقحام أخطاء جديدة. سيشرح **الضبط** (Configuration) كيفيّة نقل التفاصيل خارج الشِفرة بالكامل، حيث يمكن تغييرها بأمان وسهولة أكبر.

ستساعدك كل هذه التقنيات على كتابة الشفرات التي تنحني ولا تنكسر.

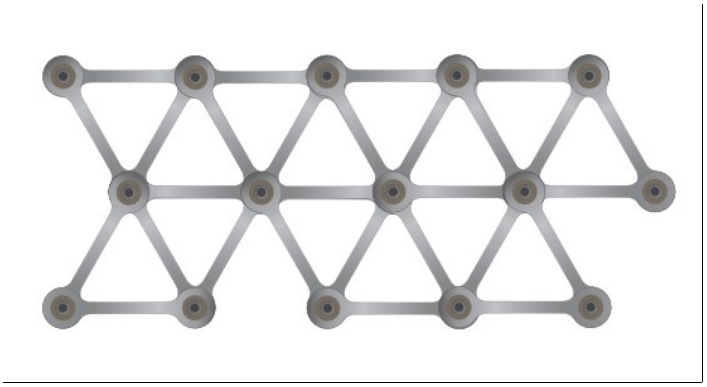
79 **[المترجم]** في هندسة البرمجيات، يتكون مسار التنفيذ أو ما يسمى "خط الأنابيب" من سلسلة من عناصر المعالجة (العمليات، التشعبات، الدوال، إلخ)، مرتبة بحيث يكون خرج كل عنصر هو دخل العنصر التالي ؛ الاسم مستوحى من خط الأنابيب المادي.

الموضوع 28. الفصل

> عندما نحاول اختيار أي شيء بمفرده، نجد أنه مرتبط بكل شيء آخر في الكون.
> ← جون موير، صيفي الأول في سييرا

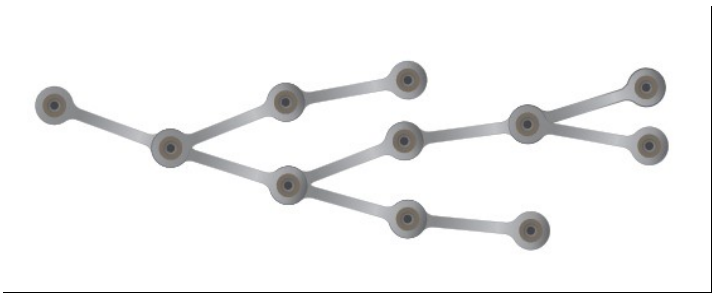
ندّعي في الموضوع 8، جوهر التصميم الجيّد، صفحة 51 أن استخدام مبادئ التصميم الجيّد سيجعل الشِفرة التي تكتبها سهلة التغيير. فالاقتران هو عدو التغيير لأنه يربط الأشياء التي يجب أن تتغير بالتوازي. وهذا يجعل التغيير أكثر صعوبة: إما أن تصرف وقتًا في تتبع جميع الأجزاء التي تحتاج إلى التغيير، أو تصرف وقتًا في التساؤل عن سبب تعطل الأشياء عند تغييرك "شيء واحد فقط" وليس الأشياء الأخرى التي اقترن بها.

عندما تصمم شيئًا تريد أن يكون صلبًا، ربما مثلا جسر أو برج، يمكنك ربط المكونات معًا:



تعمل الروابط معًا لجعل البنية صلبة.

قارن ما سبق بشيء من هذا القبيل:



هنا لا توجد صلابة بنيوية: يمكن أن تتغير الروابط الفردية بينما فقط تتكيف معها المكونات الأخرى.

عندما تصمم الجسور، فأنت تريدهم أن يحتفظوا بأشكالهم؛ أنت بحاجة إليهم ليكونوا صلبين. ولكن عندما تصمم برنامجًا تريد تغييره، فأنت على العكس تمامًا: تريد أن يكون مرنًا. ولكي يكون مرنًا، يجب أن تقترن المكونات الفردية بأقل عدد ممكن من المكونات الأخرى.

ولجعل الأمور أسوأ، يكون الاقتران متعديًا: إذا كان A مقترنًا بـ B و C، وكان B مقترنًا بـ M و N، و C بـ X و Y، فإن A يقترن فعليًا بكل من B، وC، وM، وN، وX وY.

هذا يعني أن هناك مبدأ بسيط يجب عليك اتباعه:

نصيحة ٤٤ | **الشِفرة المفصولة هي أسهل للتغيير**

بالنظر إلى أننا لا نكتب الشِفرة عادةً باستخدام العوارض والمسامير الفولاذية، فماذا يعني فصل الشِفرة؟ في هذا القسم سنتحدث عن:

- حُطَام القطار - سلاسل استدعاءات الطريقة (method)
- العولمة - مخاطر الأشياء الساكنة (static)
- الوراثة - لماذا يعد الصنف الفرعي أمرًا شديد الخَطَر.

إلى حد ما هذه القائمة صوريّة: يمكن أن يحدث الاقتران في أي وقت تقريبًا تشترك فيه شِفرتان في شيء ما، حتى تقرأ ما يلي ابق عينيك على الأنماط الضمنيّة حتى تتمكن من تطبيقها على التعليمات البرمجية الخاصة بك. واحذر من بعض أعراض الاقتران:

- التبعيات الغريبة بين الوحدات أو المكتبات غير ذات الصلة.
- تغييرات "بسيطة" على وحدة واحدة تنتشر بواسطة وحدات غير ذات صلة في النظام أو تكسر الأشياء في مكان آخر من النظام.
- مطوّرو البرامج الذين يخشون تغيير الشِفرة لأنهم غير متيقِّنين مما قد يتأثر.
- الاجتماعات التي تتطلب من الجميع الحضور لأنه لا يوجد أحد متيقِّن مِن مَن سيتأثر بالتغيير.

حطام القطار

لقد رأينا جميعًا شِفرة (وربما كتبنا) مثل هذه:

```
public void applyDiscount(customer, order_id, discount) {
    totals = customer
                .orders
                .find(order_id)
                .getTotals();
    totals.grandTotal = totals.grandTotal - discount;
    totals.discount = discount;
}
```

نأخذ مرجع لبعض الطلبات من الكائن customer، مستخدمين ذلك للعثور على طلب معين، ثم الحصول على مجموعة المبالغ الإجمالية totals لهذا الطلب.

باستخدام هذه المجاميع، نطرح الخَصْم discount من إجمالي الطلب الكلي ثم نقوم أيضًا بتحديثه بعد هذا الخَصْم.

يجتاز هذا الجزء من الشِفرة خمسة مستويات من التجريد، من العميل إلى إجمالي المبالغ. في نهاية المطاف، يجب أن تعرف شِفرة المستوى الأعلى لدينا أن كائن العميل يكشف عن الطلبات orders، وأن الطلبات لديها الطريقة find التي تأخذ معرف الطلب order id وترجع طلبًا order، وأن الكائن order يحتوي على الكائن totals الذي يحتوي على getters و setters للمجموع الكلي والخصومات. إن هذا كثير من المعرفة المضمّنة. لكن الأسوأ من ذلك، أن هناك الكثير من الأشياء التي *لا يمكن تغييرها في المستقبل* إذا كانت هذه الشِفرة ستستمر في العمل. فكما تُجمع جميع السيارات في القطار معًا، تُجمع كذلك جميع الطرائق والسمات في خطّام القطار[80].

دعنا نتخيل أن الشركة قرّرت أنه لن يكون هناك خَصْم للطلب بأكثر من 40%. أين نضع الشِفرة التي تفرض هذه القاعدة؟

يمكنك القول أنها تخصّ الدالة ApplyDiscount التي كتبناها للتو. هذا بالتأكيد جزء من الإجابة. ولكن باستخدام الشِفرة كما هي الآن، لا يمكنك معرفة أن هذه هي الإجابة **الكاملة**. يمكن لأي جزء من الشِفرة، في أي مكان، تعيين الحقول في الكائن totals، وإذا لم ينتبه المشرف على صيانة هذه الشِفرة إلى هذه الملاحظة، فلن يتم التحقق من إظهار السياسة الجديدة.

إحدى الطرق للتعامل مع ذلك هي في التفكير في المسؤوليات. بالتأكيد يجب أن يكون الكائن totals مسؤولًا عن إدارة المجاميع. ومع ذلك، فهو لا يفعل ذلك: إنه في الواقع مجرّد حاوية لمجموعة من الحقول التي يمكن لأي شخص الاستعلام عنها وتحديثها.

يتمثل إصلاح ذلك بواسطة تطبيق شيء ندعوه:

نصيحة ٤٥	أخبر، لا تسأل

ينص هذا المبدأ على أنه لا يجب اتخاذ قرارات بناءً على الحالة الداخليّة لكائن ما ثم تحديث هذا الكائن. إن القيام بذلك يدمّر تمامًا فوائد التغليف، وبواسطة ذلك، تنشر المعرفة الخاصة بالتنفيذ في جميع أنحاء الشِفرة. لذا فإن الإصلاح الأول لحطام القطار لدينا هو تفويض الخَصْم discount إلى كائن الإجمالي:

```
public void applyDiscount(customer, order_id, discount) {
    customer
        .orders
        .find(order_id)
        .getTotals()
        .applyDiscount(discount);
}
```

80 **[المترجم]** كارثة متوقعة لا مفر منها.

لدينا نفس النوع من مشكلة *أخبر لا تسأل* *Tell-don-ask (TDA)* مع كائن العميل والطلبات الخاصة به: لا ينبغي لنا جلب قائمة الطلبات والبحث عنها. بدلاً من ذلك، يجب أن نحصل على الطلب الذي نريده مباشرة من العميل:

```
public void applyDiscount(customer, order_id, discount) {
    customer
        .findOrder(order_id)
        .getTotals()
        .applyDiscount(discount);
}
```

الأمر نفسه ينطبق على كائن الطلب order والمجاميع الخاصة به. لماذا يجب أن يعرف العالم الخارجي أن تنفيذ طلب ما يستخدم كائنًا منفصلاً لتخزين مجاميعه؟

```
public void applyDiscount(customer, order_id, discount) {
    customer
        .findOrder(order_id)
        .applyDiscount(discount);
}
```

وهنا ربما نتوقف.

في هذه المرحلة، قد تفكّر في أن TDA (*أخبر لا تسأل*) ستجعلنا نضيف طريقة ApplyDiscountToOrder (order_id) للعملاء. وإذا اتُبعت بالتزام شديد، فإنها ستنجح.

لكن TDA ليس قانونًا من قوانين الطبيعة؛ إنه مجرّد نمط لمساعدتنا في التعرف على المشكلات. في هذه الحالة، نحن مرتاحون في الكشف عن حقيقة أن العميل لديه طلبات، وأنه يمكننا العثور على أحد هذه الطلبات عن طريق مطالبة العميل بها. هذا قرار عملي.

في كل تطبيق، هناك مفاهيم معيّنة عالية المستوى تكون عالميّة. في هذا التطبيق، تتضمن هذه المفاهيم "العملاء" customers و "الطلبات" orders. لا معنى لإخفاء الطلبات تمامًا داخل كائنات العملاء: لديهم وجود خاص بهم. لذلك ليس لدينا مشكلة في إنشاء واجهات برمجة التطبيقات تكشف كائنات النظام.

قانون ديميتر

غالبًا ما يتحدّث النّاس عن شيء يسمى قانون ديميتر، أو LoD، فيما يتعلق بالاقتران. LoD هي مجموعة من الإرشادات[81] كتبها إيان هولاند أواخر الثمانينيات. قام بإنشائها لمساعدة المطورين في مشروع ديميتر (Demeter) للحفاظ على دوالهم نظيفة ومفصولة أكثر.

يقول LoD أن الطريقة المعرّفة في الصنف C يجب أن تَستدعي فقط:

- طرائق مثيلات أخرى في الصنف C.
- معامِلاتها

81 لذا فهو ليس قانونًا حقًّا. إنه أشبه *بفكرة ديمتر الجيّدة المرحة*.

- طرائق في الكائنات التي تقوم بإنشائها، سواء في المكدس أو في المراكم (heap)
- المتغيرات العالمية "العمومية" (Global)

في الطبعة الأولى من هذا الكتاب، أمضينا بعض الوقت في وصف LoD. في العشرين سنة المتخللة، اختفى بريق هذه الوردة بالذات. لا نحب الآن عبارة "المتغير العمومي" global variable (لأسباب سنتناولها في القسم التالي). اكتشفنا أيضًا أنه من الصعب استخدام هذا في الممارسة العمليّة: إنه يشبه إلى حدّ ما الحاجة إلى تحليل مستند قانوني كلما استدعيت طريقة.

ومع ذلك، فإن المبدأ لا يزال سليمًا. نوصي فقط بطريقة أبسط إلى حدّ ما للتعبير عن الشيء نفسه تقريبًا:

نصيحة ٤٦ | **لا تُداخل استدعاءات الطريقة**

حاول ألا يكون لديك أكثر من "." واحدة عندما ***تريد الوصول إلى شيء ما***. والوصول إلى شيء ما يغطي أيضًا الحالات التي تستخدم فيها متغيرات وسيطة، كما في الشِفرة التالية:

```
# هذا أسلوب سيء جدا
amount = customer.orders.last().totals().amount;
# وهذا أيضا
orders = customer.orders;
last = orders.last();
totals = last.totals();
amount = totals.amount;
```

هناك استثناء كبير لقاعدة النقطة الواحدة: لا تُطبّق القاعدة إذا كانت الأشياء التي تربطها (بالنقطة) من غير المحتمل حقا أن تتغير. من الناحية العمليّة، يجب اعتبار أن أي شيء في تطبيقك من المحتمل أن يتغير. يجب اعتبار أي شيء في مكتبة الطرف الثالث متغيرًا، خاصة إذا كان مشرفو هذه المكتبة معروفين بتغيير واجهات برمجة التطبيقات بتغيير الإصدارات.بالرغم من ذلك فإن المكتبات التي تأتي مع اللغة، ربما تكون مستقرة إلى حدّ ما، لذلك سنكون سعداء بشِفرة مثل:

```
people
.sort_by {|person| person.age }
.first(10)
.map {| person | person.name }
```

عملت شِفرة روبي هذه عندما كتبنا الطبعة الأولى، قبل 20 عامًا، ومن المحتمل أنها ستبقى تعمل عندما ندخل إلى منزل المبرمجين القدامى (في أي يوم الآن ...).

السلاسل ومسارات التنفيذ (Chains and Pipelines)

نتحدث في الموضوع 30، برمجة التحويل، في الصفحة 169 عن إنشاء الدوال في مسارات التنفيذ (pipelines). تقوم مسارات التنفيذ هذه بتحويل البيانات، وتمريرها من دالة إلى أخرى. هذا ليس مشابه لاستدعاءات الطريقة في حُطام القطار، لأننا لا نعتمد على تفاصيل التنفيذ المخفيّة.

هذا لا يعني أن مسارات التنفيذ لا تُدخل بعض الاقتران: إنها تفعل ذلك. يجب أن يكون تنسيق البيانات المُرجعة من قبل إحدى الدوال في مسار التنفيذ متوافقًا مع التنسيق المقبول من قبل الدالة التالية.

تجربتنا هي أن عوائق هذا النموذج من الاقتران أقل بكثير من العوائق التي تحول دون تغيير الشِفرة في النموذج المقدّم في خُطام القطار.

شرور العولمة Globalization

تعد البيانات التي يمكن الوصول إليها عالميًا مصدرًا خبيثًا للاقتران بين مكونات التطبيق. تعمل كل قطعة من البيانات العموميّة كما لو أن **كل** طريقة في تطبيقك كسبت فجأة معاملًا إضافيًا: على كل حال، تلك البيانات العموميّة متاحة داخل كل طريقة.

تقرن البيانات العمومية الشِفرة لأسباب عدّة. الأكثر وضوحا هو أن التغيير في تنفيذ عمومي ما يحتمل أن يؤثر على جميع التعليمات البرمجية في النظام. عمليًا، التأثير محدود نوعًا ما؛ تكمن المشكلة فعليًا في معرفة أنك قد عثرت على ***كل*** الأماكن التي تحتاج إلى تغيير.

تخلق البيانات العالمية أيضًا اقترانًا عندما يتعلق الأمر بفصل الشِفرة الخاصة بك.

حققنا الكثير من فوائد إعادة استخدام الشِفرة. لقد كانت تجربتنا أن إعادة الاستخدام يجب ألا تكون مصدر قلق رئيس عند إنشاء الشِفرة، ولكن التفكير في جعل الشِفرة قابلة لإعادة الاستخدام يجب أن يكون جزءًا من روتين كتابة الشِفرة الخاص بك. عندما تجعل الشِفرة قابلة لإعادة الاستخدام، فإنك تمنحها واجهات نظيفة، وتفصلها عن باقي الشفرات. هذا يسمح لك باستخراج طريقة أو وحدة ما دون سحب أي شيء آخر معها. وإذا كانت شفرتك تستخدم البيانات العالمية، فسيصبح من الصعب فصلها عن الباقي.

سترى هذه المشكلة عند كتابة اختبارات الوحدة للشِفرة التي تستخدم البيانات العالمية. ستجد نفسك تكتب مجموعة من تعليمات الإعداد البرمجية لإنشاء بيئة عالمية فقط لمجرّد السماح بتشغيل الاختبار.

نصيحة ٤٧	تجنّب البيانات العموميّة

تتضمن البيانات العموميّة نمط المفرد Singleton

في القسم السابق كنا حريصين على التحدث عن البيانات العموميّة وليس ***المتغيرات العموميّة***. وذلك لأن الناس غالبًا ما يخبروننا "انظروا! لا متغيرات عموميّة. لقد قمتُ بتغليفها جميعًا كمثيل للبيانات في كائن مفرد (singleton object)[82] أو وحدة عامة (global module)".

82 **[المترجم]** في هندسة البرمجيات، يعد النمط المفرد نمط تصميم برامج يقيد إنشاء مثيل لصنف واحد بمثيل "فردي". مفيد عندما تكون هناك حاجة إلى كائن واحد بالضبط لتنسيق الإجراءات عبر النظام. المصطلح يأتي من المفهوم الرياضي لمفرد بالإنجليزي (singleton).
https://ar.wikipedia.org/wiki/%D8%A7%D9%84%D9%86%D9%85%D8%B7_%D8%A7%D9%84%D9%85%D9%81%D8%B1%D8%AF

حاول مرة أخرى يا سكيبي[83]. إذا كان كل ما لديك هو نمط مفرد (singleton) مع مجموعة من متغيرات الحالة المُصدَّرة، إذا فهي لا تزال بيانات عموميّة فحسب. لكن لديها فقط اسم أطول.

لذلك يأخذ الناس هذا النمط المفرد ويخفون جميع البيانات خلف الطرائق. فبدلاً من كتابة شِفرة الـ Confg.log_level يكتبون الآن Confg.log_level() أو Confg.getLogLevel(). هذا أفضل، لأنه يعني أن بياناتك العموميّة تخفي القليل من الذكاء خلفها. إذا قرّرت تغيير تمثيل مستويات السجل، يمكنك الحفاظ على التوافق بواسطة الاقتران بين الجديد والقديم في ضبط واجهة برمجة التطبيقات Config API. لكن لا يزال لديك فقط مجموعة واحدة من بيانات الضبط.

تتضمن البيانات العالمية موارد خارجية

أي مورد خارجي قابل للتغيير هو بيانات عالمية. إذا كان تطبيقك يستخدم قاعدة بيانات، أو مخزن بيانات، أو نظام مِلَفّات، أو واجهة برمجة تطبيقات للخدمة وما إلى ذلك، فقد يخاطر بالوقوع في فخ العولمة. مرّة أخرى، الحل هو التأكد تغليف هذه الموارد دائمًا خلف الشفرات التي تتحكم فيها.

نصيحة ٤٨ **إذا كان من المهم أن يكون عموميًا، فغلّفه في واجهة برمجة التطبيقات**

الوراثة تضيف اقتران

إن إساءة استخدام التصنيف الفرعي (subclassing)، حيث يرث الصنف الحالة (state) والسلوك من صنف أخرى، أمر مهم للغاية بحيث نناقشه في القسم الخاص به، الموضوع 31، ضريبة الوراثة، في الصفحة 180.

مرّة أخرى، هذا كل شيء عن التغيير

من الصعب تغيير الشِفرة المقترنة: يمكن أن يكون للتغييرات في مكان واحد تأثيرات ثانوية في مكان آخر في الشِفرة، وغالبًا ما تكون في الأماكن التي يصعب العثور عليها والتي لا تظهر إلا بعد شهر من مرحلة الإنتاج.

إبقاء شفرتك خجولة: إن التعامل مع الأشياء التي تعرفها مباشرة، سيساعد في الحفاظ على فصل تطبيقاتك، وهذا سيجعلها أكثر قابلية للتغيير.

الأقسام ذات الصلة

- الموضوع 8، جوهر التصميم الجيّد، صفحة 51
- الموضوع 9، DRY - شرور التكرار، في الصفحة 54
- الموضوع 10، التعامدية، في الصفحة 62

83 **[المترجم]** سكيبي – أحد أول الأفلام الحاصلة على جائزة الأكاديمي.

- الموضوع 11، قابلية العكس، في الصفحة 69
- الموضوع 29، شعوذة العالم الحقيقي، صفحة 159
- الموضوع 30، برمجة التحويل، في الصفحة 169
- الموضوع 31، ضريبة الوراثة، في الصفحة 180
- الموضوع 32، الضبط، في الصفحة 188
- الموضوع 33، كسر الاقتران الزمني، صفحة 193
- الموضوع 34، الحالة المشتركة حالة غير صحيحة، في الصفحة 198
- الموضوع 35، الممثلون والعمليات، في الصفحة 205
- الموضوع 36، السبّورات، صفحة 211
- نناقش Tell، لا تسأل في مقالة بناء البرامج لعام 2003 فن Enbugging[84]

84 https://media.pragprog.com/articles/jan_03_enbug.pdf

الموضوع 29. تلاعب العالم الحقيقي

> الأشياء لا تحدث فحسب، هي وُجدت لتحدث.
>
> ← جون ف.كينيدي

في الماضي، عندما كان مؤلفاك لا يزالان يتمتعان بمظهرهما الصبياني الجيّد، لم تكن أجهزة الحاسوب مرنة للغاية. عادةً ما كنا ننظم طريقة تفاعلنا معها وفقًا لقيودها.

اليوم نتوقع أكثر من ذلك: يجب أن تتكامل أجهزة الحاسوب مع عالمنا، وليس العكس. وعالمنا مضطرب: الأشياء تحدث باستمرار، وتتحرك الأشياء في كل مكان من حولنا، نغيّر عقولنا، ... والتطبيقات التي نكتبها بطريقة أو بأخرى يجب أن تنجح بما يجب عليها القيام به.

يتعلق هذا القسم بكتابة هذه التطبيقات المتجاوبة.

سنبدأ بمفهوم الحدَث (event).

الأحداث

يمثل الحدث توافر المعلومات. قد يأتي الحدث من العالم الخارجي: ينقر المستخدم على زر أو يحدّث أسعار الأسهم. وقد يكون داخليًا: تجهز نتيجة الحساب، ينتهي البحث. يمكن أن يكون أمرًا بسيطًا جدًا مثل جلب العنصر التالي في القائمة.

مهما كان مصدر الحدث، إذا كتبنا التطبيقات التي تستجيب للأحداث، وعدّلنا ما تفعله بناءً على تلك الأحداث، فستعمل هذه التطبيقات بشكل أفضل في العالم الحقيقي. سيجدها المستخدمون أكثر تفاعليّة، وستستخدم التطبيقات بنفسها الموارد بشكل أفضل.

ولكن كيف يمكننا كتابة هذه الأنواع من التطبيقات؟ دون نوع من الاستراتيجية، سنجد أنفسنا مرتبكين سريعًا، وستكون تطبيقاتنا فوضى من التعليمات البرمجية المقترنة بإحكام.

دعونا نلقي نظرة على أربع استراتيجيات مساعدة.

1. آلات الحالة المنتهية(Finite State Machines)
2. نمط المراقب (Observer Pattern)
3. النشر / الاشتراك (Publish/Subscribe)
4. البرمجة التفاعليّة والتدفقات (Streams)

آلات الحالة المنتهية[85]

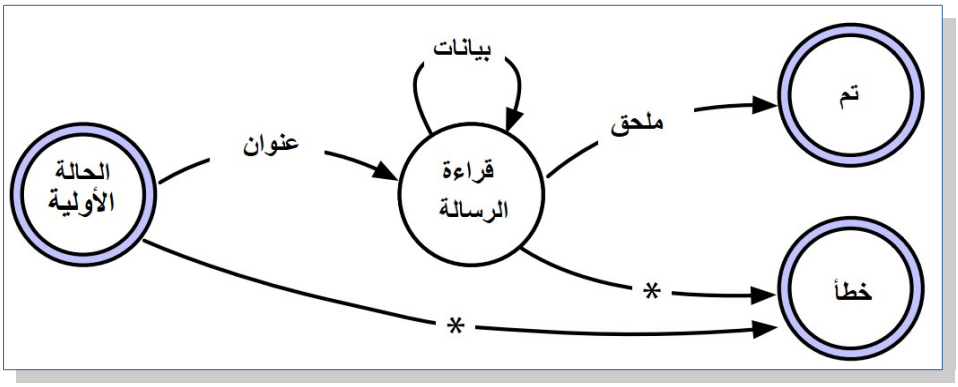
يجد ديف نفسه يكتب الشِفرة باستخدام آلات الحالات المنهية Finite State Machine (FSM) كل أسبوع تقريبًا. في كثير من الأحيان، سيكون تنفيذ FSM مجرّد بضعة أسطر من الشِفرة، ولكن هذه الأسطر القليلة تساعد على تخليصك من الكثير من الفوضى المحتملة.

يُعد استخدام FSM أمرًا سهلاً للغاية، ومع ذلك ينفر الكثير من المطورين منها. يبدو أن هناك اعتقادًا بأنها صعبة، أو أنها تُطبّق فقط إذا كنت تتعامل مع الكيانات المادية (hardware)، أو أنك بحاجة إلى استخدام مكتبة يصعب فهمها. لا شيء من هذا صحيح.

التشريح العملي لـFSM

آلة الحالة هي في الأساس مجرّد مواصَفة لكيفيّة التعامل مع الأحداث. تتكون من مجموعة من الحالات، واحدة منها هي الحالة الراهنة "الحاليّة" (current state). من أجل كل حالة، نسجّل الأحداث (events) المهمة لهذه الحالة. لكل حدث من هذه الأحداث، نحدّد الحالة الراهنة الجديدة للنظام.

على سبيل المثال، قد نتلقى رسائل متعددة الأجزاء من مقبس ويب websocket. الرسالة الأولى هي العنوان "الترويسة" (header). ويتبع ذلك أي عدد من رسائل البيانات، تليها رسالة تذييل. يمكن تمثيل هذا كـ FSM كالتالي:



نبدأ ب "الحالة الأوليّة". إذا تلقينا رسالة العنوان، فإننا ننتقل إلى حالة "قراءة الرسالة". إذا تلقينا أي شيء آخر في أثناء وجودنا في الحالة الأوليّة (السطر المعنون بعلامة النجمة) فإننا ننتقل إلى حالة "الخطأ" وننتهي.

في أثناء وجودنا في حالة "قراءة الرسالة"، يمكننا قَبُول إما رسائل البيانات، وفي هذه الحالة نواصل القراءة في نفس الحالة، أو يمكننا قَبُول رسالة ملحق "تزييل"، والتي تحولنا إلى الحالة "تم". أي شيء آخر يتسبب في الانتقال إلى الحالة "خطأ".

الشيء الأنيق في FSMs هو أنه يمكننا التعبير عنها كبيانات حقّاً. إليك جدول يمثل محلّل رسائلنا:

85 **[المترجم]** آلة الحالات المنتهية (بالإنجليزية: Finite-State Machine اختصاراً FSM) أو ببساطة آلة الحالات هي نموذج حوسبة رياضي يستخدم لتصميم دارات المنطق المتتابع والبرامج الحاسوبية.
https://ar.wikipedia.org/wiki/آلة_ذات_حالات_منتهية

الحالة	الأحداث			
	عنوان	بيانات	ملحق	غير ذلك
أوليّة	قراءة	خطأ	خطأ	خطأ
قراءة	خطأ	قراءة	تم	خطأ

تمثل الصفوف في الجدول الحالات. لمعرفة ما يجب فعله عند وقوع حدث، ابحث في الصف عن الحالة الحاليـة، ثم ابحث عن العمود الذي يمثّل الحدث، ومحتويات تلك الخليّة هي الحالة الجديدة.

الشِفرة التي تعالجها بسيطة بنفس القدر:

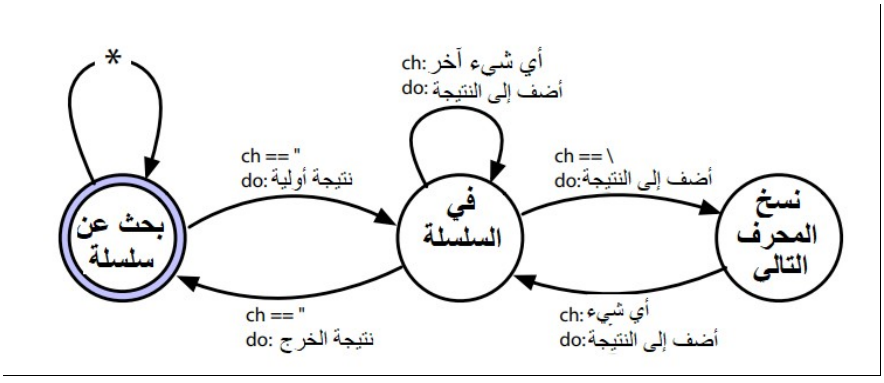
event/simple_fsm.rb
```
Line 1 TRANSITIONS = {
     - initial: {header: :reading},
     - reading: {data: :reading, trailer: :done},
     - }
   5
     - state = :initial
     - while state != :done && state != :error
     - msg = get_next_message()
  10
- end
```

الشِفرة التي تنفذ الانتقالات بين الحالات موجودة في السطر 10. وهي تفهرس جدول الانتقال باستخدام الحالة الراهنة، ثم تفهرس الانتقالات لتلك الحالة مستخدمةً نوع الرسالة. إذا لم تكن هناك حالة جديدة متطابقة، يتم تعيين الحالة إلى **خطأ** ": error".

إضافة الإجراءات Actions

إن FSM النقيّة، كالتي شاهدناها للتو، هي محلل تدفق أحداث. ناتجها الوحيد هو الحالة النهائية. يمكننا تعزيزها عن طريق إضافة الإجراءات التي تُقدَح في بعض الانتقالات.

على سبيل المثال، قد نحتاج إلى استخراج كل السلاسل المحرفيّة في مِلَف مصدر. إن السلسلة النصيّة نص بين علامتي اقتباس، لكن الخط المائل العكسي في سلسلة ما يلغي الحرف التالي، لذا فإن "\"quotes \Ignore"" هي سلسلة مفردة. إليك FSM التي تفعل ذلك:



هذه المرّة، تحتوي كل عملية انتقال (transition) على عنوانين. العنوان في الأعلى هو الحدث الذي يُشغّلها، وفي الأسفل هو الإجراء الذي يجب اتخاذه ونحن نتنقل بين الدول.

سنوضح هذا في جدول، كما فعلنا في المرّة السابقة. ومع ذلك، في هذه الحالة، يكون كل إدخال في الجدول قائمة مكوّنة من عنصرين تحتوي على الحالة التالية واسم الإجراء:

event/strings_fsm.rb
```
TRANSITIONS = {
# الإجراء المتخذ الحالة الجديدة الحالي
#-----------------------------------------------------
  look_for_string: {
     "" =>         [ :in_string,          :start_new_string ],
    :default => [ :look_for_string,  :ignore ],
  },
  in_string: {
      "" =>        [ :look_for_string,  :finish_current_string ],
     '\\' =>       [ :copy_next_char,:add_current_to_string ],
    :default => [ :in_string,          :add_current_to_string ],
  },
  copy_next_char: {
      :default =>[ :in_string,          :add_current_to_string ],
  },
}
```

لقد أضفنا أيضًا إمكانية تحديد انتقال افتراضي، يتم إجراؤه إذا كان الحدث لا يتطابق مع أي انتقالات أخرى لهذه الحالة.

الآن دعونا نلقي نظرة على الشِفرة:

event/strings_fsm.rb
```
state = :look_for_string
result = []
while ch = STDIN.getc
    state, action = TRANSITIONS[state][ch] || TRANSITIONS[state][:default]
    case action
    when :ignore
    when :start_new_string
         result = []
    when :add_current_to_string
       result << ch
    when :finish_current_string
       puts result.join
    end
end
```

هذا مشابه للمثال السابق، من حيث أننا ندور خلال الأحداث (المحارف في الإدخال)، مما يؤدي إلى حدوث انتقالات. لكنها تعمل أكثر من الشِفرة السابقة. نتيجة كل انتقال هي حالة جديدة واسم لإجراء. نستخدم اسم الإجراء لتحديد الشِفرة المراد تشغيلها قبل أن نعود للدوران ضمن الحلقة.

هذا الشِفرة أساسيّة للغاية، ولكنها تنجز المهمة. هناك العديد من الأبدال الأخرى: يمكن لجدول الانتقال استخدام وظائف مجهولة أو مؤشرات دالة للإجراءات (actions)، ويمكنك تغليف الشِفرة التي تنفّذ آلة الحالة في صنف منفصل، مع حالتها الخاصة، وما إلى ذلك.

لا داعٍ للقول أنه يجب عليك معالجة جميع انتقالات الحالة في نفس الوقت. إذا كنت تمضي خلال خطوات تسجيل مستخدم في تطبيقك، فمن المحتمل أن يكون هناك عدد من الانتقالات في أثناء إدخال تفاصيلهم، والتحقّق صحّة بريدهم الإلكتروني، والموافقة على 107 تحذيرات تشريعيّة مختلفة يجب أن تعطيها التطبيقات عبر الإنترنت الآن، وهكذا. إن إبقاء الحالة في التخزين الخارجي، واستخدامها لقيادة آلة الحالة، هي طريقة رائعة للتعامل مع هذا النوع من متطلبات سير العمل.

آلات الحالة هي بداية

لا يستخدم المطورون آلات الحالة، ونودّ أن نشجعك على البحث عن فرص لتطبيقها. لكنها لا تحلّ جميع المشكلات المرتبطة بالأحداث. لذا دعنا ننتقل إلى بعض الطرق الأخرى للنظر في مشكلات التلاعب بالأحداث.

نمط المراقب

في نمط المراقب[86] لدينا مصدر للأحداث، يُدعى الهدف أو "المراقَب" (observable) وقائمة من العملاء، "المراقِبين" (*obse rvers*) المهتمين بهذه الأحداث.

يسجّل المراقِب اهتمامه بالهدف، عادة بتمرير مرجع إلى دالة ليتم استدعاؤها. بعد ذلك، وعند وقوع الحدث، يقوم الهدف (observable) بالمرور على قائمة المراقبين واستدعاء الدالة التي مرّرها كل منهم. يُعطى الحدث كمعامل لهذا الاستدعاء.

إليك مثال بسيط في روبي. يتم استخدام وحدة Terminator لإنهاء التطبيق. قبل أن يفعل ذلك، فإنه يُخطر جميع مراقبيه بأن التطبيق سيُغلق[87] . قد يستخدمون هذا الإخطار لترتيب الموارد المؤقتة، وتخصيص البيانات، وما إلى ذلك:

event/observer.rb
```
module Terminator
    CALLBACKS = []
    def self.register(callback)
      CALLBACKS << callback
    end
    def self.exit(exit_status)
      CALLBACKS.each { |callback| callback.(exit_status) }
      exit!(exit_status)
    end
end
```

86 **[المترجم]** نمط المراقب (بالإنجليزية: Observer Pattern) هو أحد أنماط التصميم التي يدير فيها كائن يدعى باسم الهدف قائمة من الكائنات الأخرى التي تعتمد عليه وتسمى هذه الكائنات بالمراقبين، حيث ينبه هذه الكائنات عند حدوث تغيير في حالته والذي عادةً ما يحصل لدى استدعاء أحد الطرق الخاصة به.
المراقب_(نموذج_تصميم) /https://ar.wikipedia.org/wiki

87 نعم، نحن نعلم أن روبي لديها فعلًا هذه الإمكانية بواسطة دالتها at_exit

```
Terminator.register(-> (status) { puts "callback 1 sees #{status}" })
Terminator.register(-> (status) { puts "callback 2 sees #{status}" })
Terminator.exit(99)
$ ruby event/observer.rb
callback 1 sees 99
callback 2 sees 99
```

لا يتضمن إنشاء هدف (observable) الكثير من التعليمات البرمجية: يمكنك دفع مرجع دالة إلى قائمة (list)، ثم استدعاء تلك الدوال عند وقوع الحدث. هذا مثال جيد على عدم استخدام مكتبة.

تم استخدام نمط مراقب/هدف لعقود من الزمن، وقد خدمنا جيدًا. وهو منتشر بشكل خاص في أنظمة واجهة المستخدم، حيث تُستخدم الاستدعاءات المؤخّرة "الردود" (callbacks) لإبلاغ التطبيق بحدوث تفاعل ما.

لكن لدى نمط المراقب مشكلة: نظرًا لأن كل مراقب يجب عليه التسجيل مع الهدف، فإنه يُدخل اقترانًا. إضافة إلى ذلك، لأنه في التنفيذ النموذجي، يتم التعامل مع الاستدعاءات المؤخرة (callbacks) مضمّنة بواسطة الهدف، بشكل متزامن، مما قد يؤدي إلى اختناقات الأداء.

يتم حل هذه المشكلة بواسطة الاستراتيجية التالية، النشر / الاشتراك.

النشر/ الاشتراك

يعمل النشر / الاشتراك (Publish/Subscribe) اختصارًا (pubsub) على تعميم نمط المراقب، وفي نفس الوقت على حل مشكلات الاقتران والأداء.

في نموذج pubsub، لدينا *ناشرين ومشتركين*. وهم متصلين عبر قنوات (channels). تُنفَّذ القنوات في جزء منفصل من الشِفرة: في بعض الأحيان مكتبة وأحيانًا عمليّة وأحيانًا بنية تحتيّة موزّعة. تُخفى كل تفاصيل التنفيذ هذه عن الشِفرة الخاصة بك.

كل قناة لها اسم. يسجل **المشتركون** اهتمامًا بواحدة أو أكثر من هذه القنوات المحددة، ويكتب **الناشرون** الأحداث إليهم. وعلى ضد نمط المراقب، تتم معالجة الاتصال بين الناشر والمشترك خارج شفرتك، ومن المحتمل أن يكون غير متزامن.

مع أنّه يمكنك تنفيذ نظام pubsub مبدئي جدًا بنفسك، فربما لا تريد ذلك. لدى معظم مزودي الخِدْمَات السحابيّة عروض pubsub، مما يسمح لك بربط التطبيقات حول العالم. ستحتوي كل لغة شائعة على مكتبة pubsub واحدة في الأقل.

إن Pubsub هي تقنية جيّدة لفصل معالجة الأحداث غير المتزامنة. تسمح بإضافة شِفرة واستبدالها، ربما في أثناء تشغيل التطبيق، دون تغيير الشِفرة الموجودة. الجانب السلبي هو أنه قد يكون من الصعب رؤية ما يجري في نظام يستخدم pubsub بقدر كبير: لا يمكنك إلقاء نظرة على ناشر ومعرفة المشتركين المعنيين برسالة معيّنة على الفور.

بالمقارنة مع نمط المراقب، يعد pubsub مثالًا رائعًا على تقليل الاقتران عن طريق التجريد بواسطة واجهة مشتركة (القناة). ومع ذلك، فإنه لا يزال أساسًا مجرّد نظام لتمرير الرسائل. لكن إنشاء أنظمة تستجيب لمجموعات الأحداث سيتطلب أكثر من ذلك، لذا فلنلق نظرة على الطرق التي يمكننا بواسطتها إضافة بُعد زمني لمعالجة الأحداث.

البرمجة التفاعليّة والتدفقات والأحداث

إذا سبق لك استخدام جدول بيانات إلكتروني (spreadsheet)، فستكون على دراية بالبرمجة التفاعليّة. إذا كانت الخلية تحتوي على صيغة تشير إلى خلية ثانية، فإن تحديث هذه الخلية الثانية سيؤدي إلى تحديث الخلية الأولى أيضًا. تستجيب القيم تبعًا تغير القيم التي تستخدمها.

هناك العديد من أطر العمل التي يمكن أن تساعد في هذا النوع من التفاعل على مستوى البيانات: في مجال المتصفح فإن React و Vue.js هما المفضلين حاليا (ولكن، نظرًا لكونهما JavaScript، ستكون هذه المعلومات قديمة حتى قبل أن يصبح هذا الكتاب مطبوعًا).

من الواضح أنه يمكن أيضًا استخدام الأحداث لإثارة ردود الفعل في الشِفرة البرمجية، ولكن ليس من السهل بالضرورة استكشافها. وهنا تأتي التدفقات (streams).

تتيح لنا التدفقات التعامل مع الأحداث كما لو كانت مجموعة من البيانات. يبدو الأمر كما لو كان لدينا قائمة بالأحداث، والتي تصبح أطول عند وصول أحداث جديدة. جمالية ذلك هي في أنه بإمكاننا التعامل مع التدفقات تمامًا مثل أي مجموعة أخرى: يمكننا المعالجة والدمج والتصفية والقيام بجميع الأشياء الأخرى التي نعرفها جيدًا. يمكننا حتى الجمع بين تدفقات الأحداث والمجموعات العادية. ويمكن أن تكون التدفقات غير متزامنة، مما يعني أن شفرتك تحصل فرصة الاستجابة للأحداث عند وصولها.

يتم تعريف الخط الأساسي الحالي *الفعلي* للتعامل مع الأحداث التفاعلية على الموقع الإلكتروني http://reactivex.io، الذي يحدّد مجموعة من المبادئ اللغويّة ويوثّق بعض عمليات التنفيذ الشائعة. هنا سنستخدم مكتبة RxJs لجافا سكريبت.

يأخذ المثال الأول تدفقين ويضغطهما معًا: النتيجة هي تدفق جديد حيث يحتوي كل عنصر فيه على عنصر واحد من تدفق الإدخال الأول وعنصر واحد من التدفق الآخر. في مثالنا هذا، الدفق الأول هو ببساطة قائمة بأسماء الحيوانات الخمسة. التدفق الثاني أكثر إثارة للاهتمام: إنه مؤقت الفاصل الزمني الذي يُولّد حدثًا كل 500 مللي ثانية. نظرًا لأن التدفقين مضغوطين معًا، لا يتم توليد نتيجة إلا عندما تتوفر البيانات في كليهما، لذا فإنّ التدفق الناتج الخاص بنا لا يصدر قيمة سوى كل نصف ثانية.

event/rx0/index.js
```
import * as Observable from 'rxjs'
import { logValues } from "../rxcommon/logger.js"
let animals = Observable.of("ant", "bee", "cat", "dog", "elk")
let ticker = Observable.interval(500)
let combined = Observable.zip(animals, ticker)
combined.subscribe(next => logValues(JSON.stringify(next)))
```

تستخدم هذا الشِفرة دالّة تسجيل بسيطة[88] تضيف عناصر إلى قائمة في نافذة المتصفح. يُحدّد طابع زمني لكل عنصر بالمللي ثانية منذ بَدْء تشغيل البرنامَج. إليك ما يظهر من أجل شفرتنا:

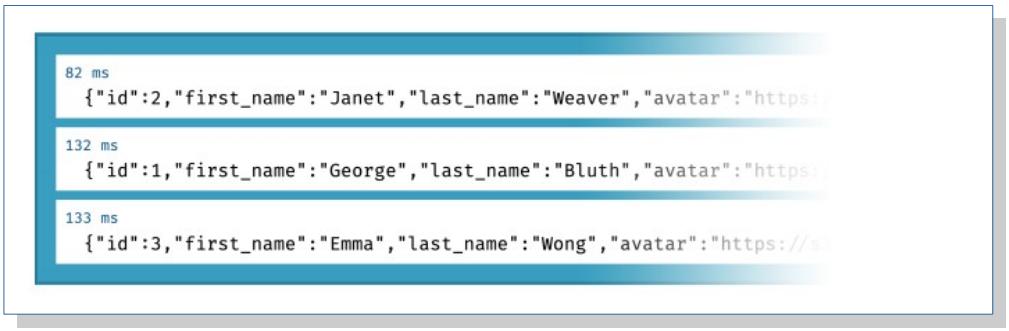
88 https://media.pragprog.com/titles/tpp20/code/event/rxcommon/logger.js

لاحظ الطوابع الزمنيّة: نحصل على حدث واحد من الدفق (stream) كل 500 مللي ثانية. يحتوي كل حدث على رَقَم تسلسلي (تم إنشاؤه بواسطة الفاصل الزمني (interval) الخاص بالهدف observable) واسم الحيوان التالي من القائمة. وعند مشاهدته مباشرة في متصفح، تظهر أسطر السجل كل نصف ثانية.

عادة ما يتم ملء تدفقات الأحداث عند وقوع الأحداث، مما يعني أن الأهداف (observables) التي تملؤها يمكن أن تعمل بالتوازي. فيما يلي مثال لجلب معلومات حول المستخدمين من موقع بعيد. سنستخدم من أجل ذلك https://reqres.in، وهو موقع عام يوفر واجهة REST مفتوحة. كجزء من واجهة برمجة التطبيقات الخاصة بها، يمكننا جلب البيانات الخاصّة بمستخدم (مزيّف) معين عن طريق إجراء طلب GET لـ «users/id» . تجلب شفرتنا المستخدمين أصحاب المعرّفات (Id) ذوات الأرقام 3 و 2 و 1:

event/rx1/index.js
```
import * as Observable from 'rxjs'
import { mergeMap } from 'rxjs/operators'
import { ajax } from 'rxjs/ajax'
import { logValues } from "../rxcommon/logger.js"
let users = Observable.of(3, 2, 1)
let result = users.pipe(
mergeMap((user) => ajax.getJSON(`https://reqres.in/api/users/${user}`))
)
result.subscribe(
resp => logValues(JSON.stringify(resp.data)),
err => console.error(JSON.stringify(err))
)
```

التفاصيل الداخلية للشِفرة ليست مهمة كثيرًا. الأمر المثير هو النتيجة الموضحة في لقطة الشاشة التالية:



انظر إلى الطوابع الزمنية: تمت معالجة الطلبات الثلاثة، أو ثلاثة تـدفقات منفصـلة، بـالتوازي، الطلب ذو المعـرف 2 هـو أول من تم إعادته استغرق 28 ميللي ثانية للعودة، وأعيد الطلبان التاليان بعد 50 و 51 مللي ثانية.

تدفقات الأحداث هي مجموعات غير متزامنة

في المثال السابق، كانت قائمة معرفات المستخدمين لدينا (في مستخدمي observable) ثابتـة. ولكن يجب ألا تكـون كـذلك. ربما نريد جمع هذه المعلومات عندما يُسجّل الأشخاص الدخول إلى موقعنا. كل ما علينا فعله هو إنشاء حـدث هـدف (observable) يحتوي على المعرف الخاص بالمستخدم عند إنشائه لجلسة العمل، واستخدام هذه الهدف بدلاً من الهدف الثابت (statc). سنجلب بعد ذلك تفاصيل عن المستخدمين في أثناء استلامنا لهذه المعرّفات، ومن المفترض تخزينها في مكان ما.

هذا تجريد قويّ للغاية: لم نعد بحاجـة إلى التفكـير في الـوقت على أنـه شـيء علينـا إدارتـه. تعمـل تـدفقات الأحـداث على توحيـد المعالجة المتزامنة وغير المتزامنة وراء واجهة برمجة تطبيقات مشتركة وملائمة.

الأحداث في كل مكان الأحداث في كل مكان. بعضـها واضـح: نقـرة زر، انتهـاء صَـلاحِيَة المـؤقت. البعض الآخـر أقـل وضـوحًا من ذلـك: شخص يسجل الدخول، سطر في مِلَفّ يطابق نمطًا. ولكن، بغض النظر عن مصدرها، يمكن أن تكون الشِـفرة الـتي يـدور حـول الأحـداث أكثر تجاوبًا ومفصولةً بشكل أفضل من نظيرتها الخطيّة.

الأقسام ذات الصلة

- الموضوع 28، الفصل، في الصفحة 151
- الموضوع 36، السبّورات، صفحة 211

تمارين

تمرين 19 (إجابة محتملة في الصفحة 105)

ذكرنا في قسم FSM أنه يمكنك نقل التنفيذ العام لآلة الحالة إلى صنفها الخاص. من المحتمل أن يُهيّئ ذاك الصنف بتمرير جدول الانتقالات والحالة الأوليّة. حاول تنفيذ مستخرج السلسلة بهذه الطريقة.

تمرين 20 (إجابة محتملة في الصفحة 326)

أي من هذه التقنيات (ربما جميعها) ستكون جيّدة للحالات التالية:

- إذا تلقيت ثلاثة أحداث تعطّل لواجهة الشبكة في غضون خمس دقائق، أخبر موظفي العمليات.
- إذا كان الوقت بعد غروب الشمس، وتم اكتشاف حركة في أسفل الدرج تليها حركة اكتُشفت في أعلى الدرج، شغل أضواء الطابق العلوي.
- تريد إخطار أنظمة إعداد التقارير المختلفة بإتمام الطلب.
- من أجل تحديد ما إذا كان العميل مؤهلاً للحصول على قرض سيارة، يحتاج التطبيق إلى إرسال الطلبات إلى ثلاث خِدمَات خلفية وانتظار الردود.

الموضوع 30. برمجة التحويل

إذا لم تتمكن من وصف ما تفعله كعملية، فأنت لا تعرف ما لذي تفعله.

← دبليو إدواردز ديمينج، (attr)

تقوم جميع البرامج بتحويل البيانات وتحويل المدخلات إلى مخرجات. ومع ذلك، عندما نفكّر في التصميم، نادرًا ما نفكّر في إنشاء التحويلات. بدلاً من ذلك، نكون قلقون بشأن الأصناف والوحدات، وبنى البيانات والخوارزميات واللغات وأطر العمل.

نعتقد أن هذا التركيز على الشِفرة غالبًا ما يخطئ الهدف: نحن بحاجة إلى العودة إلى التفكير في البرامج على أنها شيء يحوّل المدخلات إلى مخرجات. عندما نفكر بهذه الطريقة، تتلاشي الكثير من التفاصيل التي كنا قلقين حيالها في السابق. تصبح البنية أكثر وضوحًا، ومعالجة الخطأ أكثر اتساقًا، وينخفض الاقتران.

لبَدْء الاستقصاء، فلنعد بـ آلة الزمن إلى السبعينيات ونطلب من مبرمج يونكس أن يكتب لنا برنامجًا يسرد أطول خمسة مِلَفّات في شجرة الدليل (directory tree)، حيث تعني كلمة أطول "وجود أكبر عدد من الأسطر".
قد تتوقع منهم الوصول إلى محرّر ما وبَدْء الكتابة باستخدام لغة سي. ولكنهم لن يفعلوا ذلك، لأنهم يفكرون في ذلك من حيث ما لدينا (شجرة دليل) وما نريد (قائمة المِلَفّات). ثم يذهبون إلى طرفيّة ما ويكتبون شيئًا مثل:

```
$ find . -type f | xargs wc -l | sort -n | tail -5
```

هذه سلسلة من التحويلات:

find . -type f

اطبع قائمة بجميع المِلَفّات (type f-) في أو تحت الدليل الحالي (.) للخرج القياسي.

xargs wc -l

يقرأ الأسطر من المدخلات القياسية ويرتّبها لتمريرها جميعًا كوسطاء (arguments) للأمر wc -l. يحسب برنامَج wc مع الخِيار l - عدد الأسطر في كل وسطائه ويطبع كل نتيجة على أنها " count filename" إلى الخرج القياسي.

sort -n

يفرز المدخلات القياسية بافتراض أن كل سطر يبدأ برَقَم (n-)، ويطبع النتيجة إلى الخرج القياسي.

tail -5

يقرأ المدخلات القياسيّة ويطبع فقط الأسطر الخمسة الأخيرة إلى الخرج القياسي.

نفّذ هذا في دليل كتابنا وسنحصل على

```
470 ./test_to_build.pml
```

```
 487 ./dbc.pml
 719 ./domain_languages.pml
 727 ./dry.pml
9561 total
```

هذا السطر الأخير هو إجمالي عدد الأسطر في جميع المِلَفّات (وليس فقط تلك المعروضة)، لأن هذا هو ما يفعله wc. يمكننا تجريده بواسطة طلب سطر إضافي من tail، ثم تجاهل الخط الأخير:

```
$ find . -type f | xargs wc -l | sort -n | tail -6 | head -5
      470 ./debug.pml
      470 ./test_to_build.pml
      487 ./dbc.pml
      719 ./domain_languages.pml
      727 ./dry.pml
```

دعنا ننظر إلى ذلك من حيث البيانات التي تبرز بين الخطوات الفردية. يصبح متطلبنا الأصلي، " أعلى 5 مِلَفّات من حيث عدد الأسطر"، سلسلة من التحولات (كما يظهر في الشكل التالي).

← اسم الدليل

← قائمة بالمِلَفّات

← قائمة بأعداد الأسطر

← قائمة مرتبة

← أعلى خمسة + المجموع

← أعلى خمسة

الشكل 1 - مسار التنفيذ find كسلسلة من التحولات

يكاد يكون مثل خط التجميع الصناعي: غذِ البيانات الخام في أحد أطرافه وسيخرج المنتج النهائي (المعلومات) من الطرف الآخر.

ونوّد أن نفكّر في جميع الشفرات بهذه الطريقة.

نصيحة ٤٩ البرمجة تتعلّق بالشِفرة، لكن البرامج تتعلق بالبيانات

إيجاد التحويلات

في بعض الأحيان تكون أسهل طريقة للعثور على التحويلات هي البَدْء بالمتطلبات وتحديد مدخلاتها ومخرجاتها. الآن حدّدتَ الدالّة التي تمثل البرنامَج العام. يمكنك بعد ذلك وِجدان الخطوات التي تقودك من الدخل إلى الخرج. هذا هو نهج *من أعلى إلى أسفل*.

تريد على سبيل المثال إنشاء موقع ويب للأشخاص الذين يلعبون ألعاب الكلمات والتي تعثر على جميع الكلمات التي يمكن تكوينها من مجموعة من الأحرف. دخلك هنا مجموعة من الأحرف، وخرجك قائمة بالكلمات المكونة من ثلاثة أحرف والكلمات المكونة من أربعة أحرف وهكذا:

"lvyin" ← تحوّل إلى

3 => ivy, lin, nil, yin
4 => inly, liny, viny
5 => vinyl

(نعم، كلها كلمات، في الأقل وفقًا لقاموس *ماك او لإس* macOS.)

الحيلة وراء التطبيق العام بسيطة: لدينا قاموس يجمع الكلمات حسب التوقيع، يتم اختياره بحيث يكون لجميع الكلمات التي تحتوي على نفس الحروف نفس التوقيع. أبسط دالة توقيع هي مجرّد قائمة مرتبة من الأحرف في الكلمة. يمكننا بعد ذلك البحث عن سلسلة مُدخلة عن طريق إنشاء توقيع لها، ثم رؤية الكلمات في القاموس (إن وجدت) التي لها نفس التوقيع.

وعلى هذا ينقسم مكتشف *الجناس* " لُعبَة ترتيب الأحرف" إلى أربعة تحويلات منفصلة:

الخطوة	التحويل	بيانات العيّنة
الخطوة 0:	الدخل الأولي	"ylvin"
الخطوة 1:	كل المجموعات من ثلاثة أحرف أو أكثر	vin, viy, vil, vny, vnl, vyl, iny, inl, iyl, nyl, viny, vinl, viyl, vnyl, inyl, vinyl
الخطوة 2:	تواقيع المجموعات	inv, ivy, ilv, nvy, lnv, lvy, iny, iln, ily, lny, invy, ilnv, ilvy, lnvy, ilny, ilnvy
الخطوة 3:	قائمة بجميع كلمات القاموس التي تتطابق مع أي من التواقيع	ivy, yin, nil, lin, viny, liny, inly, vinyl
الخطوة 4:	الكلمات مجمعة حسب الطول	3 => ivy, lin, nil, yin 4 => inly, liny, viny 5 => vinyl 3 => ivy, lin, nil, yin

تخفيض التحويلات على طول الطريق

دعنا نبدأ بالنظر إلى الخطوة 1، التي تأخذ كلمة وتنشئ قائمة بجميع المجموعات المكونة من ثلاثة أحرف أو أكثر. يمكن التعبير عن هذه الخطوة نفسها كقائمة من التحويلات:

الخطوة	التحويل	بيانات العيّنة
الخطوة 1.0:	الدخل الأولي	"vinyl"
الخطوة 1.1:	التحويل إلى محارف	v, i, n, y, l
الخطوة 1.2:	الحصول على المجموعات الفرعيّة	[], [v], [i], ... [v,i], [v,n], [v,y], ... [v,i,n], [v,i,y], ... [v,n,y,l], [i,n,y,l], [v,i,n,y,l]
الخطوة 1.3:	فقط تلك المكونة من أكثر من 3 أحرف	[v,i,n], [v,i,y], ... [i,n,y,l], [v,i,n,y,l]
الخطوة 1.4:	إعادة التحويل إلى سلسلة	[vin,viy, ... inyl,vinyl]

لقد وصلنا الآن إلى النقطة التي يمكننا فيها بسهولة تنفيذ كل تحويل في الشِفرة (نستخدم *إكسير* في هذه الحالة):

```
function-pipelines/anagrams/lib/anagrams.ex
defp all_subsets_longer_than_three_characters(word) do
  word
  |> String.codepoints()
  |> Comb.subsets()
  |> Stream.filter(fn subset -> length(subset) >= 3 end)
  |> Stream.map(&List.to_string(&1))
end
```

ماذا عن عامل التشغيل <| ؟

تحتوي *الإكسير*، إلى جانب العديد من اللغات الوظيفيّة الأخرى، على مشغّل مسار تنفيذ (pipeline operator)، يُطلق عليه أحيانًا أنبوب أمامي أو مجرّد أنبوب[89]. كل ما يفعله هو أخذ القيمة على يساره وإدخالها كأول معامل للدالّة على يمينه، لذلك فأن

```
"vinyl" |> String.codepoints |> Comb.subsets()
```

هي نفس الكتابة التالية

```
Comb.subsets(String.codepoints("vinyl"))
```

(قد تحقن لغات أخرى قيمة مسار التنفيذ هذه كمعامل أخير للدالّة التالية – فهي تعتمد إلى حدّ بعيد على نمط المكتبات المدمجة.)

ربما تعتقد أن هذا مجرّد تبسيطات نحويّة. ولكن بطريقة حقيقيّة للغاية فإن مشغّل مسارات التنفيذ يمثّل فرصة ثوريّة للتفكير بشكل مختلف. استخدام مسار التنفيذ يعني أنك تفكّر تلقائيًا في تحويل البيانات؛ في كل مرّة ترى المعامل <| فانت ترى فعليًا مكانًا تتدفق فيه البيانات بين تحويل وآخر.

89 يبدو أن أول استخدام للمحرف <| كمسار تنفيذ يعود إلى عام 1994، في مناقشة حول لغة Isabelle / ML، أرشفة في https://blogs.msdn.microsoft.com/dsyme/2011/05/17/archeological-semiotics-the-birth-of-the-pipeline-symbol-1994

تحتوي العديد من اللغات على شيء مشابه: Elm و F لديهما <|، و كلوجور (Clojure) لديها <- و <<- (التي تعمل بشكل مختلف قليلاً)، و R لديها %>%. لدى هاسكل كلا من مشغِلي مسارات التنفيذ وتجعل من السهل الإعلان عن مشغلات جديدة. بينما نكتب هذا، هناك حديث عن إضافة معامل <| إلى جافا سكريبت.

إذا كانت لغتك الحاليّة تدعم شيئًا مشابهًا، فأنت محظوظ. إذا لم تكن كذلك، راجع اللغة X لا تحتوي على مسارات تنفيذ، في الصفحة 315.

بكافة الأحوال، عُد إلى الشِفرة.

استمر في التحويل ...

الآن انظر إلى الخطوة 2 من البرنامَج الرئيس، حيث نُحوّل المجموعات الفرعيّة إلى تواقيع. مرّة أخرى، إنه تحويل بسيط- تصبح قائمة المجموعات الفرعيّة قائمة بالتواقيع:

الخطوة	التحويل	بيانات العيّنة
الخطوة 2.0:	الدخل الأولي	vin, viy, ... inyl, vinyl
الخطوة 2.1:	التحويل إلى تواقيع	inv, ivy ... ilny, inlvy

شِفرة إكسير في القائمة التالية بنفس البساطة:

function-pipelines/anagrams/lib/anagrams.ex
```
defp as_unique_signatures(subsets) do
  subsets
  |> Stream.map(&Dictionary.signature_of/1)
end
```

نحوّل الآن قائمة التواقيع هذه: يتم تعيين كل توقيع لقائمة الكلمات المعروفة [وفقا للقاموس] ذات التوقيع نفسه، أو إلى (صفر) إذا لم تكن هناك مثل هذه الكلمات. علينا بعد ذلك أن نزيل الأصفار وأن نسوّي القوائم المتداخلة في مستوى واحد:

function-pipelines/anagrams/lib/anagrams.ex
```
defp find_in_dictionary(signatures) do
  signatures
  |> Stream.map(&Dictionary.lookup_by_signature/1)
  |> Stream.reject(&is_nil/1)
  |> Stream.concat(&(&1))
end
```

الخطوة 4، تجميع الكلمات حسب الطول، هي تحويل بسيط آخر، تحويل قائمتنا إلى خريطة (map) حيث تكون المفاتيح هي الأطوال، والقيم كلها كلمات بتلك الأطوال:

function-pipelines/anagrams/lib/anagrams.ex
```
defp group_by_length(words) do
  words
```

```
  |> Enum.sort()
  |> Enum.group_by(&String.length/1)
end
```

وضع كل شيء معًا

لقد كتبنا كل التحويلات الفرديّة. حان الوقت الآن لربطهم معًا في دالتنا الرئيسة:

```
function-pipelines/anagrams/lib/anagrams.ex
def anagrams_in(word) do
  word
  |> all_subsets_longer_than_three_characters()
  |> as_unique_signatures()
  |> find_in_dictionary()
  |> group_by_length()
end
```

هل تعمل؟ فلنجرّبها:

```
iex(1)> Anagrams.anagrams_in "lyvin"
%{
    3 => ["ivy", "lin", "nil", "yin"],
    4 => ["inly", "liny", "viny"],
    5 => ["vinyl"]
}
```

اللغة X لا تدعم مسارات التنفيذ (Pipelines)

مسارات التنفيذ موجودة منذ مدّة طويلة، ولكن فقط في اللغات المتخصّصة. لقد انتقلوا فقط إلى الاتجاه السائد مؤخرًا، ولا تزال العديد من اللغات الشائعة لا تدعم هذا المفهوم.

الخبر السار هو أن التفكير في التحويلات لا يتطلب بنية لغويّة معينة: إنها أقرب لأن تكون فلسفة تصميم. ما زلت تبني الشِفرة الخاصة بك كتحويلات، لكنك تكتبها كسلسلة من المهام:

```
const content = File.read(file_name);
const lines = find_matching_lines(content, pattern)
const result = truncate_lines(lines)
```

إنها مملّة بعض الشيء، لكنها تنجز المطلوب.

لماذا يُعدّ هذا عظيمًا جدًا؟

دعونا نلقي نظرة على جسم الدالة الرئيسة مرّة أخرى:

```
word
|> all_subsets_longer_than_three_characters()
|> as_unique_signatures()
|> find_in_dictionary()
|> group_by_length()
```

إنها ببساطة سلسلة من التحويلات المطلوبة لتلبية متطلباتنا، كل منها يأخذ المدخلات من التحويل السابق ويمرّر الناتج إلى التحويل التالي. يأتي هذا أقرب ما يمكن إلى التعليمات البرمجية للقراءة والكتابة.

ولكن هناك شيء أعمق أيضًا. إذا كانت الخلفية الخاصّة بك هي برمجة كائنيّة التوجه، فعندئذٍ تتطلب منك ردود أفعالك إخفاء البيانات وتغليفها داخل الكائنات. ثم تتجاذب هذه الأشياء ذهابًا وإيابًا، مغيّرةً حالة بعضها البعض. يقدّم هذا الكثير من الاقتران، وهو سبب كبير يجعل من الأنظمة غرضية التوجه صعبة التغيير.

نصيحة ٥٠ لا تحتفظ بالحالة، مرّرها

في النموذج التحويلي، نقلب ذلك رأساً على عقب. فبدلًا من مجموعات صغيرة من البيانات المنتشرة في جميع أنحاء النظام، فكّر في البيانات كنهرٍ قويّ، كتدفق. تصبح البيانات نظيرًا للوظائف: مسار التنفيذ هو تسلسل من التعليمات البرمجية ← البيانات ← التعليمات البرمجية ← البيانات... لم تعد البيانات مرتبطة بمجموعة معينة من الدالات، كما هو الحال في تعريف الصنف. بدلاً من ذلك، فهي حرّة في تمثيل التقدّم المتكشف لتطبيقنا في أثناء تحويل مدخلاته إلى مخرجات. هذا يعني أنه يمكننا تقليل الاقتران بشكلٍ كبير: يمكن استخدام دالّة ما (وإعادة استخدامها) في أي مكان تتطابق فيه معاملاتها مع خرج دالّة أخرى.

نعم، لا تزال هناك درجة من الاقتران، ولكن من خبرتنا، يمكن التحكم فيها بشكل أكبر من الأسلوب غرضي التوجه للقيادة والتحكم. وإذا كنت تستخدم لغة تستخدم تدقيق النوع (type checking)، فستتلقى تحذيرات وقت التَرْجَمَة عندما تحاول ربط شيئين غير متوافقين.

ماذا عن معالجة الخطأ؟

لقد نجحت تحويلاتنا حتى الآن في عالم لا يحدث فيه أي خطأ. لكن كيف يمكننا استخدامها في العالم الحقيقي؟ إذا كان بإمكاننا فقط بناء سلاسل خطيّة (linear chains)، فكيف يمكننا إضافة كل هذا المنطق الشرطيّ الذي نحتاجه للتحقّق الأخطاء؟

هناك العديد من الطرق للقيام بذلك، لكن جميعها تعتمد على اتفاقيّة أساسيّة: نحن لا نمرّر القيم الخام (raw values) بين التحويلات. بدلاً من ذلك، نغلّفها في بنية بيانات (أو نوع) والتي تخبرنا أيضًا ما إذا كانت القيمة المضمنّة صالحة. في هاسكل، على سبيل المثال، يُدعى هذا المُغلّف Maybe. وفي F و Scala يدعى " Option ".

إن كيفيّة استخدامك لهذا المفهوم خاص باللغة. عمومًا، وعلى الرغم من ذلك، هناك طريقتان أساسيّتان لكتابة الشِفرة: يمكنك التعامل مع التحقّق الأخطاء داخل التحويلات الخاصّة بك أو خارجها.

لغة الإكسير، التي استخدمناها حتى الآن، ليس لديها هذا الدعم المدمج. بالنسبة لأغراضنا، هذا شيء جيّد، حيث نُظهر التنفيذ من الألف إلى الياء. يجب أن يعمل شيء ما مشابه في معظم اللغات الأخرى.

أولاً، اختر تمثيلاً

نحتاج إلى تمثيل لغلافنا (بنية البيانات التي تحمل قيمة أو إشارة خطأ). يمكنك استخدام البنى لهذه، لكن إكسير لديه فعلًا عرف قوي جدًا: تميل الدالات إلى إرجاع مجموعة تحتوي إما على {:ok، value} أو على {:error، reason}. على سبيل المثال، يُرجع File.open إما :ok وعملية إدخال / إخراج أو:error ورمز سبب الخطأ:

```
iex(1)> File.open("/etc/passwd")
{:ok, #PID<0.109.0>}
iex(2)> File.open("/etc/wombat")
{:error, :enoent}
```

سنستخدم المجموعة :ok/:error كغلاف عند تمرير الأشياء عبر مسار تنفيذ (pipeline).

ثم تعامل معها داخل كل تحويل

دعنا نكتب دالة تُرجع جميع الأسطر في مِلَفّ يحتوي على سلسلة معطاة، مُقتطَعة إلى أول 20 محرف. نريد أن نكتبها كتحويل، لذا فإن الإدخال سيكون اسم مِلَفّ وسلسلة للتطابق، وسيكون الناتج إما مجموعة :ok مع قائمة من الأسطر أو مجموعة :error مع نوع السبب. يجب أن تبدو دالة المستوى الأعلى مشابهة لما يلي:

function-pipelines/anagrams/lib/grep.ex
```
def find_all(file_name, pattern) do
  File.read(file_name)
  |> find_matching_lines(pattern)
  |> truncate_lines()
end
```

لا يوجد تدقيق صريح للأخطاء هنا، ولكن إذا أعادت أي خطوة في مسار التنفيذ مجموعة خطأ، فسيعرض مسار التنفيذ هذا الخطأ دون تنفيذ الوظائف التالية له[90]. نقوم بذلك باستخدام مطابقة نمط إكسير:

function-pipelines/anagrams/lib/grep.ex
```
defp find_matching_lines({:ok, content}, pattern) do
  content
  |> String.split(~r/\n/)
  |> Enum.filter(&String.match?(&1, pattern))
  |> ok_unless_empty()
end
defp find_matching_lines(error, _), do: error
# ----------
defp truncate_lines({ :ok, lines }) do
  lines
  |> Enum.map(&String.slice(&1, 0, 20))
  |> ok()
end
defp truncate_lines(error), do: error
# ----------
defp ok_unless_empty([]), do: error("nothing found")
```

90 لقد أخذنا الحرية هنا. من الناحية التقنيّة ننفذ الدوال التالية. نحن فقط لا ننقد الشِفرة الموجودة في هذه الدوال.

```
defp ok_unless_empty(result), do: ok(result)
defp ok(result), do: { :ok, result }
defp error(reason), do: { :error, reason }
```

الق نظرة على الدالة find_matching_lines. إذا كان المعامل الأول هو مجموعة ok :، فإنها تستخدم المحتوى في تلك المجموعة للعثور على الأسطر التي تتطابق مع النمط. ومع ذلك، إذا لم يكن المعامل الأول هو مجموعة ok :، فسيتم تشغيل الإصدار الثاني من الدالّة، والتي تُرجع هذا المعامل فقط. وبهذه الطريقة، تقوم الدالّة ببساطة بإعادة توجيه خطأ أسفل مسار التنفيذ. ينطبق الشيء نفسه على truncate_lines.

يمكننا اللعب بهذا في وحدة التحكم (console):

```
iex> Grep.find_all "/etc/passwd", ~r/www/
{:ok, ["_www:*:70:70:World W", "_wwwproxy:*:252:252:"]}
iex> Grep.find_all "/etc/passwd", ~r/wombat/
{:error, "nothing found"}
iex> Grep.find_all "/etc/koala", ~r/www/
{:error, :enoent}
```

يمكنك أن ترى أن الخطأ في أي مكان في مسار التنفيذ يصبح على الفور قيمة لمسار التنفيذ.

أو تعامل معها في مسار التنفيذ

قد تبحث في الدالتين find_matching_lines و truncate_lines معتقدًا أننا نقلنا عبء معالجة الخطأ إلى التحويلات. ستكون على حق. في اللغة التي تستخدم مطابقة النمط في استدعاءات الدوال، مثل إكسير، يتم تقليل التأثير، لكنه لا يزال بشعًا.

سيكون من اللطيف إذا كان للإكسير نسخة من مشغّل مسار التنفيذ |> لديه معرفة بالمجموعات :ok/:error وبالتنفيذ الذي يقصر الدائرة عند حدوث خطأ [91]. لكن الحقيقة أنه لا يسمح لنا بإضافة شيء مشابه، وبطريقة تكون قابلة للتطبيق على عدد من اللغات الأخرى.

تكمن المشكلة التي نواجهها في أنه عند حدوث خطأ، لا نريد تشغيل تعليمات برمجية أكثر أسفل مسار التنفيذ، وأننا لا نريد أن تعرف هذه التعليمات أن هذا يحدث. هذا يعني أننا بحاجة إلى تأجيل تشغيل دوال مسار التنفيذ حتى نعلم أن الخطوات السابقة فيه قد نجحت. سنحتاج للقيام بذلك إلى تغييرها من استدعاءات دالة إلى قيم دالة يمكن استدعاؤها لاحقًا. إليك أحد التنفيذات:

function-pipelines/anagrams/lib/grep1.ex

```
defmodule Grep1 do
  def and_then({ :ok, value }, func), do: func.(value)
  def and_then(anything_else, _func), do: anything_else
  def find_all(file_name, pattern) do
    File.read(file_name)
    |> and_then(&find_matching_lines(&1, pattern))
    |> and_then(&truncate_lines(&1))
  end
  defp find_matching_lines(content, pattern) do
```

91 في الواقع، يمكنك إضافة عامل التشغيل هذا إلى إكسير باستخدام مرفق الماكرو الخاص به. مثال على ذلك هو مكتبة Monad في hex. يمكنك أيضًا استخدام إكسير مع البناء، ولكن بعد ذلك تفقد الكثير من معنى كتابة التحولات التي تحصل عليه باستخدام مسارات التنفيذ.

```
    content
    |> String.split(~r/\n/)
    |> Enum.filter(&String.match?(&1, pattern))
    |> ok_unless_empty()
  end
  defp truncate_lines(lines) do
    lines
    |> Enum.map(&String.slice(&1, 0, 20))
    |> ok()
  end
  defp ok_unless_empty([]), do: error("nothing found")
  defp ok_unless_empty(result), do: ok(result)
  defp ok(result), do: { :ok, result }
  defp error(reason), do: { :error, reason }
end
```

تعدّ الدالّة and_then مثالاً على دالة ربط: فهي تأخذ قيمة مغلّفة في شيء ما، ثم تطبّق دالّة على تلك القيمة، وتعيد قيمة مغلّفة جديدة. يتطلب استخدام الدالة and_then في مسار التنفيذ علامات ترقيم إضافيّة صغيرة لأنه يجب إخبار إكسير بتحويل استدعاءات الدالّة إلى قيم دالّة، ولكن هذا الجهد الإضافي يقابله حقيقة أن دوال التحويل تصبح بسيطة: يأخذ فقط كل منها قيمة (وأيّة معاملات إضافيّة) ويُعيد {:ok, new_value} أو {:error, reason}.

برمجة تحويل التحويلات

يمكن أن يكون التفكير في الشِفرة كسلسلة من التحويلات (المتداخلة) نهجًا متحرّرًا للبرمجة. يتطلب الأمر بعض الوقت لتعتاد على ذلك، ولكن بمجرّد تطوير هذه العادة، ستجد أن الشِفرة الخاصة بك تصبح أكثر نظافة، وتصبح دوالك أقصر، وتصميماتك أجمل.

جرّبها.

الأقسام ذات الصلة

- الموضوع 8، جوهر التصميم الجيّد، صفحة 51
- الموضوع 17، ألعاب شِل، الصفحة 99
- الموضوع 26، كيفية موازنة الموارد، في الصفحة 139
- الموضوع 28، الفصل، في الصفحة 151
- الموضوع 35، الممثلون والعمليات، في الصفحة 205

تمارين

تمرين 21 (إجابة محتملة في الصفحة 326)

هل يمكنك التعبير عن المتطلبات التالية كتحويل المستوى الأعلى؟ أي تحديد المدخلات والمخرجات لكل منها.

1. إضافة ضريبة الشحن والمبيعات إلى طلبية

2. قيام تطبيقك بتحميل معلومات الضبط من مِلَفّ معين
3. قيام شخص ما بتسجيل الدخول إلى تطبيق ويب

تمرين 22 (إجابة محتملة في الصفحة 327)

لقد حدّدت الحاجة إلى التحقّق حقل الإدخال وتحويله من نوع سلسلة نصيّة إلى نوع عدد صحيح قيمته بين 18 و 150. يُوصف التحول العام بواسطة

```
field contents as string
    → [validate & convert]
      → {:ok, value} | {:error, reason}
```

اكتب التحولات الفرديّة التي تقوم بالتحقّق والتحويل.

تمرين 23 (إجابة محتملة في الصفحة 125)

في **اللغة X لا تحتوي على مسارات تنفيذ**، كتبنا في الصفحة 153:

```
const content = File.read(file_name);
const lines = find_matching_lines(content, pattern)
const result = truncate_lines(lines)
```

يكتب الكثير من الناس شِفرة كائنية التوجه عن طريق ربط استدعاءات الطريقة معًا، وقد يميلون إلى كتابة ذلك كشيء من قبيل:

```
const result = content_of(file_name)
                    .find_matching_lines(pattern)
                    .truncate_lines()
```

ما الفرق بين هاتين الشفرتين؟ أيهما تعتقد أننا نفضل؟

الموضوع 31. ضريبة الوراثة

> أردتَ موزة لكنّما حصلتَ عليه هو غوريلا تمسك الموز والغابة بِرُمَّتها.
> ← جو أرمسترونج

هل تبرمج بلغة كائنيّة التوجه؟

هل تستخدم الوراثة؟

إذا كان الأمر كذلك، توقف! ربما ليس هذا ما تريد القيام به.

دعنا نرى لماذا.

القليل من الخلفيّة

ظهرت الوراثة لأول مرّة في لغة البرمجة سيمولا 67 Simula عام 1969. لقد كانت حلًا أنيقًا لمشكلة اصطفاف أنواع متعدّدة من الأحداث في نفس القائمة. كان نهج Simula هو استخدام شيء يسمى *أصناف بادئة (prefix classes)*. يمكنك كتابة شيء مثل هذا:

```
link CLASS car;
  ... implementation of car
link CLASS bicycle;
  ... implementation of bicycle
```

إن link صنف بادئ يُضيف وظائف القوائم المرتبطة. يتيح لك ذلك إضافة كل من الأصناف car و bicycle إلى قائمة الأشياء التي تنتظر عند (فلنقل) إشارة المرور. في المصطلحات الحاليّة، سيكون link هو صنف أب (parent class).

كان النموذج الذهنيّ الذي استخدمه مبرمجو *سيمولا* هو أنه تم إلحاق بيانات المثيل (instance) وتنفيذ الصنف link بتنفيذ الأصناف car و bicycle. كان يُنظر إلى الجزء link تقريبًا على أنه *حاوية* تضم car و bicycle. وقد أعطاهم ذلك صيغة من صيغ تعددية الأشكال (polymorphism): قامت كل من car و bicycle بتنفيذ واجهة (link) لأن كلاهما يحتويان على شِفرة (link).

بعد سيمولا جاء سمول توك (Smalltalk). يصف ألان كاي، أحد مبدعي سمول توك، في إجابة 2019 Quora[92] لماذا تمتلك *سمول توك* الوراثة:

> لذا، عندما صَممتُ *سمول توك-72* – وكان الأمر بمنزلة مزحة للتسلية في أثناء التفكير في *سمول توك-71* - اعتقدت أنه سيكون من الممتع استخدام ديناميكياتها المُشابهة لـ Lisp[93] لإجراء تجارِب على "البرمجة التفاضلية" (بمعنى: طرق مختلفة للإنجاز "هذا مشابه لذاك الاستثناء").

هذا تصنيف فرعي بحت للسلوك.

92 https://www.quora.com/What-does-Alan-Kay-think-about-inheritance-in-object-oriented-programming

93 **[المترجم]** لغة معالجة القوائم (LISt Processing language).

وقد تطوّر هذان الأسلوبان للوراثة (اللذان كان لهما قدر لا بأس به في الانتشار) على مدى العقود التالية. نهج سيمولا، الذي اقترح أن الوراثة هي طريقة للجمع بين الأنواع، استمر في لغات مثل سي++ و جافا. ومدرسة سمول توك، حيث كانت الوراثة منظمة ديناميكية للسلوك، شوهدت في لغات مثل روبي وجافا سكريبت.

لذا، نواجه الآن جيلًا من مطوري اللغات كائنية التوجه الذين يستخدمون الوراثة لسببين: أنهم لا يحبون الكتابة، أو أنهم يحبون الأنواع.

أولئك الذين لا يحبون الكتابة يريحون أصابعهم باستخدام الوراثة لإضافة وظائف شائعة من أصناف أساسية "أب" (base class) إلى أصناف فرعيّة "ابن" (child class): الصنف User والصنف Product كلاهما من الأصناف الفرعيّة لـ ActiveRecord :: Base.

أما أولئك الذين يحبون الأنواع يستخدمون الوراثة للتعبير عن العِلاقة بين الأصناف: Car هي نوع من Vehicle.

للأسف كلا النوعين من الوراثة لديه مشكلات.

مشكلات استخدام الوراثة لمشاركة الشِفرة

الوراثة هي اقتران. ليس فقط صنف ابن مرتبط بصنف أب، وأب الأب، وهلمّ جرا، ولكن الشِفرة التي تستخدم الابن مرتبطة أيضًا بجميع الأسلاف. إليك مثال:

```
class Vehicle
  def initialize
    @speed = 0
  end
  def stop
    @speed = 0
  end
  def move_at(speed)
    @speed = speed
  end
end
class Car < Vehicle
  def info
    "I'm car driving at #{@speed}"
  end
end
# top-level code
my_ride = Car.new
my_ride.move_at(30)
```

عندما يستدعي المستوى الأعلى my_car.move_at، تكون الطريقة التي يتم استدعاؤها موجودة في Vehicle، أب الصنف Car.

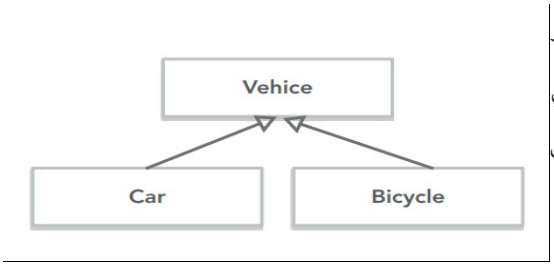
الآن يقوم المطوّر المسؤول عن Vehicle بتغيير واجهة برمجة التطبيقات، وبذلك move_at تصبح set_velocity، ويصبح متغير المثيل @speed هو @velocity.

من المتوقع أن يكسر تغيير واجهة برمجة التطبيقات عملاء الصنف Vehicle. لكن لا يكسر المستوى الأعلى: بقدر ما يتعلق الأمر باستخدام Car. ما يفعله صنف Car من حيث التنفيذ ليس له صلة بشِفرة المستوى الأعلى، لكنه لا يزال ينكسر.

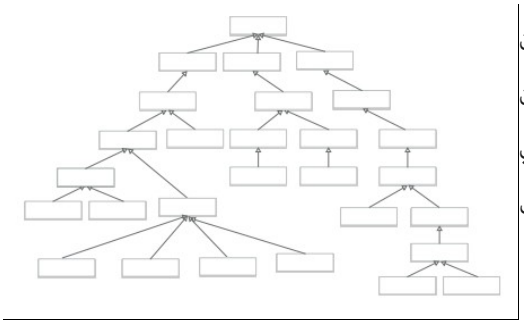
وبالمثل، فإن اسم متغير المثيل هو مجرّد تفاصيل تنفيذ داخلية بحتة، ولكن عندما يتغيّر صنف Vehicle، فإنه يكسر أيضًا (بصمت) صنف Car.

هذا كثير من الاقتران.

مشكلات استخدام الوراثة لبناء الأنواع



يرى بعض النّاس الوراثة كوسيلة لتعريف أنواع جديدة. يُظهر مخطط التصميم المفضل لديهم التسلسلات الهرميّة للصنف. يرون المشكلات بالطريقة التي ينظر بها *العلماء الفيكتوريون* إلى الطبيعة، على أنها شيء يمكن تقسيمه إلى تصنيفات.



1. لسوء الحظ، سرعان ما تنمو هذه الرسوم البيانيّة لتغطي الجدران الضخمة، تُضاف طبقة على طبقة للتعبير عن أصغر فارق من التمايز بين الأصناف. يمكن أن يجعل هذا التعقيد الإضافيّ التطبيق أكثر هشاشة، حيث يمكن أن تتموج التغييرات لأعلى ولأسفل طبقاتٍ عدّة.

والأسوأ من ذلك هو مسألة الوراثة المتعددة. قد يكون الصنف *Car* نوعًا من *Vehicle*، ولكن يمكن أن تكون أيضًا نوعًا من Asset و LoanCollateral و InsuredItem وما إلى ذلك. سيحتاج ذلك إلى وراثة متعدّدة لنمذجته بشكل صحيح.

أعطت سي++ الوراثة المتعددة سمعةً سيئةً في التسعينيات بسبب بعض دلالات الغموض المشكوك فيها. ونتيجة لذلك، لا تقدّمه العديد من اللغات كائنية التوجه الحاليّة. لذلك، حتى إذا كنت راضيًا عن البنية الشجرية معقّدة النوع، فلن تتمكن من لنمذجة نطاقك بدقّة على أي حال.

نصيحة ٥١ | **لا تدفع ضريبة الوراثة**

أبدال أفضل

دعنا نقترح ثلاث تقنيات والتي تعني أنك لن تحتاج أبدًا إلى استخدام الوراثة مرّة أخرى:

- الواجهات والبروتوكولات
- التفويض
- أصناف Mixins والسمات (traits)

الواجهات والبروتوكولات

تسمح لك معظم اللغات كائنيّة التوجّه بتحديد أن الصنف ينفّذ مجموعة أو أكثر من السلوكيات. يمكنك القول، على سبيل المثال، أن الصنف Car ينفذ السلوك Drivable والسلوك Locatable. تختلف الصيغة المستخدمة للقيام بذلك:

في جافا، قد تبدو كالتالي:

```
public class Car implements Drivable, Locatable {
   //  يجب أن تتضمن هذه الشِفرة لصنف السيارة
   // كلا من الوظيفتين
   //   Locatable و Drivable
}
```

إن Drivable و Locatable هي ما تسميه جافا *واجهات*؛ تسميها لغات أخرى *بروتوكولات*، ويطلق عليها البعض *سمات* "traits" (مع أنّ هذا ليس هو ما سنطلق عليه تسمية *سمة* لاحقًا).

تُعرّف الواجهات على النحو التالي:

```
public interface Drivable {
   double getSpeed();
   void stop();
}
public interface Locatable() {
   Coordinate getLocation();
   boolean locationIsValid();
}
```

لا تُنشئ هذه التصريحات أية شِفرة: فهي تقول ببساطة أن أي صنف ينفّذ Drivable يجب أن ينفّذ الطريقتين getSpeed و stop، والصنف الذي ينفّذ Locatable يجب أن ينفّذ getLocation و locationIsValid. هذا يعني أن تعريفنا السابق للصنف Car لن يكون مقبولًا إلا إذا تضمن هذه الطرائق الأربعة مجتمعةً.

إن ما يجعل الواجهات والبروتوكولات قويّة جدًا هو أنه يمكننا استخدامها كأنواع، وأي صنف ينفّذ الواجهة المناسبة سيكون متوافق مع هذا النوع. إذا كانت كل من Car و Phone ينفّذان Locatable، فيمكننا تخزين كليهما في قائمة عناصر locatable:

```
List<Locatable> items = new ArrayList<>();
items.add(new Car(...));
items.add(new Phone(...));
items.add(new Car(...));
// ...
```

يمكننا بعد ذلك معالجة تلك القائمة، بأمان مع العلم أن كل عنصر لديه getLocation و locationIsValid:

```
void printLocation(Locatable item) {
   if (item.locationIsValid() {
      print(item.getLocation().asString());
}
// ...
items.forEach(printLocation);
```

نصيحة ٥٢ فضّل الواجهات للتعبير عن تعدّدية الأشكال

تعطينا الواجهات والبروتوكولات تعدديّة الأشكال دون وراثة.

التفويض

تشجّع الوراثة المطورين على إنشاء أصناف تحتوي كائناتها على عدد كبير من الطرائق (methods). إذا كان لدى صنف أب 20 طريقة، ويريد الصنف الفرعي الاستفادة من اثنتين منها فقط، فسيظل لدى كائناتها 18 طريقة أخرى متسكعة جانبًا ومتاحة للاستدعاء. يفقد الصنف السيطرة على واجهته. هذه مشكلة شائعة - تصرّ العديد من أطر الثبات "الاستمرار"[94] (persistence) وواجهات المستخدم على أن مكونات التطبيق هي صنف فرعي لبعض الأصناف الأساسيّة المُورِّدة:

```
class Account < PersistenceBaseClass
end
```

يحمل الصنف Account الآن جميع واجهة برمجة تطبيقات صنف الاستمرار[95] معه. بدلًا من ذلك، تخيّل بديلًا باستخدام التفويض، كما في المثال التالي:

```
class Account
  def initialize(. . .)
    @repo = Persister.for(self)
  end
  def save
    @repo.save()
  end
end
```

لا نكشف الآن أيًا من واجهة برمجة تطبيقات إطار العمل لعملاء الصنف Account الخاص بنا: لقد أصبح هذا الاقتران مُعطلاً الآن. ولكن هناك المزيد. بما أننا الآن لم نعد مقيّدين بواجهة برمجة التطبيقات للإطار الذي نستخدمه، فنحن أحرار في إنشاء واجهة برمجة التطبيقات التي نحتاجها. نعم، كان بإمكاننا فعل ذلك من قبل، ولكننا دائمًا ما نواجه خطر تجاوز الواجهة التي كتبناها، واستخدام API المستمر بدلاً من ذلك.

الآن نحن نتحكم في كل شيء.

94 **[المترجم]** إطار الثبات هو برنامَج وسيط يساعد ويؤتمت تخزين بيانات البرنامَج في قواعد البيانات، خاصة قواعد البيانات العلائقية. يعمل كطبقة تجريد بين التطبيق وقاعدة البيانات، وعادة ما يسد أي اختلافات مفاهيميّة بينهما.

95 **[المترجم]** في علم الحاسوب، يشير مصطلح الاستمرارية "الثبات" (persistence) إلى خاصيّة الحالة (state) التي تفوق العملية التي أنشأتها. يتم تحقيق ذلك في الممارسة العملية عن طريق تخزين الحالة كبيانات في أجهزة تخزين بيانات الحاسوب. يجب أن تقوم البرامج بنقل البيانات من وإلى أجهزة التخزين ويجب أن توفر تعيينات (Mappings) من بنى بيانات لغة البرمجة الأصليّة إلى بنى بيانات جهاز التخزين. برامج تحرير الصور أو معالجات النصوص، على سبيل المثال، تحقق ثبات الحالة بواسطة حفظ مستنداتها في المِلَفّات

نصيحة ٥٣ فوّض لخِدْمَات: لديه A فالطرنيب هو A

في الواقع، يمكننا أن نخطو خطوة أخرى. لماذا يجب أن يعرف Account كيف يستمر بنفسه؟ أليست وظيفته معرفة وتطبيق قواعد عمل الحساب account ؟

```
class Account
  # لا شيء فقط أشياء خاصة بالحساب
end
class AccountRecord
  # تغليف الحساب مع إمكانية
  #  جلبه وتخزينه
end
```

نحن الآن منفصلون حقًا، ولكن هذا كان بتكلفة. سنضطر إلى كتابة المزيد من التعليمات البرمجيّة، وعادةً ما يكون بعضًا منها عبارة عن نص معياري (boilerplate): من المحتمل أن تحتاج جميع أصناف السجل لدينا إلى طريقة بحث، على سبيل المثال.

لحسن الحظ، هذا ما تفعله mixins والسمات لنا.

أصناف Mixins، السمات، التصنيفات، ملحقات المراسم، ...

كصناعة، نحن نحب إعطاء أسماء للأشياء. غالبًا ما سنعطي الشيء نفسه العديد من الأسماء. المزيد أفضل، أليس كذلك؟

هذا ما نتعامل معه عندما ننظر إلى أصناف mixins. الفكرة الأساسيّة بسيطة: نريد أن نكون قادرين على توسيع الأصناف والكائنات بوظائف جديدة دون استخدام الوراثة. لذلك نُنشئ مجموعة من هذه الوظائف، ونعطي تلك المجموعة اسمًا، ثم نقوم بطريقة ما بتوسيع صنف أو كائن معهم. عند هذه النقطة، تكون قد أنشأت صنفًا أو كائنًا جديدًا يجمع بين إمكانات الصنف الأصلي وجميع أصناف mixins الخاصّة به. ستتمكن في معظم الحالات، من عمل هذا الامتداد (extension) حتى إذا لم يكن لديك حق الوصول إلى شِفرة المصدر للصنف الذي توسعه.

حاليًا يختلف تنفيذ واسم هذه المِيزة بين اللغات.

سنميل إلى تسميتها ب mixins هنا، لكننا نريدك حقًا أن تفكّر في ذلك كمِيزة لغويّة غير موجهه(لا أدرية)[96]. الشيء المهم هو القدرة التي تمتلكها كل هذه التطبيقات: دمج الوظائف بين الأشياء الموجودة والأشياء الجديدة.

كمثال، دعنا نعود إلى مثال AccountRecord. كما تركناه، كان على AccountRecord أن يعرف عن كلا الحسابين وعن إطار عملنا المستمر "الثابت". وهي إلى ذلك بحاجة إلى تفويض جميع الطرائق في طبقة المستمرة التي أرادت عرضها على العالم الخارجي.

توفر لنا أصناف Mixins بديلاً. أولاً، يمكننا كتابة Mixins ينفّذ (على سبيل المثال) طريقتين من ثلاث طرائق قياسيّة للبحث. يمكننا بعد ذلك إضافتها إلى AccountRecord كـ Mixins. وبينما نكتب أصنافا جديدة للأشياء المستمرة، يمكننا إضافة Mixins إليها أيضًا:

96 **[المترجم]** ألا أدريّة- مذهب فلسفي لا يعتمد الإثبات ولا النفي.

```
mixin CommonFinders {
   def find(id) { ... }
   def findAll() { ... }
end
class AccountRecord extends BasicRecord with CommonFinders
class OrderRecord extends BasicRecord with CommonFinders
```

يمكننا أن نأخذ هذا أبعد من ذلك بكثير. على سبيل المثال، نعلم جميعًا أن كائنات الأعمال (business objects) الخاصة بنا تحتاج إلى شِفرة تحقّق لمنع البيانات السيئة من التسلل إلى حساباتنا.

لكن بالضبط ماذا نعني بالتحقّق؟

إذا أخذنا حسابًا، على سبيل المثال، فربما تكون هناك العديد من طبقات التحقّق المختلفة التي يمكن تطبيقها:

- التحقّق أن كلمة المرور المشفّرة (hashed) تطابق تلك التي أدخلها المستخدم
- التحقّق صحّة بيانات النموذج التي أدخلها المستخدم عند إنشاء الحساب
- التحقّق صحّة بيانات النموذج التي أدخلها شخص مسؤول بتحديث تفاصيل المستخدم
- التحقّق صحّة البيانات المُضافة إلى الحساب بواسطة مكونات النظام الأخرى
- التحقّق صحّة البيانات من أجل الاتساق قبل استمرارها.

نهج شائع (ونعتقد أنه أقل من مثالي) هو تجميع جميع عمليات التحقّق في صنف واحد (كائن الأعمال / كائن الاستمرار) ثم إضافة علامات للتحكم في إطلاق عملية التحقق المناسبة تبعًا الظروف.

نعتقد أن أفضل طريقة هي استخدام mixins لإنشاء أصناف متخصصة للمواقف المناسبة:

```
class AccountForCustomer extends Account
      with AccountValidations,AccountCustomerValidations
class AccountForAdmin extends Account
      with AccountValidations,AccountAdminValidations
```

هنا، يتضمن كلا الصنفين المشتقين عمليات تحقّق مشتركة لجميع كائنات Account. يتضمن متغير customer أيضًا عمليات تحقّق مناسبة لواجهات برمجة تطبيقات العميل، بينما يحتوي متغير المشرف (admin) على عمليات تحقّق المشرف (التي يفترض أنها أقل تقييدًا).

الآن، بواسطة تمرير مثيلات AccountForCustomer أو AccountForAdmin ذهابًا وإيابًا، تضمن شفرتنا تلقائيًا تطبيق التحقّق الصحيح.

نصيحة ٥٤	استخدم Mixin لمشاركة الوظائف

نادرًا ما يكون الجواب هو الوراثة

لقد ألقينا نظرة سريعة على ثلاثة أبدال للوراثة الطبقيّة التقليديّة:

- الواجهات والبروتوكولات
- التفويض
- أصناف Mixins والسمات

قد تكون كل طريقة من هذه الطرائق أفضل بالنسبة لك في ظروف معيّنة، اعتمادًا على ما إذا كان هدفك هو مشاركة معلومات النوع، أو إضافة الوظائف، أو مشاركة الطرائق. فكما هو الحال مع أي شيء في البرمجة، تهدف إلى استخدام التقنية التي تعبر عن نيتك بأفضل شكل.

وحاول ألا تسحب الغابة بِرُمَّتها طوال الرحلة.

الأقسام ذات الصلة

- الموضوع 8، جوهر التصميم الجيّد، صفحة 51
- الموضوع 10، التعامدية، في الصفحة 62
- الموضوع 28، الفصل، في الصفحة 151

التحديات

- في المرّة القادمة التي تجد فيها نفسك تقوم بتصنيف فرعي، خذ دقيقة لفحص الخيارات. هل يمكنك تحقيق ما تريد بواسطة واجهات و / أو تفويض و / أو mixins ؟ هل يمكنك تقليل الاقتران بواسطة القيام بذلك؟

الموضوع 32. الضبط (Configuration)

> دع كل أشيائك تأخذ أماكنها؛ دع كل جزءٍ من عملك يأخذ وقته.
> ← بنيامين فرانكلين، ثلاث عشرة فضيلة، السيرة الذاتية

عندما تعتمد الشِفرة على القيم التي قد تتغير بعد نشر التطبيق، فاحتفظ بهذه القيم خارج التطبيق. عندما يعمل تطبيقك في بيئات مختلفة، وربما لعملاء مختلفين، احتفظ بالقيم الخاصّة بالبيئة والعملاء خارج التطبيق. بهذه الطريقة، فأنت تضع معاملات لتطبيقك؛ تتكيّف الشِفرة مع الأماكن التي تُشغّلها.

نصيحة ٥٥ | **ضع معامِلات لتطبيقك مُستخدمًا الضبط الخارجي**

تتضمن الأشياء الشائعة التي قد ترغب في وضعها في بيانات الضبط ما يلي:

- اعتمادات الخِدْمَات الخارجيّة (قاعدة البيانات، واجهات برمجة التطبيقات التابعة لجهات خارجيّة، وما إلى ذلك)
- مستويات ووجهات التسجيل
- المنفذ وعنوان IP وأسم الجهاز والتجميع (cluster) الذي يستخدمه التطبيق
- معامِلات التحقّق الخاصّة بالبيئة
- المعامِلات المحدّدة خارجيًا، مثل معدلات الضرائب
- تفاصيل التنسيق الخاصّة بالموقع
- مفاتيح الترخيص

ابحث بشكلٍ أساسيّ، عن أي شيء تعرف أنه سيتعيّن عليك تغييره والذي يمكنك التعبير عنه خارج الجسم الرئيس لشفرتك، وضعه في خانة الضبط.

الضبط الثابت (Static Configuration)

تحتفظ العديد من أطر العمل، وعدد غير قليل من التطبيقات المخصّصة، **بالضبط** إما في مِلَفّات بسيطة (flat files) أو في جداول قاعدة البيانات. إذا كانت المعلومات موجودة في المِلَفّات البسيطة، فالتوجه هو استخدام نوع من تنسيق النص الواضح المُعدّ. حاليًا YAML و JSON شائعان لهذا الغرض. في بعض الأحيان، تستخدم التطبيقات المكتوبة بلغات البرمجة النصيّة (scripting languages) مِلَفّات شِفرة المصدر لأغراض خاصّة، مخصّصة لاحتواء الضبط فقط. إذا كانت المعلومات مُهيكلة، ومن المرجّح أن

يقوم العميل بتغييرها (معدلات ضريبة المبيعات، على سبيل المثال)، فقد يكون من الأفضل تخزينها في جدول قاعدة بيانات. وبالطبع، يمكنك استخدام كلا الطريقتين، وتقسيم معلومات الضبط وفقًا للاستخدام.

مهما كانت الطريقة الذي تستخدمها، فإن الضبط يُقرأ في التطبيق الخاص بك كبنية بيانات، عادةً عند بَدْء التطبيق. وعمومًا، يتم إنشاء بنية البيانات هذه عموميًا، حيث يُعتقد أن هذا يُسهّل على أي جزء من الشِفرة الوصول إلى القيم التي يحملها.

نحن نفضّل ألا تفعل هذا. بدلاً من ذلك، غلّف معلومات الضبط خلف واجهة برمجة تطبيقات (رقيقة). هذا من شأنه فصل الشِفرة الخاصّة بك عن تفاصيل تمثيل الضبط.

الضبط كخدمة

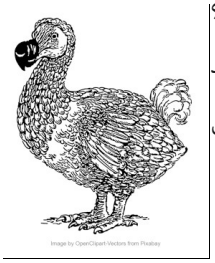
في حين أن الضبط الثابت شائع، فإننا نفضّل حاليًا نهجًا مختلفًا. ما زلنا نريد الاحتفاظ ببيانات الضبط خارج التطبيق، ولكن بدلاً من تخزينها في مِلَفّ أو قاعدة بيانات، نودّ أن نراها مخزّنة خلف واجهة برمجة تطبيقات الخدمة. ولهذا عدد من الفوائد:

- يمكن لتطبيقات متعدّدة مشاركة معلومات الضبط، مع الاستيثاق (authentication) والتحكم في الوصول مما يقيّد ما يمكن أن يراه كل منهما
- يمكن إجراء تغييرات الضبط على المستوى العموميّ
- يمكن المحافظة على بيانات الضبط عبر واجهة مستخدم متخصّصة
- تصبح بيانات الضبط ديناميكية

هذه النقطة الأخيرة، وهي أن الضبط يجب أن يكون ديناميكيًا، أمر بالغ الأهميّة مع تقدمنا نحو التطبيقات المتاحة بشكل كبير. إن الفكرة القائلة بأنه يجب علينا إيقاف وإعادة تشغيل التطبيق لتغيير معامل واحد بعيدة كل البعد عن الواقع الحديث. يمكن لمكونات التطبيق باستخدام خدمة الضبط، التسجيل للحصول على إشعارات بتحديثات المعاملات التي يستخدمونها، ويمكن أن ترسل الخدمة إليهم رسائل تحتوي على قيم جديدة في حال تغييرها ووقت حدوث هذا التغيير.

مهما كان الشكل الذي تتخذه، فإن بيانات الضبط تقود سلوك وقت تشغيل التطبيق. عندما تتغير قيم الضبط، ليست هناك حاجة لإعادة بناء الشِفرة (rebuild).

لا تكتب شِفرة الدودو Dodo-Code



دون ضبط خارجي، فإن شفرتك لن تكون متكيّفةً ومرنة كما ينبغي. هل هذا شيء سيئ؟ حسنًا، هنا في العالم الحقيقي، تموت الأنواع التي لا تتكيّف. لم يتكيّف طائر الدودو مع وجود البشر وماشيتهم في جزيرة موريشيوس، وسرعان ما انقرض[97]. كان هذا أول انقراض موثّق لأحد الأنواع على يد الإنسان.

لا تدع مشروعك (أو حياتك المهنية) تسير في طريق الدودو.

97 لم يكن من المفيد أن يضرب المستوطنون الطيور الهادئة (اقرأها: *الغبية*) حتى الموت في النوادي الرياضية.

الأقسام ذات الصلة

- الموضوع 9، DRY - شرور التكرار، في الصفحة 54
- الموضوع 14، لغات النطاق، في الصفحة 82
- الموضوع 16، قوة النص الواضح، في الصفحة 95
- الموضوع 28، الفصل، في الصفحة 151

لا تبالغ

في الإصدار الأول من هذا الكتاب، اقترحنا استخدام الضبط بدلاً من الشِفرة بطريقة مشابهة، ولكن يبدو أنه كان يجب أن نكون أكثر تحديدًا في إرشاداتنا. يمكن تؤخذ أي نصيحة إلى أقصى الحدود أو استخدامها بشكل غير ملائم، لذلك إليك بعض التحذيرات:

لا تبالغ. قرّرَ أحد عملائنا الأوائل أن يكون كل حقل في تطبيقهم قابلًا للضبط. ونتيجة لذلك، استغرق الأمر أسابيع لإجراء حتى أصغر تغيير، حيث كان عليك تنفيذ كل من الحقل والشِفرة المسؤولة عن حفظه وتعديله. كان لديهم حوالي 40،000 معامل ضبط وكابوس من كتابة الشِفرة بين أيديهم.

لا تتخذ قرارات استخدام الضبط ناشئة من الكسل. إذا كان هناك جدال حقيقي حول ما إذا كان يجب أن تعمل المِيزة بهذه الطريقة أم تلك، أو إذا كان يجب أن تكون من اختيار المستخدمين، فجرّبها بإحدى الطرق واحصل على ملاحظات على ما إذا كان هذا القرار جيّدًا.

الفصل السادس:

التسایر

6

دعنا نبدأ ببعض التعريفات حتى نكون على اتفاق: التساير (Concurrency) هو عندما يتصرّف تنفيذ شفرتين أو أكثر وكأنهما تعملان في نفس الوقت. التوازي (Parallelism) هو عندما تعملان في نفس الوقت حقًا.

للحصول على التساير، تحتاج إلى تشغيل الشِفرة في بيئة يمكنها تبديل التنفيذ بين أجزاء مختلفة من شفرتك في أثناء تشغيلها. وغالبًا ما يُنفّذ ذلك باستخدام أشياء مثل الألياف (fibers) والمسارات (threads) والعمليات (processes).

للحصول على التوازي، تحتاج إلى كيانات ماديّة (hardware) يمكنها القيام بأمرين معًا في وقتٍ واحد. قد تكون هذه الكيانات نوى متعدّدة في وحدة المعالجة المركزية، أو وحدات معالجة مركزية متعدّدة في حاسوب، أو حواسيب متعدّدة متصلة ببعضها البعض.

كل شيء متساير

يكاد يكون من المستحيل كتابة شِفرة في نظام بحجم لائق لا يحتوي على جوانب متسايرة فيه. قد تكون صريحة أو مخفيّة داخل مكتبة. يعدّ التزامن شرطًا إذا كنت تريد لتطبيقك أن يكون قادرًا على التعامل مع العالم الحقيقي، حيث الأشياء غير متزامنة (asynchronous): يتفاعل المستخدمون، وتُجلب البيانات، وتُستدعى الخِدْمَات الخارجيّة، جميعهم معًا. إذا فُرضت هذه العملية لتكون تسلسليّة، بحيث يحدث أحد الأشياء، ثم الشيء التالي، وهكذا، فستشعر أن نظامك بطيء وربما لا تستفيد بشكل كامل من قوة الأجهزة التي تعمل عليها.

في هذا الفصل سنلقي نظرة على التساير والتوازي.

غالبًا ما يتحدّث المطوّرون عن الاقتران بين أجزاء من الشِفرة. إنهم يشيرون إلى التبعيّات (dependencies)، وكيف تجعل هذه التبعيّات الأشياء صعبة التغيير.

ولكن هناك صورة آخر من الاقتران. يحدث *الاقتران الزمني (Temporal coupling)* عندما تفرض شفرتك تسلسلًا على أشياء غير مطلوبة لحلّ المشكلة المطروحة. هل تعتمد على"هذا" قبل "ذاك "؟ ليس وأنت ترغب في البقاء مرنًا. هل تصل شفرتك إلى العديد من الخِدْمَات الخلفيّة بالتتابع، واحدة تلو الأخرى؟ ليس إذا كنت تريد الاحتفاظ بعملائك. في **كسر الاقتران الزمني**، سنلقي نظرة على طرق تحديد هذا النوع من الاقتران الزمنيّ.

لماذا تعدّ كتابة التعليمات البرمجية المتسايرة والمتوازية صعبة للغاية؟ أحد الأسباب هو أننا تعلّمنا البرمجة باستخدام أنظمة متسلسلة، ولدى لغاتنا ميزات آمنة نسبيًا عند استخدامها بشكل متسلسل ولكنها تصبح مسؤوليّة بمجرّد حدوث شيئين في نفس الوقت. الحالة المشتركة (shared state) هي واحدة من أكبر المذنبين هنا. هذا لا يعني فقط المتغيرات العامّة: في أي وقت تشير فيه شفرتين أو أكثر إلى نفس قطعة البيانات القابلة للتغيير، تكون بذلك قد شاركتَ الحالة. و**الحالة المشتركة هي حالة غير صحيحة**. يصف القسم عددًا من الحلول لذلك، لكن في النهاية جميعهم عرضة للخطأ.

إذا كان ذلك يُشعرك بالحزن، فلا تيأس! هناك طرق أفضل لإنشاء تطبيقات متسايرة. أحدها هو استخدام نموذج الممثل (actor model)، حيث تتواصل العمليات المستقلة، التي لا تشارك البيانات، عبر قنوات باستخدام دلالات محدّدة وبسيطة. نتحدّث عن كل من النظريّة والممارسة لهذا النهج في **الممثلون والعمليات**.

أخيرًا، سنلقي نظرة على **السبورات** "بلاك بورد" (Blackboards). وهي أنظمة تعمل كمزيج من متجر الكائنات ووسيط ذكي للنشر / الاشتراك (publish/subscribe). في شكلها الأصليّ، لم تنتشر أبدًا حقًا. لكننا نشهد اليوم المزيد والمزيد من تطبيقات الطبقات الوسيطة مع دلالات شبيهة بالسبورات. توفّر هذه الأنواع من الأنظمة قدرًا كبيرًا من الفصل عند استخدامها بشكلٍ صحيح.

كانت الشِفرة المتسايرة والمتوازية غريبةً. الآن هي مطلوبة.

الموضوع 33. كسر الاقتران الزمنيّ

قد تتساءل "ما هو الاقتران الزمني؟" إنه بخصوص الوقت.

إن الوقت هو أحد الجوانب التي غالبًا ما يتم تجاهلها في هيكليات البرمجيات. الوقت الوحيد الذي يشغلنا هو الوقت على الجدول الزمنيّ، والوقت المتبقي حتى نشحن البرمجيّة– ولكن ليس هذا ما نتحدث عنه هنا. بدلاً من ذلك، نحن نتحدث عن دور الوقت كعنصر تصميم للبرنامَج نفسه. هناك جانبان للوقت مهمّان لنا: التزامن (الأشياء تحدث في نفس الوقت) والترتيب (المواضع النسبيّة للأشياء في الوقت المناسب).

لا نتعامل عادةً مع البرمجة مع وضع أي من هذه الجوانب في الاعتبار. عندما يجلس النّاس لأول مرّة لتصميم بنية أو كتابة برنامَج، تميل الأشياء إلى أن تكون خطيّة. هذه هي الطريقة التي يفكّر بها معظم النّاس – ***افعل هذا*** ومن ثم ***افعل ذلك*** دائمًا. لكن التفكير بهذه الطريقة يؤدي إلى الاقتران الزمني: الاقتران في الوقت المناسب. يجب استدعاء الطريقة **أ** دائمًا قبل الطريقة **ب**؛ يمكن تشغيل تقرير واحد فقط في كل مرّة؛ يجب أن تنتظر حتى يتم إعادة رسم الشاشة قبل استلام نقرة الزر. يجب أن يحدث هذا قبل ذاك.

إن هذا النهج ليس مرنًا للغاية، وليس واقعيًا جدًا.

نحتاج إلى السماح بالتزامن والتفكير في فصل أي وقت أو تبعات طلب. يمكننا بواسطة القيام بذلك، الحصول على إمكانية الوصول وتقليل أي تبعيات تستند إلى الوقت في العديد من مجالات التطوير: تحليل سير العمل والهيكلية والتصميم والنشر. ستكون النتيجة أنظمة أسهل في التفكير فيها، والتي من المحتمل أن تكون أسرع استجابةً وأكثر موثوقيّة.

البحث عن التزامن

نحتاج في العديد من المشروعات إلى نمذجة التطبيق وتحليل سير عمله كجزء من التصميم. نودّ معرفة ما يمكن أن يحدث في نفس الوقت، وما يجب أن يحدث بترتيب صارم. تتمثل إحدى طرق القيام بذلك في ضبط سير العمل باستخدام تدوين (notation) مثل *مخطط النشاط* (activity diagram).[98]

نصيحة ٥٦ **حلّل سير العمل لتحسين التساير.**

يتكون مخطط النشاط من مجموعة من الإجراءات المرسومة كمستطيلات مستديرة الأطراف. يؤدي السهم الخارج من إجراءً ما إما إلى إجراء آخر (والذي يمكن أن يبدأ بمجرّد اكتمال الإجراء الأول) أو إلى خط سميك يسمى شريط التزامن. بمجرّد اكتمال جميع الإجراءات التي تؤدي إلى شريط التزامن، يمكنك بعد ذلك المتابعة على طول أي أسهم يغادر الشريط. ويمكن بَدْء الإجراء الذي لا توجد أسهم مؤدية إليه في أي وقت.

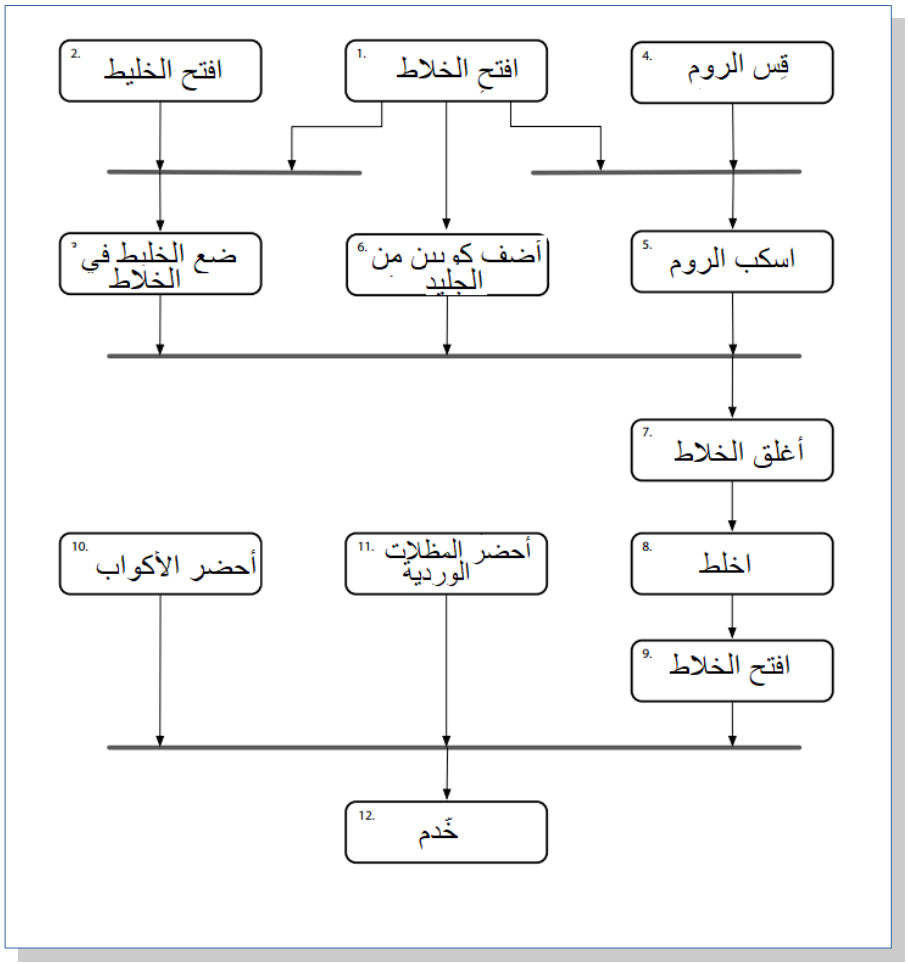
يمكنك استخدام الرسوم البيانيّة للأنشطة لزيادة التوازي إلى أقصى حد بواسطة تحديد الأنشطة التي **يمكن** إجراؤها بالتوازي، ولكن لم تمثّل.

على سبيل المثال، قد نكتب البرنامَج من أجل روبوت تحضير *بينا كولادا* (Piña Colada). قيل لنا أن الخطوات هي

1. افتح الخلاط	7. أغلق الخلاط
2. افتح خليط بينا كولادا	8. تسييل لمدة 1 دقيقة
3. ضع الخليط في الخلاط	9. افتح الخلاط
4. قِس نصف كوب من الروم الأبيض	10. أحضر الأكواب
5. اسكب الروم (rum)	11. أحضر المظلات الورديّة
6. إضف 2 كوب من الجليد	12. خدّم

على أي حال، سيفقد النادل وظيفته إذا اتّبعت هذه الخطوات، واحدة تلو الأخرى، بالترتيب. مع أنّ وصف هذه الإجراءات بشكل متسلسل، فإنه يمكن تنفيذ العديد منها بالتوازي. سنستخدم مخطّط النشاط التالي لاكتشاف سبب محتمل للتزامن والتفكير فيه.

98 مع أنّ UML قد تلاشى تدريجيًا، إلا أن العديد من مخططاته الفردية لا تزال موجودة بأي وسيلة، بما في ذلك مخطط النشاط المفيد للغاية. لمزيد من المعلومات حول جميع أنواع مخططات UML، راجع UML Distilled: دليل موجز إلى لغة نمذجة الكائنات القياسية [Fow04]



يمكن أن تلفت هذه الصورة الانتباه لرؤية مكان وجود التبعيات فعليًا. في هذه الحالة، يمكن أن تحدث جميع المهام ذات المستوى الأعلى (1 و2 و4 و10 و11) بشكل متساير مقدمًا. ويمكن أن تحدث المهام 3 و5 و6 بالتوازي في وقت لاحق. إذا كنت في مسابقة صنع بينا كولادا، فقد تؤدي هذه التحسينات إلى إحداث فرق كبير.

فرص التساير

تُظهر مخططات الأنشطة المجالات المحتملة للتساير، ولكن ليس لديها ما تقوله فيما إذا كانت هذه المناطق تستحق الاستغلال. على سبيل المثال، في مثال بينا كولادا، سيحتاج النادل خمسة أذرع ليتمكن من تشغيل جميع المهام الأوليّة المحتملة في وقت واحد.

وهنا يأتي دور التصميم. عندما ننظر إلى الأنشطة، ندرك أن الرَقَم 8، التسييل، سيستغرق دقيقة. خلال ذلك الوقت، يمكن للنادل الحصول على الأكواب والمظلات (النشاطان 10 و 11) وربما لا يزال هناك وقت لخدمة عميل آخر.

تنسيق أسرع

هذا الكتاب مكتوب بنص واضح (plain text). لبناء النسخة المراد طباعتها، أو الكتاب الإلكتروني، أو أيًا كان، يتم تغذية هذا النص بواسطة سلسلة تنفيذ (pipeline) من المعالجات. يبحث بعضها عن تراكيب معيّنة (اقتباسات المراجع، وإدخالات الفهرسة، وترميز خاص للنصائح، وما إلى ذلك). تعمل معالجات أخرى على المستند كليًا.

يجب على العديد من المعالجات في مسار التنفيذ الوصول إلى المعلومات الخارجيّة (قراءة المِلَفّات وكتابة المِلَفّات ومسارات التنفيذ عبر البرامج الخارجيّة). كل هذا العمل البطيء نسبيًا يمنحنا الفرصة لاستغلال التساير: في الواقع تُنفّذ كل خطوة في مسار التنفيذ بشكل متساير، تقرأ من الخطوة السابقة وتكتب إلى الخطوة التالية.

إضافة إلى ذلك، بعض أجزاء العمليّة تستخدم المعالج بكثافة نسبيًا. واحدة منها هي تحويل الصيغ الرياضيّة. لأسباب تاريخيّة مختلفة، يمكن أن تستغرق كل معادلة ما يصل إلى 500 مللي ثانية للتحويل. لتسريع الأمور، نستفيد من التوازي. ونظرًا لأن كل صيغة مستقلة عن الأخرى، فإننا نحول كل منها في عملية موازية خاصة بها وجمع النتائج مرّة أخرى إلى الكتاب بمجرّد توفرها.

نتيجة لذلك، يُبني الكتاب بشكل أسرع بكثير على أجهزة متعددة النوى.

(ونعم، لقد اكتشفنا فعلًا عددًا من أخطاء التساير في مسار التنفيذ على طول الطريق ...)

وهذا ما نبحث عنه عندما نصمم التساير. نأمل في العثور على الأنشطة التي تستغرق وقتًا، ولكن ليس الوقت في شفرتنا. الاستعلام عن قاعدة بيانات، والوصول إلى خدمة خارجيّة، في انتظار إدخال المستخدم: كل هذه الأشياء عادة ما تعطّل برنامجنا لحين اكتمالها. وهذه كلها فرص للقيام بشيء أكثر إنتاجية من إهدار وقت وحدة المعالجة المركزية.

فرص التوازي

تذكّر الفرق: التساير هو آليّة برمجيّة، والتوازي هو مصدر قلق للأجهزة. إذا كان لدينا العديد من المعالجات، إما محليًا أو عن بُعد، فعندئذ إذا كان بإمكاننا تقسيم العمل فيما بينها، فيمكننا بذلك تقليل الوقت الإجمالي الذي تستغرقه الأشياء.

الأشياء المثالية لتقسيم هذه الطريقة هي أجزاء من العمل مستقلة نسبيًا - حيث يمكن لكل منها المضي قدمًا دون انتظار أي شيء من الأجزاء الأخرى. النمط الشائع هو أخذ جزء كبير من العمل، وتقسيمه إلى قطع مستقلة، ومعالجة كل منها بالتوازي، ثم دمج النتائج.

هناك مثال مثير للاهتمام على ذلك في الممارسة العمليّة وهو الطريقة التي يعمل بها المترجم للغة الإكسير. عندما يبدأ، يقسّم المشروع الذي يقوم ببنائه إلى وحدات، ويترجم كل منها بالتوازي. في بعض الأحيان تعتمد الوحدة على وحدة أخرى، وفي هذه الحالة تتوقف ترجمتها مؤقتًا حتى تتاح نتائج بناء الوحدة الأخرى. عندما تكتمل وحدة المستوى الأعلى، فهذا يعني أنه تمت تَرْجَمَة جميع التبعيات. والنتيجة هي تَرْجَمَة سريعة يستفيد من جميع النوى المتاحة.

تحديد الفرص هو الجزء السهل

بالعودة إلى تطبيقاتك. لقد حدّدنا الأماكن التي ستستفيد فيها من التساير والتوازي. الآن للجزء الصعب: كيف يمكننا تنفيذه بأمان. وهو موضوع بقيّة الفصل.

الأقسام ذات الصلة

- الموضوع 10، التعامدية، في الصفحة 62
- الموضوع 26، كيفية موازنة الموارد، في الصفحة 139
- الموضوع 28، الفصل، في الصفحة 151
- الموضوع 36، السبّورات، صفحة 211

تحديات

- كم عدد المهام التي تؤديها بالتوازي عندما تستعد للعمل في الصباح؟ هل يمكنك التعبير عن هـذا في مخطـط نشـاط UML؟ هل يمكنك وِجدان طريقة للاستعداد بشكل أسرع عن طريق زيادة التساير؟

الموضوع 34. الحالة المشتركة حالة غير صحيحة

أنت في مطعمك المفضل. تنتهي من دورتك التعليميّة الرئيسة، وتسأل نادلك عما إذا كان هناك فطيرة تفاح متبقية. ينظر من فوق كتفه، ويرى قطعة واحدة في صندوق العرض، ويقول نعم. تطلبها وتتنهد برضا.

وفي الوقت نفسه، وعلى الجانب الآخر من المطعم، يسأل عميل آخر نادلتهم عن نفس السؤال. تبحث هي أيضًا، وتؤكّد أن هناك قطعة، وذاك الزَّبُون يطلبها.

سيصاب أحد الزبائن بخيبة أمل.

بدل صندوق العرض إلى حساب مصرفي مشترك، وحول أفراد طاقم الخدمة (النذل) إلى أجهزة نِقَاط البيع. قرّرت أنت وشريكك شراء هاتف جديد في نفس الوقت، ولكن هناك ما يكفي فقط في الحساب لهاتف واحد. أحدكم - المصرف أو المتجر أو أنت - لن يكون سعيدًا جدًا.

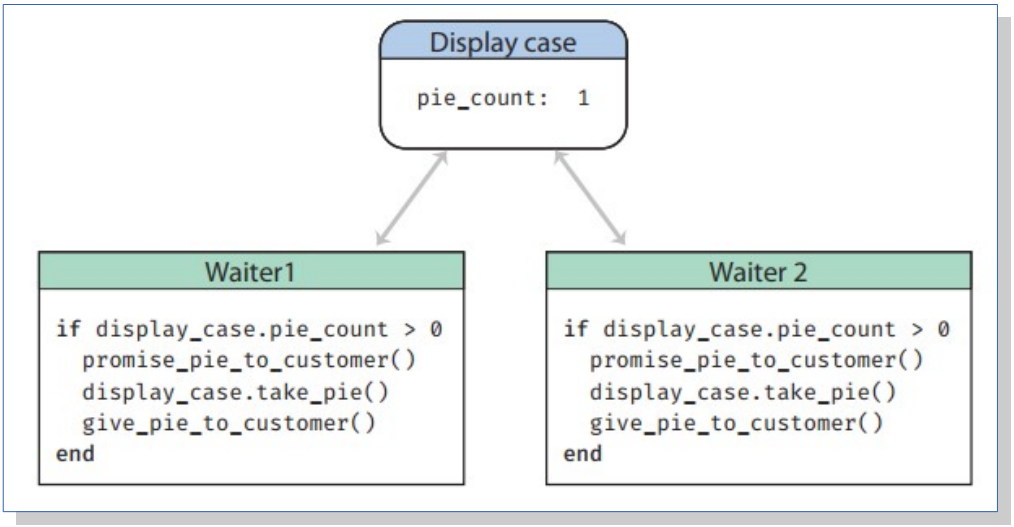
نصيحة ٥٧	**الحالة المشتركة هي حالة غير صحيحة.**

المشكلة هي في الحالة المشتركة. نظر كل خادم في المطعم في صندوق العرض دون النظر إلى الآخر. نظر كل جهاز نقطة بيع في رصيد الحساب دون النظر إلى الجهاز الآخر.

التحديثات غير القابلة للتجزئة"غير ذرية" Nonatomic[99]

دعونا نلقي نظرة على مثال العشاء كما لو كان شِفرة:

99 **[المترجم]** تكون العملية التي تعمل على الذاكرة المشتركة ذرية (atomic) إذا اكتملت في خطوة واحدة بالنسبة للمسارات الأخرى. عندما يتم تنفيذ مخزن ذري على متغير مشترك، لا يمكن لأي مؤشر ترابط آخر أن يلاحظ التعديل نصف الكامل. عندما يتم تنفيذ حمل ذري على متغير مشترك، فإنه يقرأ القيمة بِرُمَّتها كما ظهرت في لحظة واحدة من الزمن. الأحمال والمخازن غير الذرية لا تقدم تلك الضمانات. دون هذه الضمانات، ستكون البرمجة الخالية من القفل مستحيلة، حيث لا يمكنك أبدًا السماح للمسارات المختلفة بمعالجة متغير مشترك في نفس الوقت. يمكننا صياغتها كقاعدة: في أي وقت يعمل فيه مساران على متغير مشترك بشكل متزامن، وتقوم إحدى هذه العمليات بالكتابة، يجب أن يستخدم كلا المسارين العمليات الذرية.



يعمل النادلان بشكل متساير (وفي الحياة الواقعيّة بالتوازي). دعونا نلقي نظرة على شفراتهم:

```
if display_case.pie_count > 0
  promise_pie_to_customer()
  display_case.take_pie()
  give_pie_to_customer()
end
```

يحصل النادل 1 على العدد الحالي للفطائر، ويجد أنه يساوي واحد. يَعِد الزَّبُون بالفطيرة. ولكن عند هذه النقطة (اللحظة)، تعمل النادلة 2. هي أيضا ترى أن عدد الفطائر هو واحد وتقدم نفس الوعد لزبائنها. ثم يلتقط أحدهما آخر فطيرة، ويدخل النادل الآخر في نوع من حالة الخطأ (التي ربما تنطوي على الكثير من التذلل).

المشكلة هنا ليست أن عمليتين يمكن أن تكتبا إلى نفس الذاكرة. تكمن المشكلة في أنه لا يمكن لأي من العمليتين أن تضمن أن وجهة نظرها لتلك الذاكرة متسقة. بشكل فعّال، عندما ينفّذ النادل display_case.pie_count()، يقوم بنسخ القيمة من صندوق العرض (display case) إلى ذاكرته. إذا تغيرت القيمة في صندوق العرض، فهذا يعني أن ذاكرتهم (التي يستخدمونها لاتخاذ القرارات) أصبحت قديمةً الآن.

يحدث كل هذا لأن عملية جلب ثم تحديث عدد الفطائر ليست عملية قابلة للتجزئة: يمكن أن تتغير القيمة الأساسيّة في الوَسَط.

فكيف نجعلها قابلة للتجزئة "ذريّة"؟

الإشارات وأشكال الإقصاء المتبادل الأخرى

الإشارة (semaphore) هي ببساطة شيء لا يملكه سوى شخص واحد في كل مرّة. يمكنك إنشاء إشارة، ثم استخدامها للتحكم في الوصول إلى بعض الموارد الأخرى. في مثالنا، يمكننا إنشاء إشارة للتحكم في الوصول إلى صندوق الفطيرة، وتبني الاتفاقيّة التي تُتيح لأي شخص يريد تحديث محتويات حالة الفطيرة أن يفعل ذلك فقط إذا كان يحمل تلك الإشارة.

لنفترض أن المطعم قرّر إصلاح مشكلة الفطائر باستخدام إشارة فعليّة (ماديّة). يضعون جنّي بلاستيكي (plastic Leprechaun) على صندوق الفطيرة. قبل أن يتمكن أي نادل من بيع فطيرة، يجب أن يمسكوا بالجني في أيديهم. بمجرّد الانتهاء من طلبهم (وهو ما يعني تسليم الفطيرة إلى الطاولة)، يمكنهم إعادة الجنّي إلى مكانه لحراسة كنز الفطائر، وليصبح على استعداد للتوسط في الطلب التالي.

دعونا نلقي نظرة على هذا في التعليمات البرمجية. تقليديًا، كانت عملية الاستيلاء على الإشارة تدعى P، وكانت عملية تحريرها تسمى V.[100] واليوم نستخدم مصطلحات مثل القُفْل / *إلغاء القُفْل*، و*المطالبة* / *التحرير*، وما إلى ذلك.

```
case_semaphore.lock()
if display_case.pie_count > 0
  promise_pie_to_customer()
  display_case.take_pie()
  give_pie_to_customer()
end
case_semaphore.unlock()
```

تفترض هذه الشِفرة أنه تم إنشاء إشارة وتخزينها فعلًا في متغير case_semaphore.

دعنا نفترض أن كلا النادلين ينفّذان الشِفرة في نفس الوقت. كلاهما يحاولان قفل الإشارة، ولكن ينجح واحد منهما فقط. الذي يحصل على الإشارة يستمر في العمل كالمعتاد. والذي لا يحصل على الإشارة يُعلّق حتى تصبح الإشارة متاحة (ينتظر النادل ...). عندما يكمل النادل الأول إنجاز الطلبية يفتح قُفْل الإشارة ويتابع النادل الثاني في العمل. يرون الآن أنه لا يوجد فطيرة في الصندوق، ويعتذرون للزَّبُون.

هناك بعض المشكلات مع هذا النهج. ولعل الأمر الأكثر أهميّة هو أن هذه الفكرة لا تنجح إلاّ لان كل من يصل إلى صندوق الفطائر يوافق على اتفاقية استخدام الإشارة. إذا نسي شخص ما (أي إن بعض المطورين يكتب شِفرة لا تتبع الاتفاقية)، فسنعود عندها إلى الفوضى.

اجعل الموارد قابلة للمناقلة

التصميم الحالي رديء لأنه يفوض مسؤولية حماية الوصول إلى صندوق الفطائر للأشخاص الذين يستخدمونها. دعنا نغيرها لمركزة ذاك التحكم. للقيام بذلك، علينا تغيير واجهة برمجة التطبيقات حتى يتمكن النُدُل (جمع نادل) من التحقّق العدد وأخذ شريحة من الفطيرة باستدعاء واحد:

```
slice = display_case.get_pie_if_available()
if slice
  give_pie_to_customer()
end
```

لجعل هذا يعمل، نحتاج إلى كتابة طريقة تعمل كجزء من صندوق العرض نفسه:

100 يأتي الاسمان P و V من الأحرف الأولى للكلمات الهولندية. ولكن هناك بعض النقاش حول الكلمات بالضبط. اقترح مخترع التقنية، Edsger Dijkstra، كلاً من التمرير passering والتمديد prolaag لـ P، و vrijgave وربما verhogen لـ V.

```
def get_pie_if_available()                #####
    if @slices.size > 0                      #
      update_sales_data(:pie)                        #
      return @slices.shift                  #
    else                                    # شِفرة خاطئة!
      False                                 #
    end                                     #
end                                 ####
```

توضح هذه الشِفرة مفهومًا خاطئًا شائعًا. لقد نقلنا وصول الموارد إلى مكان مركزي، ولكن لا يزال من الممكن استدعاء طريقتنا من عدة مسارات (threads) متسايرة، لذلك ما زلنا بحاجة إلى حمايتها باستخدام إشارة (semaphore):

```
def get_pie_if_available()
  @case_semaphore.lock()
  if @slices.size > 0
     update_sales_data(:pie)
    return @slices.shift
  else
     false
   end
   @case_semaphore.unlock()
end
```

حتى هذه الشِفرة قد لا تكون صحيحة. إذا أثار pdate_sales_data استثناءً، فلن يُلغى قُفْل الإشارة أبدًا، وستتوقّف جميع عمليات الوصول المستقبلية لصندوق الفطائر إلى أجل غير مسمّى. نحن بحاجة لمعالجة ذلك:

```
def get_pie_if_available()
   @case_semaphore.lock()
   try {
      if @slices.size > 0
          update_sales_data(:pie)
        return @slices.shift
      else
        false
      end
   }
  ensure {
  @case_semaphore.unlock()
}
end
```

نظرًا لأن هذا يُعدّ خطأ شائعا، فإن العديد من اللغات توفر لك مكتبات تتعامل مع هذا الأمر:

```
def get_pie_if_available()
  @case_semaphore.protect() {
    if @slices.size > 0
       update_sales_data(:pie)
      return @slices.shift
    else
       false
    end
  }
end
```

المناقلات متعددة الموارد

قام مطعمنا بتركيب ثلاجة الآيس كريم. إذا طلب الزَّبُون وضع الفطائر، فسيحتاج النادل إلى التحقّق توفر كل من الفطيرة والآيس كريم.

يمكننا تغيير شِفرة النادل إلى شيء مثل:

```
slice = display_case.get_pie_if_available()
scoop = freezer.get_ice_cream_if_available()
if slice && scoop
  give_order_to_customer()
end
```

لكن هذا لن ينجح. ماذا يحدث إذا طالبنا بشريحة من الفطائر، ولكن عندما نحاول الحصول على غَرفة من الآيس كريم نجد أنه لا يوجد أي منها؟ نحن الآن نترك بعض الفطائر التي لا يمكننا فعل أي شيء بها (لأن عملائنا يجب أن يكون لديهم الآيس كريم معها). وحقيقة أننا نحتفظ بالفطيرة تعني أنها ليست في الصندوق، لذا فهي غير متاحة لبعض الزبائن الآخرين الذين (كونهم أصوليون) لا يريدون الآيس كريم معها.

يمكننا إصلاح ذلك عن طريق إضافة طريقة إلى الحالة التي تتيح لنا إرجاع شريحة من الفطيرة. سنحتاج إلى إضافة معالجة استثناء لضمان عدم الاحتفاظ بالموارد في حالة فشل شيء ما:

```
 slice = display_case.get_pie_if_available()
if slice
  try {
       scoop = freezer.get_ice_cream_if_available()
      if scoop
          try {
              give_order_to_customer()
          }
        rescue {
            freezer.give_back(scoop)
          }
         end
      }
     rescue {
        display_case.give_back(slice)
      }
end
```

مرّة أخرى، هذا أقل من مثالي. الشِفرة الآن قبيحة حقًا: من الصعب معرفة ما تفعله فعلًا: يتم إخفاء منطق الأعمال في جميع أعمال التدبير المنزلي.

في السابق، أصلحنا ذلك عن طريق نقل شِفرة معالجة المورد إلى المورد نفسه. هنا، وعلى الرغم من ذلك، لدينا موردان. هل نضع الشِفرة في صندوق العرض أم في الثلاجة؟

نعتقد أن الجواب "لا" لكلا الخيارين. النهج العملي هو القول بأن "فطيرة التفاح" هي مصدرها. سننقل هذه الشِفرة إلى وحدة نمطيّة جديدة، وبعد ذلك يمكن للعميل أن يقول "أحضر لي فطيرة التفاح مع الآيس كريم" وإما أن ينجح أو يفشل.

بالطبع، في العالم الحقيقي، من المحتمل أن يكون هناك العديد من الأطباق المركبة مثلها، ولا ترغب في كتابة وحدات جديدة لكل منها. بدلاً من ذلك، ربما تريد نوعًا من عناصر القائمة تحتوي على مراجع لمكوناتها، ومن ثم يكون لديك طريقة get_menu_item عامّة تتغير بتغير كل منها.

التحديثات غير القابلة للتناقل

يتم إيلاء الكثير من الاهتمام للذاكرة المشتركة كمصدر لمشكلات التساير، ولكن في الواقع يمكن أن تظهر المشكلات في أي مكان حيث تشارَك شِفرة التطبيق الخاص بك في الموارد القابلة للتغيير: المِلَفّات وقواعد البيانات والخِدمَات الخارجيّة، وما إلى ذلك. عندما يتمكن مثيلين أو أكثر من الشِفرة من الوصول إلى مورد ما في نفس الوقت، فأنت تعاين مشكلة محتملة.

في بعض الأحيان، لا يكون المورد واضحًا. في أثناء كتابة هذا الإصدار من الكتاب، حدثنا سلسلة الأدوات للقيام بالمزيد من العمل بالتوازي باستخدام المسارات (threads). وهذا تسبب في فشل البناء، ولكن بطرق غريبة وأماكن عشوائية. كان الخيط المشترك بين جميع الأخطاء هو أنه تعذّر العثور على المِلَفّات أو الأدلة، مع أنّها كانت في المكان الصحيح تمامًا.

لقد تتبعنا ذلك إلى مكانين في الشِفرة، مما غيّر الدليل الحالي مؤقتًا. في الإصدار غير المتوازي، كانت حقيقة أن هذه الشِفرة أعادت الدليل إلى الوراء جيدًا بما فيه الكفاية. ولكن في النسخة المتوازية، سيغيّر أحد المسارات الدليل، وبعد ذلك، في هذا الدليل، سيبدأ مسار آخر في العمل. يتوقّع هذا المسار أن يكون في الدليل الأصليّ، ولكن نظرًا لأن الدليل الحالي مشترك بين المسارات، فلم يكن الحال كذلك.

تثير طبيعة هذه المشكلة نصيحة أخرى:

نصيحة ٥٨	**الفشل العشوائي غالبًا ما يكون عبارة عن مشكلات تساير.**

أنواع أخرى من الوصول الحصري

معظم اللغات لديها دعم مكتبة لبعض أنواع الوصول الحصري إلى الموارد المشتركة. قد يطلقون عليه اسم أدوات المزامنة "mutexes"[101] (للاستبعاد المتبادل) أو المراقبين أو الإشارات. يتم تنفيذ كل ذلك كمكتبات.

ومع ذلك، فإن بعض اللغات لديها دعم التساير المدمج في اللغة نفسها. تفرض رست (Rust)، على سبيل المثال، مفهوم ملكيّة البيانات؛ متغير أو معامل واحد فقط يمكنها الاحتفاظ بمرجع إلى أي قطعة معيّنة من البيانات القابلة للتغيير في وقت واحد.

يمكنك أيضًا أن تجادل بأن اللغات الوظيفيّة، مع ميلها إلى جعل جميع البيانات غير قابلة للتغيير، تجعل التزامن أبسط. ومع ذلك، ما زالت تواجه نفس التحدّيات، لأنه في مرحلة ما تضطر إلى دخول في العالم الحقيقي القابل للتغيير.

101 **[المترجم]** استبعاد التشارك أو إقصاء التشارك (بالإنجليزية: Mutual exclusion :mutex) أداة مزامنة تستعملها بعض الخوارزميات المستخدمة في البرمجة لتجنب الاستخدام المتزامن للموارد المشتركة، مثل الوصول لمتغير عام، قد تقوم به بعض المقاطع الحرجة. المقطع الحرج هو جزء من البرمجية حيث تسعى عملية أو مسار للوصول فيه إلى مورد مشترك. المقطع الحرج في حد ذاته ليس آلية أو خوارزمية للاستبعاد المتبادل. خوارزميات استبعاد التشارك تسمح بضبط الوصول للبيانات. مثلًا، حين يتم تنفيذ روتين معين مرّة واحدة في نفس الوقت فلا يقبل التوازي.

دكتور، إنه يؤلم ...

إذا لم تخرج بأي شيء آخر من هذا القسم، فخذ هذا: التساير في بيئة موارد مشتركة أمر صعب، وإدارته بنفسك محفوفة بالتحدّيات.

وهذا هو السبب في أننا نوصي باستخدام النكتة القديمة:

دكتور، يؤلمني عندما أفعل ذلك.

حسنًا لا تفعل ذلك.

يقترح القسمان التاليان طرقًا بديلة للحصول على فوائد التساير دون ألم.

الأقسام ذات الصلة

- الموضوع 10، التعامدية، في الصفحة 62
- الموضوع 28، الفصل، في الصفحة 151
- الموضوع 38، البرمجة بالمصادفة، في الصفحة 223

الموضوع 35. الممثلون والعمليات

> دون كُتّاب، لن تُكتب القصص،
> دون ممثّلين، لا يمكن إحياء القصص.
> ← انجي ماري ديلسانتي

يُقدّم الممثّلون والعمليات طرقًا مثيرة للاهتمام لتنفيذ التساير دون عبء مزامنة الوصول إلى الذاكرة المشتركة.

ولكن قبل أن نخوض في ذلك، نحتاج إلى تحديد ما نعنيه. وسيبدو هذا أكاديميًا. لا تخف، سنعمل على اجتياز كل ذلك في مدّة قصيرة.

- الممثّل (actor) هو معالج افتراضي مستقل مع حالته المحليّة (والخاصّة). كل ممثل لديه صندوق بريد. عندما تظهر رسالة في صندوق البريد ويكون الممثل خاملاً (idle)، فإنه يبدأ في الحياة ويعالج الرسالة. عند الانتهاء من المعالجة، يعالج رسالة أخرى في صندوق البريد، أو إذا كان صندوق البريد فارغًا، فإنه يعود إلى وضع النوم (sleep).

  عند معالجة الرسالة، يمكن للممثل إنشاء ممثلين آخرين، وإرسال رسائل إلى ممثلين آخرين يعرفهم، وإنشاء حالة جديدة والتي ستصبح بدورها الحالة الحاليّة عند معالجة الرسالة التالية.

- العمليّة (process)، عادةً ما تكون معالج افتراضيّ للأغراض العامّة، وغالبًا ما تُنفّذ بواسطة نظام التشغيل لتسهيل التساير. يمكن أن تقيّد العمليات (بالاتفاقيّة) لتتصرف مثل الممثلين، وهذا هو نوع العمليّة التي نعنيه هنا.

الممثلون يجب أن يكونوا متسايرين

هناك بعض الأشياء التي لن تجدها في تعريف الممثلين:

- لا يوجد شيء واحد متحكّم فيه. لا شيء يحدّد ما سيحدث بعد ذلك، أو ينسق نقل المعلومات من البيانات الأوليّة إلى الإخراج النهائي.
- الحالة الوحيدة في النظام محفوظة في الرسائل وفي الحالة المحليّة لكل ممثل. لا يمكن فحص الرسائل إلا بواسطة قراءتها من قبل المستلم، ولا يمكن الوصول إلى الحالة المحليّة خارج الممثل.
- جميع الرسائل هي باتجاه واحد – مفهوم الرد غير موجود. إذا كنت تريد أن يقوم أحد الممثلين بإرجاع استجابة، فعليك تضمين عنوان صندوق البريد الخاص بك في الرسالة التي ترسلها إليه، وسوف يقوم (في النهاية) بإرسال الرّد كمجرّد رسالة أخرى إلى صندوق البريد هذا.
- يقوم الممثّل بمعالجة كل رسالة حتى اكتمالها، ويعالج رسالة واحدة فقط في كل مرّة.

ونتيجة لذلك، ينفَّذ الممثّلون بشكل متساير وغير متزامن ولا يشتركون في شيء. إذا كان لـديك مـا يكفي من المعالجـات الماديّـة، يمكنك تشغيل ممثل على كل منها. إذا كان لديك معالج واحد، فيمكن في وقت ما من التشغيل معالجـة تبـديل السـياق بينهمـا. في كلتـا الحالتين، فإن الشِفرة التي تعمل في الممثلَين هي نفسها.

نصيحة ٥٩	استخدم الممثلين من أجل التساير دون حالة مشتركة.

ممثل بسيط

دعنا ننفّذ العشاء باستخدام الممثلين. في هذه الحالة، سيكون لدينا ثلاثة ممثلين (الزَّبُون والنادل وصندوق الفطائر).

سيبدو تدفق الرسالة الإجمالي كما يلي:

- نحن (كنوع من الوجود الخارجي يشبه القدرة العظيمة) نخبر الزبائن أنهم جائعون
- استجابةً لذلك، سيطلبون من النادل الحصول على فطيرة
- سيطلب النادل من صندوق الفطائر الحصول على بعض الفطائر من أجل الزَّبُون
- إذا كان صندوق الفطائر ما يزال يحتوي على فطيرة متاحة، فسوف يرسـلها إلى الزَّبُـون، ويُخطـر النـادل أيضًـا بإضـافتها إلى الفاتورة
- في حال عدم وجود فطيرة، يخبر الصندوق النادلَ بذلك، ويعتذر النادل للزَّبُون

لقد اخترنا تنفيذ الشِـفرة في جافـا سـكريبت باسـتخدام مكتبـة Nact.[102] أضـفنا مُغلِّفـا صـغيرًا إلى هـذا يـتيح لنـا كتابـة الممثلين ككائنات بسيطة، حيث تكون المفاتيح (keys) هي أنواع الرسائل التي يتلقاها والقيم (values) هي دوال تُنفّذ عند تلقي هذه الرسالة المعيّنة. (معظم أنظمة الممثلين لديهم بنية مماثلة، لكن التفاصيل تعتمد على اللغة المضيفة.)

لنبدأ مع الزَّبُون. يمكن للزَّبُون استقبال ثلاث رسائل:

- أنت جائع (مرسلة من السياق الخارجي)
- هناك فطيرة على الطاولة (مرسلة من قبل صندوق الفطائر)
- عذرًا، لا يوجد فطيرة (مرسلة من النادل)

إليك الشِفرة:

concurrency/actors/index.js

```
const customerActor = {
    'hungry for pie': (msg, ctx, state) => {
        return dispatch(state.waiter,
                { type: "order", customer: ctx.self, wants: 'pie' })
    },
```

102 https://github.com/ncthbrt/nact

```
    'put on table': (msg, ctx, _state) =>
        console.log(`${ctx.self.name} sees "${msg.food}" appear on the table`),
    'no pie left': (_msg, ctx, _state) =>
        console.log(`${ctx.self.name} sulks…`)
}
```

الحالة المثيرة للاهتمام هي عندما نتلقى رسالة "تواق للفطيرة"، حيث نرسل رسالة إلى النادل. (سنرى كيف يعرف الزَّبُون عن ممثل النادل قريبًا.)

إليك شِفرة النادل:

concurrency/actors/index.js

```
const waiterActor = {
   "order": (msg, ctx, state) => {
     if (msg.wants == "pie") {
        dispatch(state.pieCase,
          { type: "get slice", customer: msg.customer, waiter: ctx.self })
      }
     else {
         console.dir(`Don't know how to order ${msg.wants}`);
     }
  },
  "add to order": (msg, ctx) =>
      console.log(`Waiter adds ${msg.food} to ${msg.customer.name}'s order`),
   "error": (msg, ctx) => {
       dispatch(msg.customer, { type: 'no pie left', msg: msg.msg });
       console.log(`\nThe waiter apologizes to ${msg.customer.name}: ${msg.msg}`)
 }
};
```

عندما يتلقى رسالة "الطلب" ('order') من الزَّبُون، فإنه يتحقّق لمعرفة ما إذا كان الطلب من أجل فطائر. إذا كان الأمر كذلك، يرسل طلبًا إلى صندوق الفطائر، ويمرّر المراجع لنفسه وللزَّبُون.

صندوق الفطائر له حالة: مجموعة من جميع شرائح الفطائر التي يضمها. (مرّة أخرى، نرى كيف يتم إعداد ذلك قريبًا.) عندما يتلقى رسالة "الحصول على شريحة" من النادل، فإنه ينظر فيما إذا كان لديه أي شرائح متبقيّة. إذا وُجِدَت، فإنه يمرّر الشريحة إلى الزَّبُون، ويخبر النادل بتحديث الطلب، ويعيد في النهاية حالة محدّثة، تحتوي على شريحة واحدة أقل. إليك الشِفرة:

concurrency/actors/index.js

```
const pieCaseActor = {
   'get slice': (msg, context, state) => {
     if (state.slices.length == 0) {
        dispatch(msg.waiter,
            { type: 'error', msg: "no pie left", customer: msg.customer })
     return state
      }
     else {
        var slice = state.slices.shift() + " pie slice";
         dispatch(msg.customer,
              { type: 'put on table', food: slice });
         dispatch(msg.waiter,
             { type: 'add to order', food: slice, customer: msg.customer });
     return state;
      }
  }
```

```
}
```

مع أنّك غالبًا ما تجد أن الممثلين بدأوا ديناميكيًا من قبل ممثلين آخرين، في حالتنا سنبقي الأمر بسيطًا ونبدأ الممثلين لدينا يدويًا. سنقوم أيضا بتمرير كل حالة أولية:

- يحصل صندوق الفطائر على القائمة الأولية لشرائح الفطائر التي يحتوي عليها
- سنعطي النادل إشارة إلى حالة الفطيرة
- سنقدم للزبائن مرجع إلى النادل

concurrency/actors/index.js
```
const actorSystem = start();
let pieCase = start_actor(
    actorSystem,
    'pie-case',
    pieCaseActor,
    { slices: ["apple", "peach", "cherry"] });
let waiter = start_actor(
    actorSystem,
    'waiter',
     waiterActor,
     { pieCase: pieCase });

let c1 = start_actor(actorSystem, 'customer1',
     customerActor, { waiter: waiter });
let c2 = start_actor(actorSystem, 'customer2',
      customerActor, { waiter: waiter });
```

وأخيرًا ننطلق. زبائننا جشعون. يطلب الزَّبُون 1 ثلاث شرائح من الفطيرة، بينما يطلب الزَّبُون 2 الحصول على شريحتين:

concurrency/actors/index.js
```
dispatch(c1, { type: 'hungry for pie', waiter: waiter });
dispatch(c2, { type: 'hungry for pie', waiter: waiter });
dispatch(c1, { type: 'hungry for pie', waiter: waiter });
dispatch(c2, { type: 'hungry for pie', waiter: waiter });
dispatch(c1, { type: 'hungry for pie', waiter: waiter });
sleep(500)
    .then(() => {
       stop(actorSystem);
     })
```

عندما نشغّلها، يمكننا رؤية الممثلين يتواصلون .[103] قد الترتيب الذي تراه مختلفًا:

```
$ node index.js
يرى الزَّبُون 1 "شريحة فطيرة التفاح" تظهر على الطاولة

يرى الزَّبُون 2 "شريحة فطيرة الخوخ" تظهر على الطاولة

يضيف النادل شريحة فطيرة التفاح إلى طلب الزَّبُون 1
```

103 لتشغيل هذه الشِفرة، ستحتاج أيضًا إلى دوال المُغلّف، والتي لا تظهر هنا. يمكنك تنزيلها من
https://media.pragprog.com/titles/tpp20/code/concurrency/actors/index.js

يضيف النادل شريحة فطيرة الخوخ إلى طلب الزَّبُون 2

يرى الزَّبُون 1 "شريحة فطيرة الكرز" تظهر على الطاولة

يضيف النادل شريحة فطيرة الكرز إلى طلب الزَّبُون 1

يعتذر النادل للزَّبُون 1: لم يتبق فطيرة

يحرد الزَّبُون 1 ...

يعتذر النادل للزَّبُون 2: لم يتبق فطيرة

يحرد الزَّبُون 2 ...

لا يوجد تساير واضح

في نموذج الممثل، ليست هناك حاجة لكتابة أي شِفرة للتعامل مع التساير، حيث لا توجد حالة مشتركة. ليست هناك حاجة أيضًا إلى كتابة شِفرة في منطق نهاية إلى نهاية صريح "افعل هذا، افعل ذلك"، حيث يعمل الممثلون على حلها بأنفسهم بناءً على الرسائل التي يتلقونها.

لا يوجد أي ذكر للهندسة الأساسيّة. تعمل هذه المجموعة من المكونات بشكل جيّد بنفس القدر على معالج واحد أو على نوى متعددة أو على أجهزة شبكية متعددة.

إرلانج تضبط الأمر

تعدّ لغة إرلانج (Erlang) ووقت التشغيل أمثلة رائعة على تنفيذ الممثل (مع أنّ مخترعي إرلانج لم يقرؤوا ورقة الممثل الأصليّة). تستدعي إرلانج عمليات الممثلين، لكنها ليست عمليات نظام تشغيل منتظمة. بدلاً من ذلك، وتمامًا مثل الممثلين الذين ناقشناهم، فإن عمليات إرلانج خفيفة الوزن (حيث يمكنك تشغيل الملايين منها على جهاز واحد)، ويتواصلون عن طريق إرسال الرسائل. وكل واحد منهم معزول عن الآخرين، لذلك لا يوجد تشارك للحالة.

إضافة إلى ذلك، يطبّق وقت تشغيل إرلانج نظام إشراف، يدير عمر العمليات، ويحتمل أن يعيد تشغيل عملية ما أو مجموعة من العمليات في حالة الفشل. كما يوفّر إرلانج أيضًا تحميل الشِفرة الساخنة: يمكنك استبدال الشِفرة في نظام شغّال دون إيقاف هذا النظام. ويدير نظام إرلانج بعضًا من أكثر الشفرات الموثوقة في العالم، وغالبًا ما يشير إلى توافريّه شبه تامة "تسع تسعات" (99.9999999%)

لكن لغة إرلانج (وهي من ذريّة Elixir) ليست فريدة - هناك تطبيقات ممثل (actor implementations) لمعظم اللغات. فكّر في استخدامها للتطبيقات المتسايرة.

الأقسام ذات الصلة

- الموضوع 28، الفصل، في الصفحة 151
- الموضوع 30، برمجة التحويل، في الصفحة 169
- الموضوع 36، السبّورات، صفحة 211

تحديات

- هل لديك حاليًا شِفرة تستخدم الاستبعاد المتبادل لحماية البيانات المشتركة. لماذا لا تجرب نموذجًا أوليًا من نفس الشِفرة المكتوبة باستخدام الممثلين؟
- تدعم شِفرة الممثل الخاصة بالعشاء فقط طلب شرائح الفطيرة. وسعها للسماح للزبائن بطلب وضع الفطائر، مع وكلاء منفصلين يديرون شرائح الفطيرة ومغارف الآيس كريم. رتّب الأشياء بحيث تتعامل مع الموقف الذي ينفَّد فيه أحدهما أو الآخر.

الموضوع 36. السبّورات

> إن الكتابة على الجدار ...[104]
> ← دانيال 5 (المرجع)

تأمّل كيف يمكن للمحقّقين استخدام السبّورة لتنسيق وحل التحقيق في جرائم القتل. تبدأ كبيرة المفتشين بوضع سَبُّورَة كبيرة في غرفة الاجتماعات. تكتب سؤالًا واحدًا عليها:

> *دمبتي* (شخصيّة مذكّرة لبيضة[105]): هل كان ذلك حادثًا؟ أم قتلًا؟

هل سقط *هامبتي* حقا، أم تم دفعه؟ قد يقدّم كل محقّق مساهمات في لغز القتل المحتمل هذا عن طريق إضافة حقائق، أقوال شهود، أي أدلّة جنائية قد تظهر، وما إلى ذلك. ومع تراكم البيانات، قد يلاحظ المحقّق وجود صلة ما وينشر تلك الملاحظة أو التكهّن أيضًا. تستمر هذه العملية، على مدى كل نوبات العمل، مع العديد من الأشخاص والوكلاء المختلفين، حتى يتم إغلاق القضية. هناك عيّنة للسَّبُّورَة معروضة في الشكل في الصفحة التالية.

بعض الميزات الرئيسة لنهج السَّبُّورَة هي:

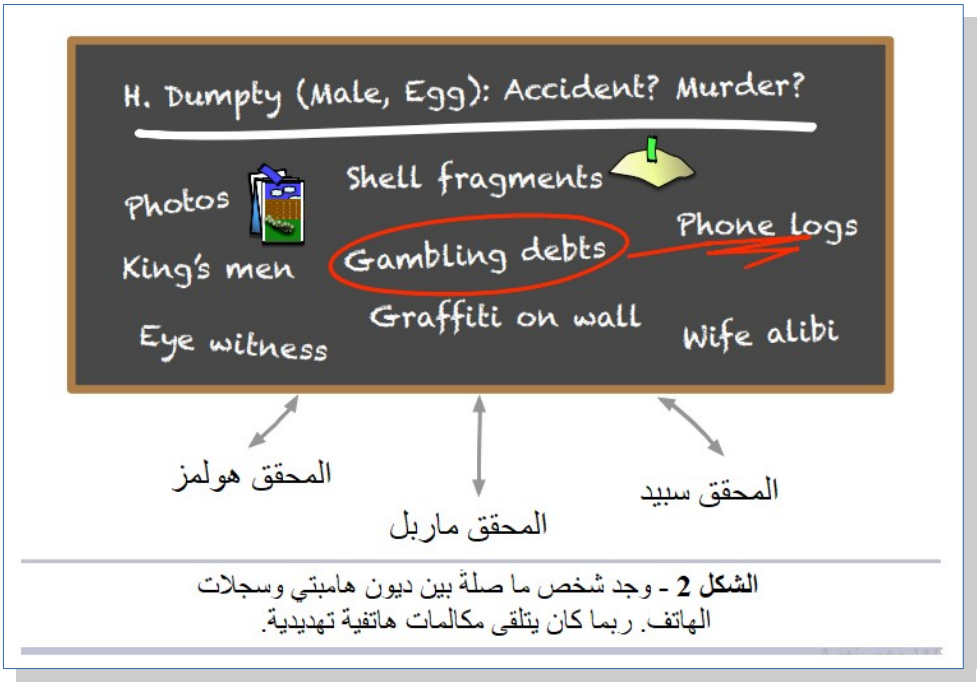
- لا يحتاج أيّ من المحقّقين إلى معرفة وجود أي محقّق آخر - فهم يراقبون السَّبُّورَة للحصول على معلومات جديدة، ويضيفون نتائجهم.
- قد يتم تدريب المحقّقين في تخصّصات مختلفة، وقد يكون لديهم مستويات مختلفة من التعليم والخبرة، وقد لا يعملون حتى في نفس المنطقة. إنهم يشتركون في الرغبة في حلّ القضية، ولكن هذا كل شيء.
- قد يأتي ويذهب محقّقون مختلفون خلال العملية، وقد يعملون في نوبات مختلفة.
- لا توجد قيود على ما يمكن وضعه على السَّبُّورَة. قد تكون صورًا وجملًا وأدلّةً ماديّة، وما إلى ذلك.

104 **[المترجم]** يستخدم هذا المثل عادة للدلالة على وجود إشارات واضحة على أن الحالة ستصبح أصعب أو وجود مفاجأة غير سارّة قريبة.

105 **[المترجم]** هامبتي دمبتي (بالإنجليزية: Humpty Dumpty) هي شخصيّة خياليّة على شكل بيضة تمشي على حائط لا متناهي، ترتبط هذه الشخصية بأغنية الأطفال الشهيرة. ففي دقيقة ما، كان دمبتي يجلس ببراءة على الحائط، يهتم بشؤونه الخاصّة، وفي الدقيقة التالية وجد جانبًا مرميًا على الأرض، وقد تحطّم إلي مليون قطعة. لم يتمكن جميع رجال الملك من تجميع هامبتي مرّة أخرى.الشيء الوحيد المتبقي هو فحص المشهد ومعرفة ما إذا كان ذلك حادثًا أم انتحارًا؟ هل قام أحدهم بتدبير مؤامرة ضد هامبتي دمبتي؟

هذا شكل من أشكال تساير عدم *التدخل* (*laissez faire*). حيث المحقّقون هم عمليات، وكلاء، ممثلون مستقلون، وما إلى ذلك. يُخزّن البعض حقائق على السَّبُورَة. ويزيل البعض الآخر الحقائق عنها، ربما يجمعها أو يعالجها، ويضيف المزيد من المعلومات إلى السَّبُورَة. تساعدهم السَّبُورَة تدريجيًا في الوصول إلى نتيجة.

استُخدمت أنظمة السَّبُورَة القائمة على الحاسوب في الأصل في تطبيقات الذكاء الاصطناعي حيث كانت المشكلات التي يتعيّن حلّها كبيرة ومعقّدة - التعرّف على الكلام، وأنظمة التفكير المستندة إلى المعرفة، وما إلى ذلك.



الشكل 2 - وجد شخص ما صلة بين ديون هامبتي وسجلات الهاتف. ربما كان يتلقى مكالمات هاتفية تهديدية.

كانت *ليندا لديفيد جيلرتر* (David Gelernter's Linda) أحدى أولى أنظمة السَّبُورَة. حيث كانت تُخزّن الحقائق بنمط قوائم (typed tuples). يمكن للتطبيقات كتابة قوائم (tuples) جديدة في ليندا، والاستعلام عن القوائم الموجودة باستخدام شكل من أشكال مطابقة الأنماط.

في وقت لاحق جاءت الأنظمة الموزعة الشبيهة بالسبورات مثل JavaSpaces و T Spaces. يمكنك باستخدام هذه الأنظمة، تخزين كائنات جافا النشطة - وليس البيانات فقط - على السَّبُورَة، واستردادها عن طريق المطابقة الجزئيّة للحقول (عبر القوالب وحرف البدل[106]) أو عن طريق الأنواع الفرعيّة. على سبيل المثال، افترض أن لديك نوع مؤلف (Author)، وهو نوع فرعيّ من شخص (Person). يمكنك البحث عن السَّبُورَة التي تحتوي على كائنات Person باستخدام قالب Author بحيث تكون قيمة الحقل LastName هي "شكسبير". "ستحصل على بيل شكسبير المؤلّف، ولكن ليس فريد شكسبير البستاني.

نعتقد أن هذه الأنظمة لم تنطلق أبدًا حقًا، لأننا نؤمن جزئيًا أن الحاجة إلى نوع من المعالجة التعاونيّة المتسايرة لم تتطوّر بعد.

106 **[المترجم]** في البرمجيات، يعد حرف البدل نوعًا من العناصر النائبة التي يمثلها حرف واحد، مثل علامة النجمة (*)، والتي يمكن تفسيرها على أنها عدد من المحارف أو سلسلة فارغة. غالبًا ما تُستخدم في عمليات البحث عن المِلَفّات، وبذلك لا يلزم كتابة الاسم بالكامل.

السَّبُورَة في أثناء العمل

لنفترض أننا نكتب برنامجًا لقبول ومعالجة طلبات الرهن أو القروض. إن القوانين التي تحكم هذه المنطقة معقّدة للغاية، حيث الحكومات الفيدرالية وحكومات الولايات والحكومات المحلية لها رأيها. يجب على المقرض أن يُثبت أنه كشف عن أشياء معيّنة، ويجب أن يطلب معلومات معيّنة – ولكن لا يجب طرح أسئلة أخرى معيّنة، وهكذا دواليك.

بالإضافة إلى مشكلات القانون المعمول به، لدينا أيضًا المشكلات التالية التي يجب التعامل معها:

- يمكن أن تصل الردود بأي ترتيب. على سبيل المثال، قد تستغرق الاستعلامات الخاصّة بفحص الائتمان أو البحث عن العنوان مدّة زمنيّة كبيرة، بينما قد تتوفر عناصر مثل الاسم والعنوان على الفور.
- يمكن أن يتم جمع البيانات من قبل أشخاص مختلفين، موزّعين على مكاتب مختلفة، في مناطق زمنيّة مختلفة.
- قد تقوم بعض الأنظمة بجمع بعض البيانات تلقائيًا. قد تصل هذه البيانات بشكل غير متزامن أيضًا.
- ومع ذلك، قد تظلّ بعض البيانات معتمدة على بيانات أخرى. على سبيل المثال، ربما لا تتمكن من بَدْء البحث عن عنوان لسيارة حتى تحصل إثبات الملكية أو التأمين.
- قد يثير وصول بيانات جديدة أسئلة وسياسات جديدة. لنفترض أن فحص الائتمان يعود بتقرير أقل وهجًا؛ الآن أنت بحاجة إلى هذه الأشكال الخمسة الإضافيّة وربما عيّنة دم.

يمكنك محاولة التعامل مع كل تركيبة وظروف ممكنة باستخدام نظام سير العمل. توجد العديد من هذه الأنظمة، ولكن يمكن أن تكون معقّدة ومُجهدة للمبرمج. مع تغير اللوائح، يجب إعادة تنظيم سير العمل: قد يضطر الناس إلى تغيير إجراءاتهم وقد يكون من الضروري إعادة كتابة التعليمات البرمجية المتأصلة.

تُعدّ السَّبُورَة، إلى جانب محرك القواعد الذي يُغلّف المتطلبات القانونية، حلًا أنيقًا للصعوبات الموجودة هنا. ترتيب وصول البيانات غير ذات صلة: عندما تُنشر حقيقة ما، يمكن أن يؤدي إلى قدح القواعد المناسبة. يتم التعامل مع التعليقات بسهولة أيضًا: يمكن أن تُعرض أي مجموعة من القواعد على السَّبُورَة وتتسبّب في قدح المزيد من القواعد القابلة للتطبيق.

نصيحة ٦٠	استخدم السبورات لتنسيق سير العمل.

يمكن أن تكون أنظمة المراسلة شبيهة بالسبورات

في أثناء كتابة هذا الإصدار الثاني، أُنشئت العديد من التطبيقات باستخدام خِدْمَات صغيرة منفصلة، تتواصل جميعها عبر أحد أشكال نظام المراسلة. أنظمة المراسلة هذه (مثل Kafka و NATS) تقوم بأكثر بكثير من مجرّد إرسال البيانات من A إلى B. على وجه الخصوص، فهي توفر الثبات (في شكل سجل الأحداث) والقدرة على استرداد الرسائل بواسطة شكل مطابقة الأنماط. هذا يعني أنه يمكنك استخدامها كنظام سَّبُورَة و / أو كمنصّة يمكنك بواسطتها تشغيل مجموعة من الممثلين.

لكن الأمر ليس بهذه البساطة ...

يزيل نهج الممثل و / أو السَّبُورَة و / أو الخِدمَات المصغرة للهيكلة نوع كامل من مشكلات التوافق المحتملة من تطبيقاتك. لكن هذه الفائدة تأتي بتكلفة. حيث يصعب التفكير في هذه الأساليب، لأن الكثير من العمل غير مباشر. ستجد أنه من المفيد الاحتفاظ بمخزون مركزي من تنسيقات الرسائل و / أو واجهات برمجة التطبيقات، خاصّة إذا كان بإمكان المستودع إنشاء الشِفرة والوثائق لك. ستحتاج أيضًا إلى أدوات جيدة لتتمكن من تتبع الرسائل والحقائق في أثناء تقدمها عبر النظام. (أحد الأساليب المفيدة هو إضافة معرّف تتبع فريد عند تهيئة دالة عمل معيّنة ثم نشرها على جميع الممثلين المعنيين. ستتمكن بعد ذلك من إعادة بناء ما يحدث بواسطة مِلَفّات السجل.)

أخيرًا، يمكن أن تكون هذه الأنواع من الأنظمة أكثر صعوبة في النشر والإدارة، حيث يوجد المزيد من الأجزاء المتحركة. إلى حد ما يتم تعويض ذلك بحقيقة أن النظام أكثر دِقَّة، ويمكن تحديثه عن طريق استبدال الممثلين الفرديين، وليس النظام بِرُمَّته.

الأقسام ذات الصلة

- الموضوع 28، الفصل، في الصفحة 151
- الموضوع 29، تلاعب العالم الحقيقي، صفحة 159
- الموضوع 33، كسر الاقتران الزمني، صفحة 193
- الموضوع 35، الممثلون والعمليات، في الصفحة 205

تمارين

تمرين 24 (إجابة محتملة في الصفحة 328)

هل يكون نظام على غرار السَّبُورَة مناسبًا للتطبيقات التالية؟ لما، أو لما لا؟

معالجة الصورة. قد ترغب في الحصول على عدد من العمليات المتوازية لالتقاط أجزاء من الصورة ومعالجتها وإعادة الصورة المكتملة مرّة أخرى.

تقويم المجموعة (Group calendaring). لديك أشخاص متوزعين في جميع أنحاء العالم، في مناطق زمنيّة مختلفة، ويتحدّثون لغات مختلفة، محاولين جدولة اجتماع.

أداة مراقبة الشبكة. يقوم النظام بجمع إحصائيات الأداء وجمع تقارير الأعطال، التي يستخدمها الوكلاء للبحث عن المشكلات في النظام.

تحديات

• هل تستخدم أنظمة السَّبُّورَة في العالم الحقيقي - لوحة الرسائل بجوار الثلاجة، أو السَّبُّورَة الكبيرة في العمل؟ ما الذي يجعلها فعّالة؟ هل تم نشر الرسائل في أي وقت بتنسيق متجانس؟ هل هذا يُهم؟

الفصل السابع:

حين تبرمج

تقول الحكمة التقليدية أنّه بمجرّد أن يكون المشروع في مرحلة كتابة الشِفرة، فإن العمل يكون ميكانيكيًا في الغالب، أي تحويل التصميم إلى تعليمات قابلة للتنفيذ. نعتقد أن هذا التوجّه هو السبب الأكبر الوحيد لفشل مشروعات البرمجيات، حيث ينتهي الأمر بالعديد من الأنظمة إلى أن تصبح قبيحة، أو غير فعّالة، أو سيئة التنظيم، أو غير قابلة للإصلاح، أو مجرّد خطأ واضح.

كتابة الشِفرة ليست عملًا ميكانيكيًا. إذا كان الأمر كذلك، فإن جميع أدوات CASE التي علّق عليها الناس آمالهم قديمًا في أوائل الثمانينيات كانت ستحلّ محلّ المبرمجين منذ مدّة طويلة. هناك قرارات يجب اتخاذها كل دقيقة - قرارات تتطلب تفكيرًا وحكمًا دقيقين في حال كان البرنامَج الناتج يرغب في التمتع بحياة طويلة ومنضبطة ومنتجة.

ليست كل القرارات واعية حتى. يمكنك تسخير غرائزك وأفكارك اللاواعية بشكل أفضل عندما تستمع إلى جذع دماغك "دماغ الزواحف" (Lizard Brain)[107] لديك. سنرى كيفيّة الاستماع بعناية أكبر ونلقي نظرة على طرق الاستجابة الفعّالة لهذه الأفكار المزعجة أحيانًا.

لكن الاستماع إلى غرائزك لا يعني أنه يمكنك فقط الطيران اعتمادًا على الطيار الآلي. يُبرمج المطورون الذين لا يفكرون بفاعليّة في شفرتهم البرمجية عن طريق المصادفة- قد تعمل الشِفرة، ولكن لا يوجد سبب محدّد لذلك. في البرمجة بالمصادفة، ندعو إلى مشاركة أكثر إيجابيّة في عملية كتابة الشِفرة.

مع أن معظم التعليمات البرمجيّة التي نكتبها تُنفّذ بسرعة، إلاّ أننا نقوم أحيانًا بتطوير خوارزميات لديها القدرة على إبطاء حتى أسرع المعالجات. في سرعة الخوارزمية، نناقش طرق تقدير سرعة تنفيذ الشِفرة، ونقدّم بعض النصائح حول كيفيّة اكتشاف المشكلات المحتملة قبل حدوثها.

يفكّر المبرمجون العمليون بشكل نقدي في جميع الشفرات، بما في ذلك الشِفرة الخاصّة بنا. نرى باستمرار مجالًا للتحسين في برامجنا وتصميماتنا. في إعادة البناء، نلقي نظرة على التقنيات التي تساعدنا على تطوير الشِفرة الحاليّة باستمرار في أثناء تقدمنا.

إن الاختبار لا يتعلق بالعثور على الأخطاء، بل يتعلق بالحصول على ملاحظات على شفرتك: جوانب التصميم، وواجهة برمجة التطبيقات، والاقتران، وما إلى ذلك. وهذا يعني أن الفوائد الرئيسة للاختبار تتحقّق عندما تفكّر في الاختبارات وتكتبها، وليس فقط عند إجرائها. سنستكشف هذه الفكرة في اختبر لتكتب الشِفرة.

107 **[المترجم]** Lizard Brain: الجزء الأكثر بدائية من الدماغ؛ جذع الدماغ. (بالامتداد) أي جزء من نفسية الشخص أو شخصيته يهيمن عليه الغريزة أو الاندفاع بدلاً من التفكير العقلاني.

ولكن بالطبع عندما تختبر الشِفرة الخاصة بك، قد تُحضر تحيُزاتك الخاصّة إلى المَهمة. في الاختبار المعتمد على الخاصية، سنرى كيفيّة جعل الحاسوب يقوم ببعض الاختبارات واسعة النطاق من أجلك وكيفيّة التعامل مع الأخطاء الحتميّة التي تظهر.

من المهم أن تكتب شفرةً يسهُل قراءتها ويسهل التفكير فيها. إنه عالم قاسي هناك، مملوء بالممثلين السيئين الذين يحاولون بنشاط اقتحام نظامك وإحداث الضرر. سنناقش بعض الأساليب والنُهج الأساسيّة للغاية لمساعدتك على ابقَ آمنًا هناك.

أخيرًا، أحد أصعب الأشياء في تطوير البرمجيات هو تسمية الأشياء. علينا أن نُسمي الكثير من الأشياء، ومن نَواح كثيرة، تُحدّد الأسماء التي نختارها الحقيقة التي ننشئها. أنت بحاجة إلى البقاء على دراية بأي انحراف دلاليّ محتمل في أثناء البرمجة.

يمكن لمعظمنا قيادة السيارة إلى حدّ بعيد باستخدام وضع الطيار الآلي "لا إراديا". نحن لا نأمر قدمنا بشكل صريح بالضغط على دواسة، أو ذراعنا لتوجيه العجلة - نحن نفكر فقط "تمهّل واستدارة لليمين". ومع ذلك، فإن السائقين الجيّدين والآمنين يراجعون الموقف باستمرار، ويتحقّقون المشكلات المحتملة، ويضعون أنفسهم في مواقع جيّدة في حالة حدوث ما هو غير متوقع. وينطبق الشيء نفسه على عملية كتابة الشِفرة - قد يكون أمرًا روتينيًا إلى حدّ بعيد، ولكن الحفاظ على ذكائك حاضرًا يمكن أن يمنع حدوث كارثة.

الموضوع 37. استمع إلى حدسك

> يمكن فقط للبشر أن ينظروا مباشرة إلى شيء ما، ولديهم كل المعلومات التي يحتاجونها لعمل تنبؤ دقيق، وربما حتى يجرون التنبؤ الدقيق للحظات، ومن ثم يقولون أنّه ليس كذلك.
> ← جافين دي بيكر، هدية الخوف

عمل *غافين دي بيكر* في الحياة يساعد النّاس على حماية أنفسهم. ويلخّص كتابه "هدية الخوف: وإشارات البقاء الأخرى التي تحمينا من العنف [de98] رسالته. أحد الموضوعات الرئيسة التي يمرّ بها الكتاب هو أننا كبشر متقدمين تعلمنا تجاهل جانبنا الحيواني. غرائزنا، جذع دماغنا. ويدّعي أن معظم الأشخاص الذين تعرضوا للهجوم في الشارع يشعرون بعدم الراحة أو التوتر قبل الهجوم. هؤلاء النّاس يخبرون أنفسهم أنهم سخيفون. ثم يبرز لهم الشخص من المدخل المظلم ....

الغرائز هي ببساطة استجابة للأنماط المعبأة في دماغنا اللاواعي. بعضها فطريّ، والبعض الآخر يتم تعلّمه بواسطة التكرار. كلما اكتسبت خبرة كمبرمج، يضع عقلك طبقات من المعرفة الضمنيّة: الأشياء التي تعمل، والأشياء التي لا تعمل، والأسباب المحتملة لنوع ما من الخطأ، وكل الأشياء التي تلاحظها طوال أيامك. هذا هو الجزء من عقلك الذي يضغط على مفتاح حفظ المِلَفّ عندما تتوقف للدردشة مع شخص ما، حتى عندما لا تدرك أنك تقوم بذلك.

تتشارك الغرائز أيًا كان مصدرها، في شيء واحد: ليس لديها كلمات. الغرائز تجعلك تشعر، لا أن تفكّر. وهكذا عندما تُقدح الغريزة، لا ترى مصباحًا متوهجًا مع لافتة حوله. تشعر بدلاً من ذلك، بالتوتر أو الغضب أو تشعر بأن هذا عمل كثير للغاية.

الحيلة هي ملاحظة حدوثها أولًا، ومن ثم معرفة السبب. لنلقِ نظرة أولاً على موقفين شائعين يحاول فيه جذع دماغك الداخلي إخبارك بشيء ما. ثم سنناقش كيف يمكنك السماح لهذا الدماغ الغريزي بالخروج من غلافه الواقي.

الخوف من الصفحة الفارغة

يخشى الجميع الشاشة الفارغة، ذلك المؤشر الوامض الوحيد الذي تحيط به مجموعة كاملة من الـ*لا شيء*. يمكن أن يكون بَدْء مشروع جديد (أو حتى وحدة نمطيّة جديدة في مشروع قائم) تجربة مخيفة. كثير منا يفضّل تأجيل تقديم الالتزام الأولي للبَدْء.

نعتقد أن هناك مشكلتان تسببان ذلك، وأن لكليهما نفس الحل.

إحدى المشكلات هي أن جذع الدماغ يحاول إخبارك بشيء ما؛ هناك نوع من الشك يكمن تحت سطح الإدراك. وهذا مهم.

بصفتك مطوّرًا، كنت تجرب الأشياء وترى أيها نجح وأيها لم ينجح. لقد كنت تراكم الخبرة والحكمة. عندما تشعر بشك مزعج، أو تواجه بعض التردد عند مواجهة مَهمة، قد تكون تلك التجربة تحاول التحدث إليك. احترمها. قد لا تكون قادرًا على وضع إصبعك على الخطأ تمامًا، ولكن امنحها بعض الوقت، ومن المحتمل أن تتبلور شكوكك في شيء أكثر صلابة، شيء يمكنك معالجته. دع غرائزك تساهم في أدائك.

المشكلة الأخرى هي أكثر ابتذالا قليلاً: قد تخشى ببساطة أنك سترتكب خطأ.

وهذا خوف مُبرّر. نحن المطورون نضع الكثير من أنفسنا في الشِفرة الخاصة بنا؛ يمكننا أن نعدّ الأخطاء في هذه الشِفرة كأثر على كفاءتنا. ربما هناك عنصر من متلازمة المحتال[108] أيضًا؛ فقد نعتقد أن هذا المشروع يتجاوزنا. ولا يمكننا رؤية طريقنا حتى النهاية؛ سنصل إلى هذه النقطة ثم نُجبر على الاعتراف بأننا خسرنا.

محاربة الذات

في بعض الأحيان، تتطاير التعليمات البرمجية مباشرة من دماغك إلى المحرّر: تصبح الأفكار *بتات* دون جهد.

في أيام أخرى، تبدو كتابة الشِفرة كالمشي صعودًا في الوحل. يتطلب اتخاذ كل خطوة جهداً هائلاً، وكل ثلاث خطوات تنزلق خطوتين للخلف.

ولكن، كونك محترفًا، أيها الجندي، تخطو خطوة بعد خطوة موحلة: لديك عمل تقوم به. لسوء الحظ، هذا على الأرجح ضد ما يجب عليك فعله.

تحاول شفرتك إخبارك بشيء ما. إنها تقول أن هذا أصعب مما ينبغي. ربما تكون البنية أو التصميم خاطئًا، أو ربما أنت تحلّ المشكلة الخاطئة، أو ربما تقوم فقط بإنشاء أخطاء تعادل مزرعة نمل. مهما كان السبب، فإن جذع دماغك يستشعر ردود الفعل من الشِفرة، ويحاول بشدة أن يجعلك تستمع.

108 **[المترجم]** يمكن تعريف متلازمة المحتال كمجموعة من مشاعر النقص التي تستمر مع أنّ النجاح واضح. يعاني "المحتالون" من الشك الذاتي المزمن والشعور بالاحتيال الفكري الذي يتجاوز أي مشاعر للنجاح أو دليل خارجي على كفاءتهم.

كيف ستحدث جذع الدماغ

نتحدث كثيرًا عن الاستماع إلى غرائزك، إلى جذع دماغك اللاوعي. التقنيات هي نفسها دائما.

نصيحة ٦١ **استمع إلى غرائزك أولاً، أوقف ما تفعله.**

امنح نفسك القليل من الوقت والمساحة للسماح لعقلك بتنظيم نفسه. توقّف عن التفكير في الشِفرة، وافعل شيئًا لا يستدعي التفكير إلى حدّ ما، بعيدًا عن لوحة المفاتيح. تمشى وتناول الغداء وتحدّث مع شخص ما. ربما تُبيّت الأمر. دع الأفكار تتغلغل خلال طبقات دماغك بمفردها: لا يمكنك فرضها. في نهاية المطاف قد تتدفق هذه الأفكار إلى المستوى الواعي، وتصبح لديك واحدة من هذه *آها! وجدتها*.

إن لم ينجح ذلك، فحاول حل المشكلة من الخارج. اصنع رُسُوم الشعار المبتكرة حول الشِفرة التي تكتبها، أو اشرحها لزميلك في العمل (ويفضّل أن يكون شخصًا غير مبرمج)، أو لبطّتك المطاطيّة. اعرض أجزاء مختلفة من عقلك لهذه المشكلة، وتحقّق مما إذا كان لدى أي منها ممسك أفضل للشيء الذي يزعجك.

لقد فقدنا تتبع عدد المحادثات التي أجريناها حيث كان أحدنا يشرح مشكلة للآخر وفجأة صاح قائلا "أوه! بالتأكيد! " وعاد لإصلاحها.

ولكن ربما جربت هذه الأشياء، وما زلت عالقًا. حان وقت العمل. نحتاج أن نخبر دماغك أن ما أنت على وشك القيام به لا يهم. ونحن نقوم بذلك عن طريق إنشاء النماذج الأوليّة.

حان وقت اللعب!

أمضى *آندي* و *ديف* ساعات في النظر في المخازن المؤقتة (buffers) الفارغة للمحرّر. سنكتب بعض الشِفرة، ثم ننظر إلى السقف، ثم نحصل على مشروب آخر، وبعدها نكتب المزيد من الشِفرة، ثم نذهب لقراءة قصّة مضحكة عن قطة ذات طرفين، ثم نكتب المزيد من الشِفرة، ثم نحدّد - الكل / حذف ونبدأ مرّة أخرى. ومرّة أخرى. ومرّة أخرى.

وعلى مرّ السنين وجدنا اختراقًا (hack) في الدماغ يبدو أنه يعمل. أخبر نفسك أنك بحاجة إلى نموذج أوليّ لشيء ما. إذا كنت تواجه شاشة فارغة، فابحث عن بعض جوانب المشروع التي تريد استكشافها. ربما تستخدم إطار عمل جديد، وتريد أن ترى كيف يربط البيانات. أو ربما تكون خوارزمية جديدة، وتريد استكشاف كيفيّة عملها في الحالات الحديّة (edge cases). أو ربما تريد تجربة أنماط مختلفة من تفاعل المستخدم.

إذا كنت تعمل على شِفرة موجودة وتُدفع للخلف، فقم بإخفائها في مكان ما واعمل نموذج أوليّ لشيء مشابه بدلاً من ذلك.

قم بما يلي:

1. اكتب عبارة " أنا نموذج أوليّ" على ورقة لاصقة، وألصقها على جانب الشاشة.
2. ذكّر نفسك أن النماذج الأوليّة تهدف إلى الفشل. وذكّر نفسك بأن النماذج الأوليّة يتم التخلّص منها حتى ولو لم تفشل. ليس هناك جانب سلبي في القيام بذلك.
3. في المخزن المؤقت الفارغ للمحرّر، أنشئ تعليق يصف في جملة واحدة ما تريد تعلّمه أو القيام به.

4. ابدأ كتابة الشِفرة.

إذا تسلّلت الشكوك إليك، فانظر إلى الملاحظة الملصقة.

إذا تبلور هذا الشك المزعج فجأة، في منتصف كتابتك للشِفرة، إلى مصدر قلق قويّ، فبادر إلى معالجته.

إذا وصلت إلى نهاية التجربة وما زلت تشعر بعدم الارتياح، فابدأ مرّة أخرى بالمشي والتحدّث وخذ وقتا للراحة.

ولكن، في تجربتنا، في مرحلة ما خلال النموذج الأوليّ، ستندهش عندما تجد نفسك تتناغم مع الموسيقى الخاصّة بك، وتستمتع بشعور إنشاء الشِفرة. ستتبخّر العصبيّة، مُستبدلةً بشعورٍ من الإصرار: فلنفعل ذلك!

في هذه المرحلة، تعرف ماذا تفعل. احذف جميع شِفرة النموذج الأوليّ، وتخلّص من الملاحظة الملصقة، واملأ مخزن المحرّر الفارغ هذا بشِفرة جديدة مشرقة ولامعة.

ليس شفرتك فقط

جزء كبير من عملنا هو التعامل مع الشفرات الموجودة، والتي غالباً ما يكتبها أشخاص آخرون. هؤلاء النّاس لديهم غرائز مختلفة بالنسبة لك، لذا فإنّ القرارات التي يتخذونها ستكون مختلفة. ليس بالضرورة أسوأ؛ فقط مختلفة.

يمكنك قراءة التعليمات البرمجيّة الخاصّة بهم آليا، مع المرور على الأشياء التي تبدو مهمّة وتدوين الملاحظات. إنه عمل روتيني، لكنّه يعمل.

أو يمكنك تجربةُ تجربةٍ معيّنة. عندما تكتشف أشياء تم القيام بها بطريقة تبدو غريبة، دونها. استمر في القيام بذلك، وابحث عن الأنماط. إذا تمكنت من رؤية ما دفعهم لكتابة الشِفرة بهذه الطريقة، فقد تجد أن مهمّة فهمها تصبح أسهل كثيرًا. ستكون قادرًا بوعي على تطبيق الأنماط التي طبّقوها ضمنيًا.

وقد تتعلّم شيئًا جديدًا على طول الطريق.

ليس مجرد شِفرة

تعلّم الاستماع إلى حدسك عند البرمجة هو مهارة مهمة لتعزيزها. لكنها تنطبق على الصورة الأكبر بشكلٍ جيّد. في بعض الأحيان يبدو التصميم خاطئًا، أو بعض المتطلبات تجعلك تشعر بعدم الارتياح. توقّف وحلّل هذه المشاعر. إذا كنت في بيئة داعمة، فعبّر عنها بصوت عالٍ. اكتشفها. من المحتمل أن يكون هناك شيء يكمن في هذا المدخل المظلم. استمع إلى غرائزك وتجنّب المشكلة قبل أن تقفز إليك.

الأقسام ذات الصلة

- الموضوع 13، النماذج الأولية والملاحظات الملحقة، في الصفحة 78
- الموضوع 22، دفاتر يوميات الهندسة، في الصفحة 121
- الموضوع 46، حل الألغاز المستحيلة، في الصفحة 277

تحديات

• هل لديك شيء تعرف أنه يجب عليك فعله، ولكنك تؤجله لأنه يشعرك بالخوف قليلاً أو لأنـه صـعب؟ طبّـق التقنيـات الموجـودة في هذا القسم. اضبط المؤقت الزمني لمدّة ساعة، ربما ساعتين، وعِد نفسك أنه عندما يدّق الجرس ستحذف ما قمت به. ماذا تعلمت؟

الموضوع 38. البرمجة بالمصادفة

هل سبق لك مشاهدة أفلام الحرب الأبيض والأسود القديمة؟ يتقدّم الجندي المرهق بحذرٍ خارج منطقة التمشيط. هناك مسحٌ للأمام: هل هناك أي ألغام أرضيّة، أم أنها آمنة للعبور؟ لا توجد أي علامات تدلّ على أنه حقل ألغام - لا توجد علامات أو أسلاك شائكة أو حفر. يدفع الجندي الأرض أمامه بحرابه ويجفل، متوقعًا انفجارًا. لا يوجد شيء. لذا فهو يمضي بشق الأنفس خلال الحقل لبعض الوقت، وهو يحث ويتلمس الأرض أمامه خلال سيره. في نهاية المطاف، يقتنع بأن الحقل آمن، يستقيم ويسير بفخر إلى الأمام، ليتفجر مباشرة إلى أشلاء.

لم تكشف تحقيقات الجندي الأوليّة عن الألغام أي شيء، لكن كان هذا مجرّد حظ. لقد توصّل إلى نتيجة خاطئة - مع نتائج كارثيّة.

بصفتنا مطوّرين، نعمل أيضًا في حقول الألغام. هناك المئات من الأشراك تنتظرنا كل يوم. تذكّر قصة الجندي، يجب أن نكون حذرين من استخلاص استنتاجات كاذبة. يجب أن نتجنّب البرمجة بالمصادفة - الاعتماد على الحظ والنجاحات العرضيّة - لمصلحة *البرمجة المتعمدة*.

كيفية البرمجة بالمصادفة

لنفترض أن *فريد* مُنح مهمة برمجيّة. يكتب *فريد* بعض التعليمات البرمجيّة، ويجرّبها، ويبدو أنها تعمل. يكتب *فريد* المزيد من التعليمات، ويجرّبها، ويبدو أنها لا تزال تعمل. بعد عدّة أسابيع من كتابة الشِفرة بهذه الطريقة، يتوقف البرنامَج فجأة عن العمل، وبعد ساعات من محاولة إصلاحه، لا يزال السبب مجهولًا له. قد يقضي *فريد* وقتًا طويلًا في مطاردة هذا الجزء من الشِفرة دون أن يكون قادرًا على إصلاحه. بغض النظر عما يفعله *فريد*، لا يبدو أن هذا الجزء يعمل بشكل صحيح على الإطلاق.

لا يعرف *فريد* سبب فشل الشِفرة *لأنه لم يكن يعرف سبب نجاحها أساسا*. لقد بدت وكأنها تعمل، اعتمادًا على "الاختبار" المحدود الذي قام به *فريد*، لكن ذلك كانت مجرّد مصادفة. وجّه *فريد* الاتهام إلى النسيان مدعومًا بثقة زائفة. الآن، قد يعرف معظم الأشخاص الأذكياء شخصًا مثل *فريد*، لكننا نحن نعرفهم بشكل أفضل. نحن لا نعتمد على المصادفات، أليس كذلك؟

في بعض الأحيان قد نفعل ذلك. أحيانا قد يكون من السهل جدًا الخلط بين مصادفة سعيدة وخِطّة هادفة. دعونا نلقي نظرة على بعض الأمثلة.

حوادث التنفيذ (Accidents of Implementation)

إن حوادث التنفيذ هي أشياء تحدث ببساطة لأن هذه هي الطريقة التي تُكتب بها الشِفرة حاليًا. ينتهي بك الأمر بالاعتماد على خطأ غير موثّق أو شروط حديّة.

افترض أنك استدعيت روتينًا مستخدمًا بياناتٍ سيئة. يستجيب الروتين بطريقة معيّنة، وأنت تكتب الشِفرة بناءً على تلك الاستجابة. لكن المؤلف لم يكن ينوي أن يعمل الروتين بهذه الطريقة - لم يفكّر في ذلك حتى. عندما يتم "إصلاح" الروتين، قد تتعطل شفرتك. في الحالة الحدّية، قد لا يكون الروتين الذي استدعيته مصممًا للقيام بما تريد، ولكن يبدو أنه يعمل بشكل جيّد. إن استدعاء الأشياء بالترتيب الخاطئ، أو في السياق الخاطئ، هي مشكلة ذات صلة.

هنا يبدو أن فريد يحاول باستماتة عرض شيء ما على الشاشة باستخدام إطار عمل تحويل (rendering) خاص لواجهة المستخدم الرسوميّة GUI:

```
paint();
invalidate();
validate();
revalidate();
repaint();
paintImmediately();
```

لكن هذه الروتينات لم تصمّم أبداً لتُستدعى بهذه الطريقة؛ مع أنّها تعمل، إلا أنها مجرّد مصادفة.

ولكي يزيد الطين بلّة، عندما يُرسَم المشهد أخيرًا، لن يحاول فريد العودة وإزالة الاستدعاءات الزائفة. "إنها تعمل الآن، من الأفضل تركها ما يكفي لوحدها ..."

من السهل أن تنخدع بهذه الطريقة من التفكير. لماذا يجب أن تخاطر بالعبث بشيء يعمل؟ حسنًا، يمكننا التفكير في عدة أسباب:

- ربما هو لا يعمل فعلًا - قد يبدو وكأنه يعمل.
- قد تكون الحالة الحدّية التي تعتمد عليها مجرّد حادث. وفي ظروف مختلفة (دقّة شاشة مختلفة، المزيد من نوى وحدة المعالجة المركزية)، قد يتصرف بشكل مختلف.
- قد يتغير السلوك غير الموثّق مع الإصدار التالي من المكتبة.
- الاستدعاءات الإضافيّة وغير الضرورية تجعل شفرتك أبطأ.
- الاستدعاءات الإضافيّة تزيد من خطر إدخال أخطاء جديدة خاصّة بها.

بالنسبة إلى الشِفرة التي تكتبها والتي سيستدعيها الآخرون، يمكن أن تساعد المبادئ الأساسيّة للوحدات النمطيّة الجيّدة وإخفاء التنفيذ خلف واجهات صغيرة وموثّقة جيّدًا. ويمكن أن يساعد العقد المحدّد جيّدًا (راجع الموضوع 23، التصميم حسب العقد، في الصفحة 125) في إزالة سوء الفهم.

بالنسبة للروتينات التي تستدعيها أنت، اعتمد فقط على السلوك الموثّق. إذا لم تستطع، لسببٍ من الأسباب، فوثّق افتراضك جيّدًا.

أليس قريبا بما يكفي

لقد عملنا ذات مرّة على مشروع كبير يبلّغ عن البيانات التي يتم تغذيتها من عدد كبير جدًا من وحدات جمع بيانات الأجهزة في المجال. امتدت هذه الوحدات عبر الولايات والمناطق الزمنيّة، ولأسباب لوجستية وتاريخيّة مختلفة، تم ضبط كل وحدة على التوقيت المحلي [109]. نتيجة لتضارب تفسيرات المنطقة الزمنيّة وعدم الاتساق في سياسات تحديد التوقيت الصيفي، كانت النتائج دائمًا خاطئة، ولكن فقط بمقدار بواحد. اعتاد المطورون في المشروع على إضافة واحد فقط أو طرح واحد للحصول على الإجابة الصحيحة،

109 ملاحظة من ندوب المعركة: هناك سبب لوجود "تنسيق التوقيت العالمي" UTC . استخدمه.

معتبرين أنه كان خاطئ بمقدار واحد في هذه الحالة فقط. وبعد ذلك، سترى الدالة التالية القيمة على أنها خاطئة بمقدار واحد من ناحية أخرى، وتعيدها كما كانت.

لكن حقيقة أن ذلك كان "فقط" في بعض الأحيان هو مجرد مصادفة، ما أدى إلى إخفاء عيب أعمق وأكثر جوهرية. فدون نموذج مناسب للتعامل مع الوقت، فإن قاعدة الشِفرة الكبيرة بِرُمَّتها كانت قد تطورت بمرور الوقت إلى كتلة غير مقبولة من تعليمات +1 و -1. في نهاية المطاف، لم يكن أي منها صحيحًا وتم إلغاء المشروع.

أنماط الأشباح (Phantom Patterns)

إن البشر مُصمَّمون لرؤية الأنماط والأسباب، حتى عندما تكون مجرّد مصادفة. على سبيل المثال، يتناوب القادة الروس دائمًا بين الأصلع والمشعر: تغلّب رئيس دولة أصلع (أو أصلع واضح) في روسيا على رئيس غير أصلع ("مشعر")، والعكس بالعكس، لمدة 200 عام تقريبًا.[110]

ولكن في حين أنك لن تكتب شِفرة تعتمد على كون الرئيس الروسي التالي أصلعًا أو مشعرًا، فإننا نفكّر في بعض المجالات بهذه الطريقة طوال الوقت. يتوقّع المقامرون الأنماط في أرقام اليانصيب، أو ألعاب النرد، أو الروليت، في حين أنها في الواقع أحداث مستقلّة إحصائيًا. في مجال المالية، فإن تداول الأسهم والسندات حافلة بنفس القدر من المصادفة بدلاً من الأنماط الفعليّة والملموسة.

قد يكون مِلَفّ السجل الذي يُظهر خطأً متقطعًا كل 1000 طلب (request) حالة تعارض يصعب تشخيصها، أو قد يكون خطأ قديمًا واضحًا. والاختبارات التي يبدو أنها تنجح على جهازك ولكن ليس على الخادم قد تشير إلى وجود اختلاف بين البيئتين، أو ربما يكون مجرّد مصادفة.

لا تفترض ذلك، أثبته.

حوادث السياق

قد أن يكون لديك "حوادث السياق" أيضًا. افترض أنك تكتب وحدة نمطيّة. فقط لأنك تقوم حاليًا بكتابة الشِفرة لبيئة واجهة المستخدم الرسوميّة، هل يجب أن تعتمد الوحدة على واجهة المستخدم الرسوميّة الموجودة؟ هل تعتمد على المستخدمين المتحدثين باللغة الإنجليزية؟ المستخدمين المتعلمين؟ ما الذي تعتمد عليه أيضًا ويكون غير مضمون؟

هل تعتمد على كون الدليل الحاليّ قابلاً للكتابة؟ على متغيرات بيئة معينة أو مِلَفّات الضبط الموجودة؟ في الوقت الذي يكون فيه الخادم دقيقًا - ما مدى التسامح؟ هل تعتمد على توفر الشبكة وسرعتها؟

عند نسخ التعليمات البرمجيّة من الإجابة الأولى التي عثرت عليها في الشبكة، هل أنت متأكّد أن السياق الخاص بك هو نفسه؟ أم أنك تبني شِفرة "عبادة البضائع"[111] (cargo cult)، فقط لتقليد الشكل دون المحتوى؟[112]

110 https://en.wikipedia.org/wiki/Correlation_does_not_imply_causation

111 **[المترجم]** نظام إيماني يقوم على الوصول المتوقع لأرواح الأسلاف في السفن التي تجلب شحنات من الطعام وغيرها.

112 انظر الموضوع 50، جوز الهند لا يفي بالغرض، في الصفحة 296.

إن العثور على إجابة تصادف أنها مناسبة ليس كمثل العثور على الإجابة الصحيحة.

نصيحة ٦٢	لا تبرمج بالصدفة.

افتراضات ضمنيّة

يمكن أن تُضلِل الصدف على جميع المستويات - من توليد المتطلبات حتى الاختبار. الاختبار محفوف بشكل خاص بالسببيات المزيفة والنتائج المصادفة. من السهل أن نفترض أن **س** يسبب **ع**، لكن كما قلنا في الموضوع 20، تنقيح الأخطاء، في الصفحة 110: لا تفترض ذلك، أثبته.

وعلى جميع المستويات، يعمل الناس مع وضع العديد من الافتراضات في الاعتبار - ولكن نادرًا ما توثّق هذه الافتراضات وغالبًا ما تكون متعارضة بين المطوّرين المختلفين. إن الافتراضات التي لا تستند إلى حقائق راسخة هي لعنة كل المشروعات.

كيف تبرمج عمدًا

نحن نرغب في قضاء وقت أقل في إنتاج الشِفرة، والتقاط الأخطاء وإصلاحها في أقرب وقت ممكن من دورة التطوير، وإنشاء أخطاء أقل للبَدء. من المفيد أن نتمكن من البرمجة عمدًا:

- كن دائمًا على دراية بما تفعله. ترك فريد الأمور تخرج عن السيطرة ببطء، حتى انتهى به الحال في الغليان، مثل الضفدع في الصفحة 34.
- هل يمكنك شرح الشِفرة بالتفصيل لمبرمج أصغر؟ إذا لم يكن الأمر كذلك، فربما أنت تعتمد على الصدف.
- لا تكتب شِفرة في الظلام. ابنِ تطبيقًا لا تفهمه بشكل كامل، أو استخدم تقنية لا تفهمها، وسيكون من المحتمل أن تُضلّل من قبل مصادفة. إذا لم تكن متأكّدًا من سبب نجاحها، فلن تعرف سبب فشلها.
- انطلق من الخطة، سواء كانت تلك الخِطّة في رأسك، أو مكتوبة على ظهر منديل كوكتيل، أو على لوح أبيض.
- اعتمد فقط على الأشياء الموثوقة. لا تعتمد على الافتراضات. إذا لم تتمكن من معرفة ما إذا كان هناك شيء موثوق به، افترض الأسوأ.
- وثّق الافتراضات الخاصّة بك. يمكن أن يساعد الموضوع 23، التصميم حسب العقد، في الصفحة 125، في توضيح افتراضاتك في ذهنك، وكذلك يساعدك في إيصالها للآخرين.
- لا تختبر شفرتك فقط، بل اختبر افتراضاتك أيضًا. لا تخمن، جرّبها فعلًا. اكتب تأكيدًا لاختبار افتراضاتك (راجع الموضوع 25، البرمجة التوكيدية، في الصفحة 135). إذا كان تأكيدك صحيحًا، فقد حسنت التوثيق في شفرتك. وإذا اكتشفت أن افتراضك خاطئا، فعد نفسك محظوظًا.

- أعط الأولوية جهودك. اقض الوقت في الجوانب المهمة؛ وهي على الأرجح الأجزاء الصعبة. إذا لم يكن لديك أساسيات أو بنية تحتيّة صحيحة، فإن الأجراس والصافرات البرّاقة لن تكون ذات صلة.
- لا تكن عبداً للتاريخ. لا تدع الشفرات الحالية تملي الشفرات المستقبلية. يمكن استبدال جميع الشفرات إذا لم تعد مناسبة. حتى في البرنامَج الواحد، لا تدع ما قمت به فعلًا يقيد ما ستفعله بعد ذلك - كن مستعدًا لإعادة البناء (انظر الموضوع 40، إعادة البناء، في الصفحة 235). قد يؤثر هذا القرار على الجدول الزمنيّ للمشروع. الافتراض هو أن التأثير سيكون أقل تكلفة من تكلفة عدم إجراء التغيير.[113]

إذا بدا لك في المرّة القادمة أن شيئًا ما يعمل، لكنك لا تعرف السبب، تأكّد أنه ليس مجرّد مصادفة.

الأقسام ذات الصلة

- الموضوع 4، حساء الحصى والضفادع المغليّة، في الصفحة 34
- الموضوع 9، DRY - شرور التكرار، في الصفحة 54
- الموضوع 23، التصميم حسب العقد، في الصفحة 125
- الموضوع 34، الحالة المشتركة حالة غير صحيحة، في الصفحة 198
- الموضوع 43، ابق آمنًا هناك، في الصفحة 255

تمارين

تمرين 25 (إجابة محتملة في الصفحة 328)

تمنحك التغذية الراجعة للبيانات من البائع مصفوفة من القوائم (tuples) التي تمثل أزواج القيمة الرئيسة. سيحتفظ مفتاح DepositAccount بسلسلة من رَقَم الحساب بالقيمة المقابلة:

```
[
...
{:DepositAccount, "564-904-143-00"}
...
]
```

لقد عمِلت بشكل مثالي في الاختبار على أجهزة الحاسوب المحمولة المطوّرة رباعية النواة وعلى جهاز البناء المكوّن من 12 نواة، ولكن على خوادم الإنتاج التي تعمل في حاويات، استمرت في الحصول على أرقام حسابات خاطئة. ماذا يحدث هنا؟

تمرين 26 (إجابة محتملة في الصفحة 329)

113 يمكنك أيضًا الذّهاب بعيدًا جدًا هنا. عرفنا ذات مرّة مطورًا أعاد كتابة كل مصدر حصل عليه لأنه كان لديه اصطلاحات التسمية الخاصة به.

أنت تكتب شِفرة برنامَج تشغيل تلقائي للتنبيهات الصوتية، وعليك إدارة قاعدة بيانات لمعلومات الاتصال. يحدّد الاتحاد الدُّوَليّ للاتصالات أن أرقام الهاتف يجب ألا تزيد عن 15 رقمًا، لذا يمكنك تخزين رَقَم هاتف جهة الاتصال في حقل رَقَمي مضمون لاستيعاب 15 رقمًا في الأقل. لقد اختبرته جيدًا في جميع أنحاء أمريكا الشَّمالية، ويبدو أن كل شيء على ما يرام، ولكن فجأة تصلك مجموعة من الشكاوى من أجزاء أخرى من العالم. لماذا ؟

تمرين 27 (إجابة محتملة في الصفحة 329)

لقد كتبت تطبيقًا يزيد مقادير الوصفات الشائعة لغرفة طعام سفينة سياحيّة تتسع لـ 5000 شخص. لكنك تتلقى شكاوى بأن التحويلات ليست دقيقة. تتحقق، وتجد أن الشِفرة تستخدم صيغة التحويل 16 كوبًا إلى غالون. هذا صحيح، أليس كذلك؟

الموضوع 39. سرعة الخوارزمية

في الموضوع 15، التقدير، في الصفحة 88، تحدثنا عن تقدير أشياء مثل الوقت الذي يستغرقه المشي عبر المدينة، أو المدّة التي سيستغرقها المشروع حتى النهاية. ومع ذلك، هناك نوع آخر من التقدير الذي يستخدمه المبرمجون العمليون يوميًا تقريبًا: تقدير الموارد التي تستخدمها الخوارزميات - الوقت والمعالج والذاكرة وما إلى ذلك.

غالبًا ما يكون هذا النوع من التقدير حاسمًا. عند الاختيار بين طريقتين لفعل شيء ما، أيهما تختار؟ أنت تعرف كم من الوقت يحتاج تشغيل برنامجك مع 1000 سجل، ولكن كيف سنزيده إلى 1000000؟ ما هي أجزاء الشِفرة التي تحتاج إلى تحسين؟

اتضح أن هذه الأسئلة يمكن الإجابة عليها في كثير من الأحيان باستخدام الفطرة السليمة، وبعض التحليل، وطريقة لكتابة التقريبات (approximations) تسمى *تدوين Big-O*.

ماذا نعني بتقدير الخوارزميات؟

تتعامل معظم الخوارزميات غير البديهية مع نوع ما من المدخلات المتغيرة - فرز n سلاسل نصية أو قلب مصفوفة n × m أو فك تشفير رسالة بمفتاح بطول n-bit. عادةً ما يؤثر حجم هذا الإدخال على الخوارزمية: فكلما زاد حجم الإدخال، زادت مدّة التشغيل أو زادت الذاكرة المستخدمة.

إذا كانت العلاقة دائمًا خطيّة (بحيث يزيد الوقت بالتناسب المباشر مع قيمة n)، فلن يكون هذا القسم مهمًا. ومع ذلك، فإن معظم الخوارزميات ليست خطيّة. والخبر السار هو أن العديد منها خطيّة جزئيا. البحث الثنائي، على سبيل المثال، لا يحتاج إلى النظر إلى كل مرشح عند العثور على تطابق. الخبر السيئ هو أن الخوارزميات الأخرى أسوأ بكثير من الخطيّة. حيث تزداد أوقات التشغيل أو متطلبات الذاكرة بشكل أسرع مقارنة بـ n. قد تستغرق الخوارزمية التي تحتاج دقيقة لمعالجة عشرة عناصر عمرًا لمعالجة 100 عنصر.

نجد أنه عندما نكتب أي شيء يحتوي على حلَقات أو استدعاءات متكرّرة، فإننا نتحقّق دون وعي من متطلبات وقت التشغيل والذاكرة. نادرًا ما تكون هذه عمليّة رسميّة، ولكنها تأكيد سريع على أن ما نقوم به معقول في ظل تلك الظروف. ومع ذلك، نجد أحيانًا أنفسنا نجري تحليلاً أكثر تفصيلًا. ذلك عندما يكون تدوين Big-O مفيدًا.

تدوين Big-O

تدوين o الكبيرة[114] (Big-O)، O() مكتوبة، هو طريقة رياضية للتعامل مع التقريب. عندما نكتب أن روتين فرز معين يفرز n سجل في زمن $O(n^2)$، فإننا نقول ببساطة أن أسوأ وقت مستغرق يختلف حسب مربع n. ضاعف عدد السجلات، وسيزداد الوقت أربعة أضعاف تقريبًا. فكر بـ O على غرار هذا المعنى.

114 **[المترجم]** تمثيل O الكبرى: يتطرّق الرمز O الكبير لمجموعة من الدوال التي تتعلق فيما بينها بالتسارع بالنسبة للأعداد الطبيعية، وعمومًا توجد عدة تمثيلات لكل منها مفهومه الخاص وقد نشط استخدام هذا التمثيل في تحليل سرعة الخوارزميات وذلك لأن حساب عدد العمليات التي تنفذها خوارزمية ما قد يكون مستحيلاً في بعض الأحيان مع وجود كثير من الأمور التي تؤثر على عدد العمليات لذا فإن إعطاء تقريب لعدد العمليات التي تقوم بها الخوارزمية أكثر راحةً لنا وتمثيل O الكبير يتيح هذا الأمر بسهولة.

يضع تدوين O() حدًا أعلى لقيمة الشيء الذي نقيسه (الوقت والذاكرة وما إلى ذلك). إذا قلنا أن دالة تستغرق وقت $O(n^2)$، فإننا نعلم أن الحدّ الأعلى للوقت الذي تستغرقه لن يزداد أسرع من n^2. في بعض الأحيان نأتي بدوال O() معقّدة إلى حدّ ما، ولكن نظرًا لأن مصطلح الترتيب الأعلى سيسيطر على القيمة مع زيادة n، فإن القاعدة هي إزالة جميع المصطلحات ذات الترتيب المنخفض، وعدم تكبّد العناء لعرض أي عوامل مضاعفة ثابتة:

$O(\frac{n^2}{2}+3n)$ بالضبط مثل $O(\frac{n^2}{2})$ بالضبط مثل $O(n^2)$

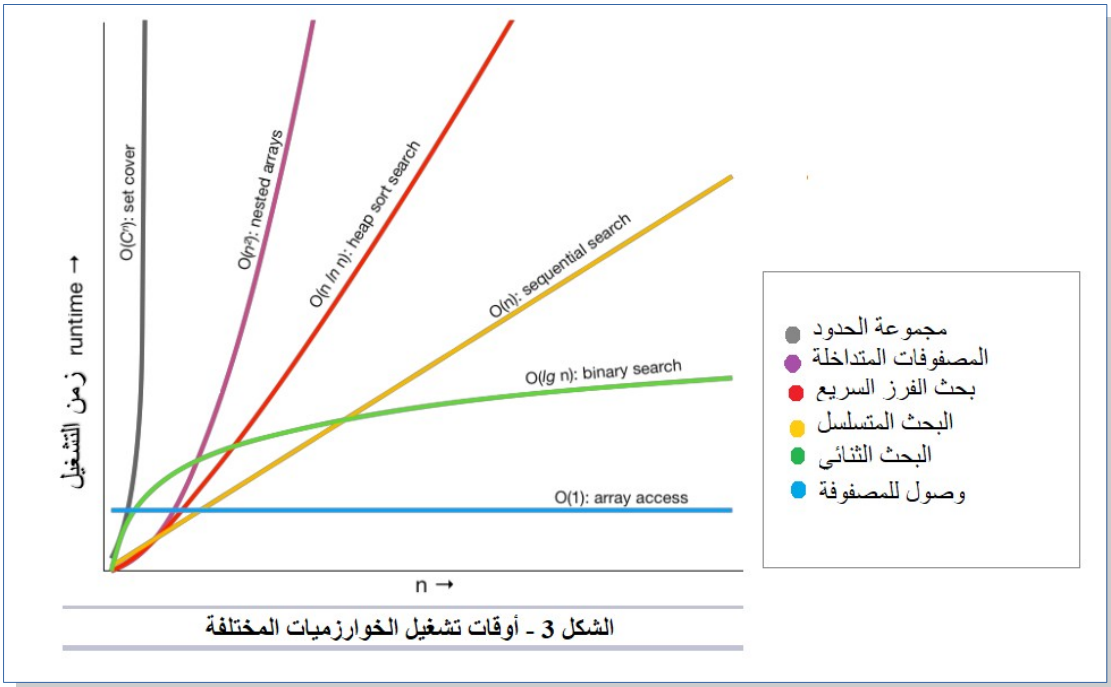
في الواقع هذه مِيزة في تدوين O() - قد تكون خوارزمية $O(n^2)$ أسرع 1000 مرّة من خوارزمية $O(n^2)$ أخرى، لكنك لن تعرفها من التدوين. لن تعطيك Big-O أرقامًا فعليّة للوقت أو الذاكرة أو أيا كان: إنها تخبرك ببساطة كيف ستتغير هذه القيم مع تغير المدخلات.

يوضح الشكل 3، أوقات تشغيل الخوارزميات المختلفة، في الصفحة 226 العديد من تدوينات O() الشائعة التي ستصادفها، إلى جانب رسم بياني يقارن أوقات تشغيل الخوارزميات في كل فئة. من الواضح أن الأمور تبدأ بسرعة في الخروج عن السيطرة بمجرّد أن نتخطى $O(n^2)$.

على سبيل المثال، لنفترض أن لديك روتينًا يستغرق ثانية واحدة لمعالجة 100 سجل. كم من الوقت ستستغرق معالجة 1000؟ إذا كان شفرتك O (1)، فسيستغرق الأمر ثانية واحدة. إذا كانت O (lg n)، فربما تنتظر حوالي ثلاث ثوانٍ. سيظهر O (n) زيادة خطيّة إلى عشر ثوان، بينما سيستغرق O (n lg n) حوالي 33 ثانية. إذا كنت غير محظوظ بما يكفي لامتلاك روتين $O(n^2)$، فاسترح لمدّة 100 ثانية في أثناء قيامه بأشيائه. وإذا كنت تستخدم خوارزمية أسيّة $O(2^n)$، فقد ترغب في صنع فنجان من القهوة - يجب أن ينتهي روتينك في حوالي 10^{263} سنة. دعنا نعرف كيف ينتهي الكون.

لا ينطبق تدوين O() على الوقت فقط؛ يمكنك استخدامه لتمثيل أي موارد أخرى تستخدمها خوارزمية. على سبيل المثال، غالبًا ما يكون من المفيد أن تكون قادرًا على نمذجة استهلاك الذاكرة (انظر التمارين للحصول على مثال).

O (1)	ثابت (عنصر وصول في مصفوفة، تعليمات بسيطة)
O (lg n)	لوغاريتمي (بحث ثنائي). قاعدة اللوغاريتم لا تُهم، لذلك فهو يعادل O (lg n) .
O (n)	خطيّ (البحث المتسلسل)
O (n lg n)	أسوأ من الخطيّ، ولكن ليس أسوأ بكثير. (متوسط وقت التشغيل السريع، الفرز السريع)
$O(n^2)$	قانون التربيع (ترتيب الاختيار والإدخال)
$O(n^3)$	التكعيب (ضرب مصفوفتين n × n)
$O(C^n)$	أسيّ (مشكلة البائع المتجول، تجزئة المجموعة)



الشكل 3 - أوقات تشغيل الخوارزميات المختلفة

تقدير الحس السليم

يمكنك تقدير ترتيب العديد من الخوارزميات الأساسية باستخدام الحس السليم.

الحلَقات البسيطة

إذا كانت حلقة بسيطة تعمل من 1 إلى n، فمن المحتمل أن تكون الخوارزمية هي O (n) - الوقت يزيد خطيًا مع زيادة n. تشمل الأمثلة عمليات البحث الشامل، والعثور على أكبر قيمة في المصفوفة، وإنشاء تدقيق المجموع (checksum).[115]

يعمل تدقيق المجموع بواسطة خوارزمية تقرأ البيانات وتوليد عدد ثابت من البتات (تبعاً للخوارزمية)، وهذه البتات تُستخدم للمقارنة مع ناتج تدقيق المجموع التالي بحيث يجب أن يتطابق الناتجان إن بقيت البيانات سليمة دون أي تغيير.

بعض من أشهر خوارزميات تدقيق المجموع هي: CRC، تدقيق مجموع فلتشر، SHA-1، MD5.

الحلَقات المتداخلة

إذا قمت بعمل تداخل لحلقة داخل حلقة أخرى، فإن الخوارزمية تصبح O (m × n)، حيث m و n هما حدّا الحلقتين. يحدث هذا عادةً في خوارزميات الفرز البسيطة، مثل الفرز الفقاعي[116]، حيث تقوم الحلقة الخارجيّة بمسح كل عنصر في المصفوفة بدوره، وتعمل الحلقة الداخليّة على مكان وضع هذا العنصر في النتيجة المفروزة. تميل خوارزميات الفرز إلى أن تكون O (n2).

115 **[المترجم]** تدقيق المجموع بالإنجليزية:(Checksum) هو شكل من أشكال فحص صحة البيانات، وهو أحد الإجراءات المبسطة للتحقق من سلامة البيانات المرسلة عبر شبكة (كالإنترنت مثلاً) أو المخزنة في وسيطة ما (قرص مدمج مثلاً)، وهي تسمح باكتشاف وجود الأخطاء في هذه البيانات.

116 **[المترجم]** خوارزمية الترتيب الفقاعي هي خوارزميةٌ مبنيةٌ على فكرة المقارنة المتكرّرة بين زوجٍ من العناصر المتجاورة، وتبديل مواقعهم إذا كانوا في ترتيبٍ خاطئ.

القطع الثنائي (Binary chop)

إذا خفّضت الخوارزمية الخاصّة بك مجموعة الأشياء التي تراها إلى النصف في كل دورة حول الحلقة، فمن المرجّح أن تكون لوغاريتمية، O (lg n). يمكن أن يكون ذلك البحث الثنائي لقائمة مصنّفة، واجتياز شجرة ثنائية، والعثور على أول مجموعة بت في كلمة آلة "مفردة تخزين" (machine word).

فرّق تسد

الخوارزميات التي تقسم عمل المدخلات إلى نصفين بشكل مستقل، ثم تجمع النتيجة يمكن أن تكون O (n lg n). المثال الكلاسيكي لذلك هو التصنيف السريع، والذي يعمل عن طريق تقسيم البيانات إلى نصفين وفرز كل منها بشكل متكرر. مع أنّ تدهور سلوك O (n^2) عمليا عند تغذيتها بالمدخلات المفرزة، فإن متوسط وقت التشغيل السريع هو O (n lg n).

التوافقية (Combinatoric)

عندما تبدأ الخوارزميات في النظر في التباديل (permutations) بين الأشياء، قد تخرج أوقات تشغيلها عن السيطرة. هذا لأن التباديل تنطوي على قيم "عاملية"(factorials) (حيث 5! = 5 × 4 × 3 × 2 × 1 = 120 تبديل في الأرقام من 1 إلى 5). حدّد خوارزمية مجمّعة لخمسة عناصر: سيستغرق تشغيلها ست مرّات أطول من وقت تشغيلها من أجل ستة، و 42 مرّة أطول من وقت تشغيلها من أجل سبعة. تتضمن الأمثلة خوارزميات للعديد من المشكلات الصعبة المعترف بها - مشكلة البائع المتجول[117]، تعبئة الأشياء على النحو الأمثل في حاوية، وتقسيم مجموعة من الأرقام بحيث يكون لكل مجموعة نفس الإجمالي، وهكذا. في كثير من الأحيان، يتم استخدام الاستدلال لتقليل أوقات تشغيل هذه الأنواع من الخوارزميات في نطاقات مشكلة محدّدة.

سرعة الخوارزمية من الناحية العمليّة

من غير المحتمل أن تقضي وقتًا طويلاً خلال مسيرتك المهنية في كتابة روتينات الفرز. من المحتمل أن تتفوق تلك الروتينات الموجودة في المكتبات المتاحة لك أصلًا على أي شيء قد تكتبه دون بذل جهد كبير. ومع ذلك، فإن الأنواع الأساسيّة من الخوارزميات التي وصفناها سابقًا تعود لتظهر مرارًا وتكرارًا. عندما تجد نفسك تكتب حلقة بسيطة، فأنت تعلم أن لديك خوارزمية O (n). إذا كانت هذه الحلقة تحتوي على حلقة داخليّة، فأنت تنظر إلى O (m × n). يجب أن تسأل نفسك عن حجم هذه القيم. إذا كانت الأرقام محدودة، فستعرف كم سيستغرق تشغيل الشِفرة. إذا كانت الأرقام تعتمد على عوامل خارجيّة (مثل عدد السجلات في التشغيل دفعة واحدة أو عدد الأسماء في قائمة بالأشخاص)، فقد ترغب في التوقّف والنظر في تأثير القيم الكبيرة على تشغيلك استهلاك الوقت أو الذاكرة.

117 **[المترجم]** مسألة البائع المتجول (بالإنجليزية: Travelling salesman problem) هي إحدى أهم المسائل في علم التعقيد الحسابي ونظرية المخططات، ونص المسألة هو: وصل تاجر إلى دولة فيها س مدينة ويريد البائع أن يزور كل مدينة في الدولة مرّة واحدة فقط وبأقل وقت سفر بين المدن. بالرغم من بساطة عرض المسألة فقد تبيّن أن هذه المسألة هي إحدى المسائل التي لا يُعرف لها خوارزمية تحلّها بسرعة، أي أنه إذا كان هناك فقط 50 مدينة حينها يتطلّب الأمر أكثر من ألف عام لإيجاد الحل!

نصيحة ٦٣ قدّر ترتيب خوارزمياتك.

هناك بعض الأساليب التي يمكنك اتباعها لمعالجة المشكلات المحتملة. إذا كان لديك خوارزمية $O(n^2)$، فحاول العثور على نهج فرّق تَسُد والذي سينقلك إلى $O(n \lg n)$.

إذا لم تكن متأكّدًا من المدّة التي ستستغرقها الشِفرة الخاصّة بك، أو مقدار الذاكرة التي ستستخدمها، فحاول تشغيلها، أو تغيير عدد سجلات الإدخال أو أي شيء من المحتمل أن يؤثر على وقت التشغيل. ثم ارسم النتائج. ستحصل باكرًا على تصوّر جيّد عن شكل المنحنى. هل ينحني لأعلى، أم أنه خط مستقيم، أم ينحسر مع زيادة حجم الإدخال؟ يجب أن تمنحك ثلاث أو أربع نِقَاط فكرةً ما.

ضع في اعتبارك أيضًا ما تفعله في الشِفرة نفسها. قد تكون حلقة $O(n^2)$ البسيطة أفضل أداءً من حلقة $O(n \lg n)$ معقّدة للقيم الأصغر من n، خاصّةً إذا كانت خوارزمية $O(n \lg n)$ لها حلقة داخليّة مكلفة.

لا تنس وَسَط كل هذه النظرية، أن هناك اعتبارات عمليّة أيضًا. قد يبدو وقت التشغيل أنه يزداد خطيًا من أجل مجموعات الإدخال الصغيرة. ولكن غذي الشِفرة بملايين السجلات وفجأة يتدهور الوقت مع بَدْء النظام في الانهيار. إذا اختبرت روتين الفرز باستخدام مفاتيح الإدخال العشوائية، فقد تُفاجأ في المرّة الأولى التي تواجه فيها الإدخال المطلوب. حاول تغطية كل من الأسس النظريّة والعمليّة. وبعد كل هذا التقدير، فإن التوقيت الوحيد المهم هو سرعة الشِفرة الخاص بك، التي تعمل في بيئة الإنتاج، مع بيانات حقيقية. وهذا ما يؤدي إلى نصيحتنا التالية.

نصيحة ٦٤ اختبر تقديراتك.

إذا كان الحصول على مواعيد زمنيّة دقيقة أمرًا صعبًا، فاستخدم *code profilers* لحساب عدد المرّات تنفيذ الخطوات المختلفة في خوارزميتك، وارسم هذه الأشكال مقابل حجم الإدخال.

الأفضل ليس دائما أفضل

تحتاج أيضًا إلى أن تكون عمليًا في اختيار الخوارزميات المناسبة - الأسرع ليس دائمًا الأفضل لأداء العمل. فبالنسبة لمجموعة صغيرة من المدخلات، سيعمل فرز الإدراج البسيط بشكل مشابه للفرز السريع، وسيستغرق وقتًا أقل في الكتابة والتصحيح. تحتاج أيضًا إلى توخي الحذر إذا كانت الخوارزمية التي تختارها لها تكلفة إعداد عالية. بالنسبة لمجموعات المدخلات الصغيرة، قد يختصر هذا الإعداد وقت التشغيل ويجعل الخوارزمية غير مناسبة.

احذر أيضًا من التحسين المبكر. من الجيد دائمًا التأكّد أن الخوارزمية هي فعلًا عنق الزجاجة قبل استثمار وقتك الثمين في محاولة تحسينها.

الأقسام ذات الصلة

- الموضوع 15، التقدير، في الصفحة 88

تحديات

- يجب أن يكون لدى كل مطوّر فكرة عن كيفيّة تصميم وتحليل الخوارزميات. كتب *روبرت سيدجويك* سلسلة من الكتب التي يمكن الوصول إليها حول هذا الموضوع (الخوارزميات [SW11] مقدمة لتحليل الخوارزميات [SF13] وغيرها). نوصي بإضافة أحد كتبه إلى مجموعتك، والتركيز على قراءتها.
- بالنسبة لأولئك الذين يحبّون تفاصيل أكثر مما يوفره Sedgewick ، اقرأ كتب Donald Knuth النهائية عن *فن برمجة الحاسوب*، والتي تحلل مجموعة واسعة من الخوارزميات.

  - فن برمجة الحاسوب، المجلد الأول: الخوارزميات الأساسية [Knu98]
  - فن برمجة الحاسوب، المجلد 2: الخوارزميات شبه العددية[Knu98a]
  - فن برمجة الحاسوب، المجلد الثالث: الفرز والبحث [Knu98b]
  - فن برمجة الحاسوب، المجلد 4 أ: الخوارزميات التوافقية، الجزء الأول [Knu11].

- في التمرين الأول التالي، نلقي نظرة على فرز مصفوفات الأعداد الصحيحة الطويلة. ما هو التأثير إذا كانت المفاتيح أكثر تعقيدًا، وكان ارتفاع المقارنة الرئيسة مرتفعًا؟ هل تؤثر البنية الأساسيّة على كفاءة خوارزميات الفرز، أم أن الفرز الأسرع هو دائمًا أسرع؟

تمارين

تمرين 28 (إجابة محتملة في الصفحة 329)

كتبنا شِفرة مجموعة من إجراءات الفرز البسيطة[118] في Rust. شغلها على أجهزة مختلفة متاحة لك. هل تتبع أرقامك المنحنيات المتوقعة؟ ما الذي يمكنك استنتاجه من السرعات النسبية لأجهزتك؟ ما هي تأثيرات إعدادات تحسين المترجم (compiler) المختلفة؟

تمرين 29 (إجابة محتملة في الصفحة 330)

في تقدير الحس السليم، في الصفحة 231 ، زعمنا أن القطع الثنائي هي O (lg n). هل يمكنك إثبات ذلك؟

تمرين 30 (إجابة محتملة في الصفحة 330)

في الشكل 3، أوقات تشغيل الخوارزميات المختلفة، في الصفحة 231، زعمنا أن O (lg n) هو نفس O (log10n) (أو في الواقع اللوغاريتمات لأي قاعدة). هل يمكنك أن تشرح لماذا؟

118 https://media-origin.pragprog.com/titles/tpp20/code/algorithm_speed/sort/src/main.rs

الموضوع 40. إعادة البناء

> كل ما أراه هو التغيير والانحلال...
> ← H. F. Lyte، ابق معي

مع تطوّر البرنامَج، سيصبح من الضروري إعادة التفكير في القرارات السابقة وتجديد أجزاء من الشِفرة. إن هذه العملية طبيعيّة تمامًا. تحتاج الشِفرة إلى التطور؛ فهي ليست شيئًا ثابتًا.

لسوء الحظ، فإنّ الاستعارة الأكثر شيوعًا لتطوير البرمجيات هي بناء المباني. يستخدم عمل برتراند ماير التقليدي "*بناء البرمجيات كائنية التوجه*" *[Mey97]* المصطلح "بناء البرمجيات"، حتى مؤلفاك المتواضعان حررا عمود "بناء البرمجيات" لبرمجيات IEEE في أوائل العقد الأول من القرن الحادي والعشرين.[119]

لكن استخدام البناء كاستعارة يعني الخطوات التالية:

1. يرسم مهندس معماريّ مخططات أوليّة.
2. يحفر متعهدو البناء الأساسات، ويبنون البنية الفوقيّة، والأسلاك والبراغي، ووضع اللمسات النهائية.
3. ينتقل إليه المستأجرون ويعيشون في سعادة دائمة بعدها، مطالبين بصيانة المباني لإصلاح أي مشكلات.

حسنًا، لا تعمل البرمجيات بهذه الطريقة تمامًا. حيث تشبه البرمجيات البستنة (gardening) بدلاً من البناء - فهي أكثر حياةً من الخرسانة. تزرع أشياء كثيرة في حديقة وفقًا لخطة وشروط أوليّة. يزهر بعضها، والبعض الآخر مقدّر له أن ينتهي به المطاف كسماد. يمكنك نقل المزروعات ذات الصلة بالنسبة لبعضها البعض للاستفادة من تفاعل الضوء والظل والرياح والمطر. تُقسم النباتات المفرطة في النمو أو تُقلّم، وقد تُنقل الألوان التي المتعارضة إلى مواقع أكثر جماليّة. تُسحب الأعشاب الضارة وتُسمّد المزروعات التي تحتاج إلى بعض المساعدة الإضافيّة. أنت تراقب باستمرار صحّة الحديقة، وتقوم بإجراء تعديلات (على التربة والنباتات والتخطيط) حسب الحاجة.

يشعر رجال الأعمال بالراحة تجاه استعارة تشييد المباني: فهي علميّة أكثر من البستنة، وقابلة للتكرار، وهناك تسلسل هرمي لإعداد التقارير للإدارة، وما إلى ذلك. لكننا لا نبني ناطحات سحاب- فنحن لسنا مقيّدين بحدود الفيزياء والعالم الحقيقي.

إن استعارة البستنة أقرب بكثير إلى واقع تطوير البرمجيات. ربما هناك روتين معين أصبح كبيرًا جدًا، أو أنه يحاول إنجاز الكثير - يجب تقسيمه إلى قسمين. الأشياء التي لا تعمل كما هو مخطط لها يجب إزالتها أو تشذيبها.

تُعرف إعادة كتابة التعليمات البرمجية وإعادة صياغتها وإعادة هيكلتها إجمالاً باسم إعادة الهيكلة (restructuring). ولكن هناك مجموعة فرعيّة من هذا النشاط أصبحت تعرف باسم إعادة البناء (refactoring).

119 حسنًا نعم، عبرنا عن مخاوفنا بشأن العنوان.

تم تعريف إعادة البناء [Fow19] بواسطة مارتن فاولر على أنها:

> تقنية منضبطة لإعادة هيكلة مجموعة موجودة من التعليمات البرمجية، وتغيير بنيتها الداخليّة دون تغيير سلوكها الخارجي.

الأجزاء الحاسمة من هذا التعريف هي:

1. النشاط منضبط وليس كليّ
2. لا يتغير السلوك الخارجي؛ هذا ليس الوقت المناسب لإضافة ميزات.

لا يُقصد من إعادة البناء أن تكون نشاطًا مُميزًا وعاليًا، يتم مرّة واحدة من مدّة لأخرى، مثل الحرث تحت الحديقة بِرُمَّتها من أجل إعادة زراعتها. وبدلا من ذلك، تعدّ إعادة البناء نشاطًا يوميًا، حيث تتخذ خطوات صغيرة منخفضة المخاطر، أشبه بإزالة الأعشاب الضّارة وجرفها. بدلاً من أن يكون شاملًا، يعيد كتابة الشِفرة بِرُمَّتها، يعد هذا أسلوبًا مستهدفًا ودقيقًا للمساعدة في الحفاظ على تغيير الشِفرة بسهولة.

ومن أجل ضمان عدم تغير السلوك الخارجي، فأنت بحاجة إلى اختبار وحدة مؤتمت جيّد للتحقّق سلوك الشِفرة.

متى يجب عليك إعادة البناء؟

عندما تكون قد تعلمت شيئًا ما، عندما تفهم شيئًا أفضل مما كنت تفعله العام الماضي أو الأمس أو حتى قبل عشر دقائق فقط.

ربما صادفتك حجر عثرة لأن الشِفرة لم تعد مناسبةً تمامًا، أو لاحظت شيئين يجب دمجهما حقًا، أو أي شيء آخر يُظن أنه "خطأ"، لا تتردد في تغييره. ليس هناك وقت أفضل من الوقت الحالي. أي عدد من هذه الأشياء قد يتسبب في جعل الشِفرة مؤهلة لإعادة البناء:

الازدواجية

لقد اكتشفتَ انتهاكًا لمبدأ DRY.

تصميم غير متعامد

لقد اكتشفت شيئًا يمكن جعله أكثر تعامدًا.

معرفة قديمة (غير محدّثة)

تتغير الأمور، وتنحرف المتطلبات، وتزيد معرفتك بالمشكلة. تحتاج الشِفرة لمواكبة ذلك.

الاستخدام

مع استخدام النظام من قبل أشخاص حقيقيين في ظل ظروف حقيقيّة، فإنّك تدرك أن بعض الميزات أصبحت الآن أكثر أهميّة مما كان يُعتقد سابقًا، وربما ميزات" يجب أن تكون" مهمّة لم تعد كذلك.

الأداء

تحتاج إلى نقل الوظائف من منطقة من النظام إلى أخرى لتحسين الأداء.

اجتياز الاختبارات

نعم. بجديّة. قلنا أن إعادة البناء يجب أن تكون نشاطًا صغيرًا، مدعومًا باختبارات جيّدة. لذلك عندما إضافتك مقدار صغيرة من الشِفرة، واجتياز اختبار إضافيّ واحد، يصبح لديك الآن فرصة رائعة للتعمّق فيما كتبته للتو ولترتيبه.

إن إعادة بناء شفرتك- تحريك الوظائف وتحديث القرارات السابقة - هو فعلًا تمرين في إدارة الألم. دعونا نواجه الأمر، قد يكون تغيير الشِفرة المصدريّة مؤلمًا جدًا: لقد كان يعمل، ربما من الأفضل أن نترك "جيد بما يكفي" وشأنه. يتردد العديد من المطورين في الدخول وإعادة فتح جزء من الشِفرة لمجرّد أنها غير صحيحة بشكلٍ كامل.

تعقيدات العالم الحقيقي

نتيجة لذلك تذهب إلى زملائك في الفريق أو إلى العميل وتقول، "هذه الشِفرة تعمل، لكنني بحاجة إلى أسبوع آخر لإعادة بنائها بالكامل."

لا يمكننا كتابة ردهم.

غالبًا ما يتم استخدام ضغط الوقت كعذر لعدم إعادة البناء. لكن هذا العذر لا يدوم: أخفق إعادة البناء الآن، وسيكون هناك وقت أطول بكثير لإصلاح المشكلة في انتظارك - عندما يكون هناك المزيد من التبعيات للتعامل معها. هل يكون هناك المزيد من الوقت المتاح؟ لا.

قد ترغب في شرح هذا المبدأ للآخرين باستخدام القياس الطبيّ: فكّر في الشِفرة التي تحتاج إلى إعادة البناء على أنها "نمو". يتطلب إزالته تدخلًا جراحيًا. يمكنك التدخل الآن وإخراجها وهي لا تزال صغيرة. أو يمكنك الانتظار بينما تنمو وتنتشر، ولكن إزالتها ستكون أكثر تكلفة وأكثر خطورة. انتظر لمدّة أطول، وقد تفقد المريض تمامًا.

نصيحة ٦٥	أعد البناء مبكرًا، أعد البناء مرارًا.

يمكن أن يكون الضرر الجانبي في التعليمات البرمجية قاتلًا مع مرور الوقت (انظر الموضوع 3، اعتلاج البرمجيات، في الصفحة 31). فإعادة الهيكلة، وكما هو الحال مع معظم الأشياء، من السهل القيام بها عندما تكون المشكلات صغيرة، كنشاط مستمر في أثناء كتابة الشِفرة. يجب ألا تكون بحاجة إلى "أسبوع لإعادة بناء" جزء من الشِفرة - فهذه إعادة كتابة كاملة. إذا كان هذا المستوى من الارتباك ضروريًا، فقد لا تتمكن من القيام بذلك على الفور. بدلاً من ذلك، تأكّد جدولته. تأكّد أن مستخدمي الشِفرة المتأثرة يعرفون أنه من المقرّر إعادة كتابتها وكيف يمكن أن يؤثر ذلك عليهم.

كيف تُعيد البناء؟

بدأت إعادة البناء في مجتمع سمول توك Smalltalk، وكانت قد بدأت للتو في اكتساب جَمهور أوسع عندما كتبنا الطبعة الأولى من هذا الكتاب، ربما بفضل أول كتاب رئيس عن إعادة البناء (إعادة البناء: تحسين تصميم الشِفرة الحالية [Fow19]، الآن هو في نسخته الثانية).

إن إعادة البناء هي في جوهرها إعادة تصميم. يمكن إعادة تصميم أي شيء صمّمته أنت أو غيرك في فريقك في ضوء الحقائق الجديدة، والفهم الأعمق، والمتطلبات المتغيرة، وما إلى ذلك. ولكن إذا تعديت تمزيق كميات هائلة من الشِفرة في حماستك المتهورة، فقد تجد نفسك في وضع أسوأ مما كنت عليه عندما بدأت.

من الواضح أن إعادة البناء نشاط يجب القيام به ببطء وتأني وحذر. يقدّم مارتن فاولر النصائح البسيطة التالية حول كيفيّة إعادة البناء دون إلحاق ضرر أكبر من النفع:[120]

1. لا تحاول إعادة البناء وإضافة الوظائف في نفس الوقت.
2. تأكّد أن لديك اختبارات جيّدة قبل أن تبدأ إعادة بنائها. قم بإجراء الاختبارات قدر الإمكان. بهذه الطريقة ستعرف بسرعة ما إذا كانت التغييرات قد عطّلت شيئًا ما.
3. اتخذ خطوات قصيرة ومدروسة: انقل حقل من صنف إلى آخر، قسّم طريقة (method)، أعد تسمية متغير. غالبًا ما تتضمن إعادة البناء إجراء العديد من التغييرات الموضعيّة التي تؤدي إلى تغيير على نطاق أوسع. إذا أبقيت خطواتك صغيرة، واختبرت بعد كل خطوة، فسوف تتجنب التنقيح المطول.[121]

إعادة البناء التلقائية

لاحظنا في الإصدار الأول، أن "هذه التقنية لم تظهر بعد خارج عالم سمول توك، ولكن من المحتمل أن يتغير هذا ..." وفعلًا، فقد حصل ذلك، حيث إن إعادة البناء التلقائية متوفرة في العديد من بيئات التطوير المتكاملة (IDEs) ومعظم اللغات السائدة.

يمكن لـ IDEs هذه إعادة تسمية المتغيرات والأساليب، وتقسيم روتين طويل إلى متغير أصغر، ونشر التغييرات المطلوبة تلقائيًا، والسحب والإفلات لمساعدتك في نقل الشِفرة، وما إلى ذلك.

سنتحدث أكثر عن الاختبار على هذا المستوى في الموضوع 41، اختبر لتكتب الشِفرة، في الصفحة 240، والاختبار على نطاق أوسع في اختبار قاسٍ ومستمر، في الصفحة 301، ولكن نقطة السيد *فاولر* في الحفاظ على اختبارات الانحدار الجيدة هي المفتاح لإعادة البناء بأمان.

إذا كان عليك أن تذهب إلى ما هو أبعد من إعادة البناء وتنتهي بتغيير السلوك أو الواجهات الخارجيّة، فيمكن أن يساعد ذلك في كسر البناء عن عمد: يجب أن تفشل تَرْجَمَة (compile) العملاء القدامى لهذه الشِفرة. بهذه الطريقة ستعرف ما يحتاج إلى التحديث.

في المرّة القادمة التي ترى فيها جزء من التعليمات البرمجيّة ليست كما يجب أن تكون، أصلحها. تحكم في الألم: إن كان الأمر مؤلمًا الآن، فإنه سيصبح مؤلمًا أكثر في وقت لاحق، يمكنك أيضًا التخلّص منه. تذكّر دروس الموضوع 3، اعتلاج البرمجيات، في الصفحة 31: لا تعش مع النوافذ المكسورة.

120 رصدت في الأصل في UML المقطر: دليل موجز للغة نمذجة الكائن القياسية [Fow00].

121 هذه نصيحة ممتازة عمومًا (انظر الموضوع 27، لا تتجاوز مصابيحك الأمامية، في الصفحة 146).

الأقسام ذات الصلة

- الموضوع 3، اعتلاج البرمجيات، في الصفحة 31
- الموضوع 9، DRY - شرور التكرار، في الصفحة 54
- الموضوع 12، الرصاصات الخطّاطة، في الصفحة 73
- الموضوع 27، لا تتجاوز مصابيحك الأمامية، في الصفحة 146
- الموضوع 44، تسمية الأشياء، في الصفحة 262
- الموضوع 48، جوهر التطوير الرشيق، في الصفحة 284

الموضوع 41. اختبر لتكتب الشِفرة

كُتبت الطبعة الأولى من هذا الكتاب في أوقات أكثر بدائية، في وقتٍ لم يكن يكتب فيه معظم المطورين أي اختبارات - لما الإزعاج، حسب اعتقادهم، إن العالم سينتهي في عام 2000 على أي حال.

في هذا الكتاب، كان لدينا قسم حول كيفيّة إنشاء شِفرة من السهل اختبارها. لقد كانت طريقة مخادعة لإقناع المطورين بكتابة الاختبارات فعلًا.

نحن الآن في أوقات أكثر استنارة. وإن كان لا يزال هناك مطورون لا يكتبون الاختبارات، إلاّ أنهم في الأقل يعرفون أنه يجب عليهم أن يكتبوا.

ولكن لا تزال هناك مشكلة. عندما نسأل المطورين عن سبب كتابة الاختبارات، ينظرون إلينا كما لو أننا سألنا للتو ما إذا كانوا لا زالوا يشفرون باستخدام البطاقات المثقّبة وسيقولون "للتأكّد عمل الشِفرة"، مع كلمة "غبي" غير منطوقة في نهاية الكلام. ونحن نعتقد أن هذا الأمر خاطئ.

إذاً ما المهم في الاختبار في نظرنا؟ وكيف نعتقد أنه يجب عليك الشروع به؟

لنبدأ بالتصريح الجريء:

نصيحة ٦٦	لا يتعلق الاختبار بإيجاد الأخطاء.

نحن نعتقد أن الفوائد الرئيسة للاختبار تتحقّق عندما تفكّر في الاختبارات وتكتبها، وليس عند إجرائها.

التفكير في الاختبارات

إنه صباح الاثنين وتتهيأ لبَدْء العمل على بعض الشفرات الجديدة. يجب أن تكتب شيئًا يستعلم من قاعدة البيانات لإرجاع قائمة بالأشخاص الذين يشاهدون أكثر من 10 مقاطع فيديو أسبوعيًا على موقع "أطرف مقاطع الفيديو الخاصة بغسل الأطباق في العالم".

يمكنك تشغيل المحرّر والبَدْء بكتابة الدالة التي تؤدي الاستعلام:

```
def return_avid_viewers do
    # ... hmmm ...
end
```

قف! كيف تعرف أن ما أنت على وشك فعله هو شيء جيّد؟

الجواب هو أنه لا يمكنك معرفة ذلك. لا أحد يعرف. لكن التفكير في الاختبارات يمكن أن يجعلها أكثر احتمالًا. إليك كيفيّة عمل ذلك.

ابدأ بالتخيّل أنك قد انتهيت من كتابة الدالة وعليك الآن اختبارها. كيف يمكنك أن تفعل ذلك؟ حسنًا، قد ترغب في استخدام بعض بيانات الاختبار، مما يعني على الأرجح أنك تريد العمل في قاعدة بيانات تتحكم فيها. حاليًا يمكن لبعض أطر العمل التعامل مع

ذلك من أجلك، وتشغيل الاختبارات على قاعدة بيانات الاختبار، ولكن في حالتنا، هذا يعني أننا يجب أن نمرّر نسخة "مثيل" قاعدة البيانات إلى دالتنا بدلاً من استخدام قاعدة بيانات عامة، لأن ذلك يسمح لنا بتغييرها في أثناء الاختبار:

```
def return_avid_users(db) do
```

ثم علينا أن نفكّر في كيفيّة ملء بيانات الاختبار تلك. يسأل المتطلب عن "قائمة بالأشخاص الذين يشاهدون أكثر من 10 مقاطع فيديو في الأسبوع". لذا ننظر إلى مخطط قاعدة البيانات للحقول التي قد تساعدنا في ذلك. نجد حقلين محتملين في جدول "من-شاهد-ماذا" (who-watched-what): هما open_video و finish_video. نحتاج من أجل كتابة بيانات الاختبار الخاصّة بنا، إلى معرفة المجال الذي يجب استخدامه. لكننا لا نعرف ما يعنيه هذا المتطلب، واتصال العمل الخاص بنا قد انتهى. دعنا فقط نغش وننقل اسم الحقل (مما سيسمح لنا باختبار ما لدينا، ومن المحتمل تغييره لاحقًا):

```
def return_avid_users(db, qualifying_field_name) do
```

بدأنا بالتفكير في اختباراتنا، ودون كتابة سطر واحد من الشِفرة، قمنا فعلًا باكتشافين واستخدمناهما لتغيير واجهة برمجة التطبيقات الخاصّة بطريقتنا (method).

اختبارات تقود كتابة الشِفرة

في المثال السابق، جعلنا التفكير في الاختبار نقلّل من الاقتران في الشِفرة (عن طريق تمرير اتصال قاعدة بيانات بدلاً من استخدام اتصال عام) ونزيد إمكانية الوصول (عن طريق جعل اسم الحقل الذي نختبر معاملا). إن التفكير في كتابة اختبار لطريقتنا جعلنا نلقي نظرة عليه من الخارج، كما لو كنا عملاء للشِفرة، وليس مؤلفيها.

نصيحة ٦٧ **الاختبار هو أول مستخدم لشفرتك.**

نعتقد أن هذه ربما تكون أكبر فائدة يقدمها الاختبار: الاختبار هو التغذية الراجعة الحيويّة التي توجّه برمجتك.

من الصعب اختبار الدالة أو الطريقة المقترنة بإحكام بشِفرة أخرى، لأنه يجب عليك إعداد كل تلك البيئة قبل أن تتمكن من تشغيل طريقتك. لذا فإن جعل أغراضك قابلة للاختبار يقلّل أيضًا من اقترانها.

وقبل أن تتمكن من اختبار شيء ما، عليك أن تفهمه. قد يبدو هذا سخيفًا، ولكن في الواقع لقد انطلقنا جميعًا في جزء ما من الشِفرة بناءً على فهمٍ غامض لما كان علينا القيام به. نؤكّد لأنفسنا أننا سنعمل على حلّها ونحن نمضي قدما. أوه، وسنضيف جميع الشفرات لدعم الشروط الحدّية لاحقًا أيضًا. أوه، ومعالجة الخطأ. وتنتهي الشِفرة بخمس مرات أطول مما ينبغي لأنها مليئة بالمنطق الشرطي والحالات الخاصّة. ولكن سلّط ضوء الاختبار على هذه الشِفرة، وستصبح الأمور أكثر وضوحًا. إذا فكّرت في اختبار الشروط الحدّية وكيف سيعمل ذلك قبل بَدْء كتابة الشِفرة، فقد تجد الأنماط في المنطق الذي سيُبسّط الدّالة. وإذا فكّرت في حالات الخطأ التي ستحتاج إلى اختبارها، فسوف تبني دالتك وفقًا لذلك.

التطوير المعتمد على الاختبار[122]

هناك مدرسة للبرمجة تقول أنه، بالنظر لجميع فوائد التفكير في الاختبارات مقدمًا، فلماذا لا تمضي قدمًا وتكتبها مقدمًا أيضًا؟ إنهم يمارسون شيئًا يسمى التطوير المعتمد على الاختبار أو (Test-Driven Development) اختصارًا TDD. ستري أيضًا تحت مسمى *تطوير الاختبار أولًا* (test-first development).[123]

الدورة الأساسية لـ TDD هي:

1. حدّد جزءًا صغيرًا من الوظائف التي تريد إضافتها.
2. اكتب اختبارًا ينجح بمجرّد تنفيذ هذه الوظيفة.
3. أجرِ جميع الاختبارات. تحقق أن الفشل الوحيد هو ذاك الذي كتبته للتو.
4. اكتب أقل كميّة من التعليمات البرمجية المطلوبة لاجتياز الاختبار، وتحقّق أن الاختبارات تعمل الآن بشكل نظيف.
5. أعد بناء الشِفرة: تحقق مما إذا كانت هناك طريقة لتحسين ما كتبته للتو (الاختبار أو الدالة). تأكّد أن الاختبارات لا تزال تنجح عند الانتهاء.

الفكرة هي أن هذه الدورة يجب أن تكون قصيرة جدًا: مسألة دقائق، بحيث تكتب اختبارات باستمرار ثم تبدأ في العمل.

نرى فائدة كبيرة في TDD للأشخاص الذين بدؤوا للتو في الاختبار. إذا اتبعت عمل TDD، فستضمن أن لديك دائمًا اختبارات لشفرتك. وهذا يعني أنك ستفكّر دائمًا في اختباراتك.

ومع ذلك، فقد رأينا أيضًا أن النّاس أصبحوا عبيدًا لـ TDD. يتجلى ذلك في طرق متعدّدة:

- يقضون فترات طويلة من الوقت لضمان أن لديهم تغطية اختبار بنسبة 100٪ دائمًا.
- لديهم الكثير من الاختبارات الزائدة. على سبيل المثال، قبل كتابة صنف (class) للمرّة الأولى، سيكتب العديد من أتباع TDD أولاً اختبارًا فاشلًا يشير ببساطة إلى اسم الصنف. يفشل، ثم يكتبون تعريف صنف فارغ وينجح. ولكن الآن لديك اختبار لا يفعل شيئًا على الإطلاق؛ سيشير الاختبار التالي الذي تكتبه أيضًا إلى الصنف، وعلى هذا يجعل الاختبار الأول غير ضروري. وسيكون هناك المزيد من الأشياء لتغييرها إذا تغير اسم الصنف لاحقًا. وهذا مجرّد مثال بسيط.
- تميل تصميماتهم إلى البَدء من الأسفل والعمل في طريقهم إلى الأعلى. (انظر من أسفل إلى أعلى مقابل من أعلى إلى أسفل مقابل الطريقة التي يجب أن تفعلها بها، في الصفحة 218.)

122 **[المترجم]** تطوير موجه بالاختبار: هو مصطلح يطلق على إحدى عمليات تطوير البرمجيات التي تعتمد على تكرار دورة تطوير قصيرة جداً: بدايةً، يقوم المبرمج بكتابة حالة فحص ذاتية فاشلة ويجب إلى حالة الفحص هذه أن تعرّف تحسينا معينا أو وظيفة جديدة. ومن ثم يكتب الشِفرة التي تجعل حالة الفحص ناجحة وأخيرا يعيد تصنيع الشِفرة كي تتلاءم مع المعايير. يُنسب تطوير أو "إعادة اكتشاف" هذه التقنية إلى *كينت بيك* لتحسين جودة تصميم البرمجيات وتقوية ثقة المبرمج في صحة تطويره. هذه الطريقة مرتبطة بمبدأ "الاختبار أولاً" في البرمجة القصوى.

123 يجادل بعض الأشخاص بأن تطوير الاختبار أولا والتطوير المعتمد على الاختبار هما شيئان مختلفان، قائلين إن نوايا الاثنين مختلفة. ومع ذلك، من الناحية التاريخية، كان الاختبار الأول (الذي يأتي من برمجة إكستريم) مطابقًا لما يسميه الناس الآن TDD.

بكل الوسائل مارس TDD. ولكن، إذا فعلت ذلك، فلا تنس التوقف بين الحين والآخر والنظر إلى الصورة الكبيرة. من السهل أن تنخدع برسالة "تم اجتياز الاختبارات" الخضراء، وكتابة الكثير من التعليمات البرمجية التي لا تجعلك أقرب إلى الحل.

TDD: أنت بحاجة إلى معرفة وجهتك

تسأل النكتة القديمة "كيف تأكل فيل؟" الخط النهائي: "قضمة واحدة في كل مرّة." وغالبًا ما يُروّج لهذه الفكرة على أنها فائدة من TDD. عندما لا تستطيع فهم المشكلة بِرُمَّتها، اتخذ خطوات صغيرة، اختبار واحدة في كل مرّة. ومع ذلك، يمكن أن يؤدي هذا النهج إلى تضليلك، مما يشجعك على التركيز على المشكلات السهلة وصقلها إلى ما لا نهاية بينما تتجاهل السبب الحقيقي الذي تكتب شفرته. حدث مثال مثير للاهتمام على ذلك في عام 2006، عندما بدأ رون جيفريز، وهو شخصيّة رائدة في حركة التطوير الرشيق، سلسلة من مشاركات المدونة التي وثّقت كتابة الشِفرة المعتمدة على الاختبار التجريبيّ من Sudoko solver.[124] بعد خمس مشاركات (post s)، قام بتحسين تمثيل اللوحة الأساسيّة، مُعيدًا البناء عدّة مرّات حتى أصبح سعيدًا بنموذج الكائن (object model). ولكن بعد ذلك تخلّى عن المشروع. من المثير للاهتمام قراءة مشاركات المدونة بالترتيب، ومشاهدة كيف يمكن لشخص ذكيّ أن ينحرف بواسطة التفاصيل الدقيقة، المعزّزة ببريق اجتياز الاختبارات.

من أسفل إلى أعلى مقابل من أعلى إلى أسفل مقابل الطريقة التي يجب عليك القيام بها

سابقا عندما كانت الحوسبة شابة وخالية من الهموم، كانت هناك مدرستان للتصميم: من الأعلى إلى الأسفل ومن الأسفل إلى الأعلى. قال أتباع منهج من أعلى لأسفل أنه يجب أن تبدأ بالمشكلة العامة التي تحاول حلها وتعمل على تقسيمها إلى عدد صغير من القطع. ثم تقسّم كل منها إلى قطع أصغر، وهكذا، حتى تنتهي بقطع صغيرة بما يكفي للتعبير عنها على شكل تعليمات برمجيّة.

أما من الأسفل إلى الأعلى فيبنون الشِفرة مثل بناء المنزل. يبدأون من الأسفل، وينتجون طبقة من الشِفرة تعطيهم بعض التجريدات الأقرب إلى المشكلة التي يحاولون حلها. ثم يضيفون طبقة أخرى، مع تجريدات عالية المستوى. يستمرون حتى الطبقة النهائية وهي تجريد يحل المشكلة. "اجعلها كذلك...."

لا تعمل أي من المدرستين في الواقع، لأن كلاهما يتجاهل أحد أهم جوانب تطوير البرمجيات: نحن لا نعرف ما نقوم به عندما نبدأ. يفترض الأشخاص من أعلى إلى أسفل أنهم يستطيعون التعبير عن المتطلبات بالكامل مقدّمًا: لا يمكنهم ذلك. ويفترض الأشخاص من الأسفل إلى الأعلى أنهم قادرون على بناء قائمة الملخصات التي ستأخذهم في نهاية المطاف إلى حل واحد من المستوى الأعلى، ولكن كيف يمكنهم تحديد وظائف الطبقات عندما لا يعرفون إلى أين يتجهون؟

نصيحة ٦٨ ابنِ من نهاية إلى نهاية، وليس من أعلى إلى أسفل أو من أسفل إلى أعلى.

نحن نؤمن بشدة أن الطريقة الوحيدة لبناء البرامج هي البناء على نحوٍ متزايد. ابنِ أجزاء صغيرة من وظائف النهاية إلى نهاية (End-to-End)، وتعرّف إلى المشكلة في أثناء تقدّمك. طبّق ما تتعلمه في أثناء متابعة كتابة الشِفرة، واشرك العملاء في كل خطوة، واطلب منهم توجيه العملية.

124 https://ronjeffries.com/categories/sudoku. شكر كبير لرون للسماح لنا باستخدام هذه القصة.

وعلى النقيض من ذلك، يصف بيتر نورفيج نهجًا بديلاً[125] يبدو مختلفًا تمامًا في طبيعته: بدلاً من أن يكون معتمدًا على الاختبارات، فهو يبدأ بفهم أساسي لكيفيّة حل هذه الأنواع من المشكلات تقليديًا (باستخدام نشر القيود)، ثم يركّز على تحسين خوارزميته. يعالج تمثيل اللوحة الأساسية (board) في عشرات الأسطر من الشِفرة والتي تُستمد مباشرة من مناقشته للتدوين.

يمكن أن تساعد الاختبارات بالتأكيد في دفع عجلة التطوير. ولكن وكما هو الحال مع كل تحرك، ما لم يكن لديك وجهة في ذهنك، فيمكن أن ينتهي بك الأمر بالسير في حلَقات مفرغة.

العودة إلى الشِفرة

لطالما كان التطوير المعتمد على المكونات هدفًا رفيعًا لتطوير البرمجيات.[126] والفكرة هي في أن مكونات البرامج العامّة يجب أن تكون متاحة ومدمجة بنفس السهولة التي تُدمج بها الدوائر المتكاملة المشتركة (ICs). ولكن هذا لا ينجح إلا إذا كانت المكونات التي تستخدمها معروفة بموثوقيتها، وإذا كان لديك الفولتية "الجهد" المشتركة ومعايير الاتصال والتوقيت وما إلى ذلك.

صَمّمت الرقائق (Chips) لتُختبر - ليس فقط في المصنع، وليس فقط عند تثبيتها، ولكن أيضًا في الميدان عند نشرها. قد تحتوي الرقائق والأنظمة الأكثر تعقيدًا على مِيزة اختبار ذاتي مدمج (BIST) كامل يشغّل بعض عمليات التشخيص على المستوى الأساسي داخليًا أو آلية الوصول إلى الاختبار (TAM) التي توفر تسخيرًا للاختبار يسمح للبيئة الخارجية بتوفير المحفزات (stimuli) وجمع الاستجابات من الشريحة.

يمكننا فعل الشيء نفسه في البرمجيات. وكما هو الحال مع زملائنا في مجال الأجهزة المادية، فإننا نحتاج إلى بناء قابلية الاختبار في البرمجيات منذ البداية، واختبار كل جزء بدقّة قبل محاولة ربطها معًا.

اختبار الوحدة

يكافئ الاختبار على مستوى الشريحة للأجهزة تقريبًا اختبار الوحدة في البرنامَج - الاختبار الذي يتم على كل وحدة على حِدَةٍ للتحقق من سلوكها. يمكننا الحصول على شعور أفضل حول كيفيّة تفاعل الوحدة في العالم الواسع الكبير بمجرّد اختبارها بشكل كامل في ظل ظروف خاضعة للرقابة (حتى الظروف المفتعلة).

اختبار وحدة البرمجيات هو شِفرة تختبر وحدة نمطية. سيؤسس اختبار الوحدة عادةً نوعًا من البيئة الاصطناعية، ثم يستدعي إجراءات روتينية في الوحدة النمطية المُختبرة. ويتحقّق النتائج المُرجعة، إما مقابل القيم المعروفة أو مقابل النتائج من الدورات السابقة لنفس الاختبار (اختبار الانحدار).

125 http://norvig.com/sudoku.html

126 لقد كنا نحاول منذ عام ١٩٨٦ في الأقل، عندما صاغ كوكس و نوفوبيلسكي مصطلح "البرمجيات المتكاملة" في كتابهما Objective-C Object-Oriented Programming: An Evolutionary Approach [CN91]. البرمجة كائنية التوجه: نهج تطوري Object-Oriented Programming.

لاحقًا، عندما نجمع "برمجيات دارات متكاملة" (software ICs) لدينا في نظام كامل، سيكون لدينا ثقة في أن الأجزاء الفردية تعمل كما هو متوقع، وبعد ذلك يمكننا استخدام نفس أدوات اختبار الوحدة لاختبار النظام كليًا. نتحدّث عن هذا الفحص على نطاق واسع للنظام في اختبار قاسٍ ومستمر، في الصفحة 301.

قبل أن نصل إلى هذا الحدّ، نحتاج إلى تحديد ما يجب اختباره إلى مستوى الوحدة. تاريخيًا، ألقى المبرمجون بضع بيتات عشوائية من البيانات على الشِفرة، ونظروا إلى عبارات الطباعة، وأسموها اختبارا. يمكننا أن نفعل ما أفضل من ذلك بكثير.

اختبار العقد

نود التفكير في اختبار الوحدة على أنه اختبار للعقد (انظر الموضوع 23، التصميم حسب العقد، في الصفحة 125). نريد كتابة حالات اختبار تضمن أن وحدة معينة تفي بعقدها. سيخبرنا هذا شيئين: ما إذا كانت الشِفرة تفي بالعقد، وما إذا كان العقد يعني ما نعتقد أنه يعني. نريد اختبار أن الوحدة توفر الوظائف التي تَعد بها عبر مجموعة واسعة من حالات الاختبار والشروط الحدّية.

ماذا يعني هذا في الممارسة العملية؟ لنبدأ بمثال عددي بسيط:

إجرائية الجذر التربيعي. عقدها الموثّق بسيط:

الشروط المسبقة:

```
argument >= 0;
```

الشروط اللاحقة:

```
((result * result) - argument).abs <= epsilon*argument;
```

يخبرنا هذا بما يجب اختباره:

- مرّر مُعطى (argument) سلبي وتيقن من رفضه.
- مرّر معطى يساوي صفر للتأكد أنه مقبول (هذه قيمة حدّية).
- مرّر القيم بين الصفر والحد الأقصى للمعطى وتحقّق أن الفرق بين مربع النتيجة والمعطى الأصلي أقل من حد صغير مُعطى (إبسيلون "epsilon").

معتمدين على هذا العقد، ومفترضين بأن إجرائيتنا تتحقق الشروط المُسبقة واللاحقة الخاصة به، يمكننا كتابة نص اختبار أساسي لاستعمال دالة الجذر التربيعي.

ثم يمكننا استدعاء هذا الروتين لاختبار دالة الجذر التربيعي:

```
assertWithinEpsilon(my_sqrt(0), 0)
assertWithinEpsilon(my_sqrt(2.0), 1.4142135624)
assertWithinEpsilon(my_sqrt(64.0), 8.0)
assertWithinEpsilon(my_sqrt(1.0e7), 3162.2776602)
assertRaisesException fn => my_sqrt(-4.0) end
```

إنه اختبار بسيط للغاية. في العالم الحقيقي، من المرجح أن تعتمد أي وحدة غير بديهية على عدد من الوحدات الأخرى، فكيف نذهب لاختبار المجموعة؟

لنفترض أن لدينا وحدة A تستخدم DataFeed و LinearRegression. من أجل ذلك، سنختبر:

1. عقد DataFeed بالكامل
2. عقد LinearRegression بالكامل
3. عقد A، الذي يعتمد على العقود الأخرى ولكنه لا يعرضها بشكل مباشر

يتطلب هذا النمط من الاختبار اختبار المكونات الفرعيّة للوحدة أولاً. بمجرّد التحقق من المكونات الفرعيّة، يمكن اختبار الوحدة نفسها.

إذا نجحت اختبارات DataFeed و LinearRegression، ولكن فشل اختبار A، يمكننا التأكّد تمامًا من أن المشكلة في A، أو في استخدام A لأحد هذه المكونات الفرعيّة.

تُعد هذه التقنية طريقة رائعة لتقليل جهد التصحيح: يمكننا التركيز بسرعة على المصدر المحتمل للمشكلة في الوحدة النمطيّة A، وعدم إضاعة الوقت في إعادة فحص مكوناتها الفرعيّة.

لماذا نذهب لكل هذه المشكلات؟ قبل كل شيء، نريد تجنب إنشاء "قنبلة موقوتة" - شيء موضوع حولنا دون أن يلاحظه أحد وينفجر في لحظة حرجة لاحقًا في المشروع. بواسطة التأكيد على الاختبار مقابل العقد، يمكننا محاولة تجنب أكبر عدد ممكن من تلك الكوارث.

نصيحة ٦٩	صمّم لتختبر.

اختبار مرتجل

لكيلا يتم الخلط مع "الاختراق الفردي"(odd hack)، فإن الاختبار المرتجل هو عندما نشغل الوخز على الشِفرة يدويًا. قد يكون هذا بسيطًا مثل console.log()، أو جزءًا من الشِفرة التي يتم إدخالها بشكل تفاعلي في منقح أخطاء أو بيئة تطوير متكاملة IDE أو REPL.

في نهاية جَلسة تنقيح الأخطاء، تحتاج إلى إضفاء الطابع الرسمي على هذا الاختبار المرتجل. إذا كُسرت الشِفرة مرّة واحدة، فمن المحتمل أن تكسر مرّة أخرى. لا تتخلص من الاختبار الذي أنشأته للتو: أضفه إلى ترسانة اختبار الوحدة الموجودة.

بناء نافذة اختبار

حتى أفضل مجموعات الاختبارات من غير المحتمل أن تجد جميع الأخطاء؛ هناك شيء يتعلق بالظروف الرطبة الدافئة لبيئة الإنتاج التي يبدو أنها تخرج البق من الأعمال الخشبيّة.

هذا يعني أنك ستحتاج غالبًا إلى اختبار برمجيّة ما بمجرّد نشرها- مع بيانات واقعيّة خلال الأوردة. على عكس لوحة الدوائر الإلكترونية أو الشريحة، ليس لدينا دبابيس اختبار في البرمجية، ولكن يمكننا تقديم طرق عرض مختلفة للحالة الداخليّة للوحدة، دون استخدام المنقّح (الذي قد يكون غير مريح أو مستحيل في تطبيق الإنتاج).

مِلَفّات السجل التي تحتوي على رسائل التتبع هي إحدى هذه الآليات. يجب أن تكون رسائل السجل بتنسيق منتظم ومتسق؛ قد ترغب في تحليلها تلقائيًا لاستنتاج وقت المعالجة أو المسارات المنطقيّة التي اتخذها البرنامَج. التشخيصات التي تم تنسيقها بشكل سيئ أو غير متناسق هي مجرّد "تنفيس"- من الصعب قراءتها ومن الصعب تحليلها.

هناك آليّة أخرى للحصول على شِفرة التشغيل الداخلي وهي تسلسل "مفتاح التشغيل السريع" (hot-key) أو عنوان URL السحريّ. عند الضغط على هذا المزيج من المفاتيح، أو الوصول إلى عنوان URL، تنبثق نافذة التحكم في التشخيص مع رسائل الحالة وما إلى ذلك. إن هذا ليس شيئًا تكشفه عادةً للمستخدمين النهائيين، ولكن يمكن أن يكون مفيدًا جدًا لمكتب المساعدة.

وبشكل أعم، يمكنك استخدام مفتاح المِيزة لتمكين التشخيص الإضافي لمستخدم معين أو فئة معينة من المستخدمين.

ثقافة الاختبار

ستُختبر جميع البرمجيات التي تكتبها - إن لم يكن من قبلك أنت وفريقك، فمن قبل المستخدمين النهائيين - لذلك يمكنك التخطيط لاختبارها جيدًا. يمكن للقليل من التفكير المُسبق أن يقطع شوطًا كبيرًا نحو تقليل تكاليف الصيانة واتصالات مكتب المساعدة.

لديك فعلًا عدد قليل من الخيارات:

- الاختبار أولاً
- الاختبار في أثناء
- لا اختبار

الاختبار أولًا، بما في ذلك التصميم المعتمد على الاختبار، ربما يكون خيارك الأفضل في معظم الظروف، لأنه يضمن إجراء الاختبار. ولكن في بعض الأحيان لا يكون ذلك مناسبًا أو مفيدًا، لذلك يمكن أن يكون الاختبار في أثناء البرمجة أمرًا احتياطيًا جيدًا، حيث تكتب بعض التعليمات البرمجية، وتعبث بها، وتكتب الاختبارات لها، ثم تنتقل إلى البت التالي. غالبًا ما يُطلق على الخِيار الأسوأ "الاختبار لاحقًا"، ولكن من تمازح؟ "الاختبار لاحقًا" يعني حقًا "لا اختبار".

تعني ثقافة الاختبار أن جميع الاختبارات تنجح طوال الوقت. إن تجاهل مجموعة من الاختبارات التي "تفشل دائمًا" تجعل من السهل تجاهل جميع الاختبارات، وتبدأ الحلقة المفرغة (راجع الموضوع 3، اعتلاج البرمجيات، في الصفحة 31).

اعتراف

من المعروف أنني (ديف) أخبر الناس أنني لم أعد أكتب الاختبارات. أفعل ذلك جزئيًا لزعزعة إيمان أولئك الذين حولوا الاختبار إلى دين. وأقول جزئيًا لأنها (إلى حدّ ما) صحيحة.

أعمل في مجال كتابة الشِفرة منذ 45 عامًا، وأكتب اختبارات تلقائية لأكثر من 30 عامًا منها. إن التفكير في الاختبار مبنيّ على الطريقة التي أتعامل بها مع كتابة الشِفرة. شعرت بالراحة. وتصرّ شخصيتي على أنه عندما يبدأ شيء ما في الشعور بالراحة يجب أن أنتقل منه إلى شيء آخر.

في هذه الحالة قرّرت التوقف عن كتابة الاختبارات لبضعة أشهر ومعرفة ما فعلته بشفرتي. لدهشتي، كان الجواب "ليس الكثير". لذلك قضيت بعض الوقت لمعرفة السبب.

أعتقد أن الإجابة هي أن (بالنسبة لي) معظم فائدة الاختبار تأتي من التفكير في الاختبارات وتأثيرها على الشِفرة. وبعد القيام بذلك لمدّة طويلة، يمكنني أن أفكّر في ذلك دون كتابة اختبارات في الواقع. كانت شفرتي لا تزال قابلة للاختبار؛ لم يتم اختبارها.

لكن هذا يتجاهل حقيقة أن الاختبارات هي أيضًا طريقة للتواصل مع المطورين الآخرين، لذلك أقوم الآن بكتابة اختبارات للشفرات المشتركة مع الآخرين أو التي تعتمد على خصائص التبعيات الخارجية.

يقول آندي أنه لا ينبغي لي تضمين هذا الشريط الجانبي. ويخشى من أن يغري المطورين عديمي الخبرة بعدم الاختبار. إليك حل وَسَط:

هل يجب أن تكتب الاختبارات؟ نعم. ولكن بعد قيامك بذلك لمدة 30 عامًا، لا تتردد في التجربة قليلًا لمعرفة أين تكمن الفائدة بالنسبة لك.

تعامل مع شِفرة الاختبار بنفس الاهتمام الذي توليه لأي شِفرة إنتاج. حافظ على انفصالها ونظافتها وقوتها. لا تعتمد على أشياء غير موثوقة (راجع الموضوع 38، البرمجة بالمصادفة، في الصفحة 223) مثل الموضع المطلق للأدوات في نظام واجهة المستخدم الرسوميّة، أو الطوابع الزمنيّة الدقيقة في سجل الخادم، أو الصياغة الدقيقة لرسائل الخطأ. اختبار هذه الأنواع من الأشياء سيؤدي إلى اختبارات هشّة.

نصيحة ٧٠	**اختبر برمجيتك، أو سيختبرها مستخدموك.**

الاختبار جزء من البرمجة. لم يبق شيء للإدارات أو الموظفين الآخرين.

الاختبار والتصميم والتشفير - كلها برمجة.

الأقسام ذات الصلة

- الموضوع 27، لا تتجاوز مصابيحك الأمامية، في الصفحة 146
- الموضوع 51، مجموعة البادئ العملية، في الصفحة 300

الموضوع 42. الاختبار المعتمد على الخاصيّة

> ثِق، ولكن تحقّق (Доверяй, но проверяй)
>
> ← مثل روسي

نوصي بكتابة اختبارات الوحدة لدوالك. يمكنك القيام بذلك بواسطة التفكير في الأشياء النموذجيّة التي قد تخلق مشكلة، بناءً على معرفتك بالشيء الذي تختبره.

مع أنّ ذلك، فهناك مشكلة صغيرة ولكنها كبيرة الاحتمال تكمن في تلك القَفْرة. إذا كتبتَ الشِفرة الأصليّة وكتبتَ الاختبارات، فهل من الممكن التعبير عن افتراض غير صحيح في كليهما؟ تجتاز الشِفرة الاختبارات، لأنها تفعل ما يفترض أن تفعله اعتمادًا على فهمك.

إحدى الطرق للقيام بذلك هي جعل أشخاص مختلفين يكتبون الاختبارات عندما تكون الشِفرة قيد الاختبار، لكننا لا نحبّذ ذلك: كما قلنا في الموضوع 41، اختبر لتكتب الشِفرة، في الصفحة 240، إحدى أكبر فوائد التفكير في الاختبارات هي الطريقة تُخبر بها عن الشِفرة التي تكتبها. تفقِد ذلك عندما ينفصل عمل الاختبار عن عمل كتابة الشِفرة.

بدلاً من ذلك، نحن نفضّل بديلاً، حيث يقوم الحاسوب، الذي لا يشاركك تصوراتك المُسبقة، ببعض الاختبارات لك.

العقود والثوابت والخصائص

في الموضوع 23، التصميم حسب العقد، في الصفحة 125، تحدثنا عن فكرة أن الشِفرة لديها عقود تفي بها: أنت تستوفي الشروط عندما تُغذيها بالمدخلات، وسوف تقدّم لك هي ضمانات معيّنة حول المخرجات المُنتَجة.

هناك أيضًا ثوابت الشِفرة (invariants)، وهي الأشياء التي تظل صحيحة حول جزء من الحالة (state) عندما تُمرّر بواسطة دالة ما. على سبيل المثال، إذا فرزت قائمة، فسيكون للنتيجة نفس عدد العناصر كالقائمة الأصليّة – الطول هو ثابت (invariant).

بمجرّد أن ننتهي من عقودنا وثوابتنا (التي سنجمعها معًا ونستدعي الخصائص(properties))، يمكننا استخدامها لأتمتة اختبارنا. ما ينتهي به الأمر يسمى الاختبار المعتمد على الخاصيّة.

نصيحة ٧١	استخدم الاختبارات المعتمدة على الخصائص للتحقّق صحّة افتراضاتك.

كمثال تجريبيّ، يمكننا بناء بعض الاختبارات لقائمتنا المفروزة. لقد أنشأنا فعلًا خاصيّة واحدة: القائمة التي تم فرزها هي بنفس حجم القائمة الأصلية. يمكننا أيضًا أن نذكّر أنه لا يمكن أن يكون أي عنصر في النتيجة أكبر من العنصر الذي يليه.

يمكننا الآن التعبير عن ذلك في الشِفرة. معظم اللغات لديها نوع من أطر الاختبار المعتمد على الخاصيّة. هذا المثال موجود في بايثون، ويستخدم أداة Hypothesis[127] و pytest[128]، لكن المبادئ منتشرة جدًا.

هنا الشِفرة المصدرية الكاملة للاختبارات:

```
proptest/sort.py
from hypothesis import given
import hypothesis.strategies as some
@given(some.lists(some.integers()))
def test_list_size_is_invariant_across_sorting(a_list):
    original_length = len(a_list)
    a_list.sort()
    assert len(a_list) == original_length
@given(some.lists(some.text()))
def test_sorted_result_is_ordered(a_list):
    a_list.sort()
    for i in range(len(a_list) - 1):
        assert a_list[i] <= a_list[i + 1]
```

ليك ما يحدث عند تشغيلها:

```
$ pytest sort.py
======================== test session starts ========================
...
plugins: hypothesis-4.14.0
sort.py ..                                                    [100%]
===================== 2 passed in 0.95 seconds ======================
```

ليس هناك الكثير من الدراما. ولكن، خلف الكواليس، أجرت Hypothesis كلا من اختباراتنا مئة مرّة، مُجتازةً قائمة مختلفة في كلّ مرّة. سيكون للقوائم أطوال متفاوتة، وسيكون لها محتويات مختلفة. يبدو الأمر كما لو أننا أعددنا 200 اختبار فردي باستخدام 200 قائمة عشوائية.

اختبار توليد البيانات

كما هو الحال مع معظم مكتبات الاختبار المعتمدة على الخصائص، تمنحك Hypothesis لغة صغيرة لوصف البيانات التي يجب توليدها. تعتمد اللغة على استدعاءات لدوال في وحدة "hypothesis. strategies"، التي أطلقنا عليها اسم some ، فقط لأنه يقرأ بشكل أفضل.

إذا كتبنا:

```
@given(some.integers())
```

ستُنفّذ دالة الاختبار الخاصّة بنا عدّة مرات. وسيُمرّر عدد صحيح مختلف في كل مرّة. إذا كتبنا بدلاً من ذلك ما يلي:

```
@given(some.integers(min_value=5, max_value=10).map(lambda x: x * 2))
```

سنحصل عندها على الأرقام الزوجيّة بين 10 و 20.

127 **[المترجم]** Hypothesis هي مكتبة اختبار متقدمة لبايثون. تتيح لك كتابة الاختبارات التي تم تحديدها بمصدر من الأمثلة، ثم تولد أمثلة بسيطة ومفهومة تجعل اختباراتك تفشل. يتيح لك هذا العثور على المزيد من الأخطاء في التعليمات البرمجية مع عمل أقل.

128 **[المترجم]** Pytest هو إطار اختبار يسمح لنا بكتابة شفرات الاختبار باستخدام *أسلوب الثعبان*. يمكنك كتابة شِفرة لاختبار أي شيء مثل قاعدة البيانات، API، حتى واجهة المستخدم إذا كنت تريد. ولكن يتم استخدام pytest بشكل أساسي في الصناعة لكتابة اختبارات APIs.

يمكنك أيضًا إنشاء أنواع، بحيث أن

```
@given(some.lists(some.integers(min_value=1), max_size=100))
```

ستكوّن قوائم بأرقام طبيعية يبلغ طولها 100 عنصر على الأكثر.

لا يُفترض أن تكون هذه دورة تعليميّة حول إطار عمل معيّن، لذلك سوف نتخطى مجموعة من التفاصيل الممتعة وبدلاً من ذلك سنلقي نظرة على مثال حقيقي.

وِجدان افتراضات سيئة

نحن نكتب نظامًا بسيطًا لمعالجة الطلبات ومراقبة المخزون (لأن هناك دائمًا متسع لآخر). يقوم بنمذجة مستويات المخزون بكائن Warehouse "المستودع". يمكننا الاستعلام عن مستودع ما لمعرفة ما إذا كان هناك شيء ما في المخزون، وإزالة أشياء من المخزون، والحصول على مستويات المخزون الحاليّة.

إليك الشِفرة:

proptest/stock.py
```
class Warehouse:
  def __init__(self, stock):
      self.stock = stock
  def in_stock(self, item_name):
      return (item_name in self.stock) and (self.stock[item_name] > 0)
  def take_from_stock(self, item_name, quantity):
      if quantity <= self.stock[item_name]:
          self.stock[item_name] -= quantity
      else:
          raise Exception("Oversold {}".format(item_name))
  def stock_count(self, item_name):
          return self.stock[item_name]
```

كتبنا اختبار الوحدة الأساسي، والذي ينجح:

proptest/stock.py
```
def test_warehouse():
      wh = Warehouse({"shoes": 10, "hats": 2, "umbrellas": 0})
      assert wh.in_stock("shoes")
      assert wh.in_stock("hats")
      assert not wh.in_stock("umbrellas")
      wh.take_from_stock("shoes", 2)
      assert wh.in_stock("shoes")
      wh.take_from_stock("hats", 2)
      assert not wh.in_stock("hats")
```

ثم كتبنا دالة تعالج طلبًا لطلب عناصر من المستودع. ترجع بقائمة (tuple) حيث يكون العنصر الأول إما "موافق"(ok) أو "غير متوفر" (not available)، متبوعًا بالعنصر والكمية المطلوبة. كتبنا أيضًا بعض الاختبارات، ونجحوا:

proptest/stock.py
```
def order(warehouse, item, quantity):
  if warehouse.in_stock(item):
      warehouse.take_from_stock(item, quantity)
      return ( "ok", item, quantity )
  else:
      return ( "not available", item, quantity )
```

proptest/stock.py

```
def test_order_in_stock():
    wh = Warehouse({"shoes": 10, "hats": 2, "umbrellas": 0})
    status, item, quantity = order(wh, "hats", 1)
    assert status == "ok"
    assert item == "hats"
    assert quantity == 1
    assert wh.stock_count("hats") == 1
def test_order_not_in_stock():
    wh = Warehouse({"shoes": 10, "hats": 2, "umbrellas": 0})
    status, item, quantity = order(wh, "umbrellas", 1)
    assert status == "not available"
    assert item == "umbrellas"
    assert quantity == 1
    assert wh.stock_count("umbrellas") == 0
def test_order_unknown_item():
    wh = Warehouse({"shoes": 10, "hats": 2, "umbrellas": 0})
    status, item, quantity = order(wh, "bagel", 1)
    assert status == "not available"
    assert item == "bag
```

يبدو من الخارج أن كل شيء على ما يرام. ولكن قبل شحن الشِفرة، دعنا نضيف بعض اختبارات الخاصيّة.

هناك شيء واحد نعرفه وهو أن المخزون لا يمكن أن يظهر ويختفي عبر مناقلاتنا. هذا يعني أنه إذا أخذنا بعض العناصر من المستودع، فيجب أن يكون الرَقَم الذي أخذناه بالإضافة إلى الرَقَم الموجود حاليًا في المستودع يساوي الرَقَم الأصلي الموجود في المستودع. في الاختبار التالي، نجري اختبارنا باستخدام مُعامل العنصر المختار عشوائيًا من "قبعة" (hat) أو "حذاء" (shoe) والكمية المختارة من 1 إلى 4:

proptest/stock.py

```
@given(item = some.sampled_from(["shoes", "hats"]),
       quantity = some.integers(min_value=1, max_value=4))
def test_stock_level_plus_quantity_equals_original_stock_level(item, quantity):
    wh = Warehouse({"shoes": 10, "hats": 2, "umbrellas": 0})
    initial_stock_level = wh.stock_count(item)
    (status, item, quantity) = order(wh, item, quantity)
    if status == "ok":
        assert wh.stock_count(item) + quantity == initial_stock_level
```

دعنا نشغله:

```
$ pytest stock.py
. . .
stock.py:72:

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
stock.py:76: in test_stock_level_plus_quantity_equals_original_stock_level
    (status, item, quantity) = order(wh, item, quantity)
stock.py:40: in order
    warehouse.take_from_stock(item, quantity)

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
self = <stock.Warehouse object at 0x10cf97cf8>, item_name = 'hats'
quantity = 3
    def take_from_stock(self, item_name, quantity):
    if quantity <= self.stock[item_name]:
        self.stock[item_name] -= quantity
    else:
>   raise Exception("Oversold {}".format(item_name))
E   Exception: Oversold hats
stock.py:16: Exception
```

```
---------- Hypothesis ----------
Falsifying example:
    test_stock_level_plus_quantity_equals_original_stock_level(
        item='hats', quantity=3)
```

انفجرت الشِفرة في warehouse.take_from_stock: حاولنا إزالة ثلاث قبعات من المستودع، ولكن لدينا اثنين فقط في المخزون.

وجد اختبار الخاصية لدينا افتراضًا خاطئًا: تتحقق دالة in_stock فقط من وجود عنصر واحد في الأقل من العناصر المحددة في المخزون. نحن نحتاج بدلاً من ذلك إلى التأكد أنه لدينا ما يكفي لإكمال الطلب:

```
    proptest/stock1.py
    def in_stock(self, item_name, quantity):
➤   return (item_name in self.stock) and (self.stock[item_name] >= quantity)
```

ونقوم أيضًا بتغيير دالة "الطلب" order:

```
    proptest/stock1.py
    def order(warehouse, item, quantity):
➤   if warehouse.in_stock(item, quantity):
        warehouse.take_from_stock(item, quantity)
        return ( "ok", item, quantity )
    else:
        return ( "not available", item, quantity )
```

والآن ينجح اختبار الخاصية لدينا.

غالبا ما تفاجئك الاختبارات المعتمدة على الخصائص

استخدمنا في المثال السابق اختبارًا معتمدًا على الخاصيّة للتحقّق تعديل مستويات المخزون بشكل صحيح. وجد الاختبار خطأً، ولكن لم يكن ذلك الخطأ متعلقًا بتعديل مستوى المخزون. بدلاً من ذلك، وجد خطأً في دالة in_stock.

هذه هي القوة والإحباط في الاختبار القائم على الخاصية. إنه قوي لأنك تعد بعض القواعد لتوليد المدخلات، وبإعداد بعض التأكيدات للتحقق من صحة المخرجات، ثم دعه ينفجر. أنت لا تعرف أبدا ما سيحدث. قد يمر الاختبار. قد يفشل التوكيد. أو قد تفشل الشِفرة تمامًا لأنها لا تستطيع التعامل مع المدخلات التي حصلت عليها.

والإحباط هو في أنه قد يكون من الصعب تحديد ما لذي فشل.

اقتراحنا هو أنه عندما يفشل اختبار قائم على الخاصية، اكتشف ماهي المعاملات التي كان يجتازها إلى دالة الاختبار، ثم استخدم هذه القيم لإنشاء اختبار وحدة منفصل ومنتظم. يقوم اختبار الوحدة بعمل شيئين لك. أولاً، يتيح لك التركيز على المشكلة دون إجراء جميع الاستدعاءات الإضافيّة في شفرتك بواسطة إطار الاختبار المعتمد على الخاصيّة. ثانيًا، يعمل اختبار الوحدة هذا بمنزلة اختبار انحدار. نظرًا لأن الاختبارات المعتمدة على الخصائص تولّد قيمًا عشوائية تُمرّر إلى الاختبار الخاص بك، فلا يوجد ما يضمن استخدام نفس القيم في المرّة التالية التي تقوم فيها بإجراء الاختبارات. يضمن اختبار الوحدة الذي يفرض استخدام هذه القيم أن هذا الخطأ لن يمر.

تساعد الاختبارات المعتمدة على الخاصية أيضًا تصميمك

قلنا حين تحدثنا عن اختبار الوحدة، أن إحدى الفوائد الرئيسة هي الطريقة التي جعلتك تفكر في الشِفرة: اختبار الوحدة هو العميل الأول لواجهة برمجة التطبيقات الخاصة بك.

وينطبق الشيء نفسه على الاختبارات المعتمدة على الخاصيّة، ولكن بطريقة مختلفة قليلًا. يجعلونك تفكّر في الشِفرة الخاصة بك من حيث الثوابت والعقود؛ تفكّر فيما يجب ألا يتغير، وما يجب أن يكون صحيحًا. هذه الرؤية الإضافيّة لها تأثير سحري على التعليمات البرمجيّة الخاصّة بك، وإزالة الحالات الحدّية وتسليط الضوء على الدوال التي تترك البيانات في حالة غير متناسقة.

نحن نعتقد أن الاختبار المعتمد على الخاصية مكمّل لاختبار الوحدة: فهما يعالجان اهتمامات مختلفة، وكل منهما يحقّق فوائده الخاصّة. إذا لم تكن تستخدمهما حاليًا، فجرّبهما.

الأقسام ذات الصلة

- الموضوع 23، التصميم حسب العقد، في الصفحة 125
- الموضوع 25، البرمجة التوكيدية، في الصفحة 135
- الموضوع 45، هوّة المتطلبات، في الصفحة 268

تمارين

تمرين 31 (إجابة محتملة في الصفحة 330)

انظر إلى مثال المستودع. هل هناك أي خصائص أخرى يمكنك اختبارها؟

تمرين 32 (إجابة محتملة في الصفحة 331)

تقوم شركتك بشحن الآلات. تأتي كل آلة في صندوق، وكل صندوق على شكل مستطيل. تختلف الصناديق في الحجم. مهمتك هي كتابة بعض التعليمات البرمجية لحزم أكبر عدد ممكن من الصناديق في طبقة واحدة تناسبها في شاحنة التسليم. خرج شفرتك هو قائمة بجميع الصناديق. تعطي القائمة من أجل كل صندوق الموقع في الشاحنة، إلى جانب العرض والارتفاع. ما هي خصائص الناتج التي يمكن اختبارها؟

تحدّيات

فكّر في الشِفرة الذي تعمل عليها حاليًا. ما هي الخصائص: العقود والثوابت؟ هل يمكنك استخدام إطار الاختبار المعتمد على الخاصية للتحقّق ذلك تلقائيًا؟

الموضوع 43. ابق آمنًا هناك

> الأسوار الجيّدة تصنع جيرانًا جيّدين.
>
> ← روبرت فروست، *ترميم الجدار*

في مناقشة الطبعة الأولى من اقتران الشِفرة، أدلينا ببيانٍ جريء وساذج: "لسنا بحاجة إلى أن نكون مذعورين مثل الجواسيس أو المنشقين". لقد كنا مخطئين. في الواقع، أنت بحاجة إلى أن تكون مذعورًا كل يوم.

بينما نكتب هذا، تمتلئ الأخبار اليومية بقصص عن خروقات البيانات المدمرة، والأنظمة المسروقة، والاحتيال الإلكتروني. مئات الملايين من السجلات المسروقة في نفس الوقت، ومليارات مليارات الدولارات من الخسائر والإصلاحات - وتنمو هذه الأرقام بسرعة كل عام. ليس السبب في ذلك في الغالبية العظمى من الحالات أن المهاجمين كانوا أذكياءً بشكلٍ رهيب، أو حتى مؤهلين تأهيلًا غامضًا.

بل لأن المطورين كانوا مهملين.

الـ 90٪ الأخرى

خلال كتابتك للشِفرة، قد تمرّ بعدّة دورات من "إنه يعمل!" و "لماذا لا يعمل ذلك؟" مع بعض العبارات أحيانا "لا يمكن أن يحدث..."[129] وبعد عدة تلال ومطبّات من هذا التقدّم الشاق، من السهل أن تقول لنفسك، "ها هي كلها تعمل!" وتعلن أن الشِفرة قد أُنجزت. بالطبع، لم تُنجز بعد. لقد انتهيت بنسبة ٩٠٪، ولكن الآن مازال لديك نسبة الـ ٩٠٪ *الأخرى* المتبقية.

إن الشيء التالي الذي يجب عليك فعله هو تحليل الشِفرة من أجل الطرق التي قد تسير فيها بشكل خاطئ وإضافتها إلى مجموعة الاختبار الخاصّة بك. ستفكر في أشياء مثل تمرير المعاملات السيئة، أو التسرب (leaking) أو نقص الموارد؛ شيء من هذا القبيل.

في الأيام الخوالي، ربما كان هذا التقييم للأخطاء الداخلية كافيًا. ولكن اليوم فهو ليس سوى البداية، لأنه بالإضافة إلى الأخطاء من المسببات الداخلية، تحتاج إلى التفكير في كيفيّة قيام ممثل خارجي بإفساد النظام عمداً. ولكن ربما تحتج، "أوه، لن يهتم أحد بهذه الشِفرة، إنها ليست مهمةً، ولا أحد يعرف عن هذا الخادم حتى..." إنه عالمٌ كبير هناك، ومعظمه متصل. سواء أكان ذلك الطفل الملول على الجانب الآخر من الكوكب، أو الإرهاب الذي ترعاه الدولة، أو العصابات الإجرامية، أو التجسس الجماعي، أو حتى حاقد سابق، فهم موجودون ويستهدفونك. يُقاس زمن بقاء نظام قديم وغير مُقرصن على الشبكة المفتوحة بالدقائق - أو حتى أقل.

الأمن بواسطة الغموض لا ينجح.

مبادئ الأمن الأساسيّة

لدى المبرمجين الواقعيين كميّة صحيّة من الزعر. نحن نعلم أن لدينا عيوبًا وحدودًا، وأن المهاجمين الخارجيين سيغتنمون أيّة ثغرة نتركها للإخلال بأنظمتنا. سيكون لبيئات التطوير والنشر الخاصّة بك احتياجاتها الخاصّة التي تركّز على الأمان، ولكن هناك عدد من المبادئ الأساسيّة التي يجب أن تضعها في اعتبارك دائمًا:

129 انظر تنقيح الأخطاء.

1. تقليل مساحة سطح الهجوم
2. مبدأ الامتياز الأقل
3. تأمين الافتراضيات (Secure Defaults)
4. تشفير البيانات الحساسة
5. الحفاظ على تحديثات الأمان

دعونا نلقي نظرة على كل منها.

تقليل مساحة سطح الهجوم

مساحة سطح الهجوم في النظام هي مجموع كل نِقَاط الوصول التي يمكن للمهاجم بواسطتها إدخال البيانات أو استخراج البيانات أو تنفيذ خدمة ما. وفيما يلي بعض الأمثلة على ذلك:

يؤدي تعقيد الشِفرة إلى عوامل للهجوم

إن تعقيد الشِفرة يجعل سطح الهجوم أكبر، مع المزيد من الفرص للآثار الجانبيّة غير المتوقعة. فكّر في الشِفرة المعقّدة على أنها تجعل مساحة السطح أكثر مساميّةً وأكثر عرضةً للعدوى. مرّة أخرى، الشِفرة البسيطة والأصغر هي الأفضل. شِفرة أقل تعني أخطاء أقل، وفرص أقل لثغرة أمنيّة معيقة. الشِفرة الأبسط والأكثر إحكامًا والأقل تعقيدًا أسهل في التفكير فيها ويسهل اكتشاف نِقَاط ضعفها المحتملة.

بيانات الإدخال هي عامل هجوم

لا تثق مطلقًا بالبيانات الواردة من كيان خارجي، عقّمها دائمًا قبل تمريرها إلى قاعدة بيانات، أو إلى عرض التحويل (rendering)، أو المعالجات الأخرى[130]. يمكن أن تساعد بعض اللغات في ذلك. في روبي، على سبيل المثال، تُلوّن المتغيرات التي تحتفظ بإدخال خارجي، مما يحدّ من العمليات التي يمكن إجراؤها عليها. على سبيل المثال، يبدو أن هذه الشِفرة تستخدم أداة wc للإبلاغ عن عدد الأحرف في مِلَفّ يتم توفير اسمه في وقت التشغيل:

safety/taint.rb
```
puts "Enter a file name to count: "
name = gets
system("wc -c #{name}")
```

يمكن للمستخدم الشرير أن يُلحق ضررًا مثل هذا:

```
Enter a file name to count:
test.dat ; rm -rf /
```

130 تذكر صديقنا العزيز، طاولات بوبي الصغيرة (little Bobby Tables)(https://xkcd.com/327)؟ في أثناء إعادة التفكير، ألق نظرة على https://bobby-tables.com، الذي يسرد طرق تعقيم البيانات التي مُرّرت إلى استعلامات قاعدة البيانات.

ومع ذلك، سيؤدي تعيين مستوى الأمان إلى 1 إلى تشويه البيانات الخارجية، مما يعني أنه لا يمكن استخدامه في سياقات شديد الْخَطَر:

```
➤ $SAFE = 1
puts "Enter a file name to count: "
name = gets
system("wc -c #{name}")
```

والآن عندما ننفّذها، يُقبض علينا متلبسين:

```
$ ruby taint.rb
Enter a file name to count:
test.dat; rm -rf /
code/safety/taint.rb:5:in `system': Insecure operation - system (SecurityError)
from code/safety/taint.rb:5:in `main
```

الخِدْمَات غير الخاضعة للاستيثاق هي عوامل هجوم بحكم طبيعتها، يمكن لأي مستخدم في أي مكان في العالم الاتصال بالخِدْمَات غير الخاضعة للاستيثاق، لذا فإنه بمنعك لأي معالجة أخرى أو تقييدك لها تكون بذلك قد أوجدت على الفور فرصة لهجوم حجب الخدمة (denial-of-service attack)[131] على أقل تقدير. هناك عدد غير قليل من انتهاكات البيانات العامّة للغاية التي حدثت مؤخرًا بسبب قيام المطورين عن طريق الخطأ بوضع البيانات في متاجر بيانات غير موثّقة ومقروءة للجمهور في السحابة.

الخِدْمَات الموثّقة هي عامل هجوم

حافظ على عدد المستخدمين المخوّلين في الحد الأدنى تمامًا. استبعد المستخدمين والخِدْمَات القديمة أو غير المستخدمة أو المنتهية الصَّلاحِيَة. عُثر على العديد من الأجهزة التي تم تمكين الشبكة فيها تحتوي على كلمات مرور افتراضيّة بسيطة أو حسابات إدارية غير مستخدمة وغير محميّة. إذا اختُرق حساب لديه بيانات اعتماد النشر، فسيكون منتجك بِرُمَّته معرضًا للخطر.

بيانات الخرج هي عامل هجوم

هناك قصة (ربما مِلَفّقة) عن نظام أبلغ بشكل خاطئ عن رسالة خطأ أن كلمة المرور يستخدمها مستخدم آخر. لا تعطي معلومات. تأكّد أن البيانات التي تبلغ عنها مناسبة للتفويض الممنوح لذلك المستخدم. اقتطع أو موّه المعلومات التي يحتمل أن تكون خطرة مثل أرقام الضمان الاجتماعي أو أرقام الهويات الحكوميّة الأخرى.

معلومات التنقيح هي ناقل هجوم

131 **[المترجم]** هجمات الحرمان من الخِدْمَات أو هجوم حجب الخدمة (بالإنجليزية: Denial of Service Attacks) هي هجمات تتم عن طريق إغراق المواقع بسيل من البيانات غير اللازمة يتم إرسالها عن طريق أجهزة مصابة ببرامج (في هذه الحالة تسمى DDOS Attacks) تعمل على نشر هذه الهجمات بحيث يتحكم فيها القراصنة والعابثين الإلكترونيين لمهاجمة الشبكة (الإنترنت) عن بعد، بإرسال تلك البيانات إلى المواقع بشكل كثيف، مما يسبب بطء الخِدْمَات أو ازدحاماً مرورياً بهذه المواقع ويسبب صعوبة وصول المستخدمين لها نظراً لهذا الاكتظاظ.

لا يوجد شيء يثلج الصدر مثل رؤية تتبع مكدّس كامل مع بيانات على جهاز الصراف الآلي المحلي، أو كشك المطار، أو صفحة الويب المتعطلة. يمكن أن تؤدي المعلومات المصمّمة لتسهيل عملية التنقيح إلى تسهيل الاختراق أيضًا. تأكّد أن أي "نافذة اختبار" (تمت مناقشتها في الصفحة 221) وتقارير استثناءات وقت التشغيل محميّة من أعين التجسس.[132]

نصيحة ٧٢	**حافظ على بساطة الشِفرة، وقلّل أسطح الهجوم.**

مبدأ الامتياز الأقل

هناك مبدأ رئيس آخر هو استخدام أقل قدر من الامتياز لأقصر وقت ممكن. بعبارة أخرى، لا تحصُل تلقائيًا على أعلى مستوى أذونات، مثل الجذر (root) أو المسؤول (Administrator). إذا كانت هناك حاجة لهذا المستوى العالي، خذه، وقم بأدنى قدر من العمل، وانزل عن إذنك بسرعة لتقليل المخاطر. يعود هذا المبدأ إلى أوائل السبعينيات:

> يجب أن يعمل كل برنامَج وكل مستخدم ذو امتياز للنظام باستخدام أقل قدر من الامتياز اللازم لإكمال المهمة. - جيروم سولتزر، اتصالات ACM، 1974.

خذ مثلا برنامَج "تسجيل الدخول" على الأنظمة المشتقة من يونكس. تُنفَّذ في البداية بامتيازات الجذر (root). بمجرّد الانتهاء من استيثاق المستخدم الصحيح، فإنه يُخفّض الامتياز عالي المستوى إلى امتياز المستخدم.

لا ينطبق هذا فقط على مستويات امتياز نظام التشغيل. هل يقوم تطبيقك بتنفيذ مستويات مختلفة من الوصول؟ هل هي أداة غير حادة مثل " administrator" مقابل " user"؟ إذا كان الأمر كذلك، فكّر في شيء أكثر دقّة، حيث تُقسّم الموارد الحسّاسة إلى تصنيفات (categories) مختلفة، ويكون لدى المستخدمين الفرديين أذونات لبعض هذه التصنيفات فقط.

تتبع هذه التقنية نفس نوع فكرة تقليل مساحة السطح- تقليل نطاق عوامل الهجوم، حسب الوقت ومستوى الامتياز. في هذه الحالة، القلّة مطلوبة أكثر.

تأمين الافتراضيات

يجب أن تكون الإعدادات الافتراضية في تطبيقك، أو للمستخدمين على موقعك، هي القيم الأكثر أمانًا. قد لا تكون هذه هي القيم الأكثر ملاءمة للمستخدم أو الأكثر قيمة، ولكن من الأفضل السماح لكل فرد بأن يقرّر بنفسه المفاضلة بين الأمان والراحة.

على سبيل المثال، قد يكون الإعداد الافتراضي لإدخال كلمة المرور هو إخفاء كلمة المرور عند إدخالها، مع استبدال كل حرف بعلامة النجمة. إذا كنت تدخل كلمة مرور في مكان عام مزدحم، أو يتم عرضها أمام جَمهور كبير، فهذا الأعداد الافتراضي معقول. ولكن

132 أثبتت هذه التقنية نجاحها على مستوى شريحة وحدة المعالجة المركزية (CPU)، حيث تستهدف عمليات الاستغلال المعروفة تنقيح الأخطاء والتسهيلات الإدارية. بمجرّد تصدعها، تترك الآلة بِرُمّتها مكشوفة.

قد يرغب بعض المستخدمين في رؤية كلمة المرور مكتوبة، ربما لإمكانية الوصول. إذا كان هناك خطر ضئيل أن ينظر شخص ما من فوق كتفهم، فهذا خِيار معقول لهم.

تشفير البيانات الحساسة

لا تترك معلومات التعريف الشخصيّة أو البيانات المالية أو كلمات المرور أو بيانات الاعتماد الأخرى في نصٍ واضح، سواء في قاعدة بيانات أو في مِلَفّ خارجي آخر. إذا تعرضت البيانات للكشف، سيوفّر التشفير مستوى إضافيًا من الأمان.

في التحكم في الإصدار، نوصي بشدة بوضع كل ما يلزم للمشروع تحت التحكم في الإصدار. حسنا، كل شيء تقريبًا. إليك أحد الاستثناءات الرئيسة لهذه القاعدة:

لا تتحقق الأسرار أو مفاتيح واجهة برمجة التطبيقات أو مفاتيح SSH أو كلمات مرور التشفير أو بيانات الاعتماد الأخرى إلى جانب شِفرة المصدر في التحكم في الإصدار.

يجب إدارة المفاتيح والأسرار بشكل منفصل، عمومًا عبر مِلَفّات الضبط أو متغيرات البيئة كجزء من البناء والنشر.

الحفاظ على تحديثات الأمان

يمكن أن يكون تحديث أنظمة الحاسوب بمنزلة ألم كبير. تحتاج إلى هذا التصحيح الأمني (security patch) - ولكن كأثر جانبي فهو يعطّل جزءًا من تطبيقك. يمكنك أن تقرّر الانتظار وتأجيل التحديث حتى وقت لاحق. هذه فكرة فظيعة، لأن نظامك الآن عرضةٌ لاستغلالٍ معروف.

نصيحة ٧٣	طبّق تصحيحات الأمان بسرعة.

تؤثّر هذه النصيحة على كل جهاز متصل بالإنترنت، بما في ذلك الهواتف والسيارات والأجهزة المحمولة وأجهزة الحاسوب الشخصية وأجهزة المطورين وآلات البناء (build machines) وخوادم الإنتاج والصور السحابية. كل شيء. وإذا كنت تعتقد أن هذا لا يهم حقًا، فتذكر فقط أن أكبر اختراق للبيانات في التاريخ (حتى الآن) كانت بسبب الأنظمة التي كانت متأخرة في تحديثاتها.

لا تدع ذلك يحدث لك.

كلمات المرور غير المطابقة للنمط (Password Antipatterns)

إحدى المشكلات الأساسيّة للأمن هي أن الأمن الجيّد في كثير من الأحيان يتعارض مع الحس السليم أو الممارسة الشائعة. على سبيل المثال، قد تعتقد أن متطلبات كلمة المرور الصارمة ستزيد أمان تطبيقك أو موقعك. ستكون مُخطئا.

ستقلّل سياسات كلمة المرور الصارمة من مستوى الأمان لديك. هذه قائمة قصيرة من الأفكار السيئة للغاية، إلى جانب بعض التوصيات من المعهد القومي للمعايير والتقنية:[1]

- لا تقيّد طول كلمة المرور على أقل من 64 حرفًا. توصي NIST بـ 256 كحد أقصى لطول جيّد.
- لا تقتطع كلمة المرور التي اختارها المستخدم.
- لا تقيّد الأحرف الخاصة مثل [ ] ( ) ؛ &٪ $ # أو /. انظر الملاحظة عن طاولات بوبي (Bobby Tables) سابقًا في هذا القسم. إذا كانت الأحرف الخاصة في كلمة المرور الخاصة بك ستعرض نظامك للخطر، فأنت تواجه مشكلات أكبر. تدعو NIST لقبول جميع أحرف ASCII والطباعة والمساحة و اليونيكود (Unicode).
- لا تقدم تلميحات لكلمة المرور للمستخدمين غير الخاضعين للاستيثاق، أو المطالبة بأنواع معيّنة من المعلومات (على سبيل المثال، "ما اسم حيوانك الأليف الأول؟").
- لا تُعطّل وظيفة اللصق في المتصفح. إن تعطيل وظائف المستعرضين ومديري كلمات المرور لا يجعل نظامك أكثر أمانًا، بل إنه في الواقع يدفع المستخدمين إلى إنشاء كلمات مرور أبسط وأقصر يسهل اختراقها. يتطلب كل من NIST في الولايات المتحدة والمركز الوطني للأمن السيبراني في المملكة المتحدة على وجه التحديد أدوات التحقّق للسماح بعمل وظيفة اللصق لهذا السبب.
- لا تفرض قواعد صياغة أخرى. على سبيل المثال، لا تفرض أي مزيج معين من الأحرف الكبيرة والصغيرة، أو الأعداد، أو الأحرف الخاصّة، أو تمنع تكرار الأحرف، وما إلى ذلك.
- لا تطلب بشكل تعسفي من المستخدمين تغيير كلمات المرور الخاصّة بهم بعد مرور بعض الوقت. لا تفعل ذلك إلا لسبب وجيه (على سبيل المثال، إذا كان هناك خرق).

تريد تشجيع كلمات المرور الطويلة والعشوائية بدرجة عالية من الاعتلاج. يحد وضع قيود مصطنعة من الاعتلاج ويشجع على عادات كلمة المرور السيئة، مما يجعل حسابات المستخدم عرضة للاستيلاء عليها.

---

1. *منشور NIST Special 800-63B: إرشادات الهُوِيَّة الرَقَمية*: الاستيثاق وإدارة دورة الحياة، وهي متاحة مجانًا عبر الإنترنت على https://doi.org/10.6028/NIST.SP.800-63b

الحس العام مقابل التشفير

من المهم أن تضع في اعتبارك أن الحس السليم قد يخزلك عندما يتعلق الأمر بمسائل التشفير. القاعدة الأولى والأكثر أهميّة عندما يتعلق الأمر بالتشفير لا تفعل ذلك بنفسك أبدًا.[133] بمجرّد دخولك إلى عالم التشفير، حتى أصغر الأخطاء وأقلها أهميّة يمكن أن يؤثر على كل شيء: من المحتمل أن تُكسر خوارزمية التشفير الذكيّة الجديدة محليّة الصنع من قبل خبير في دقائق.

لا تريد إجراء التشفير بنفسك.

كما قلنا في مكان آخر، لا تعتمد إلا على أشياء موثوق بها: يتم فحصها جيدًا، وفحصها بدقّة، وصيانتها بشكلٍ جيّد، وتحديثها بشكلٍ متكرر، ويفضّل أن تكون مكتبات وأطر عمل مفتوحة المصدر.

بالإضافة إلى مهام التشفير البسيطة، ألق نظرة فاحصة على الميزات الأخرى المتعلقة بالأمان في موقعك أو تطبيقك. خذ الاستيثاق، على سبيل المثال.

تحتاج من أجل تنفيذ تسجيل الدخول الخاص بك باستخدام كلمة المرور أو الاستيثاق البيومتري (biometric authentication) إلى فهم كيفيّة عمل التجزئة والأملاح، وكيفيّة استخدام المخترقين لأشياء مثل جداول قوس قزح (Rainbow)[134]، ولماذا لا يجب استخدام MD5 أو SHA1، ومجموعة من المخاوف الأخرى. حتى إذا حصلت على كل ذلك بشكل صحيح، في نهاية اليوم تظل مسؤولاً عن الاحتفاظ بالبيانات والحفاظ عليها آمنة، مع مراعاة أي تشريعات جديدة والتزامات قانونية جديدة.

أو يمكنك اتباع النهج الواقعي والسماح لشخص آخر بالقلق بشأنها واستخدام موفر استيثاق تابع لجهة خارجية. قد تكون هذه خدمة جاهزة يتم تشغيلها داخل الشركة، أو قد تكون طرفًا ثالثًا في السحابة. غالبًا ما تكون خِدْمَات الاستيثاق متاحة من موفري البريد الإلكتروني أو الهاتف أو وسائل التواصل الاجتماعي، التي قد تكون أو لا تكون مناسبة لتطبيقك. على أي حال، يقضي هؤلاء الأشخاص كل أيامهم في الحفاظ على أنظمتهم آمنة، وهم أفضل حالًا منك.

ابق آمنا هناك.

الأقسام ذات الصلة

- الموضوع 23، التصميم حسب العقد، في الصفحة 125
- الموضوع 24، البرامج الميتة لا تكذب، في الصفحة 133
- الموضوع 25، البرمجة التوكيدية، في الصفحة 135
- الموضوع 38، البرمجة بالمصادفة، في الصفحة 223

133 إلا إذا كان لديك درجة الدكتوراه في التشفير، حتى في تلك الحالة فقط مع مراجعة الأقران الرئيسة، والتجارب الميدانية واسعة النطاق مع حصيلة أخطاء، وميزانية للصيانة على المدى الطويل.

134 **[المترجم]** جدول قوس قزح هو قائمة بجميع تباديل النص الصريح الممكنة لكلمات المرور المشفرة الخاصة بخوارزمية تجزئة معينة. جميع أنظمة الحاسوب التي تتطلب المصادقة المستندة إلى كلمة المرور تخزّن قواعد بيانات كلمات المرور المرتبطة بحسابات المستخدمين، وعادةً ما تكون مشفّرة بدلاً من النص العادي كإجراء أمني.

- الموضوع 45، هوّة المتطلبات، في الصفحة 268

الموضوع 44. تسمية الأشياء

> بداية الحكمة هي تسمية الأشياء بمسمياتها الصحيحة.
>
> ← كونفوشيوس

ما عَلاقة الاسم؟ عندما نبرمج، يكون الجواب هو "كل العَلاقة!"

نحن ننشئ أسماءً للتطبيقات والأنظمة الفرعيّة والوحدات والدوال والمتغيرات - ننشئ باستمرار أشياء جديدة ونعطيها أسماءً. وهذه الأسماء مهمّة جدًا، لأنها تكشف الكثير عن نيّتك واعتقادك.

نعتقد أنه يجب تسمية الأشياء وفقًا للدور الذي تلعبه في الشِفرة الخاصّة بك. هذا يعني أنه كلما أنشأت شيئًا ما، فأنت بحاجة إلى التوقف لوهلة والتفكير في "ما دافعي لإنشاء هذا؟"

إن هذا سؤال قوي، لأنه يُخرجك من عقليّة حل المشكلات الفورية ويجعلك تنظر إلى الصورة الأكبر. عندما تفكّر في دور المتغير (variable) أو الدّالة، فإنك تفكّر فيما هو مميز فيها، وفيما يمكنها فعله، وما الذي تتفاعل معه. غالبًا ما نجد أنفسنا ندرك أن ما كنا على وشك القيام به ليس له معنى، كل ذلك لأننا لم نستطع التوصل إلى اسم مناسب.

هناك بعض العلم خلف فكرة أن الأسماء ذات مغزىً عميق. اتضح أنه يمكن للدماغ قراءة وفهم الكلمات بسرعة كبيرة: أسرع من العديد من الأنشطة الأخرى. هذا يعني أن الكلمات لها أولوية معينة عندما نحاول فهم شيء ما. يمكن إثبات ذلك باستخدام تأثير ستروب Stroop.[135]

انظر إلى اللوحة التالية. تحتوي قائمة بأسماء الألوان (أو الظلال) يعرض كل منها بلون (أو ظل) لكن ليس بالضرورة أن تتطابق الأسماء والألوان. إليك الجزء الأول من التحدي- قل بصوت عالٍ اسم كل لون كما هو مكتوب:[136]

135 دراسات التدخل في ا

136 لدينا نسختان من هذه اللوحة. تستخدم أحدهما ألوانًا مختلفة، والأخرى تستخدم ظلالًا رمادية. إذا كنت ترى هذا باللونين الأبيض والأسود وتريد الإصدار الملون، أو إذا كنت تواجه مشكلة في تمييز الألوان وترغب في تجربة الإصدار الرمادي، فانتقل إلى https://pragprog.com/the-pragmatic-programmer/stroop-effect.

كرّر ذلك الآن، ولكن بدلاً من ذلك قل بصوت عالٍ اسم اللون المستخدم لرسم الكلمة. أصعب، أليس كذلك؟ من السهل أن تكون بارعًا في القراءة، ولكن الحال أكثر صعوبةً عند محاولة التعرف على الألوان.

يعامل دماغك الكلمات المكتوبة على أنها شيء يجب احترامه. نحن بحاجة للتأكّد أن الأسماء التي نستخدمها ترقى إلى ذلك المستوى.

لنلق نظرة على مثالين:

- نحن نصادق "استيثاق" (authenticating) الأشخاص الذين يصلون إلى موقعنا الذي يبيع المجوهرات المصنوعة من بطاقات الرسوميات القديمة:

```
let user = authenticate(credentials)
```

المتغير هو user لأنه *دائمًا* user. لكن لماذا؟ إنه لا يعني شيئا. ماذا لو أسميناه customer أو buyer ؟ وبهذه الطريقة نحصل على رسائل تذكير مستمرة بينما نكتب الشِفرة لما يحاول هذا الشخص القيام به، ولما يعنيه ذلك لنا.

- لدينا مثيل للطريقة (method) التي تطبق خصم على الطلب:

```
public void deductPercent(double amount)
// ...
```

هنا يوجد شيئين. أولاً، إن deductPercent هو ***ما*** يفعله وليس ***لماذا*** يفعله. ثم إن اسم المعامل amount مُضلل في أحسن الأحوال: هل هو مبلغ ثابت، نسبة مئوية؟

ربما سيكون هذا أفضل:

```
public void applyDiscount(Percentage discount)
// ...
```

اسم الطريقة الآن يوضح قصدها. لقد غيّرنا أيضا المعامل من نوع double إلى Percentage ، وهو النوع الذي عرّفناه. لا نعرف بالنسبة لك ما هو، ولكن عند التعامل مع النسب المئوية، لا نعرف أبدًا ما إذا كان من المفترض أن تتراوح القيمة بين 0 و 100 أو 0.0 و 1.0. باستخدام وثائق نوع تتوقعها الدالة.

- لدينا وحدة تقوم بأشياء مثيرة للاهتمام باستخدام أرقام فيبوناتشي. أحد هذه الأشياء هو حساب ترتيب الرَقَم (n^{th}) في المتتالية. توقف وفكّر فيما ستسمي به هذه الدالة.

معظم الناس الذين نسألهم سيطلقون عليها اسم fib. يبدو معقولًا، ولكن تذكر أنه عادةً ما ستُستدعى في سياق الوحدة الخاصة بها، لذلك سيكون الاستدعاء Fib.fib (n). ماذا لو نسميها بـ of أو n^{th} بدلاً من ذلك:

```
Fib.of(0) # => 0
Fib.nth(20) # => 4181
```

عند تسمية الأشياء، فأنت تبحث باستمرار عن طرق لتوضيح ما تعنيه، وسيؤدي هذا التوضيح إلى فهم أفضل لشفرتك في *أثناء كتابتها*.

ومع ذلك، يجب ألا تكون جميع الأسماء مرشحة لنيل جائزة أدبية.

الاستثناء الذي يثبت القاعدة

بينما نسعى جاهدين من أجل الوضوح في الشِفرة، فإن العلامة التجارية هي أمر مختلف تمامًا.

هناك تقاليد راسخة مفادها أن المشروعات وفرق المشروعات يجب أن يكون لها أسماء "ذكيّة" غامضة. أسماء البوكيمون، والأبطال الخارقين في Marvel، والتدييات اللطيفة، وشخصيات ملك الخواتم "Lord of the Rings"، وغيرها من الأسماء.

حرفيًا

احترام الثقافة

هناك شيئان صعبان فقط في علوم الحاسوب: إبطال ذاكرة التخزين المؤقت (cache invalidation) وتسمية الأشياء.

معظم نصوص الحاسوب التمهيدية ستنبّهك إلى عدم استخدام متغيرات الأحرف الفردية مثل i أو j أو k.[137]

نعتقد أنهم مخطئون. نوعًا ما.

في الواقع، يعتمد ذلك على ثقافة اللغة أو بيئة البرمجة المعيّنة. في لغة البرمجة سي، يُستخدَم i و j و k بشكل تقليدي كمتغيرات لزيادة الحلقة، وتستخدم s للسلسلة المحرفيّة، وما إلى ذلك. إذا كنت تبرمج في تلك البيئة، فهذا ما اعتدت رؤيته وسيكون من اللغط (ثَمّ الخطأ) انتهاك هذه القاعدة. من ناحية أخرى، فإن استخدام ذاك العرف في بيئة مختلفة حيث لا يُتوقع وجوده هو أمر خاطئ. لن تفعل شيئًا شنيعًا مثل مثال كلوجور هذا الذي يُسند سلسلة للمتغير i:

```
(let [i "Hello World"]
(println i))
```

تفضّل بعض مجتمعات اللغات نمط camelCase، باستخدام أحرف كبيرة مُضمنة، بينما يفضل البعض الآخر snake_case مع تسطير سفلي مضمن لفصل الكلمات. بالطبع ستقبل اللغات نفسها أنماط أخرى، ولكن هذا لا يجعلها صحيحة. احترم الثقافة المحليّة.

تسمح بعض اللغات بمجموعة فرعية من معيار الترميز (Unicode) في الأسماء. استشعر ما يتوقعه المجتمع قبل أن تنساق إلى أسماء مثل ɹəsn أو εξέρχεται.

الاتساق

يشتهر إيمرسون بمقولته "الاتساق الأحمق هو بعبع العقول الصغيرة ..."، لكن إيمرسون لم يكن ضمن فريق من المبرمجين.

137 هل تعلم لماذا يستخدم i عادة كمتغير حلقة؟ تأتي الإجابة منذ أكثر من 60 عامًا، عندما كانت المتغيرات التي تبدأ بـ I إلى N أعدادًا صحيحة في فورتران (FORTRAN) الأصلية. وكانت فورتران بدورها متأثرة بالجبر.

كل مشروع له مفرداته الخاصّة: الكلمات الاصطلاحية التي لها معنى خاص للفريق. يعني "الطلب" (Order) شيئًا ما لفريق يُنشئ متجرًا عبر الإنترنت، وشيئًا آخر مختلفًا تمامًا لفريق يرسم تطبيقه رُسُوم بيانية لنَسَب الجماعات الدينية. من المهم أن يعرف كل فرد في الفريق ما تعنيه هذه الكلمات، وأن يستخدموها باستمرار.

إحدى الطرق هي التشجيع على الكثير من التواصل. إذا قام كل شخص بإقران (pair) البرامج، وتبديل الأزواج بشكل متكرر، فإن المصطلحات ستنتشر تدريجيًا.

هناك طريقة أخرى وهي أن يكون لديك مسرد للمشروع، يسرد المصطلحات التي تحمل معنى خاص للفريق. إنها وثيقة غير رسميّة، ربما يتم الاحتفاظ بها على ويكي، ربما مجرّد بطاقات فهرسة على الحائط في مكان ما.

بعد مدّة، سيكون للغة المشروع حياة خاصّة بها. ومع ارتياح الجميع للمفردات، ستتمكن من استخدام المصطلحات كاختصار، مُعبّرًا عن الكثير من المعنى بدقة وإيجاز. (هذا بالضبط ما تعنيه لغة النمط.)

إعادة التسمية كذلك أصعب

هناك مشكلتان صعبتان في علوم الحاسوب: إبطال ذاكرة التخزين المؤقت، وتسمية الأشياء، وأخطاء "بمقدار 1" (off-by-one errors).

إن الأشياء تتغير، بغض النظر عن مقدار الجهد الذي بذلته مقدمًا. يعاد بناء الشِفرة، فترات الاستخدام، ويتغيّر المعنى بمهارة. إذا لم تكن متيقظًا لتحديث الأسماء في أثناء التنقل، فيمكن أن تهوي سريعًا إلى كابوس أسوأ بكثير من الأسماء عديمة المعنى: الأسماء المضلّلة. هل سبق وأن شرح لك شخص ما التناقضات في الشِفرة مثل "الروتين المُسمى getData يقوم فعلًا بكتابة البيانات إلى مِلَفّ أرشفة"؟

كما كنا نناقش في اعتلاج البرمجيات، عندما تكتشف مشكلة، أصلحها هنا والآن. وعندما ترى اسمًا لم يعد يعبر عن القصد، أو مضللاً أو مربكًا، فأصلحه. لديك اختبارات انحدار كاملة، لذلك ستكتشف أي حالات ربما فاتتك.

نصيحة ٧٤ **سمّ بشكل جيد؛ أعد التسمية عند الحاجة.**

إذا لم تتمكن لسببٍ أو لآخر من تغيير الاسم الخطأ الآن، فلديك مشكلة أكبر: انتهاك ETC (راجع **جوهر التصميم الجيّد**). أصلح ذلك أولاً، ثم غيّر الاسم المسيء. اجعل إعادة التسمية أمرًا سهلاً، وافعل ذلك كثيرًا.

وإلا، سيكون عليك أن تشرح للأشخاص الجدد في الفريق أن getData يكتب فعلًا البيانات في مِلَفّ، وعليك فعل ذلك بشكلٍ مباشر.

الأقسام ذات الصلة

- الموضوع 3، اعتلاج البرمجيات، في الصفحة 31
- الموضوع 40، إعادة البناء، في الصفحة 235
- الموضوع 45، هوّة المتطلبات، في الصفحة 268

تحديات

- عندما تجد دالة أو طريقة تحمل اسمًا عامًا جدًا، حاول إعادة تسميتها للتعبير عن جميع الأشياء التي تفعلها حقًا. وبذلك تصبح هدفًا أسهل لإعادة البناء.
- في أمثلتنا، اقترحنا استخدام أسماء أكثر تحديدًا مثل *المشتري* بدلاً من *المستخدم* الأكثر تقليديّة وعموميّة. ما الأسماء الأخرى التي تستخدمها عادة والتي يمكن أن تكون أفضل؟
- هل تتطابق الأسماء في نظامك مع مصطلحات المستخدم في المجال؟ إذا لم يكن كذلك، لماذا؟ هل هذا يسبب تنافرًا معرفيًا من نمط *تأثير ستروب* للفريق؟
- هل من الصعب تغيير الأسماء في نظامك؟ ما الذي يمكنك فعله لإصلاح تلك النافذة المكسورة؟

الفصل الثامن:

قبل بَدْء المشروع

8

في بداية المشروع، ستحتاج أنت والفريق إلى معرفة المتطلبات. فمجرد إخبار المستخدمين لك بما يجب فعله أو الاستماع إليهم غير كافٍ: اقرأ **هوّة المتطلبات** وتعلم كيفيّة تجنّب الأشراك والمخاطر الشائعة.

إن الحكمة التقليدية وإدارة القيود هما موضوعا **حل الألغاز المستحيلة**. سواء كنت تنقذ المتطلبات أو التحليل أو كتابة الشِفرة أو الاختبار، فستظهر مشكلات صعبة. في معظم الأوقات، لن تكون هذه المشكلات بالصعوبة التي تبدو عليها في البداية.

وعندما يأتي هذا المشروع المستحيل، نودّ أن نلجأ إلى سلاحنا السري: **العمل معًا**. لا نعني بـ "العمل معًا"، مشاركة مستند متطلبات ضخم، أو إرسال نسخ رسائل بريد إلكتروني مكثفة أو اجتماعات لا نهاية لها. إنما نعني حل المشكلات معًا في أثناء البرمجة. سنوضح لك من تحتاجه وكيف تبدأ.

مع أنّ بيان التطوير الرشيق "آجايل" (Agile) يبدأ بـ "الأفراد والتفاعلات على العمليات والأدوات"، فإن جميع المشروعات "الذكية" تقريبًا تبدأ بمناقشة ساخرة حول العملية التي سيختارونها والأدوات التي سيستخدمونها. ولكن بغض النظر عن مدى جودة التفكير في الأمر، وبصرف النظر عن "أفضل الممارسات" التي يتضمنها، فلا توجد طريقة يمكن أن تحل محل التفكير. فأنت لا تحتاج إلى أي عملية أو أداة معينة، ما تحتاجه هو "**جوهر التطوير الرشيق**".

بواسطة حل هذه القضايا الحرجة قبل بَدء المشروع، يمكنك أن تكون في وضع أفضل لتجنب "شلل التحليل" والبَدء فعلًا - وإكمال - مشروعك الناجح.

الموضوع 45. هوّة المتطلبات

> يتحقق الكمال، ليس عندما لا يتبقى شيء لإضافته
> ولكن عندما لا يتبقى شيء لحذفه...
> ^ أنطوان دي سانت إكسوبيري، رياح، ورمال، ونجوم، 1939

تشير العديد من الكتب والبرامج التعليمية إلى تجميع المتطلبات كمرحلة مبكرة من المشروع. تبدو كلمة "تجميع" (gathering) وكأنها تشير إلى قبيلة من المحللين السعداء الذين يبحثون عن شذرات الحكمة الملقاة على الأرض من حولهم بينما تُعزف السمفونية الرعوية بلطف في الخلفية. توحي كلمة "تجميع" أن المتطلبات موجودة فعلًا - ما عليك سوى العثور عليها ووضعها في سلتك مُنتشيًا وأنت في طريقك.

إن الأمر ليس بهذا الشكل مطلقًا. فنادرًا ما تكمن المتطلبات على السطح. عادة، ما تُدفن عميقاً تحت طبقات من الافتراضات والمفاهيم الخاطئة والسياسات. والأسوأ من ذلك، أنها غالبًا غير موجودة على الإطلاق.

نصيحة ٧٥ لا أحد يعرف بالضبط ما يريد.

أسطورة المتطلبات

في الأيام الأولى للبرمجيات، كانت أجهزة الحاسوب أكثر قيمةً (من حيث التكلفة المُستهلَكة لكل ساعة) من الأشخاص الذين عملوا معها. لقد وفرنا المال بواسطة محاولة تصحيح الأمور من المرّة الأولى. كان جزء من هذه العملية يحاول تحديد بالضبط ما سنجعل الآلة تفعله. سنبدأ بالحصول على مواصفات المتطلبات، ونجمعها في مستند تصميم، ثم في مخططات انسيابية وشِفرة زائفة (pseudo code)، وأخيرًا في شِفرة. قبل إدخالها إلى جهاز الحاسوب، كنا نقضي وقتًا في فحصها.

يكلف هذا الكثير من المال. وهذه التكلفة تعني أن الناس حاولوا أتمتة شيء ما فقط عندما عرفوا بالضبط ما يريدون. ونظرًا لأن الأجهزة المبكرة كانت محدودة نوعا ما، فإن نطاق المشكلات التي تم حلها كان محدودًا: كان بالإمكان فعلًا فهم المشكلة بِرُمَّتها قبل أن بَدْء العمل.

لكن العالم الحقيقي ليس كذلك. فهو فوضوي ومتضارب وغير معروف. في هذا العالم، تكون المواصفات الدقيقة لأي شيء نادرة، إن لم تكن مستحيلة تمامًا.

هنا يأتي دور المبرمجين. مهمتنا هي مساعدة الناس على فهم ما يريدون. في الواقع، ربما تكون هذه هي السمة الأكثر قيمة لنا. وهي جديرة بالتكرار.

نصيحة ٧٦ المبرمجون يساعدون الناس على فهم ما يريدون.

البرمجة كعلاج

دعونا نتصل بالأشخاص الذين يطلبون منا كتابة البرمجيات لعملائنا.

يأتي إلينا العميل النموذجي بحاجة. قد تكون هذه الحاجة استراتيجية، ولكن من المرجح أن تكون مسألة تكتيكية: استجابة لمشكلة حالية. ربما تكون هذه الحاجة لتغيير نظام موجود أو قد يطلب شيئًا جديدًا. سيتم التعبير عنها أحيانا من ناحية تجارية، وأحيانا أخرى من ناحية فنية.

غالبًا ما يكون الخطأ الذي يرتكبه المطورون الجدد هو أخذ بيان الحاجة هذا وتنفيذ حل له.

حسب خبرتنا، فإن بيان الحاجة الأولي هذا ليس متطلبًا مطلقًا. قد لا يدرك العميل ذلك، لكنه حقًا يدعو للاستكشاف.

لنأخذ مثالاً بسيطًا.

أنت تعمل لدى ناشر كتب ورقيّة وإلكترونيّة. وقد أُعطيت متطلبًا جديدًا:

يجب أن يكون الشحن مجانيًا على جميع الطلبيات التي تكلف 50 دولارًا أو أكثر.

توقّف للحظة وتخيّل نفسك في هذا الموقف. ما أول ما يتبادر إلى الذهن؟

من الجيّد جدًا أن يكون لديك أسئلة:

- هل يشمل مبلغ الـ 50 دولارًا الضرائب؟
- هل يشمل مبلغ الـ 50 دولارًا رسوم الشحن الحاليّة؟
- هل يجب أن يكون مبلغ الـ 50 دولارًا للكتب الورقيّة، أم يمكن أن تشمل الطلبية أيضًا كتبًا إلكترونيّة؟
- ما نوع الشحن المقدّم؟ الأولوية؟ برّي؟
- ماذا عن الطلبيات الدوليّة؟
- كم مرة سيتغير حد الـ 50 دولارًا في المستقبل؟

هذا ما نفعله نحن. عندما نُعطي شيئًا يبدو بسيطًا، فإننا نُتعب الناس بواسطة البحث عن الحالات الحديّة والسؤال عنها.

من المحتمل أن يكون العميل قد فكّر فعلًا في بعض هذه الأشياء، وافترضَ للتو أن التنفيذ سيعمل بهذه الطريقة. إن مجرّد طرح السؤال يدفع هذه المعلومات إلى السطح.

ولكن من المحتمل أن تكون الأسئلة الأخرى أشياء لم يفكّر فيها العميل من قبل. هذا هو المكان الذي تصبح فيه الأشياء مثيرة للاهتمام، وحيث يتعلم المطوّر الجيّد أن يكون دبلوماسيًا.

أنت: كنا نتساءل عن المبلغ الإجمالي البالغ 50 دولارًا. هل يشمل ذلك ما نفرضه عادةً على الشحن؟

العميل: بالطبع. إنه المبلغ الإجمالي الذي سيدفعونه لنا.

أنت: هذا أمر رائع وبسيط أن يفهمه عملاؤنا: يمكنني رؤية الاستقطاب. لكن يمكنني أن أرى بعض العملاء الأقل وجدانية يحاولون التلاعب بهذا النظام.

العميل: كيف ذلك؟

أنت: حسنًا، لنفترض أنهم اشتروا كتابًا بثمن 25 دولارًا، ثم حددوا الشحن بين عشية وضحاها "العاجل"، وهو الخيار الأكثر تكلفة. من المحتمل أن تكون تكلفة ذلك الشحن حوالي 30 دولارًا، مما يجعل قيمة الطلبية بِرُمَّتها 55 دولارًا. سنجعل الشحن مجانيًا، وسيحصلون مع شحنٍ بين عشية وضحاها على كتاب بقيمة 25 دولارًا مقابل 25 دولارًا فقط.

(عند هذه النقطة يتوقف المطوّر ذو الخبرة. قدّم الحقائق، ودع العميل يتخذ القرارات)

العميل: أوه. لم يكن هذا بالتأكيد ما قصدته. سنخسر المال في هذه الطلبيات. ما هي الخيارات؟

وهذا يبدأ الاستكشاف. يتمثل دورك هنا في تفسير ما يقوله العميل وإرجاع النتائج إليه. إنها عمليّة فكريّة وإبداعيّة في نفس الوقت: أنت تفكر سريعًا وتساهم في حل من المحتمل أن يكون أفضل من حل قد ينتجه أحدكما (أنت أو العميل) بمفرده.

المتطلبات هي عملية (Process)

في المثال السابق، أخذ المطوّر المتطلبات وأعاد نتيجة التغذية الراجعة (feedback) إلى العميل. وهذا ما بـدأ الاستكشـاف. في أثناء هذا الاستكشاف، من المحتمل أن تأتي بمزيد من التغذية الراجعة حيث يتعامل العميل مـع حلـول مختلفـة. هـذا هـو واقـع تجميـع المتطلبات:

نصيحة ٧٧ **يتم تعلّم المتطلبات في حلقة التغذية الراجعة.**

مهمتك هي مساعدة العميل على فهم عـواقب متطلباتـه المعلنـة. يمكنـك فعل ذلـك عن طريـق توليـد تغذيـة راجعـة، والسـماح لهم (لعملائك) باستخدام تلك الملاحظات لتحسين تفكيرهم.

في المثال السابق، كان من السهل التعبير عن التغذية الراجعة بالكلمات. في بعض الأحيان لا يكون الحال كذلك. وأحيانًـا لن تعـرف بصراحة ما يكفي عن المجال لتكون محددًا بهذا الشكل.

في حالات كهذه، يعتمـد المـبرمجون العمليـون على "هـل هـذا مـا قصـدته؟" مدرسـة التغذيـة الراجعة. نحن ننتج نمـاذج بـالحجم الطبيعي "مجسمات" (mockups)[138] ونماذج أولية (prototypes)، ونترك العميـل يلعب بهـا. ومن الناحيـة المثاليـة الأشـياء التـي ننتجها مرنة بما فيه الكفاية بحيث نتمكن من تغييرها خلال مناقشاتنا مع العميـل، ممـا يـتيح لنـا الـرد على "ليس هـذا مـا قصـدته" مـع "أكثر شبها بهذا؟"

في بعض الأحيان يمكن تُجمّع هذه النماذج (mockups) معًا في غضون ساعة أو نحـو ذلـك. من الواضـح أنهـا مجـرد محفّـز من أجل الحصول على فكرة.

لكن الحقيقة هي أن كل العمل الذي نقوم به هو في الواقع شكل من أشكال (mockup). حتى في نهاية المشـروع، مـا زلنـا نفسـر ما يريده عملاؤنا. في الواقع، بحلول هذه المرحلة، من المحتمل أن يكـون لـدينا المزيـد من العملاء: موظفـو ضـمان الجـودة، والعمليـات، والتسويق، وربما حتى مجموعات الاختبار من العملاء.

لذا فإن المبرمج العملي ينظر إلى المشروع كلّه على أنه تمرين لجمع المتطلبات. لهذا السبب نفضـل التكـرارات القصـيرة؛ تلـك الـتي تنتهي بملاحظات العملاء المباشرة. هذا يُبقينا على المسار الصحيح، ويتأكّد تقليل الـوقت الضـائع إلى الحـد الأدنى في حـال ذهابنـا في الاتجاه الخاطئ.

انظر إلى الموضوع من وجهة نظر العميل

هناك طريقة بسيطة لقراءة أفكار عملائك قليلا ما تُستخدم: أن تصبح عميلاً. هـل تكتب نظامًـا لمكتب المسـاعدة؟ اقضِ عـدّة أيـام في مراقبة الهواتف مع موظف دعم متمرس. هل تقوم بأتمتة نظام مراقبة المخزون اليدوي؟ اعمل في المستودع لمدة أسبوع.[139]

138 **[المترجم]** في التصنيع والتصميم، mockups، هو نموذج بالحجم الطبيعي أو بالحجم الكامل لتصميم أو جهاز، يستخدم للتـدريس، والعـرض، وتقييم التصـميم، والـترويج، وأغـراض أخـرى. Mockup هـو نمـوذج أولي إذا كـان يـوفر في الأقـل جـزءًا من وظـائف النظـام ويسـمح باختبـار التصميم. يستخدم المصممون Mockups بشكل أساسي للحصول على تعليقات (تغذية راجعة) من المستخدمين.

139 هل يبدو الأسبوع وكأنه وقت طويل؟ الأمر في الحقيقة ليس كذلك، خاصـة عنـدما تنظـر إلى العمليـات الـتي يشـغل/يتخـذ فيهـا كـل من الإدارة والعاملين عوالم مختلفة. ستمنحك الإدارة وجهة نظر واحدة حول كيفية عمل الأشياء، ولكن عندما على أرض الواقع، ستجد واقعا مختلفا تمامًـا

فبالإضافة إلى حصولك على نظرة ثاقبة حول كيفية استخدام النظام فعليًا، ستندهش من الكيفيّة التي يساعد بها طلب "هل يمكنني الجلوس لمدة أسبوع في أثناء قيامك بعملك؟" على بناء الثقة ووضع أساس للتواصل مع عملائك. فقط تذكّر ألا تقف في الطريق!

نصيحة ٧٨	اعمل مع المستخدم لتفكّر مثله.

مدّة جمع ردود الأفعال (feedback) هي أيضًا وقت مناسب لبَدء بناء عَلاقة بقاعدة عملائك، ومعرفة توقعاتهم وآمالهم بالنسبة للنظام الذي تقوم ببنائه. انظر الموضوع 52، أبهج مستخدميك، في الصفحة 307 للمزيد.

المتطلبات مقابل السياسة

لنتخيل أنه في أثناء مناقشة نظام الموارد البشرية، يقول العميل "يمكن لمشرفي الموظف وقسم شؤون الموظفين فقط معاينة سجلات ذلك الموظف". هل يُعدّ هذا البيان شرطًا حقًا ؟ ربما في الوقت الحاضر، لكنه يدمج سياسة الأعمال في بيان بحت.

سياسة العمل؟ المتطلبات؟ إنه تمييز دقيق نسبيًا، لكن سيكون له آثار عميقة على المطورين. إذا تم ذكر المتطلب على أنه "يمكن للمشرفين والموظفين فقط عرض سجل الموظف"، فقد ينتهي الأمر بالمطور إلى كتابة شِفرة اختبار صريح في كل مرّة يصل فيها التطبيق إلى هذه البيانات. ومع ذلك، إذا كانت العبارة "يمكن للمستخدمين المصرّح لهم فقط الوصول إلى سجل الموظف"، فمن المحتمل أن يقوم المطوّر بتصميم وتنفيذ نوع من نظام التحكم في الوصول. عندما تتغير السياسة (وسيحدث ذلك)، ستحتاج البيانات الوصفية (Metadata) لهذا النظام فقط إلى التحديث. في الواقع، يقودك جمع المتطلبات بهذه الطريقة بشكل طبيعي إلى نظام مدروس جيدًا لدعم البيانات الوصفيّة.

في الواقع، هناك قاعدة عامّة هنا:

نصيحة ٧٩	السياسة هي بيانات وصفيّة.

نفّذ الحالة العامة، مُستخدما معلومات السياسة كمثال على نوع الشيء الذي يحتاج النظام إلى دعمه.

المتطلبات مقابل الواقع

في مقال نشر في مجلة Wired في يناير 1999[140]، وصف المنتج والموسيقي Brian Eno قطعة رائعة من التِقَانَة - لوحة الدمج النهائية "الميكسر" (mixer). يفعل أي شيء يبدو أنه يمكن القيام به. ومع ذلك، بدلاً من السماح للموسيقيين بصنع موسيقى أفضل، أو إنتاج تسجيل أسرع أو أقل تكلفة، فإن ذلك يعيق الطريق؛ يعطل العملية الإبداعية.

لمعرفة السبب، عليك إلقاء نظرة على كيفيّة عمل مهندسي التسجيل. يوازنون الأصوات بشكل حدسيّ. طوّروا على مرّ السنين، حلقة تغذية راجعة فطرية بين آذانهم وأطراف أصابعهم - خافت منزلق، ومقابض دوّارة، وما إلى ذلك. ومع ذلك، فإن واجهة لوحة الدمج(mixer) الجديد لم تستغل تلك القدرات. بدلاً من ذلك، أجبرت مستخدميها على الكتابة على لوحة المفاتيح أو النقر بالماوس.

-واقعا ستستغرق وقتًا لاستيعابه.

140 https://www.wired.com/1999/01/eno/

كانت الوظائف التي قدّمتها شاملة، ولكن كانت مجمّعة بطرق غير مألوفة وغريبة. تم إخفاء الوظائف التي يحتاجها المهندسون أحيانًا خلف أسماء غامضة، أو تم إنجازها بواسطة مجموعات غير بديهية من المرافق الأساسية.

يوضح هذا المثال أيضًا إيماننا بأن الأدوات الناجحة تتكيف مع الأيدي التي تستخدمها. يأخذ جمع المتطلبات الناجح ذلك في الاعتبار. وهذا هو السبب في أن التغذية الراجعة المبكرة، مع النماذج الأولية أو الرصاصات الخطاطة، ستسمح لعملائك بقول "نعم، إنها تفعل ما أريد، ولكن ليس كما أريد."

توثيق المتطلبات

نعتقد أن أفضل توثيق للمتطلبات، وربما يكون توثيق المتطلبات *الوحيد*، هو شِفرة العمل.

لكن هذا لا يعني أنه يمكنك الانصراف دون توثيق فهمك لما يريده العميل. هذا يعني فقط أن هذه المستندات ليست قابلة للتسليم: فهي ليست شيئًا تعطيه العميل للتوقيع عليه. بدلاً من ذلك، فهي مجرد إرشادات للمساعدة في توجيه عملية التنفيذ.

مستندات المتطلبات ليست للعملاء

في الماضي، كان كل من *آندي* و *ديف* يعملان في مشروعات أنتجت متطلبات مفصّلة بشكل لا يصدق. توسعت هذه المستندات الأساسيّة في شرح أوليّ للعميل مدّته دقيقتان لما كان مطلوبًا، مما أدى إلى إنتاج روائع بسمك بوصة مليئة بالرسوم البيانية والجداول. تم تحديد الأمور لدرجة أنه لم يكن هناك مجال تقريبًا للغموض في التنفيذ. ونظرًا لاستخدام أدوات قوية بما فيه الكفاية، يمكن أن تكون الوثيقة في الواقع البرنامَج النهائي.

كان إنشاء هذه الوثائق خطأً لسببين. أولاً، كما ناقشنا، لا يعرف العميل حقًا ما يريده مُسبقًا. لذلك عندما نأخذ ما يقوله ونوسّعه إلى ما يكاد أن يكون وثيقة قانونية، فإننا نبني قلعة معقدة للغاية على الرمال المتحركة.

قد تقول "ولكن بعد ذلك نأخذ المستند إلى العميل ويوقع عليه. نحن نحصل على تغذية راجعة ". وهذا يقودنا إلى المشكلة الثانية مع مواصفات المتطلبات هذه: العميل لا يقرأها أبدًا.

يستخدم العميل المبرمجين لأنه بينما يُحفَّز العميل بواسطة حل مشكلة عالية المستوى وغامضة إلى حدّ ما، فإن المبرمجين يهتمون بكل التفاصيل والفروق الدقيقة. وثيقة المتطلبات مكتوبة للمطورين، وتحتوي على معلومات وتفاصيل دقيقة تكون أحيانًا غير مفهومة وغالبًا ما تكون مملة للعميل.

أرسل مستند متطلبات مكون من 200 صفحة، ومن المرجح أن يقدّر وزنه العميل ليقرر ما إذا كان وزنه كافيًا ليكون مهمًا، فقد يقرأ أول فقرتين (وهذا هو سبب تسمية أول فقرتين دائمًا بـ *ملخص الإدارة*)، وربما يقلّب باقي الصفحات، ويتوقف أحيانًا عندما يكون هناك مخطط أنيق.

إن هذا لا يقلل من شأن العميل. لكن منحهم مستندًا تقنيًا كبيرًا يشبه إعطاء المطور العادي نسخة من *الإلياذة* في هوميروس اليوناني (Homeric Greek) ومطالبتهم بكتابة شِفرة لعبة الفيديو منها.

مستندات المتطلبات هي للتخطيط

لذلك نحن لا نؤمن بوثيقة المتطلبات الضخمة والثقيلة جدًا (بما يكفي لصعق الثور). ومع ذلك، نحن نعلم أنه يجب تدوين المتطلبات، وذلك ببساطة لأن المطورين في الفريق يحتاجون إلى معرفة ما سيفعلونه.

ما الشكل الذي يتخذه ذلك؟ نحن نفضل شيئًا يمكن وضعه في بطاقة فهرسة حقيقية (أو افتراضية). غالبًا ما تسمى هذه الأوصاف القصيرة *قصص المستخدمين* (user stories). يصفون ما يجب أن يفعله جزء صغير من التطبيق من منظور المستخدم لتلك الوظيفة.

عند كتابتها بهذه الطريقة، يمكن وضع المتطلبات على السَّبُّورَة وتحريكها لإظهار كل من الحالة والأولوية.

قد تعتقد أن بطاقة فهرسة واحدة لا يمكنها الاحتفاظ بالمعلومات اللازمة لتنفيذ أحد مكونات التطبيق. ستكون على حق. وهذا جزء من الموضوع. بالحفاظ على بيان المتطلبات هذا قصيرًا، فإنك تشجع المطورين على طرح أسئلة توضيحيّة. أنت تعزز عملية التغذية الراجعة بين العملاء والمبرمجين قبل وأثناء إنشاء كل جزء من الشِفرة.

الإفراط في المواصفات

هناك خطر كبير آخر في إنتاج مستند المتطلبات وهو التحديد الشديد. المتطلبات الجيّدة مجرّدة. عندما يتعلق الأمر بالمتطلبات، فإن أبسط بيان يظهر بدقة احتياجات العمل هو الأفضل. هذا لا يعني أنك يمكن أن تكون غامضًا - يجب عليك التقاط الثوابت الدلاليّة الأساسيّة كمتطلبات، وتوثيق ممارسات العمل المحدّدة أو الحاليّة كسياسة.

المتطلبات ليست هيكلة. المتطلبات ليست تصميمًا، ولا هي واجهة المستخدم. المتطلبات حاجة.

رقاقة بسكويت نعناع إضافية ...

يُعزى فشل العديد من المشروعات إلى زيادة النطاق- المعروف أيضًا باسم تضخّم الميزات أو السمات الزاحفة أو زحف المتطلبات. هذا جانب من متلازمة الضفدع المغليّ في الموضوع 4ـ حساء الحصى والضفادع المغليّة، في الصفحة 34. ماذا يمكننا أن نفعل لمنع المتطلبات من التسلل إلينا؟

الجواب (مرّة أخرى) هو التغذية الراجعة. إذا كنت تعمل مع العميل في تكرارات مع تغذية راجعة مستمرة، فسيختبر العميل بشكل مباشر تأثير فكرة "مِيزة واحدة أخرى فقط". سيرون بطاقة قصة أخرى تظهر على السَّبُّورَة، وسيجب مساعدتهم في اختيار بطاقة أخرى لإفساح المجال للانتقال إلى التكرار التالي. تعمل الملاحظات في كلا الاتجاهين.

احتفظ بمسرد مصطلحات

بمجرد أن تبدأ في مناقشة المتطلبات، سيستخدم المستخدمون وخبراء المجال مصطلحات معيّنة لها معنى محدّد لهم. قد يفرقون بين "الزَّبُون" (client) و "العميل" (customer)، على سبيل المثال. سيكون من غير المناسب بعد ذلك استخدام أي من الكلمتين بشكل عرضيّ في النظام.

أنشئ مسردًا لمشروعك وحافظ عليه - مكان واحد يعرّف جميع المصطلحات والمفردات المحدّدة المستخدمة في المشروع. يجب على جميع المشاركين في المشروع، من المستخدمين النهائيين إلى فريق الدعم، استخدام المسرد لضمان الاتساق. هذا يعني أن المسرد يحتاج إلى أن يكون متاحًا على نطاق واسع - حجّة جيّدة للتوثيق عبر الإنترنت.

نصيحة ٨٠	استخدم مسرد مصطلحات للمشروع.

من الصعب أن تنجح في مشروع إذا دعا المستخدمون والمطورون الشيء نفسه بأسماء مختلفة، أو الأسوأ من ذلك، إذا أشاروا إلى أشياء مختلفة بالاسم نفسه.

الأقسام ذات الصلة

- الموضوع 5، برمجيات جيّدة كفاية، في الصفحة 36
- الموضوع 7، تواصل!، في الصفحة 44
- الموضوع 11، قابلية العكس، في الصفحة 69
- الموضوع 13، النماذج الأولية والملاحظات الملحقة، في الصفحة 78
- الموضوع 23، التصميم حسب العقد، في الصفحة 125
- الموضوع 43، ابق آمنًا هناك، في الصفحة 255
- الموضوع 44، تسمية الأشياء، في الصفحة 262
- الموضوع 46، حل الألغاز المستحيلة، في الصفحة 277
- الموضوع 52، أبهج مستخدميك، في الصفحة 307

تمارين

تمرين 33 (إجابة محتملة في الصفحة 331)

أي مما يلي من المحتمل أن يكون متطلبات حقيقية؟ أعد صياغة تلك التي لا تجعلهم أكثر فائدة (إن أمكن).

1. يجب أن يكون وقت الاستجابة أقل من 500 مللي ثانية.
2. سيكون للنوافذ المشروطة[141] خلفية رمادية.
3. سيتم تنظيم التطبيق على هيئة عدد من عمليات الواجهة الأمامية (of front-end processes) وخادم واجهة خلفية (back-end server).
4. إذا قام مستخدم بإدخال أحرف غير رَقَمية في حقل رَقَمي، فسيقوم النظام بتمييز خلفية الحقل ولن يقبلها.
5. يجب أن تكون الشِفرة والبيانات الخاصة بهذا التطبيق المضمن في نطاق 32 ميجا بايت.

141 **[المترجم]** وهي نوافذ فرعية تظهر أمام النافذة الرئيسة، حيث لا يمكن للمستخدم الضغط على أي عناصر تحكم أو إدخال أي معلومات في النافذة الرئيسة حتى يتم إغلاق هذه النافذة.

تحديات

- هل يمكنك استخدام البرمجية التي تكتبها؟ هل من الممكن أن يكون لديك شعور جيد بالمتطلبات دون أن تكون قادرًا على استخدام البرمجية بنفسك؟
- اختر مشكلة غير متعلقة بالحاسوب تحتاج إلى حلها حاليًا. أنشئ متطلبات لحل غير حاسوبي.

الموضوع 46. حل الألغاز المستحيلة

ذات مرّةٍ ربطَ غورديوس، ملك فريجيا، عقدةً لا يمكن لأحد حلها. قيل أن من يحلّ لغز العقدة الغوردية سيحكم كل آسيا. وهكذا توالى الناس حتى أتى الإسكندر الأكبر، وقطع العقدة بسيفه. تفسير مختلف قليلاً للمتطلبات فقط، هذا كل شيء.... وقد انتهى به الأمر إلى حكم معظم آسيا.

ستجد نفسك بين الحين والآخر عالقًا في منتصف مشروع ما حين يواجهك لغز صعب حقًا: جزء من المواجهة التي لا يمكنك التعامل معها، أو ربما بعض التعليمات البرمجية التي تغدو أصعب بكثير مما كنت تعتقد. ربما تبدو مستحيلةً. ولكن هل هي حقا صعبة كما تبدو؟

فكّر في ألغاز العالم الحقيقي - تلك القطع الصغيرة الملتوية من الخشب أو الحديد المطاوع أو البلاستيك التي يبدو أنها تظهر كهدايا لعيد الميلاد أو في مبيعات المرآب. كل ما عليك فعله هو إزالة الحلقة، أو تركيب القطع على شكل حرف T في الصندوق، أو أيًا كان.

لذلك تسحب الحلقة، أو تحاول وضع قطع الـ T في الصندوق، لتكتشف سريعا أن الحلول الواضحة لا تنجح. لا يمكن حل اللغز بهذه الطريقة. ولكن مع أنّ ذلك واضح، فإنه لا يمنع الناس من تجرِبة الشيء نفسه - مرارًا وتكرارًا - التفكير في ضرورة وجود طريقة ما.

بالطبع لا يوجد. الحل يكمن في مكان آخر. سر حل اللغز هو تحديد القيود الحقيقية (غير المتخيّلة) وإيجاد حل لها. بعض القيود مطلقة. بعضها الآخر مجرّد أفكار مُسبقة. يجب احترام القيود المطلقة، مهما بدت بغيضة أو غبيّة.

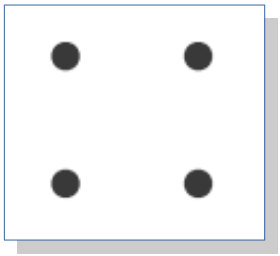
من ناحية أخرى، وكما أثبت الإسكندر، قد لا تكون بعض القيود الظاهرة قيودًا حقيقية على الإطلاق. يمكن أن تكون العديد من مشكلات البرمجيات مخادعة.

درجات الحرية

تشجعنا العبارة الرنانة الشائعة "التفكير خارج الصندوق" على التعرف على القيود التي قد لا تكون قابلة للتطبيق وتجاهُلها. لكن هذه العبارة ليست دقيقة تمامًا. إذا كان "الصندوق" هو حدود القيود والشروط، فإن الحيلة تكمن في العثور على هذا الصندوق والذي قد يكون أكبر بكثير مما تعتقد.

مفتاح حل الألغاز هو التعرف على القيود المفروضة عليك والتعرف على درجات الحرية التي تتمتع بها، فبفضلهم ستجد الحل المناسب لك. هذا هو السبب في أن بعض الألغاز فعّالة للغاية؛ يمكنك رفض الحلول المحتملة بسهولة شديدة.

على سبيل المثال، هل يمكنك توصيل جميع النِقَاط في اللغز التالي والعودة إلى نقطة البداية برسم ثلاثة خطوط مستقيمة فقط - دون رفع قلمك عن الورقة أو إعادة تتبع خطواتك (Math Puzzles & Games [Hol92])؟



يجب عليك تحدي أي مفاهيم مسبقة وتقييم ما إذا كانت قيودًا حقيقية وصارمة وملزمة أم لا.

الأمر لا يتعلق بما إذا كنت تفكر داخل الصندوق أو خارجه. تكمن المشكلة في *وِجدان* الصندوق - تحديد القيود الحقيقية.

نصيحة ٨١	**لا تفكّر خارج الصندوق اعثر عليه.**

عندما تواجه مشكلة مستعصية، عدّد جميع السبل الممكنة أمامك. لا *تستبعد* أي شيء منها، مهما بدا أنه غير صالحٍ للاستخدام أو غبي. الآن راجع القائمة واشرح لماذا لا يمكن اتباع مسار معين. هل أنت واثق؟ هل يمكنك *إثبات* ذلك؟

تأمّل في حصان طروادة - حل جديد لمشكلة مستعصية. كيف تدخل القوات إلى مدينة مسوّرة دون أن تُكتشَف؟ يمكنك أن تراهن على أنه تم رفض حل "بواسطة الباب الأمامي" في البداية باعتباره انتحار.

صنّف وحدّد أولويات القيود الخاصة بك. عندما يبدأ النجارون مشروعًا ما، يقطّعون القطع الأطول أولاً، ثم يقطعون القطع الأصغر من الخشب المتبقي. وبنفس الطريقة، نريد تحديد القيود الأكثر تقييدًا أولًا، وملائمة القيود المتبقية داخلها.

بالمناسبة، حل لغز النِقَاط الأربعة السابق معروض في نهاية الكتاب في الصفحة 331.

ابتعد عن طريقتك الخاصة!

أحيانًا تجد نفسك تعمل على حل مشكلة تبدو أصعب بكثير مما كنت تعتقد أنها يجب أن تكون عليه. ربما تشعر وكأنك تسير في الطريق الخطأ - يجب أن تكون هناك طريقة أسهل من هذه! ربما تكون متأخرًا عن الجدول الزمني الآن، أو حتى تشعر باليأس من إمكانية تشغيل النظام لأن هذه المشكلة بالذات "مستحيلة الحل".

إنّه الوقت المثالي للقيام بشيء آخر لمدّة من الزمن. اعمل على شيء مختلف. اذهب وتمش مع كلبك. بيّت الأمر.

يدرك عقلك الواعي المشكلة، لكن عقلك الواعي غبي جدًا حقًا (لا أقصد الإساءة). لذلك حان الوقت لمنح عقلك الحقيقي، تلك الشبكة العصبية الترابطية المذهلة التي تكمن تحت وعيك، بعض المساحة. ستندهش من عدد المرّات التي ستظهر فيها الإجابة في عقلك عندما تشتت انتباهك عن عمد.

إذا بدا هذا غامضًا جدًا بالنسبة لك، فالأمر ليس كذلك. يُبلغ تقرير *علم النفس اليوم (Psychology Today)*[142]:

142 https://www.psychologytoday.com/us/blog/your-brain-work/201209/stop-trying-solve-problems

لتبسيط الأمر - كان أداء الأشخاص الذين تشتت انتباههم في مهمة حل المشكلات المعقدة أفضل من الأشخاص الذين بذلوا جهدًا واعيًا.

إذا كنت لا تزال غير مستعد لترك المشكلة لبعض الوقت، فمن المحتمل أن يكون أفضل شيء تالي هو إيجاد شخص ما لشرحها لـه. غالبًا ما يؤدي مجرد الإلهاء بالحديث عنها إلى التنوير.

اطلب منهم أن يطرحوا عليك أسئلة مثل:

- لماذا تحل هذه المشكلة؟
- ما فائدة حلها؟
- هل المشكلات التي تواجهها تتعلق بحالات حدّية؟ هل يمكنك استبعادها؟
- هل هناك مشكلة أبسط ذات صلة يمكنك حلها؟

هذا مثال آخر على "البط المطاطي" في الممارسة العملية.

الثروة تفضل العقل المستعد

يُروى إن لويس باستور قال:

Dans les Champs de l'observation le hasard ne favise que les esprits préparés.

(عندما يتعلق الأمر بالمراقبة، فالثروة تفضل العقل المستعد).

وهذا صحيح لحل المشكلات أيضًا. فمن أجل الحصول عللحظات من هذه الـ "*وجدتها*!"، يحتـاج دماغـك اللاواعي إلى الكثـير من المواد الخام؛ الخبرات السابقة التي يمكن أن تساهم في الإجابة.

هناك طريقة رائعة لتغذية عقلك وهي إعطائه ملاحظات على ما يصلح وما لا يصلح في أثناء قيامك بعملـك اليـومي. وقـد وصـفنا طريقة رائعة للقيام بذلك باستخدام دفتر يوميات الهندسة (الموضوع 22، دفاتر يوميات الهندسة، في الصفحة 121).

وتذكر دائمًا النصيحة على غلاف *دليل المسافرإلى المجرة Hitchhiker's Guide to the Galaxy*: **لا تخف**.

الأقسام ذات الصلة

- الموضوع 5، برمجيات جيّدة كفاية، في الصفحة 36
- الموضوع 37، استمع إلى حدسك، في الصفحة 218
- الموضوع 45، هوة المتطلبات، في الصفحة 268

- كتب آندي كتابًا كاملاً عن هذا النوع من الأشياء: *التفكير العملي والتعلم: أعد بناء دماغك Pragmatic Thinking and Learning: Refactor Your Wetware* [Hun08].

تحديات

- ألق نظرة فاحصة على أي مشكلة صعبة تواجهها اليوم. هل يمكنك قطع العقدة الغوردية؟ هل عليك فعل ذلك بهذه الطريقة؟ هل عليك فعلها أصلا؟
- هل تم تسليمك مجموعة من القيود عند قيامك بتسجيل الدخول إلى مشروعك الحالي؟ هل ما زالت جميعها قابلة للتطبيق، وهل تفسيرها لا يزال صالحًا؟

الموضوع 47. العمل معا

لم ألتق أبدًا بشخص يرغب في قراءة 17000 صفحة من الوثائق، وإذا وُجد، فسأقتله لإخراجه من المخزون الجيني.
➤ جوزيف كوستيلو، *رئيس الإيقاع* President of Cadence

لقد كان واحدا من تلك المشروعات "المستحيلة"، من النوع الذي تسمع عنه والذي يبدو مبهجًا ومرعبًا في نفس الوقت. كان النظام القديم يقترب من نهاية صلاحيته، وكانت الأجهزة المادية تختفي فعليًا، وكان لابد من تصميم نظام جديد يتناسب تمامًا مع السلوك (غير الموثّق غالبًا). ستمر مئات الملايين من الدولارات من أموال الأشخاص الآخرين عبر هذا النظام، وكان الموعد النهائي من البداية حتى النشر في حدود أشهر.

في هذا المكان التقى آندي وديف لأول مرّة. مشروع مستحيل بموعد نهائي مثير للسخرية. كان هناك شيء واحد فقط جعل المشروع ناجحًا للغاية. كانت الخبيرة التي أدارت هذا النظام لسنوات جالسة هناك في مكتبها، مقابل القاعة مباشرةً مقابل غرفة التطوير الصغيرة الخاصة بنا. كانت جاهزة باستمرار للأسئلة والتوضيحات والقرارات والعروض التجريبية.

نوصي في هذا الكتاب بالعمل من كثب مع المستخدمين؛ هم جزء من فريقك. في مشروعنا الأول معًا، تدربنا على ما يمكن أن يسمى الآن البرمجة الزوجية "البرمجة بالأزواج"[143] (pair programming) أو البرمجة الجماعية "الغوغائية" (mob programming): يقوم أحد الأشخاص بكتابة التعليمات البرمجية بينما يقوم واحد أو أكثر من أعضاء الفريق بالتعليق والتفكير وحل المشكلات معًا. إنها طريقة قوية للعمل معًا تتجاوز الاجتماعات التي لا نهاية لها والمذكرات والوثائق القانونية المحشوة والمُثمَّنة للوزن على الفائدة.

وهذا ما نعنيه حقًا بـ "العمل مع": ليس فقط طرح الأسئلة وإجراء المناقشات وتدوين الملاحظات، ولكن طرح الأسئلة وإجراء المناقشات في أثناء قيامك فعليًا بالبرمجة.

قانون كونواي

في عام 1967، قدم *ملفين كونواي* فكرة في كيفية اختراع اللجان؟ [Con68] الذي أصبح معروفًا باسم قانون كونواي:

فالمنظمات التي تصمم الأنظمة مقيدة بإنتاج تصاميم تكون نسخ من بنى الاتصالات لهذه المنظمات.

وهذا يعني أن البِنَى الاجتماعية ومسارات الاتصال للفريق والمؤسسة ستنعكس في التطبيق أو موقع الويب أو المنتج الذي يتم تطويره. أظهرت دراسات مختلفة دعمًا قويًا لهذه الفكرة. لقد شهدنا ذلك بشكل مباشر مرات لا حصر لها - على سبيل المثال، في الفِرق

143 **[المترجم]** البرمجة الزوجية هي أحد تقنيات أجايل لتطوير البرمجيات حيث يعمل اثنين من المبرمجين معا في محطة عمل واحدة. يكتب أحدهم الشِفرة المصدرية في حين يستعرض الأخر كل سطر من الشِفرة كما تمت كتابته. والشخص الذي يكتب يُسمى المحرك. ويطلق على الشخص الذي يستعرض الشِفرة المراقب (أو الملاح). ويتبادل المبرمجان الأدوار بشكل متكرر. في أثناء الاستعراض، يضع المراقب في اعتباره التوجه الاستراتيجي للعمل، والخروج بأفكار من أجل إدخال تحسينات والمشكلات المحتملة في المستقبل لمعالجتها. وهذا يتيح الفرصة للمحرك في تركيز كل اهتمامه على الجوانب "التكتيكية" لاستكمال المهمة الحالية، وذلك باستخدام المراقب كشبكة أمان ودليل.

التي لا يتحدث فيها أحد مع الآخر على الإطلاق، أدى ذلك إلى أنظمة "أنابيب المواقد"[144] (stove-pipe) المنعزلة. أو الفرق التي تم تقسيمها إلى قسمين، أدى ذلك إلى تقسيمات العميل / الخادم أو الواجهة الأمامية / الخلفية.

تقدم الدراسات أيضًا دعمًا للمبدأ العكسي: يمكنك هيكلة فريقك عن عمد بالطريقة التي تريد للشِفرة أن تظهر بها. على سبيل المثال، تظهر الفرق الموزعة جغرافيًا أنها تميل إلى المزيد من البرمجيات المعياريّة الموزعة.

ولكن الأهم من ذلك، أن فرق التطوير التي تضم مستخدمين ستنتج برمجيات تظهر بوضوح تلك المشاركة، والفرق غير المكترثة ستظهر ذلك أيضًا.

البرمجة الزوجية

البرمجة الزوجية هي إحدى ممارسات البرمجة الفائقة (eXtreme Programming) التي أصبحت شائعة خارج نظام البرمجة الفائقة XP نفسها. في البرمجة الزوجية، يقوم أحد المطورين بالكتابة على لوحة المفاتيح بينما لا يفعل الآخر ذلك. يعمل كلاهما على حل المشكلة معًا، ويمكنهما تبديل مهام الكتابة حسب الحاجة.

هناك العديد من الفوائد للبرمجة الزوجية. يجلب الأشخاص المختلفون خلفيات وخبرات مختلفة، وتقنيات وأساليب مختلفة لحل المشكلات، ومستويات مختلفة من التركيز والانتباه لأي مشكلة بعينها. يجب أن يركّز المطور الذي يعمل ككاتب على لوحة المفاتيح على التفاصيل ذات المستوى المنخفض لبناء الجملة وأسلوب كتابة الشِفرة، في حين أن المطور الآخر له الحرية في النظر في النطاق والقضايا ذات المستوى الأعلى. في حين أن هذا قد يبدو اختلافا بسيطًا، تذكر فقط أننا نحن البشر نملك قدر كبير من النطاق الترددي للدماغ. إن العبث بكتابة الكلمات والرموز الخفيّة التي سيقبلها المترجم على مضض يأخذ قدراً لا بأس به من قوة المعالجة الخاصة بنا. إن توفر عقل كامل لمطورٍ ثانٍ في أثناء المهمة يجلب الكثير من القوة العقلية للدعم.

يساعد ضغط الندّية الملازم للشخص الثاني في مواجهة لحظات الضعف والعادات السيئة لتسمية المتغيرات foo وما شابه. فتكون أقل ميلًا إلى اتباع طريق مختصر محرج عندما يشاهد شخص آخر بنشاط ما تفعل، مما يؤدي أيضًا إلى برمجيات عالية الجودة.

البرمجة الجماعية

وإذا كان عقلان أفضل من واحد، فماذا عن وجود دزينة من الأشخاص المختلفين يعملون جميعًا على نفس المشكلة في نفس الوقت، مع كاتب واحد؟

إن البرمجة الجماعية(الغوغائية)، وعلى الرغم من التسمية، إلا أنها لا تتضمن مشاعل أو مذراة "شوك الحقول". إنها امتداد للبرمجة الزوجية التي تتضمن أكثر من مطورين اثنين فقط. يذكر المؤيدون نتائج رائعة باستخدام الغوغائية لحل المشكلات الصعبة. يمكن أن تضم الغوغائية بسهولة أشخاصًا لا يعتبرون عادةً جزءًا من فريق التطوير، بما في ذلك المستخدمين ورعاة المشروع والمختبرين. في الواقع، في مشروعنا الأول "المستحيل" معًا، كان مشهدًا مألوفًا لأحدنا أن يكتب بينما يناقش الآخر المشكلة مع خبير الأعمال لدينا. كانت مجموعة صغيرة من ثلاثة أفراد.

144 **[المترجم]** يتعامل تطوير مواقد الأنابيب أو ما يسمى: "أنظمة المدخنة" مع تصميم وإنشاء نظام لحل مشكلة فورية دون النظر إلى الاتصال المستقبلي أو التكامل مع الأنظمة الأخرى. لا تشارك تطبيقات مواقد الأنابيب الموارد أو البيانات مع التطبيقات الأخرى. يستحضر المصطلح صورة أنابيب الموقد التي ترتفع فوق المباني، ويعمل كل منها على حدة.

قد تفكر في البرمجة الجماعية على أنها تعاون وثيق مع الكتابة المباشرة للشِفرة.

ماذا علي أن أفعل؟

إذا كنت تبرمج حاليًا بشكل منفرد فقط، فربما تجرب البرمجة الزوجية. امنحها أسبوعين في الأقل، بضع ساعات فقط في كل مرّة، حيث ستشعر بالغرابة في البداية. ومن أجل طرح أفكار جديدة أو تشخيص المشكلات الشائكة، فربما تجرب جَلسة برمجة جماعيّة.

إذا كنت تقوم فعلًا بالبرمجة الزوجية أو الجماعية، فمن المشمول بذلك؟ هل هم المطورين فقط أم تسمح لأعضاء فريقك الموسّع بالمشاركة: المستخدمين، المختبرين، الرعاة ...؟

وكما هو الحال مع جميع أشكال التعاون، تحتاج إلى إدارة الجوانب البشرية منه بالإضافة إلى الجوانب التقنية. إليك بعض النصائح للبَدْء:

- ابنِ الشِفرة، وليس الأنا. لا يتعلق الأمر بمن هو ألمع. لكلٍ منّا أوقات جيدة وأخرى سيئة.
- ابدأ صغيرا. ببرمجة جماعية مع 4-5 أشخاص فقط، أو ابدأ ببضعة أزواج من المبرمجين فقط، في جلسات قصيرة.
- انتقد الشِفرة وليس الشخص. تبدو عبارة "دعونا نلقي نظرة على هذه الكتلة" أفضل بكثير من عبارة "أنت مخطئ".
- استمع وحاول فهم وجهات نظر الآخرين. الاختلاف ليس خطأ.
- أجرِ استعادات متكررة لمحاولة التحسين في المرّة القادمة.

تعد البرمجة في نفس المكتب أو عن بُعد، بمفردك أو في أزواج أو في مجموعات، طرقًا فعّالة للعمل معًا لحل المشكلات. إذا قمت أنت وفريقك بذلك بإحدى الطرق فقط، فقد ترغب في تجربة أسلوب مختلف. لكن لا تقفز إليه فقط بشكل ساذج: فهناك قواعد واقتراحات وإرشادات لكل من أنماط التطوير هذه. على سبيل المثال، باستخدام البرمجة الجماعية، يمكنك تبديل كاتب الآلة كل 5-10 دقائق.

قم ببعض القراءة والبحث، في كل من الكتب المدرسية وتقارير الخبرة، وتعرّف على المزايا والعثرات التي قد تواجهها. قد ترغب في البَدْء بكتابة شِفرة تمرين بسيط، وليس القفز مباشرة إلى أصعب شِفرة إنتاج لديك.

ولكن مهما كان الأمر، دعنا نقترح نصيحة أخيرة:

نصيحة ٨٢	**لا تدخل في الشِفرة لوحدك.**

الموضوع 48. جوهر التطوير الرشيق

أنت تستمر في استخدام هذه الكلمة، لكني لا أعتقد أنها تعني ما تعتقد أنت أنها تعنيه.

➤ إينيغو مونتويا، الأميرة العروس

رشيق (Agile) هي صفة: إنها تصف كيف تفعل شيئًا ما. يمكنك أن تكون مطورًا رشيقًا.

يمكنك أن تكون في فريق يتبنى ممارسات رشيقة، فريق يستجيب التغيير والنكسات بخفة. الرشاقة هي أسلوبك وليست أنت.

نصيحة ٨٣	**الرشاقة (Agile) ليست اسما؛ الرشاقة هي كيف تفعل الأشياء.**

بينما نكتب هذا، بعد ما يقرب من 20 عامًا من بَدء بيان تطوير البرمجيات الرشيقة Manifesto for Agile Software Development[145]، نرى العديد والعديد من المطورين يطبقون قيمه بنجاح. نرى العديد من الفرق الرائعة التي تجد طرقًا لأخذ هذه القيم واستخدامها لتوجيه ما يفعلونه، وكيفية تغيير ما يفعلونه.

لكننا نرى أيضًا جانبًا آخر من الرشاقة. نرى فِرقًا وشركات تتوق إلى حلول جاهزة: Agile-in-a-Box. ونرى العديد من الاستشاريين والشركات سعداء للغاية لبيعهم ما يريدون. نرى الشركات تتبنى المزيد من طبقات الإدارة، وإعداد التقارير الرسمية، والمزيد من المطورين المتخصصين، والمزيد من المسميات الوظيفية الفاخرة التي لا تعني إلّا "شخص لديه حافظة وساعة توقيت".[146]

نشعر أن العديد من الأشخاص فقدوا المعنى الحقيقي للرشاقة، ونودّ رؤية الناس يعودون إلى الأساسيات.

تذكر القيم من هذا البيان:

نحن نكتشف طرقًا أفضل لتطوير البرمجيات بواسطة تنفيذها ومساعدة الآخرين على فعل ذلك. بواسطة هذا العمل أصبحنا نقدّر ما يلي:

- **الأفراد والتفاعلات** على العمليات والأدوات
- **البرمجيات العاملة** على وثائق شاملة
- **تعاون العملاء** على التفاوض على العقد
- **الاستجابة للتغيير** على اتباع خِطَّة

أي، بينما توجد قيمة في العناصر الموجودة على اليسار، فإننا نقدر العناصر الموجودة على اليمين أكثر.

145 https://agilemanifesto.org

146 لمزيد من المعلومات حول مدى سوء هذا النهج، راجع استبداد المقاييس [Mul18] The Tyranny of Metrics.

من الواضح أن أي شخص يبيع لك شيئًا يزيد من أهمية الأشياء الموجودة على اليسار على الأشياء الموجـودة على اليمين لا يقـدّر نفس الأشياء التي فعلناها نحن وكُتّاب هذا البيان الآخرون.

وأي شخص يبيع لك حلًّا جاهزًا فهو لم يقرأ البيان التمهيدي. يتم تحفيز القيم وإعلامها بواسطة العمل المسـتمر للكشـف عن طـرق أفضل لإنتاج البرمجيات. هذه ليست وثيقة ثابتة. إنها اقتراحات لعملية منتجة.

لا يمكن أن تكون هناك عملية رشيقة

في الواقع، عندما يقول أحدهم "افعل هذا وستكون رشيقًا"، فهو مخطئ. وفقا للـتعريف.

لأن الرشاقة، سواء في العالم المادي أو في عالم تطوير البرمجيات، تدور حول الاسـتجابة للتغيـير، والاسـتجابة للأمـور المجهولـة التي تواجهها بعد الانطلاق. فالغزال الجاري لا يسير في خط مستقيم. ويقوم لاعب الجمباز بإجراء مئات التصـحيحات في الثانيـة لأنهـا تستجيب للتغيرات في بيئتها وللأخطاء الطفيفة في وضع قدمه.

هذا هو الحال أيضا مع الفرق والمطورين الفرديين. لا توجـد خِطّة واحـدة يمكنـك اتباعهـا عنـد تطـوير البرمجيـات. ثلاث من القيم الأربع تخبرك بذلك. يتعلق الأمر بجمع الانطباعات (feedback)والاستجابة لها.

لا تُخبرك القيم بما يجب عليك فعله. هم يخبرونك ما الذي تبحث عنه عندما تقرر بنفسك ما يجب فعله.

دائمًا ما تكون هذه القرارات سياقية: فهي تعتمد على من تكـون أنت وعلى طبيعـة فريقـك وتطبيقـك وأدواتـك وشـركتك وعميلـك والعالم الخارجي؛ عدد كبير بشكل لا يصدق من العوامل، بعضها رئيس وبعضها تافه. لا يمكن لأي خِطّة ثابتة ومحدّدة أن تنجو من حالـة عدم اليقين هذه.

إذًا ماذا نفعل؟

لا أحد يستطيع أن يخبرك ماذا تفعل. لكننا نعتقد أنه بإمكاننا إخبارك بشيء عن الروح التي تفعلها بها. يتلخص كل ذلـك في كيفيـة تعاملك مع عدم اليقين. يقترح البيان أن تفعل ذلك بواسطة جمع التغذية الراجعة (feedback) والعمل بناءً عليها. إذًا هـا هي وصـفتنا للعمل بطريقة رشيقة:

1. تطوّر أينما كنت.
2. اتخذ أصغر خطوة ذات مغزى نحو المكان الذي تريد أن تكون فيه.
3. قيّم ما ينتهي بك الأمر إليه، وأصلح أي شيء تكسره.

كرّر هذه الخطوات حتى النهاية. واستخدمها بشكل متكرر، في كل مستوى في كل شيء تفعله.

في بعض الأحيان، يصبح القرار الذي يبدو تافهًا مهمًا عند تجمع الانطباعات.

"الآن تحتاج الشِفرة خاصتي إلى الحصول على مالك الحساب.
```
let user = accountOwner(accountID);
```
إممم ... اسم المستخدم عديم الفائدة. سأجعله مالكًا.
```
let owner = accountOwner(accountID);
```
ولكن يبدو هذا الآن زائدا عن الحاجة قليلا. ما الذي أحاول فعله هنا؟ تقول القصة أنني أرسل بريدًا إلكترونيًا إلى هذا الشخص، لذا أحتاج إلى العثور على عنوان بريده الإلكتروني. ربما لست بحاجة إلى مالك الحساب بالكامل على الإطلاق.

```
let email = emailOfAccountOwner(accountID);
```

بواسطة تطبيق حلقة التغذية الراجعة على مستوى منخفض حقًا (تسمية متغير)، قمنا فعلًا بتحسين تصميم النظام كليًا، مما قلل من الاقتران بين هذه الشِفرة والشِفرة التي تتعامل مع الحسابات.

تنطبق حلقة التغذية الراجعة أيضًا على أعلى مستوى في المشروع. لقد حدث بعض من أنجح أعمالنا عندما بدأنا العمل على متطلبات العميل، واتخذنا خطوة واحدة، وأدركنا أن ما كنا على وشك القيام به لم يكن ضروريًا، وأن الحل الأفضل لم يشمل البرمجية حتى.

تنطبق هذه الحلقة خارج نطاق مشروع واحد. يجب على الفرق تطبيقها لمراجعة العملية ومدى نجاحها. الفريق الذي لا يجرب عملياته باستمرار ليس فريقًا رشيقًا.

وهذا يقود التصميم

في الموضوع 8، جوهر التصميم الجيد، في الصفحة 51، نؤكد أن مقياس التصميم هو مدى سهولة تغيير نتيجة هذا التصميم: ينتج من التصميم الجيد شيئًا يسهل تغييره أكثر مما ينتج من التصميم السيئ.

ويوضح النقاش التالي حول الرشاقة سبب ذلك.

أنت تجري تغيير ثم تكتشف أنك لا تحبه. تقول الخطوة 3 في قائمتنا أنه يجب أن نكون قادرين على إصلاح ما نكسر. لجعل حلقة التغذية الراجعة فعّالة، يجب أن يكون هذا الإصلاح غير مؤلم قدر الإمكان. إذا لم يكن الأمر كذلك، فسنميل إلى تجاهلها وتركها غير مثبتة. نتحدث عن هذا التأثير في الموضوع 3، اعتلاج البرمجيات، في الصفحة 31. لجعل هذا الشيء المرن بالكامل يعمل، نحتاج إلى ممارسة التصميم الجيد، لأن التصميم الجيد يجعل الأشياء سهلة التغيير. وإذا كان التغيير سهلاً، فيمكننا التكيف، على كل مستوى، دون أي تردد.

تلك هي الرشاقة.

الأقسام ذات الصلة

- الموضوع 27، لا تتجاوز مصابيحك الأمامية، في الصفحة 146
- الموضوع 40، إعادة البناء، في الصفحة 235
- الموضوع 50، جوز الهند لا يفي بالغرض، في الصفحة 296

تحديات

حلقة الملاحظات البسيطة ليست مخصصة للبرمجيات فقط. فكّر في قرارات أخرى اتخذتها مؤخرًا. هل كان من الممكن تحسين أي منها بواسطة التفكير في طريقة التراجع عنها إذا لم تأخذك الأمور في الاتجاه الذي تريده؟ هل يمكنك التفكير في طرق يمكنك بواسطتها تحسين ما تفعله بواسطة جمع الانطباعات والعمل بناءً عليها؟

الفصل التاسع:

المشروعات العمليّة

9

مع بَدْء تقدم مشروعك، نحتاج إلى الابتعاد عن قضايا الفلسفة الفردية وكتابة الشِفرة لمناقشة قضايا أكبر بحجم المشروع. لن نخوض في تفاصيل إدارة المشروع، لكننا سنتحدث عن عدد قليل من المجالات الحاسمة التي يمكن أن تبني أي مشروع أو تهدمه.

فبمجرد أن يكون لديك أكثر من شخص واحد يعمل في مشروع ما، فستكون بحاجة إلى إنشاء بعض القواعد الأساسية وتفويض أجزاء من المشروع وفقًا لذلك. في الفرق العملية، سنوضح كيفيّة فعل ذلك مع مراعاة الفلسفة العملية (Pragmatic philosophy).

الغرض من نهج تطوير البرمجيات هو مساعدة الناس على العمل معًا. هل تقوم أنت وفريقك بعمل ما يناسبكم، أم أنكم تستثمرون فقط في الأدوات السطحية التافهة، ولا تحصلون على الفوائد الحقيقية التي تستحقون؟ سنرى لماذا جوز الهند لا يفي بالغرض وسنقدّم السر الحقيقي للنجاح.

وبالطبع لا شيء من ذلك مهم إذا لم تتمكن من تسليم البرمجية بشكل متسق وموثوق. هذا هو أساس الثلاثي السحري للتحكم في الإصدار والاختبار والأتمتة: مجموعة البادئ العملية.

في نهاية المطاف، فإن النجاح هو من وجهة نظر المراقب - راعي المشروع. ما يهم هو إدراك النجاح، وفي أبهِج مستخدميك، سنوضح لك كيفيّة إسعاد راعي كل مشروع.

النصيحة الأخيرة في الكتاب هي نتيجة مباشرة لكل ما يتبقى. في كبرياء وتحامل، نطلب منك التوقيع على عملك، وأن تفخر بما تفعله.

الموضوع 49. الفرق العمليّة

> في المجموعة إل (L)، يشرف *ستوفيل* على ستة مبرمجين من الدرجة الأولى، وهذا تحدّ إداري يمكن مقارنته تقريبًا برعي القطط.
>
> ^ مجلة واشنطن بوست 9 يونيو 1985

حتى في عام 1985، كانت النكتة حول رعي القطط قد بدأت بالتقادم. وبحلول وقت الطبعة الأولى في مطلع القرن، كانت قد أصبحت قديمة تمامًا. ومع ذلك فهي مستمرة، لأنها تحمل في طياتها مسحةً من الحقيقة. يشبه المبرمجون القطط إلى حدّ ما: أذكياء، قويّ الإرادة، عنيدين، مستقلين، وغالبًا ما يُبجّلون من قبل المحيطين.

لقد تناولنا في هذا الكتاب حتى الآن تقنيات عملية تساعد الفرد على أن يصبح مبرمجًا أفضل. لكن هل يمكن لهذه الطرق أن تنجح مع الفرق أيضًا، حتى بالنسبة لفرق من الأشخاص ذوي الإرادة القوية والمستقلين؟ الجواب بكل تأكيد "نعم!" هناك مزايا لكونك فردًا عمليًا، ولكن هذه المزايا تتضاعف مرّات عدّة إذا كان هذا الفرد يعمل في فريق عملي.

الفريق، من وجهة نظرنا، هو كيان صغير ومستقر في الغالب. خمسون شخصًا ليسوا فريقًا، بل هم حشد.[147] الفرق التي تسحب الأعضاء باستمرار إلى مهام أخرى ولا يعرف أحدهم الآخر ليسوا فريقًا أيضًا، فهم مجرّد غرباء يتشاركون مؤقتًا موقفا للحافلات تحت المطر.

الفريق العملي (pragmatic) صغير، أقل من 10-12 عضوًا أو نحو ذلك. نادرًا ما يأتي الأعضاء ويذهبون. الجميع يعرفون بعضهم البعض جيدًا ويثقون ببعضهم البعض ويعتمدون على بعضهم البعض.

نصيحة ٨٤	حافظ على فِرق صغيرة ومستقرة.

في هذا القسم، سنلقي نظرة سريعة على كيفيّة تطبيق التقنيات العملية على الفرق كليًا. هذه الملاحظات ليست سوى البداية. وبمجرّد أن يكون لديك مجموعة من المطورين العمليين يعملون في بيئة تمكينيّة، فسوف يطورون ويصقلون ديناميكيات فريقهم التي تعمل لمصلحتهم بسرعة.

دعونا نعيد صياغة بعض الأقسام السابقة بمصطلحات الفِرق.

لا نوافذ مكسورة

الجودة هي قضية الفريق. سيجد المطوّر الأكثر اجتهادًا والموضوع ضمن فريق قليل الاهتمام، صعوبة في الحفاظ على الحماس اللازم لحل المشكلات المقلقة. وتتفاقم المشكلة إذا عمل الفريق بنشاط على تثبيط المطور عن صرف الوقت على إجراء هذه الإصلاحات.

يجب ألا تتسامح الفرق كليًا مع النوافذ المكسورة - تلك العيوب الصغيرة التي لا يصلحها أحد. يجب أن يتحمل الفريق المسؤولية عن جودة المنتج، ودعم المطورين الذين يفهمون فلسفة عدم وجود نوافذ مكسورة والتي وصفناها في الموضوع 3، اعتلاج البرمجيات، في الصفحة 31، وتشجيع أولئك الذين لم يكتشِفوها بعد.

تحتوي بعض منهجيات الفريق على "موظف جودة" - وهو الشخص الذي يفوّضه الفريق بمسؤولية جودة المنتج. من الواضح أن هذا سخيف: لا يمكن أن تأتي الجودة إلا من مجموع المساهمات الفردية لجميع أعضاء الفريق. الجودة مُدمجة وليست مُلحقة.

الضفادع المغلية

تذكر الضفدع الموضوع في وعاء الماء، مرّة أخرى في الموضوع 4، حساء الحصى والضفادع المغليّة، في الصفحة 34؟ إنه لا يلاحظ التغيير التدريجي في بيئته وينتهي به الأمر مطبوخًا. يمكن أن يحدث الشيء نفسه للأفراد غير اليقظين. فقد يكون من الصعب مراقبة بيئتك العامّة في خضم تطور المشروع.

147 مع نمو حجم الفريق، تنمو مسارات الاتصال بمعدل O (n2)، حيث n هو عدد أعضاء الفريق. ففي الفرق الكبيرة، يبدأ التواصل بالانهيار ويصبح غير فعال.

من الأسهل على الفرق كليًا أن تغلي. حيث يفترض الأشخاص أن شخصًا آخر يتعامل مع المشكلة، أو أن قائد الفريق يجب أن يوافق على تغيير ما يطلبه المستخدم. حتى الفرق ذات النيات الحسنة يمكن أن تكون غافلة عن التغييرات المهمة في مشروعاتها.

حارب هذا. وشجّع الجميع على مراقبة البيئة بنشاط من أجل التغييرات. ابق متيقظًا ومدركًا للنطاق المتزايد، والمقاييس الزمنية المتناقصة، والميزات الإضافيّة، والبيئات الجديدة - أي شيء لم يكن في الفهم الأصلي. احتفظ بالمقاييس وفقًا للمتطلبات الجديدة.[148] لا يحتاج الفريق إلى رفض التغييرات خارج نطاق السيطرة - كل ما عليك هو أن تدرك أنها تحدث. وإلا، ستكون أنت في الماء الساخن.

جدول محفظتك المعرفيّة

في الموضوع 6ـ محفظتك المعرفية، في الصفحة 38، ألقينا نظرة على الطرق التي يجب أن تستثمر بها في محفظتك المعرفية الشخصيّة في وقتك الخاص. تحتاج الفرق التي تصبو إلى النجاح إلى التفكير في استثماراتها في المعرفة والمهارات أيضًا.

إذا كان فريقك جادًا بشأن التحسين والابتكار، فأنت بحاجة إلى جدولتهم زمنيًا. محاولة إنجاز الأشياء "عندما يتوفر وقت فراغ" تعني أنها لن تُنجز أبدًا. أيا كان نوع تراكم الأعمال غير المنجزة أو قائمة المهام أو التدفق الذي تستخدمه، لا تؤجلها لمصلحة تطوير المِيزة فقط. يعمل الفريق على أكثر من مجرد تطوير ميزات جديدة. تتضمن بعض الأمثلة المحتملة ما يلي:

صيانة الأنظمة القديمة

بينما نحب العمل على النظام الجديد اللامع، من المحتمل أن تكون هناك أعمال صيانة يجب القيام بها على النظام القديم. لقد التقينا بالفرق التي تحاول دفع هذا العمل إلى الزاوية. إذا كان الفريق مكلفًا بتنفيذ هذه المهام، فنفذها - بشكل حقيقي.

أعد صياغة العملية حسّها

لا يمكن أن يحدث التحسين المستمر إلا عندما تأخذ الوقت الكافي للنظر حولك، واكتشاف ما ينجح وما لا ينجح، ثم إجراء التغييرات (راجع الموضوع 48، جوهر التطوير الرشيق، في الصفحة 284). هناك عدد كبير جدًا من الفرق منشغلة للغاية في إنقاذ المياه بحيث لا يتوفر لديهم الوقت لإصلاح التسرب. جدوله. أصلحه.

تجارب تقنية جديدة

لا تعتمد تقنيات أو أطر عمل أو مكتبات جديدة فقط لمجرد "أن الجميع يفعل ذلك" أو بناءً على شيء رأيته في مؤتمر أو قرأته عبر الإنترنت. افحص التقنيات المرشحة عمدًا باستخدام النماذج الأولية. ضع المهام في الجدول الزمني لتجربة الأشياء الجديدة وتحليل النتائج.

148 مخطط الاحتراق أفضل لهذا من مخطط التراجع المعتاد. باستخدام مخطط الاحتراق، يمكنك أن ترى بوضوح كيف تغيّر قواعد اللعبة بوضوح.

تحسينات التعلم والمهارة

يعدّ التعلم والتحسينات الشخصية بداية رائعة، ولكن هناك العديد من المهارات تكون أكثر فعاليّة عند الانتشار على مستوى الفريق. خطط للقيام بذلك، سواء كان ذلك عند الغداء غير الرسمي أو عبر المزيد من جلسات التدريب الرسمية.

نصيحة ٨٥	جدوِلها لجعلها تحدث.

كن حاضرًا مع الفريق

من الواضح أن المطورين في الفريق يجب أن يتحدثوا مع بعضهم البعض. قدمنا بعض الاقتراحات لتسهيل ذلك في الموضوع 7، تواصل!، في الصفحة 44. ومع ذلك، فمن السهل أن ننسى أن الفريق نفسه له وجود داخل المنظمة. يحتاج الفريق ككيان إلى التواصل بوضوح مع بقية العالم.

بالنسبة إلى الغرباء، فإن أسوأ فرق المشروع هي تلك التي تبدو متوترة ومتحفظّة. فهم يعقدون اجتماعات عديمة الهيكلية، حيث لا يريد أحد التحدث. وحيث رسائل البريد الإلكتروني ووثائق المشروع الخاصة بهم في حالة من الفوضى: لا يوجد اثنتان متماثلتان، ويستخدم كل منها مصطلحات مختلفة.

تتمتع فرق المشروع العظيمة بشخصيّة مميزة. يتطلّع النّاس إلى اجتماعات معهم، لأنهم يعلمون أنهم سيرون أداءً جيّد الإعداد يجعل الجميع يشعرون بالرضا. تكون الوثائق التي ينتجونها واضحة ودقيقة ومتسقة. يتحدث الفريق بصوت واحد[149]، وربما يكون لديهم حس فكاهي.

هناك خدعة تسويقية بسيطة تساعد الفرق على التواصل كشخص واحد: إنشاء علامة تجارية (brand). عندما تبدأ مشروعًا، ابتكر اسمًا له، من الناحية المثالية شيئًا غير مألوف"جنوني". (في الماضي، سمّينا المشروعات بأسماء أشياء مثل الببغاوات القاتلة التي تفترس الأغنام والأوهام البصرية والجربوع وشخصيات الرسوم المتحركة والمدن الأسطورية). اقض 30 دقيقة في ابتكار شعار مرح واستخدمه. استخدم اسم فريقك بحرية عند التحدّث مع الناس. يبدو الأمر سخيفًا، لكنه يمنح فريقك هوية للبناء عليها، ويمنح العالم شيء لا يُنسى لربطه بعملك.

لا تكرروا أنفسكم

تحدثنا في الموضوع 9، DRY- شرور التكرار، في الصفحة 54، عن صعوبات التخلص من العمل المكرر بين أعضاء الفريق. تؤدي هذه الازدواجية إلى إهدار الجهود، ويمكن أن تؤدي إلى كابوس الصيانة. تعد أنظمة المدخنة "Stovepipe" أو ما يسمى الأنظمة "المنعزلة" (siloed) شائعة في هذه الفرق، مع تشاركية قليلة والكثير من الوظائف المكررة.

إن التواصل الجيد هو مفتاح تجنب هذه المشكلات. ونعني بكلمة "جيد" *فوري وخالي من الاحتكاك*.

149 يتحدث الفريق بصوت واحد - خارجيًا. داخليا، نشجع بشدة النقاش النشيط والقوي. يميل المطورون الجيدون إلى أن يكونوا شغوفين بعملهم.

يجب أن تكون قادرًا على طرح سؤال على أعضاء الفريق والحصول على رد فوري تقريبا. إذا كان الفريق موجودًا في مكان واحد، فقد يكون هذا بسيطًا مثل دس رأسك على الحائط المكعب أو أسفل القاعة. بالنسبة للفرق البعيدة، قد تضطر إلى الاعتماد على تطبيق مراسلة أو وسائل إلكترونية أخرى.

إذا كان عليك الانتظار لمدة أسبوع حتى يطرح اجتماع الفريق سؤالك أو مشاركة حالتك، فهذا يعني قدر كبير من الاحتكاك.[150] يعني عدم الاحتكاك أنه من السهل والمريح طرح الأسئلة ومشاركة تقدمك ومشكلاتك وآرائك و التعلم والبقاء على دراية بما يفعله زملائك في الفريق.

حافظ على الوعي للحفاظ على عدم تكرار نفسك

رصاصات الفريق الخطاطة

يتعيّن على فريق المشروع إنجاز العديد من المهام المختلفة في مناطق مختلفة من المشروع، والتعامل مع الكثير من التقنيات المختلفة. يجب أن يتم فهم المتطلبات وتصميم البنية وكتابة الشِفرة للواجهة الأمامية والخادم والاختبار. لكن من المفاهيم الخاطئة الشائعة أن هذه الأنشطة والمهام يمكن أن تحدث بشكل منفصل ومعزول. وهذا غير ممكن.

تدعو بعض المنهجيات إلى كل أنواع الأدوار والألقاب المختلفة داخل الفريق، أو تنشئ فرقًا متخصّصة منفصلة تمامًا. لكن المشكلة في هذا النهج هو أنه يقدّم بوابات وعمليات تسليم. الآن بدلاً من التدفق السلس من الفريق إلى النشر، لديك بوابات اصطناعية حيث يتوقف العمل. عمليات التسليم التي يجب أن تنتظر حتى يتم قبولها. الموافقات. ورقة العمل. تسمي المنهجية المرنة (Lean)ذلك بالنفايات، ويسعون جاهدين للقضاء عليها.

كل هذه الأدوار والأنشطة المختلفة هي في الواقع وجهات نظر مختلفة لنفس المشكلة، والفصل المصطنع بينها يمكن أن يسبب الكثير من المشكلات. على سبيل المثال، من غير المرجح أن يكون المبرمجون الذين يبعدون بمستويين أو ثلاث مستويات عن المستخدمين الفعليين لشفراتهم على دراية بالسياق الذي يتم فيه استخدام عملهم. لن يكونوا قادرين على اتخاذ قرارات مستنيرة.

باستخدام نهج الرصاصات الخطاطة، نوصي بتطوير ميزات فرديّة، مهما كانت صغيرة ومحدودة في البداية، تنتقل من نهاية إلى نهاية عبر النظام بِرُمَّته. هذا يعني أنك بحاجة إلى كل المهارات اللازمة للقيام بذلك داخل الفريق: الواجهة الأمامية، واجهة المستخدم / تجربة المستخدم (UI/UX)، الخادم، إدارة قواعد البيانات (DBA)، ضمان الجودة، إلخ، كلها مريحة ومعتادة على العمل مع بعضها البعض. باستخدام نهج الرصاصات الخطاطة، يمكنك تنفيذ أجزاء صغيرة جدًا من الوظائف بسرعة كبيرة، والحصول على تغذية راجعة فورية عن مدى جودة تواصل فريقك وتسليمه. هذا يخلق بيئة تمكنك من إجراء التغييرات وضبط فريقك والعملية بسرعة وسهولة.

نصيحة ٨٦	نظّم فرق وظيفيّة بالكامل.

ابن فِرقًا تتمكن من إنشاء شِفرة نهاية إلى نهاية، بشكل تدريجي ومتكرر.

150 التقى آندي بالفرق التي تجري مواقفهم اليومية في سكروم (Scrum) أيام الجمعة.

الأتمتة

هناك طريقة رائعة لضمان الاتساق والدقة وهي أتمتة كل ما يفعله الفريق. لماذا تتصارع مع معايير تنسيق الشِفرة عندما يستطيع المحرر أو بيئة التطوير المتكاملة IDE فعل ذلك نيابة عنك تلقائيًا؟ لماذا تُجري الاختبار اليدوي عندما يمكن للبناء المستمر تشغيل الاختبارات تلقائيًا؟ لماذا تنشر يدويًا في حين أن الأتمتة يمكن أن تفعل ذلك بنفس الطريقة في كل مرّة، بشكل متكرر وموثوق؟

الأتمتة عنصر أساسي في كل فرق المشروعات. تأكّد أن الفريق لديه مهارات في بناء الأدوات لإنشاء ونشر الأدوات التي تعمل على أتمتة تطوير المشروع ونشر الإنتاج.

اعرف متى تتوقف عن إضافة الطلاء

تذكر أن الفرق مكونة من أفراد. امنح كل عضو القدرة على التألق بطريقته الخاصة. امنحهم بنية كافية لدعمهم وللتأكد أن المشروع يقدّم قيمة. ثم، كما في مثال الرسام في برمجيات جيّدة كفاية، قاوم إغراء إضافة المزيد من الطلاء.

الأقسام ذات الصلة

- الموضوع 2، القطة أكلت شفرتي المصدرية، في الصفحة 29
- الموضوع 7، تواصل!، في الصفحة 44
- الموضوع 12، الرصاصات الخطّاطة، في الصفحة 73
- الموضوع 19، التحكم في الإصدار، في الصفحة 105
- الموضوع 50، جوز الهند لا يفي بالغرض، في الصفحة 296
- الموضوع 51، مجموعة البادئ العملية، في الصفحة 300

تحديات

- ابحث عن الفرق الناجحة خارج مجال تطوير البرمجيات. ما الذي يجعلهم ناجحين؟ هل يستخدمون أيًا من العمليات التي نُوقشت في هذا القسم؟
- في المرّة القادمة التي تبدأ فيها مشروعًا، حاول إقناع الأشخاص بإعطائه علامة تجارية. امنح مؤسستك الوقت لتعتاد الفكرة، ثم أجر تدقيق سريع لمعرفة الفرق الذي أحدثته، سواء داخل الفريق أو خارجه.
- ربما واجهت مشكلات ذات مرة مثل "إذا استغرق حفر حفرة من 4 عمال 6 ساعات، فكم من الوقت سيستغرق 8 عمال في حفرها؟" لكن في الحياة الواقعية، ما هي العوامل التي تؤثر على الإجابة إذا كان العمال يكتبون شفرةً بدلاً من ذلك؟ كم عدد السيناريوهات التي يتم فيها تقليل الوقت فعليًا؟

- اقرأ شهر الرجل الأسطوري [Bro96] بقلم فريدريك بروكس. للحصول على رصيد إضافي، اشتري نسختين حتى تتمكن من قراءته مرتين بسرعة.

الموضوع 50. جوز الهند لا يفي بالغرض

لم يسبق لسكان الجزيرة الأصليون أن رأوا طائرة من قبل، أو التقوا بأشخاص مثل هؤلاء الغرباء. ففي مقابل استخدام أراضيهم، قدم الغرباء طيورًا ميكانيكية كانت تقلع وتهبط طوال اليوم على "المدرج"، جالبةً معها ثروة مادية لا تصدق إلى موطنهم على الجزيرة. ذكر الغرباء شيئًا عن الحرب والقتال. وفي يوم من الأيام انتهى كل شيء وغادروا جميعًا، آخذين معهم ثرواتهم الغريبة.

كان سكان الجزيرة مستميتين لاستعادة ثرواتهم الطيبة، وأعادوا بناء صورة طبق الأصل من المطار وبرج المراقبة والمعدات باستخدام مواد محلية: الكُرُوم، وقشور جوز الهند، وسعف النخيل، وما إلى ذلك. لكن لسبب ما، وعلى الرغم من وجود كل شيء في مكانه، لم تأت الطائرات. لقد قلّدوا الشكل وليس المحتوى. يسمي علماء الأنثروبولوجيا ذلك بـ عبادة الشحن "البضائع".

في كثير من الأحيان، نكون نحن سكان الجزر.

من السهل والمغري الوقوع في فخ عبادة الشحن : بواسطة استثمار وبناء الأدوات التي يسهل رؤيتها، فأنت تأمل في جذب سحر العمل الأساسي. ولكن كما هو الحال مع عبادة الشحن الأصلية في ميلانيزيا،[151] فإن مطار مزيف مصنوع من قشور جوز الهند ليس بديلاً عن الشيء الحقيقي.

على سبيل المثال، لقد رأينا شخصيًا فرقًا تدعي أنها تستخدم سكروم (Scrum). ولكن، عند الفحص الدقيق، اتضح أنهم كانوا يجتمعون وقوفاً يوميًا مرّة في الأسبوع، مع تكرارات لمدة أربعة أسابيع والتي غالبًا ما تتحول إلى تكرار لمدة ستة أو ثمانية أسابيع. لقد شعروا أن هذا لا بأس به لأنهم كانوا يستخدمون أداة جدولة "رشيقة" (agile) شائعة. كانوا يستثمرون فقط في الأدوات السطحية - وحتى في ذلك الوقت، غالبًا بالاسم فقط، كما لو كان "الوقوف" أو "التكرار" نوعًا من التعويذات للخرافات. مما لا يثير الدهشة أنهم فشلوا أيضًا في اجتذاب السحر الحقيقي.

السياق مهم

هل وقعت أنت أو فريقك في هذا الفخ؟ اسأل نفسك، لماذا تستخدم طريقة التطوير هذه؟ أو إطار العمل هذا؟ أو تقنية الاختبار تلك؟ هل هي فعلًا مناسبة تمامًا للوظيفة التي تقوم بها؟ هل تعمل بشكل جيّد بالنسبة لك؟ أم أنه تم تبنيها لمجرد أنه تم استخدامها بواسطة أحدث قصة نجاح مدعومة بالإنترنت؟

هناك توجه حالي لتبني سياسات وعمليات الشركات الناجحة مثل سبوتيفي (Spotify)، نيتفليكس (Netflix)، ستريبي (Stripe)، جيت لاب (GitLab)، وغيرها. لكل منها آراءه الفريدة في تطوير البرمجيات وإدارتها. ولكن تأمل في السياق: هل أنت في نفس السوق، مع نفس القيود والفرص، والخبرات المماثلة وحجم المؤسسة، والإدارة المشابهة، والثقافة المماثلة؟ وقاعدة المستخدمين والمتطلبات المماثلة؟

لا تهبطوا عليه. فالأدوات المحددة والبنى السطحية والسياسات والعمليات والأساليب غير كافية.

نصيحة ٨٧	افعل ما ينجح وليس ما هو عصري.

151 راجع https://en.wikipedia.org/wiki/Cargo_cult

كيف تعرف "ما الذي ينجح"؟ أنت تعتمد على أهم التقنيات العملية:

جربه.

جرب الفكرة مع فريق صغير أو مجموعة من الفرق. حافظ على الأجزاء الجيّدة التي يبدو أنها تعمل بشكل جيد، وتخلص من أي شيء آخر كتفايات أو عبء. لن يقلّل أحد من مستوى عمل مؤسستك لأنها تعمل بشكل مختلف عن "سبوتيفي" أو "نيتفليكس"، لأنه حتى هم أنفسهم لم يتبعوا العمليات الحالية في أثناء النمو. وبعد سنوات من الآنَ، ومع نضج تلك الشركات ودوراتها والاستمرار في الازدهار، فإنها ستقوم بشيء مختلف مرة أخرى.

هذا هو السر الحقيقي لنجاحهم.

مقاس واحد لا يناسب أحد تماما

الغرض من منهجية تطوير البرمجيات هو مساعدة الناس على العمل معًا. كما نناقش في جوهر التطوير الرشيق، لا توجد خِطّة واحدة يمكنك اتباعها عند تطوير البرمجيات، لا سيما الخِطّة التي توصل إليها شخص آخر في شركة أخرى.

العديد من برامج الشهادات (certification) هي في الواقع أسوأ من ذلك: فهي تستند إلى قدرة الطالب على حفظ القواعد واتباعها. لكن ليس هذا ما تريده. أنت بحاجة إلى القدرة على رؤية ما وراء القواعد الحالية واستغلال إمكانات المِيزة. هذه عقلية مختلفة تمامًا عن "لكن

آجايل (agile) / إكس بي (XP)/ كانبان (Kanban)/ لين (Lean)/ سكروم(Scrum) يفعل ذلك بهذه الطريقة ..." وهلمّ جرًا.

بدلاً من ذلك، تريد أن تأخذ أفضل القطع من أي منهجية معينة وتكييفها للاستخدام. لا يوجد حجم واحد يناسب الجميع، والطرق الحالية بعيدة عن الاكتمال، لذلك ستحتاج إلى إلقاء نظرة على أكثر من طريقة واحدة شائعة.

على سبيل المثال، يحدّد سكروم بعض ممارسات إدارة المشروع، لكن سكروم في حد ذاته لا يوفر إرشادات كافية على المستوى التقني للفرق أو على مستوى المحفظة / الحكومي للقيادة. إذا، أين تبدأ؟

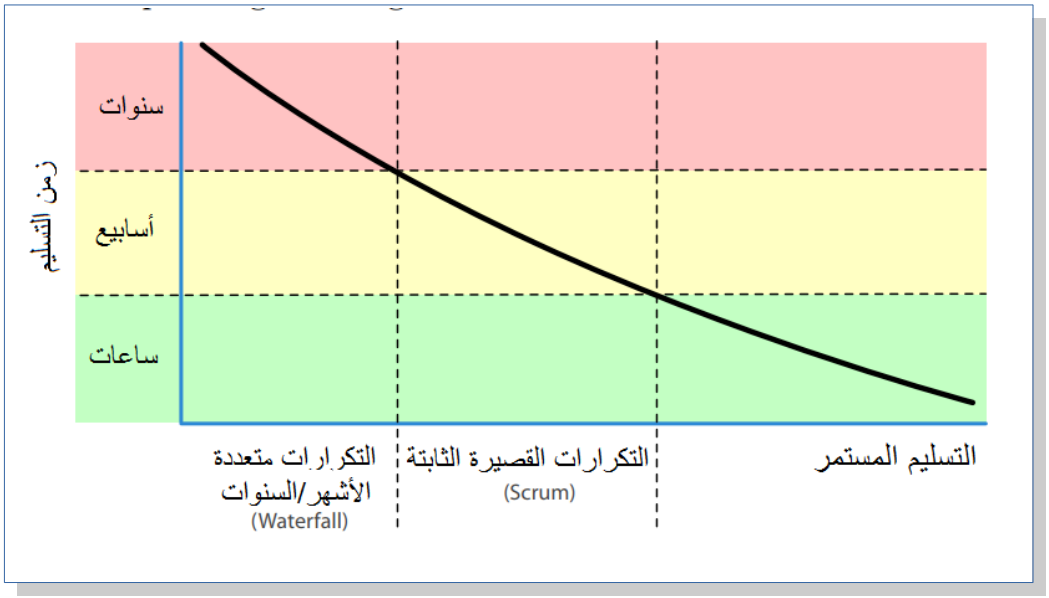
كن مثلهم!

كثيرا ما نسمع قادة تطوير البرمجيات يقولون لموظفيهم: "يتعين علينا أن نعمل مثل شركة نيتفليكس" (أو واحدة من هذه الشركات الرائدة الأخرى). بالطبع يمكنك فعل ذلك.

لكن أولا، احصل على بضع مئات الآلاف من الخوادم وعشرات الملايين من المستخدمين...

الهدف الحقيقي

الهدف بالطبع ليس "عمل سكروم" أو "عمل أجايل" أو "عمل لين" أو ما لديك. الهدف هو أن تكون في وضع يسمح بتقديم برمجية عمل تمنح المستخدمين بعض الإمكانات الجديدة في أي لحظة. ليس بعد أسابيع أو أشهر أو سنوات من الآن، بل الآن. بالنسبة للعديد من الفرق والمؤسسات، يبدو التسليم المستمر وكأنه هدف رفيع بعيد المنال، خاصة إذا كنت مثقلًا بعملية تقصُر التسليم على شهور أو حتى أسابيع. ولكن كما هو الحال مع أي هدف، فإن المفتاح هو الاستمرار في التوجه في الاتجاه الصحيح.



إذا كنت تُسلّم في سنوات، فحاول تقصير الدورة إلى أشهر. ومن شهور، اقصرها لأسابيع. ومن الإسراع لمدة أربعة أسابيع، جرب أسبوعين. من أسبوعين، جرب أسبوعا. ثم يوميا. ثم، أخيرًا، عند الطلب. لاحظ أن القدرة على التسليم عند الطلب لا تعني أنك مجبر على التسليم في كل دقيقة من كل يوم. أنت تقدم الخِدْمَات عندما يحتاجها المستخدمون، وعندما يكون من المنطقي أن تفعل ذلك من الناحية التجارية.

نصيحة ٨٨	سلّم عندما يحتاج المستخدمون.

من أجل الانتقال إلى هذا النمط من التطوير المستمر، فأنت بحاجة إلى بنية تحتيّة متينة، والتي نناقشها في الموضوع التالي، مجموعة البادئ العملية. أنت تطور في الجذع الرئيس لنظام التحكم في الإصدار الخاص بك، وليس في الفروع، وتستخدم تقنيات مثل مفاتيح التبديل للميزات لنشر ميزات الاختبار للمستخدمين بشكل انتقائي.

بمجرد أن تصبح البنية التحتيّة الخاصّة بك في حالة جيدة، عليك أن تقرر كيفية تنظيم العمل. قد يرغب المبتدئين في البَدْء بسكروم لإدارة المشروعات، بالإضافة إلى الممارسات التقنية من البرمجة الفائقة eXtreme Programming (XP). قد تتطلع الفرق الأكثر انضباطًا وخبرة إلى تقنيات كانبان (Kanban) و لين (Lean)، سواء بالنسبة للفريق وربما للمشكلات الحكومية الأكبر.

لكن لا تأخذ كلمتنا على محمل الجدّ، تحقق هذه الأساليب وجرّبها بنفسك. كن حذرًا، مع ذلك، في المبالغة في ذلك. الإفراط في الاستثمار في أي منهجية معينة يمكن أن يتركك أعمى عن الأبدال. سوف تعتادها. سرعان ما يصبح من الصعب رؤية أي طريقة أخرى. لقد أصبحت متكلسًا، والآن لا يمكنك التكيف بسرعة بعد الآن.

قد تستخدم كذلك جوز الهند.

الأقسام ذات الصلة

- الموضوع 12، الرصاصات الخطّاطة، في الصفحة 73
- الموضوع 27، لا تتجاوز مصابيحك الأمامية، في الصفحة 146
- الموضوع 48، جوهر التطوير الرشيق، في الصفحة 284
- الموضوع 49، الفِرَق العملية، في الصفحة 289
- الموضوع 51، مجموعة البادئ العملية، في الصفحة 300

الموضوع 51. مجموعة البَدْء العملية

> تتقدم الحضارة بواسطة زيادة عدد العمليات المهمة التي يمكننا أداؤها دون تفكير.
> ^ ألفريد نورث وايتهيد

عندما كانت السيارات في بداياتها، كانت التعليمات الخاصة ببَدْء تشغيل سيارة فورد طراز تي T Ford أطول من صفحتين. ولكن في السيارات الحديثة، ما عليك سوى الضغط على زر فقط - فإجراء بادئ التشغيل تلقائي ومضمون. قد يقوم الشخص الذي يتبع قائمة التعليمات بإغراق المحرك، لكن البادئ التلقائي (automatic starter) لن يفعل ذلك.

مع أنّ تطوير البرمجيات لا يزال صناعة في مرحلة الطراز تي (Model-T)، فإنه لا يمكننا تحمل تكلفة قراءة صفحتين من الإرشادات مرارًا وتكرارًا لبعض العمليات الشائعة. سواء كان الأمر يتعلق بإجراء البناء والإصدار، أو الاختبار، أو الأعمال الورقية للمشروع، أو أي مهمة أخرى متكررة في المشروع، يجب أن تكون تلقائية وقابلة للتكرار على أي جهاز مؤهل.

إضافة إلى ذلك، نريد ضمان الاتساق وقابلية التكرار في المشروع. تترك الإجراءات اليدوية الاتساق متروكًا للصدفة؛ قابلية التكرار غير مضمونة، خاصة إذا كانت جوانب الإجراء مفتوحة للتفسير من قبل أشخاص مختلفين.

بعد أن كتبنا الإصدار الأول من المبرمج العملي، أردنا إنشاء المزيد من الكتب لمساعدة الفرق على تطوير البرمجيات. لقد توصلنا إلى أنه يجب علينا أن نبدأ من البداية: ما هي العناصر الأساسية والأكثر أهمية التي يحتاجها أي فريق بغض النظر عن المنهجية أو اللغة أو مجموعة التِقَانَة. وهكذا ولدت فكرة مجموعة البادئ العملية، والتي تغطي هذه الموضوعات الثلاثة الهامة والمترابطة:

- التحكم في الإصدار
- اختبار الانحدار
- الأتمتة الكاملة

هذه هي الأرجل الثلاث التي تدعم كل مشروع. إليك الطريقة.

استخدم التحكم بالإصدار للتوجيه

كما قلنا في التحكم في الإصدار، فأنت تريد الاحتفاظ بكل ما يلزم لبناء مشروعك تحت التحكم في الإصدار. تصبح هذه الفكرة أكثر أهمية في سياق المشروع نفسه.

أولاً، يسمح لأجهزة البناء بأن تكون مؤقتة. بدلاً من آلة واحدة مهيبة ومزعجة في زاوية المكتب يخشى الجميع لمسها،[152] يتم إنشاء آلات بناء و / أو مجموعات"عناقيد" (clusters) عند الطلب كنماذج موضعية في السحابة. يخضع ضبط النشر (Deployment configuration) أيضًا للتحكم في الإصدار، لذا يمكن التعامل مع إطلاق الإصدار للإنتاج تلقائيًا.

وهذا هو الجزء المهم: على مستوى المشروع، يقود التحكم في الإصدار عملية الإنشاء والإصدار.

نصيحة ٨٩	استخدم التحكم في الإصدار لتوجيه البناءات والاختبارات والإصدارات.

أي، تُقدح البناء والاختبار والنشر عبر عمليات الإيداع أو الدفع للتحكم في الإصدار، وتُضمّن في حاوية في السحابة. ويُحدّد الإصدار إلى التدريج أو الإنتاج باستخدام علامة (tag) في نظام التحكم في الإصدار الخاص بك. تصبح الإصدارات بعد ذلك جزءًا أقل بكثير من الحياة اليومية - تسليم حقيقي مستمر، غير مرتبط بأي آلة بناء أو آلة مطور.

اختبار قاس ومستمر

يختبر العديد من المطورين بلطف، لا شعوريًا بمعرفة مكان كسر الشِفرة وتجنب نِقَاط الضعف. لكن المبرمجون العمليون مختلفون. فنحن مدفوعون للعثور على الأخطاء الموجودة لدينا الآن، لذلك ليس علينا أن نتحمل العار من الآخرين الذين يعثرون على أخطائنا لاحقًا.

يشبه العثور على الأخطاء إلى حد ما الصيد بشبكة. نستخدم الشباك الصغيرة الدقيقة (اختبارات الوحدة) لصيد أسماك المنوة الصغيرة، والشبكات الكبيرة الخشنة (اختبارات التكامل) لصيد أسماك القرش القاتلة. في بعض الأحيان، تتمكن الأسماك من الهروب، لذلك نصلح أي ثقوب نعثر عليها، على أمل اكتشاف المزيد والمزيد من العيوب المراوغة التي تسبح في محيط مشروعنا.

نصيحة ٩٠	اختبر مبكرًا، اختبر كثيرًا، اختبر تلقائيًا.

نريد أن نبدأ الاختبار بمجرّد أن يكون لدينا شِفرة. تلك الأسماك الصغيرة لديها عادة سيئة وهي أن تصبح أسماك قرش عملاقة تأكل الإنسان بسرعة كبيرة، ثم أن اصطياد سمكة قرش أصعب قليلاً. لذلك نكتب اختبارات الوحدة. الكثير من اختبارات الوحدة.

في الواقع، قد يحتوي المشروع الجيد على شِفرة اختبار أكثر من احتوائه على شِفرة الإنتاج. الوقت المستغرق لإنتاج شِفرة الاختبار هذا يستحق الجهد المبذول. ينتهي الأمر بكلفة أقل بكثير على المدى الطويل، وفي الواقع لديك فرصة لإنتاج منتج بلا عيوب تقريبا.

إضافة إلى ذلك، فإن معرفتك بأنك قد اجتزت الاختبار يمنحك درجة عالية من الثقة في أن جزءًا من التعليمات البرمجية قد "تم".

نصيحة ٩١	لن يتم الانتهاء من البرمجة حتى تُجرى جميع الاختبارات.

152 لقد رأينا هذه اليد الأولى مرات أكثر مما تعتقد.

يقوم البناء التلقائي بتشغيل جميع الاختبارات المتاحة. من المهم أن تهدف إلى "اختبار حقيقي"، بمعنى آخر، يجب أن تتطابق بيئة الاختبار مع بيئة الإنتاج بإحكام. أية ثغرات ستكون مكانًا لتكاثر الأخطاء (bugs).

قد يغطي الإصدار عدة أنواع رئيسة من اختبار البرمجيات: اختبار الوحدة؛ اختبار التكامل؛ المصادقة والتحقق من الصحة؛ واختبار الأداء.

هذه القائمة ليست كاملة بأي حال من الأحوال، وستتطلب بعض المشروعات المتخصصة أنواعًا أخرى مختلفة من الاختبارات أيضًا. لكنه يعطينا نقطة انطلاق جيدة.

اختبار الوحدة

اختبار الوحدة هو شِفرة تستخدم وحدة. لقد غطينا هذا في الموضوع 41، اختبر لتكتب الشِفرة، في الصفحة 240. إن اختبار الوحدة هو الأساس لجميع أشكال الاختبار الأخرى التي سنناقشها في هذا القسم. إذا لم تعمل الأجزاء بحد ذاتها "بشكل مستقل"، فمن المحتمل ألا تعمل بشكل جيد مجتمعة. يجب أن تجتاز جميع الوحدات التي تستخدمها اختبارات الوحدة الخاصة بها قبل أن تتمكن من المتابعة.

بمجرد اجتياز جميع الوحدات ذات الصلة لاختباراتها الفردية، تكون جاهزًا للمرحلة التالية. تحتاج إلى اختبار كيفية استخدام جميع الوحدات وتفاعلها مع بعضها البعض في جميع أنحاء النظام.

اختبار التكامل

يوضح اختبار التكامل أن الأنظمة الفرعية الرئيسة التي يتكون منها المشروع تعمل وأنها تعمل بشكل جيد مع بعضها البعض. فمع وجود عقود جيدة في مكانها المناسب واختبارها جيدًا، يمكن اكتشاف أي مشكلات في التكامل بسهولة. وبخلاف ذلك، يصبح التكامل أرضًا خصبة لتكاثر الأخطاء. في الواقع، غالبًا ما يكون أكبر مصدر منفرد للأخطاء في النظام.

إن اختبار التكامل هو في الحقيقة مجرد امتداد لاختبار الوحدة الذي وصفناه - حيث تختبر فقط كيفية احترام الأنظمة الفرعية بِزِمَّتها لعقودها.

التحقق (Verification) التأكد الصَّلاحِيَّة (Validation)

بمجرد أن يكون لديك واجهة مستخدم قابلة للتنفيذ أو نموذج أولي، فأنت بحاجة إلى الإجابة على سؤال مهم للغاية: لقد أخبرك المستخدمون بما يريدون، ولكن هل هذا فعلا ما يحتاجون؟

هل يفي بالمتطلبات الوظيفية للنظام؟ هذا، أيضا، يحتاج إلى اختبار. إن النظام الخالي من الأخطاء الذي يجيب عن السؤال الخاطئ ليس مفيدًا جدًا. كن على دراية بأنماط وصول المستخدم النهائي وكيفية اختلافها عن بيانات اختبار المطوّر (على سبيل المثال، راجع القصة حول ضربات الفرشاة في الصفحة 111).

اختبار الأداء

قد يكون اختبار الأداء (Performance) أو اختبار الإجهاد (stress) من الجوانب المهمة للمشروع أيضًا.

اسأل نفسك ما إذا كانت البرمجية تلبي متطلبات الأداء في ظل ظروف العالم الحقيقي - مع العدد المتوقع من المستخدمين، أو الاتصالات، أو المعامَلات في الثانية. هل هي قابلة للتطوير؟

بالنسبة لبعض التطبيقات، قد تحتاج إلى أجهزة أو برمجيات اختبار متخصصة لمحاكاة الحمل بشكل واقعي.

اختبار الاختبارات

نظرًا لأننا لا نستطيع كتابة برمجيات مثالية، فهذا يعني أننا لا نستطيع كتابة برمجية اختبار مثالية أيضًا. نحن بحاجة لاختبار هذه الاختبارات.

فكّر في مجموعتنا من مجموعات الاختبار لدينا كنظام أمان متطور، مصمم لدق ناقوس الخطر عند ظهور خطأ ما. كم هو أفضل اختبار نظام أمان من محاولة اختراقه؟

بعد كتابة اختبار لاكتشاف خطأ معين، تسبب في الخطأ عمداً وتأكد أن الاختبار يشتكي. هذا يضمن أن الاختبار سيكتشف الخطأ إذا حدث فعليًا.

نصيحة ٩٢	**استخدم المخبريين لاختبار اختبارك.**

إذا كنت جادًا بشأن الاختبار، فاخذ فرعًا منفصلًا من شجرة المصدر، وأدخل الأخطاء عمدا، وتحقّق أن الاختبارات ستكتشفها. وعلى مستوى أعلى، يمكنك استخدام شيء مثل Chaos Monkey[153] [154] من نتفليكس (Netflix) لتعطيل (أي "قتل") الخِدْمَات واختبار مرونة تطبيقك.

عند كتابة الاختبارات، تأكد صدور صوت الإنذارات في الوقت المناسب.

اختبر بكل معنى الكلمة

بمجرد أن تتأكد صحة اختباراتك، وتجد الأخطاء التي أنشأتها، كيف تعرف ما إذا كنت قد اختبرت قاعدة الشِفرة جيّدًا بما فيه الكفاية؟

153 **[المترجم]** أو تدعى هندسة الفوضى وهي نوع من الاختبارات للنظام البرمجي في مرحلة الإنتاج من أجل بناء الثقة في قدرة النظام على تحمل الظروف المضطربة وغير المتوقعة.

154 https://netflix.github.io/chaosmonkey

الإجابة المختصرة هي "لا تعرف" ولن تعرف ذلك أبدًا. قد تتطلع إلى تجربة أدوات تحليل التغطية التي تراقب شفرتك في أثناء الاختبار وتتبع أي من أسطر الشِفرة نُفّذ وأيها لم يُنفّذ. تساعد هذه الأدوات في منحك فكرة عامة عن مدى شمولية اختبارك، ولكن لا تتوقع أن ترى تغطية بنسبة 100٪.[155]

حتى لو أصبت بكل سطر من الشِفرة، فهذه ليست الصورة الكاملة. المهم هو عدد الحالات التي قد يتضمنها برنامَجك. الحالات لا تكافئ أسطر التعليمات البرمجية. على سبيل المثال، افترض أن لديك دالة تأخذ عددين صحيحين، يمكن أن يكون كل منهما رقمًا من 0 إلى 999:

```
 int test(int a, int b) {
return a / (a + b);
}}
```

من الناحية النظرية، تحتوي هذه الدالة المكونة من ثلاثة أسطر على 1000000 حالة منطقية، سيعمل 999999 منها بشكل صحيح وواحدة لن تعمل (عندما تكون a + b تساوي صفرًا). إن مجرد معرفة أنك نفذت هذا السطر من الشِفرة لا يخبرك بذلك — ستحتاج إلى تحديد جميع الحالات الممكنة للبرنامَج. لسوء الحظ، عمومًا هذه مشكلة صعبة حقًا. صعبة مثل "أن الشمس ستصبح كتلة صلبة باردة قبل أن تتمكن من حلها."

نصيحة ٩٣	اختبر تغطية الحالة، وليس تغطية الشِفرة.

الاختبار القائم على الخاصية

هناك طريقة رائعة لاستكشاف كيفية تعامل شفرتك مع الحالات غير المتوقعة وهي أن يكون لديك جهاز حاسوب ينشئ هذه الحالات.

استخدم تقنيات الاختبار القائمة على الخاصية لتوليد بيانات الاختبار وفقًا للعقود وثوابت الشِفرة قيد الاختبار. نغطي هذا الموضوع بالتفصيل في الموضوع 42، الاختبار المعتمد على الخاصية، في الصفحة 249.

تقوية الشبكة

أخيرًا، نودّ الكشف عن أهم مفهوم منفرد في الاختبار. إنه مفهوم واضح، ويذكر كل كتاب مدرسي تقريبًا أنه علينا القيام بذلك بهذه الطريقة. لكن ولسببٍ ما، لا تزال معظم المشروعات لا تفعل ذلك. إذا انزلق خطأ ما عبر شبكة الاختبارات الحالية، فأنت بحاجة إلى إضافة اختبار جديد لاعتراضه في المرّة القادمة.

نصيحة ٩٤	جد الخلل مرّة واحدة.

بمجرد أن يجد المختبر البشري خطأً ما، يجب أن تكون هذه هي المرّة الأخيرة التي يجد فيها المختبِر هذا الخطأ. يجب تعديل الاختبارات الآلية للتحقق من هذا الخطأ المعين منذ تلك اللحظة، وفي كل مرة، دون استثناءات، مهما كانت تافهة، وبغض النظر عن مقدار شكوى المطور وقوله، "أوه، لن يحدث هذا مرّة أخرى".

155 للحصول على دراسة شيقة للعلاقة بين تغطية الاختبار والعيوب، انظر تغطية اختبار الوحدة الأسطورية [ADSS18].

لأنه سيحدث مرّة أخرى. وليس لدينا الوقت لمطاردة الأخطاء التي كان من الممكن أن تجدها لنا الاختبارات الآلية. علينا أن نقضي وقتنا في كتابة شِفرة جديدة - وأخطاء جديدة.

الأتمتة الكاملة

كما قلنا في بداية هذا القسم، فإن التطور الحديث يعتمد على إجراءات تلقائية ومشِفرة (scripted). سواء كنت تستخدم شيئًا بسيطًا مثل تدوينات شل النصية مع rsync و ssh، أو حلول كاملة الميزات مثل Ansible أو Puppet أو Chef أو Salt، والتي لا تعتمد على أي تدخل يدوي.

كنّا ذات مرّة في موقع العميل حيث كان جميع المطورين يستخدمون نفس بيئة التطوير المتكاملة (IDE). أعطى مسؤول النظام لكل مطوّر مجموعة من التعليمات حول تثبيت حزم الوظائف الإضافية إلى بيئة التطوير هذه. ملأت هذه الإرشادات العديد من الصفحات - الصفحات المليئة بـ انقر هنا، مرّر هناك، اسحب هذا، انقر نقرًا مزدوجًا على ذاك، ثم افعلبذلك مرّة أخرى.

ليس من المستغرب أن يتم تحميل جهاز كل مطوّر بشكل مختلف قليلاً. حدثت اختلافات طفيفة في سلوك التطبيق عندما قام مطورون مختلفون بتشغيل نفس الشِفرة. قد تظهر الأخطاء على جهاز واحد محدد ولكن ليس على أجهزة أخرى. عادة ما يكشف تعقب الاختلافات في الإصدار لأي مكون عن مفاجأة.

نصيحة ٩٥ **لا تستخدم الإجراءات اليدوية.**

الناس ليسوا قابلين للتكرار مثل أجهزة الحواسيب. ويجب ألا نتوقع منهم أن يكونوا كذلك. سينفّذ البرنامَج النصي أو برنامَج الشل نفس التعليمات، بنفس الترتيب، مرّة تلو الأخرى. إنه يخضع للتحكم في الإصدار بحد ذاته، لذا يمكنك فحص التغييرات التي أجريت على إجراءات البناء / الإصدار بمرور الوقت أيضًا ("لكنها اعتادت العمل ...").

كل شيء يعتمد على الأتمتة. لا يمكنك بناء المشروع على خادم سحابي مجهول ما لم يكن البناء تلقائيًا بالكامل. ولا يمكنك النشر تلقائيًا إذا كانت هناك خطوات يدوية متضمنة. وبمجرد تقديمك للخطوات اليدوية ("لهذا الجزء فقط ...") تكون قد كسرت نافذة كبيرة جدًا.[156]

بواسطة هذه الأرجل الثلاثة للتحكم في الإصدار والاختبار القاسي والأتمتة الكاملة، سيكون لمشروعك الأساس المتين الذي تحتاجه لتتمكن من التركيز على الجزء الصعب: إسعاد المستخدمين.

الأقسام ذات الصلة

- الموضوع 11، قابلية العكس، في الصفحة 69

156 تذكر دائمًا اعتلاج البرمجيات. دائما.

- الموضوع 12، الرصاصات الخطّاطة، في الصفحة 73
- الموضوع 17، ألعاب شِل، الصفحة 99
- الموضوع 19، التحكم في الإصدار، في الصفحة 105
- الموضوع 41، اختبر لتكتب الشِفرة، في الصفحة 240
- الموضوع 49، الفِرَق العملية، في الصفحة 289
- الموضوع 50، جوز الهند لا يفي بالغرض، في الصفحة 296

تحديات

- هل عمليات الإنشاء الليليـة أو المسـتمرة خاصـتك تلقائيـة، ولكن النشـر في الإنتـاج ليس كـذلك؟ لمـاذا ا؟ مـا الـذي يميّـز هـذا الخادم؟
- هل يمكنك اختبار مشروعك تلقائيًا بالكامل؟ تضطر العديـد من الفـرق على الإجابـة بـ "لا". لمـاذا ا؟ هـل من الصـعب تحديـد النتائج المقبولة؟ ألن يجعل ذلك من الصعب الإثبات للرعاة بأن المشروع "نُفّذ"؟
- هل من الصعب للغاية اختبار منطق التطـبيق بشـكل مسـتقل عن واجهـة المسـتخدم الرسـومية؟ مـاذا يقـول هـذا عن واجهـة المستخدم الرسومية؟ عن الاقتران؟

الموضوع 52. أبهِج مستخدميك

> عندما تُبهر الناس، فإن هدفك ليس جني الأموال منهم أو حملهم على فعل ما تريد، ولكن لملئهم بسعادة كبيرة.
>
> ➤ جاي كاواساكي

هـدفنا كمطـورين هـو إرضـاء المسـتخدمين. لهـذا نحن هنـا. ليس من أجل الحصـول على بيانـاتهم، أو الاحتيـال عليهم أو إفـراغ محافظهم. وبغض النظر عن الأهداف الشائنة، فإن تقديم برمجية عاملة في الوقت المناسب ليس كافيًا.

ذلك وحده لن يُبهجهم.

لا يتحفّز المستخدمين خصوصا بواسطة الشِفرة. بدلاً من ذلك، لديهم مشكلة عمل تحتاج إلى حل في سياق أهدافهم وميزانياتهم. يؤمنون أنه بواسطة العمل مع فريقك سيكونون قادرين على حلها.

إن توقعاتهم ليست متعلقة بالبرمجيات. حتى إنهم لم يُضمّنوها في أية مواصفات يقدمونها لك (لأن هذه المواصفات ستكون غـير مكتملة حتى يقوم فريقك بتكرارها معهم مرات عدّة).

كيف تكشف عن توقعاتهم إذًا؟ اطرح سؤالًا بسيطًا:

> كيف ستعرف أننا جميعًا نجحنا بعد شهر (أو عام، أو أيًا كان) بعد الانتهاء من هذا المشروع؟

قـد تُفاجأ بالإجابـة. يمكن في الواقـع الحكم على مشـروع لتحسـين توصـيات المنتج من حيث الاحتفـاظ بـالعملاء؛ قـد يُحكم على مشروع لدمج قاعدتي بيانات من حيث جودة البيانات، أو ربما من حيث توفير التكاليف. إن هذه التوقعات الخاصة بقيمة العمل هي مـا يُهم حقًا - وليس فقط مشروع البرمجية نفسها. البرمجية ليست سوى وسيلة لتحقيق هذه الغايات.

والآن بعد أن ظهرت على السـطح بعض التوقعـات الأساسـية للقيمـة الكامنة وراء المشـروع، يمكنـك البَـدْء في التفكـير في كيفيـة تحقيقها في ضوء ذلك:

- تأكد أن هذه التوقعات واضحة تمامًا لكل فرد من في الفريق.
- عند اتخاذك للقرارات، فكّر في المسار الذي يقترب من تلك التوقعات.
- حلّل متطلبات المستخدم تحليلًا نقديًا في ضوء هذه التوقعات. اكتشفنا في العديـد من المشـروعات أن "المتطلب" المـذكور كان في الواقع مجرّد تخمين لما يمكن أن تفعله التِقَائَة، لقد كانت في الواقع خِطّة تنفيـذ لهـواة وُضـعت كوثيقـة متطلبـات. لا تخف من تقديم اقتراحات من شأنها تغيير المتطلب إذا كان بإمكانك إثبات أنها ستقرّب المشروع من الهدف.
- استمر في التفكير في هذه التوقعات في أثناء تقدمك في المشروع.

لقد وجدنا أنه كلما زادت معرفتنا بالنطاق، كلما أصبحنا قادرين بشكل أفضل على تقديم اقتراحات بشأن الأشياء الأخرى التي يمكن القيام بها لمعالجة مشكلات العمل الأساسية. نعتقد اعتقادًا راسخًا أن المطورين، الذين يتعرضون للعديد من الجوانب المختلفة للمؤسسة، يمكنهم غالبًا رؤية طرق لنسج أجزاء مختلفة من العمل معًا والتي لا تكون واضحة دائمًا للإدارات الفردية.

نصيحة ٩٦	**أبهِج المستخدمين، لا تقدم شفرةً فقط.**

إذا كنت ترغب في إسعاد عملائك، فكوّن عَلاقة بهم بحيث يمكنك المساعدة بنشاط في حل مشكلاتهم. مع أنّ توصيفك كـ "مطور برمجيات" أو "مهندس برمجيات" قد يكون مختلف قليلا، فإنه في الحقيقة يجب أن تكون "حلال مشكلات". هذا ما نفعله، وهذا هو جوهر المبرمج العملي.

نحن نحلّ المشكلات.

الأقسام ذات الصلة

- الموضوع 12، الرصاصات الخطّاطة، في الصفحة 73
- الموضوع 13، النماذج الأولية والملاحظات الملحقة، في الصفحة 78
- الموضوع 45، هوة المتطلبات، في الصفحة 268

الموضوع 53. كبرياء وتحامل

لقد أبهجتنا لمدّة طويلة بما فيه الكفاية.

^ جين أوستن، كبرياء وتحامل

المبرمجون العمليون لا يتهربون من المسؤولية. بدلاً من ذلك، فنحن نفرح بقبول التحديات وبجعل خبرتنا معروفة جيدًا. إذا كنا مسؤولين عن تصميمٍ ما أو عن جزءٍ من الشِفرة، فإننا نُنجز عملًا نفخر به.

نصيحة ٩٧	وقّع على عملك.

كان المحترفين منذ القدم فخورين بالإمضاء على أعمالهم. يجب أن تكون أنت أيضا كذلك.

لا تزال فرق المشروع مكونة من أشخاص، ومع ذلك، يمكن أن تسبب هذه القاعدة مشكلات. ففي بعض المشروعات، يمكن أن تسبب فكرة مُلكية الشِفرة مشكلات في التعاون. قد يصبح الناس محليين(متعصبين)، أو غير راغبين في العمل على عناصر الأساس المشترك. قد ينتهي الأمر بالمشروع كمجموعة من الإقطاعيات الصغيرة المنعزلة. حيث تصبح متحيزًا لمصلحة الشِفرة الخاصة بك وضد زملائك في العمل.

ليس هذا ما نريده. يجب ألا تدافع بغيرة عن شفرتك ضد المتدخلين؛ على نفس المنوال، يجب أن تعامل شِفرة الأشخاص الآخرين باحترام. تعدّ القاعدة الذهبية ("تعامل مع الآخرين كما تحب أن يعاملوك") والأساس القائم على الاحترام المتبادل بين المطورين هو أمر بالغ الأهمية لإنجاح هذه القاعدة(النصيحة).

يمكن أن يوفر إخفاء الهُوِيَّة (Anonymity)، خاصة في المشروعات الكبيرة، أرضًا خصبة للإهمال والأخطاء والكسل والشِفرة السيئة. يصبح من السهل جدًا أن ترى نفسك مجرد ترس في آلة، تُنتج أعذارًا واهية في تقارير حالات لانهائية بدلاً من إنتاج شِفرة جيدة.

مع أنّ الشِفرة يجب أن تكون مملوكة، إلا أنها يجب ألا تكون مملوكة لفرد. في الواقع، توصي شركة Kent Beck's eXtreme Programming[157] بالملكية المشتركة للشِفرة (ولكن هذا يتطلب أيضًا ممارسات إضافيّة، مثل البرمجة الزوجيّة، للحماية من مخاطر إخفاء الهُوِيَّة).

نريد أن نرى فخر الملكية. "لقد كتبت هذا، وأنا واثق من عملي." يجب أن يتم التعرف على توقيعك كمؤشر للجودة. يجب أن يرى الأشخاص اسمك على جزء من الشِفرة ويتوقعوا أن يكون متينا ومكتوبًا جيدًا ومختبرًا وموثقًا. عمل احترافي حقًا. كتبه محترف.

مُبرمجٌ عملي.

شكرا لك.

157 http://www.extremeprogramming.org

الفصل العاشر:

مُلحق

على المدى الطويل، نحن نشكل حياتنا، ونشكل أنفسنا. لا تنتهي العملية أبدًا حتى نموت. والخيارات التي نتخذها هي مسؤوليتنا الخاصة في النهاية.

^ إليانور روزفلت

ملحق

في العشرين عامًا التي سبقت الإصدار الأول، كنا جزءًا من تطور الحاسوب من حب استطلاع سطحي إلى ضرورة حديثة للشركات. في العشرين عامًا منذ ذلك الحين، نمت البرمجيات إلى ما أبعد من مجرد آلات تجارية وسيطرت حقًا على العالم. لكن ماذا يعني ذلك لنا حقًا؟

في شهر الرجل الأسطوري:مقالات في تطوير البرمجيات The Mythical Man-Month: Essays on Software Engineering [Bro96]، قال فريد بروكس: "إن المبرمج، مثل الشاعر، لا يعمل إلا القليل من الأشياء الفكريّة الخالصة. إنه يبني قلاعه في الهواء، ومن الهواء، ويخلق بجهد الخيال". نبدأ بصفحة فارغة، ويمكننا إنشاء أي شيء تقريبًا يمكننا تخيله. والأشياء التي نصنعها يمكن أن تغير العالم.

من مساعدة تويتر (Twitter) للناس في التخطيط للثورات، إلى المعالج في سيارتك الذي يعمل على منعك من الانزلاق، إلى الهاتف الذكي مما يعني أننا لم نعد بحاجة إلى تذكر التفاصيل اليومية المزعجة، فبرامجنا موجودة في كل مكان. خيالنا في كل مكان.

نحن المطورين متميزون بشكل لا يصدق. نحن حقا نبني المستقبل. إنه مقدار غير عادي من القوة. ومع هذه القوة تأتي مسؤولية غير عادية.

كم مرّة نتوقف للتفكير في ذلك؟ كم مرّة نتناقش، سواء فيما بيننا أو مع جَمهور عام، ماذا يعني هذا؟

تستخدم الأجهزة المدمجة أجهزة حاسوب أكفأ من تلك المستخدمة في أجهزة الحاسوب المحمولة وأجهزة الحاسوب المكتبية ومراكز البيانات. غالبًا ما تتحكم أجهزة الحاسوب المدمجة هذه في الأنظمة الحيوية، من محطات الطاقة إلى السيارات إلى المُعِدّات الطبيّة. حتى نظام التحكم في التدفئة المركزية أو الأجهزة المنزلية البسيطة يمكن أن يقتل شخصًا ما إذا كان سيئ التصميم أو التنفيذ. عندما تعمل على تطوير هذه الأجهزة، فإنك تتحمل مسؤوليةً مذهلة. يمكن للعديد من الأنظمة غير المضمنة أيضًا أن تحدث ضررًا كبيرًا ونفعًا كبيرًا.

يمكن لوسائل التواصل الاجتماعي أن تروّج الثورة السلمية أو أن تثير الكراهيَة البشعة. يمكن للبيانات الضخمة أن تجعل التسوق أسهل، ويمكن أن تدمّر أي أثر للخصوصية قد تعتقد أنك تملك. تتخذ الأنظمة المصرفية قرارات بشأن القروض التي تغيّر حياة الناس. ويمكن استخدام أي نظام للتطفل على مستخدميه.

لقد رأينا تلميحات عن احتمالات المستقبل المثالي، وأمثلة على العواقب غير المقصودة التي تؤدي إلى كابوس العالم المرير "ديستوبيا" (nightmare dystopias)[158]. قد يكون الفرق بين النتيجتين أكثر دقة مما تعتقد. والأمر كله بين يديك.

158 **[المترجم]** هو مجتمع خيالي، فاسد أو مخيف أو غير مرغوب فيه بطريقة ما. وقد تعني الديستوبيا مجتمع غير فاضل تسوده الفوضى.

البوصلة الأخلاقية

ثمن هذه القوة غير المتوقعة هو اليقظة. تؤثر أفعالنا بشكلٍ مباشر على النّاس. لم يعد برنامَج الهوايات على وحدة المعالجة المركزية 8 بت في المرآب، أو عملية الأعمال المجمعة المعزولة على الحاسوب الرئيس في مركز البيانات، أو حتى الحاسوب المكتبي فقط ؛ برمجياتنا تنسج نسيج الحياة اليومية الحديثة.

من واجبنا أن نسأل أنفسنا سؤالين حول كل جزء نقدمه من الشِفرة:

1. هل حميت المستخدم؟
2. هل يمكنني استخدام هذا بنفسي؟

أولاً، يجب أن تسأل "هل بذلت قُصَارَى جهدي لحماية مستخدمي هذه الشِفرة من الأذى؟" هل وضعت أحكامًا لتطبيق تصحيحات الأمان المستمرة على جهاز مراقبة الأطفال البسيط هذا؟ هل تيقنّت من أنه ولو فشل منظم الحرارة التلقائي للتدفئة المركزية، فسيظل العميل يتمتع بالتحكم اليدوي؟ هل أخزن البيانات التي أحتاجها فقط، وتشفير أي شيء شخصي؟

لا أحد كامل؛ الجميع يفتقد الأشياء بين الحين والآخر. ولكن إذا كنت لا تستطيع أن تقول بصدق أنك حاولت سرد جميع العواقب، وتأكدّت من حماية المستخدمين منها، فأنت تتحمل بعض المسؤولية عندما تسوء الأمور.

نصيحة ٩٨	**لا تتسبب في أذى.**

ثانيًا، هناك حكم متعلق بالقاعدة الذهبية: هل يسعدني أن أكون مستخدمًا لهذه البرمجية؟ هل أريد مشاركة بياناتي؟ هل أرغب في تحويل تحركاتي إلى منافذ البيع بالتجزئة؟ هل أكون سعيدًا بنقلي بهذه السيارة ذاتية القيادة؟ هل أنا مرتاح لفعل ذلك؟

تبدأ بعض الأفكار المبتكرة في الالتفاف على حدود السلوك الأخلاقي، وإذا كنت مشتركًا في هذا المشروع، فأنت مسؤول تمامًا مثل الرعاة.

بغض النظر عن عدد درجات الانفصال التي يمكنك تبريرها، تظل إحدى القواعد صحيحة:

نصيحة ٩٩	**لا تمكن المخادعين.**

تخيل المستقبل الذي تريده

الأمر متروك لك. إن خيالك، وآمالك، ومخاوفك هي التي توفر الأشياء الفكرية الخالصة التي تبني العشرين عامًا القادمة وما بعدها.

أنتم تبنون المستقبل لأنفسكم ولأحفادكم. واجبكم أن تجعلوه مستقبلًا نرغب جميعًا في العيش فيه. كن مُدركا عندما تفعل شيئًا ضد هذا المثل الأعلى، وتحلَّ بالشجاعة لتقول "لا!" تخيّل المستقبل الذي يمكن أن يكون لدينا، وامتلك الشجاعة لخلقه. ابن القلاع في الهواء كل يوم.

حياتنا رائعة

نصيحة ١٠٠	**إنها حياتك.** **شاركها. احتفي بها. ابنها.** **واستمتع!**

الملحق أ:

قائمة المراجع

قائمة المراجع

[ADSS18] Vard Antinyan, Jesper Derehag, Anna Sandberg, and Miroslaw Staron. Mythical Unit Test Coverage. IEEE Software. 35:73-79, 2018.

[And10] Jackie Andrade. What does doodling do? Applied Cognitive Psychology. 24(1):100-106, 2010, January.

[Arm07] Joe Armstrong. Programming Erlang: Software for a Concurrent World. The Pragmatic Bookshelf, Raleigh, NC, 2007.

[BR89] Albert J. Bernstein and Sydney Craft Rozen. Dinosaur Brains: Dealing with All Those Impossible People at Work. John Wiley & Sons, New York, NY, 1989.

[Bro96] Frederick P. Brooks, Jr. The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley, Reading, MA, Anniversary, 1996.

[CN91] Brad J. Cox and Andrew J. Novobilski. Object-Oriented Programming: An Evolutionary Approach. Addison-Wesley, Reading, MA, Second, 1991.

[Con68] Melvin E. Conway. How do Committees Invent? Datamation. 14(5):28-31, 1968, April.

[de98] Gavin de Becker. The Gift of Fear: And Other Survival Signals That Protect Us from Violence. Dell Publishing, New York City, 1998.

[DL13] Tom DeMacro and Tim Lister. Peopleware: Productive Projects and Teams. Addison-Wesley, Boston, MA, Third, 2013.

[Fow00] Martin Fowler. UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley, Boston, MA, Second, 2000.

[Fow04] Martin Fowler. UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley, Boston, MA, Third, 2004.

[Fow19] Martin Fowler. Refactoring: Improving the Design of Existing Code. AddisonWesley, Boston, MA, Second, 2019.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley,Reading, MA, 1995.

[Hol92] Michael Holt. Math Puzzles & Games. Dorset House, New York, NY, 1992.

[Hun08] Andy Hunt. Pragmatic Thinking and Learning: Refactor Your Wetware. The Pragmatic Bookshelf, Raleigh, NC, 2008.

[Joi94] "T.E. Joiner. Contagious depression: Existence, specifcity to depressed symptoms, and the role of reassurance seeking. Journal of Personality and Social Psychology. 67(2):287–296, 1994, August."

[Knu11] Donald E. Knuth. The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1. Addison-Wesley, Boston, MA, 2011.

[Knu98] Donald E. Knuth. The Art of Computer Programming, Volume 1: Fundamental Algorithms. Addison-Wesley, Reading, MA, Third, 1998.

[Knu98a] Donald E. Knuth. The Art of Computer Programming, Volume 2: Seminu merical Algorithms. Addison-Wesley, Reading, MA, Third, 1998.

[Knu98b] Donald E. Knuth. The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley, Reading, MA, Second, 1998.

[KP99] Brian W. Kernighan and Rob Pike. The Practice of Programming. AddisonWesley, Reading, MA, 1999.

[Mey97] Bertrand Meyer. Object-Oriented Software Construction. Prentice Hall, Upper Saddle River, NJ, Second, 1997.

[Mul18] Jerry Z. Muller. The Tyranny of Metrics. Princeton University Press,

Princeton NJ, 2018.

[SF13] Robert Sedgewick and Phillipe Flajolet. An Introduction to the Analysis of Algorithms. Addison-Wesley, Boston, MA, Second, 2013.

[Str35] James Ridley Stroop. Studies of Interference in Serial Verbal Reactions. Journal of Experimental Psychology. 18:643–662, 1935.

[SW11] Robert Sedgewick and Kevin Wayne. Algorithms. Addison-Wesley, Boston, MA, Fourth, 2011.

[Tal10] Nassim Nicholas Taleb. The Black Swan: Second Edition: The Impact of the Highly Improbable. Random House, New York, NY, Second, 2010.

[WH82] "James Q. Wilson and George Helling. The police and neighborhood safety. The Atlantic Monthly. 249[3]:29–38, 1982, March."

[YC79] Edward Yourdon and Larry L. Constantine. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Prentice Hall, Englewood Cliffs, NJ, 1979.

[You95] Edward Yourdon. When good-enough software is best. IEEE Software. 1995, May.

الملحق ب:

الإجابات المُمكنة للتمارين

أفضّل أن يكون لدي أسئلة لا يمكن الإجابة عليها على إجابات لا يمكن السؤال عنها.
^ ريتشارد فاينمان

الإجابات الممكنة للتمارين

الإجابة 1 (من التمرين 1 في الصفحة 68)

اعتمادًا على طريقتنا في التفكير، فإن الصنف Split2 يُعدّ أكثر تعامديةً. فهو يركّز على مهمته الخاصّة، ويقسّم الأسطر، ويتجاهل التفاصيل مثل من أين تأتي هذه الأسطر. وذلك لا يؤدي إلى تسهيل تطوير الشِفرة فحسب، بل يجعلها أيضًا أكثر مرونة. يمكن لـ Split2 فصل الأسطر التي تُقرأ من مِلَفّ، أو المولَّدة بواسطة روتين آخر، أو المُمرّرة عبر البيئة.

الإجابة 2 (من التمرين 2 في الصفحة 68)

لنبدأ بتأكيد: يمكنك كتابة شِفرة جيدة ومتعامدة بأي لغة تقريبًا. في الوقت نفسه، لكل لغة إغراءاتها: ميزات يمكن أن تؤدي إلى زيادة الاقتران وتقليل التعامد.

في اللغات كائنية التوجه، توفر ميزات مثل الوراثة المتعددة والاستثناءات والتحميل الزائد للمشغل (operator overloading) والتحميل الزائد للطريقة الأصل (عبر التصنيف الفرعي) فرصة كبيرة لزيادة الاقتران بطرق غير واضحة. هناك أيضًا نوع من الاقتران لأن الصنف (class) يقرن الشِفرة بالبيانات. وعادة هذا شيء جيد (عندما يكون الاقتران جيدًا، نسميه تماسكا). ولكن إذا لم تجعل أصنافك مركزة بشكل كافٍ، فقد يؤدي ذلك إلى بعض الواجهات البشعة جدًا.

في اللغات الوظيفيّة، نشجعك على كتابة الكثير من الدوال الصغيرة المنفصلة والجمع بينها بطرق مختلفة لحل مشكلتك. يبدو هذا جيّدًا من الناحية النظرية. وفي الممارسة العملية غالبًا ما يكون كذلك. ولكن هناك شكل من أشكال الاقتران يمكن أن يحدث هنا أيضًا. تعمل هذه الدوال عادةً على تحويل البيانات، مما يعني أن نتيجة إحدى الدوال يمكن أن تصبح دخلًا للدالة الأخرى. إذا لم تكن حذرًا، فإن إجراء تغيير على تنسيق البيانات الذي تولّده إحدى هذه الدوال يمكن أن يؤدي إلى فشل في مكان ما أسفل التدفق التحويلي. يمكن للغات ذات أنظمة الكتابة الجيدة أن تساعد في التخفيف من ذلك.

الإجابة 3 (من التمرين 3 في الصفحة 81)

استخدم التِقَانَة المنخفضة كحل إسعافي! ارسم بعض الرسوم الكاريكاتورية بعلامات على السَّبُّورَة - سيارة وهاتف ومنزل. ليس بالضرورة أن يكون رسما مُتقنًا؛ خطوط (stick-figure)[159] جيدة. ضع ملاحظات لاصقة تصف محتويات الصفحات المستهدفة في المناطق القابلة للنقر. ومع تقدم الاجتماع، يمكنك تحسين الرُّسُوم وأماكن الملاحظات المُلصقة.

الإجابة 4 (من التمرين 4 في الصفحة 86)

نظرًا لأننا نريد جعل اللغة قابلة للتوسيع، فسنجعل المحلّل اللغوي مستندا إلى جدول (table driven)ـ يحتوي كل إدخال في الجدول على حرف الأمر، وعلم (flag) لتقول ما إذا كانت هناك حاجة إلى وسيط، واسم الروتين المطلوب استدعاؤه للتعامل مع هذا الأمر المحدد.

lang/turtle.c
```
typedef struct {
    char cmd; /* حرف الأمر */
    int hasArg; /* يأخذ وسيط */
    void (*func)(int, int); /* روتين للاستدعاء */
} Command;
static Command cmds[] = {
   { 'P', ARG, doSelectPen },
   { 'U', NO_ARG, doPenUp },
   { 'D', NO_ARG, doPenDown },
   { 'N', ARG, doPenDir },
   { 'E', ARG, doPenDir },
   { 'S', ARG, doPenDir },
   { 'W', ARG, doPenDir }
};
```

البرنامَج الرئيس بسيط جدًا: اقرأ سطرًا، وابحث عن الأمر، واحصل على الوسيط إذا لزم الأمر، ثم استدعي دالة المعالجة.

lang/turtle.c
```
while (fgets(buff, sizeof(buff), stdin)) {
   Command *cmd = findCommand(*buff);
     if (cmd) {
         int arg = 0;
         if (cmd->hasArg && !getArg(buff+1, &arg)) {
         fprintf(stderr, "'%c' needs an argument\n", *buff);
             continue;
       }
       cmd->func(*buff, arg);
   }
}
```

159 **[المترجم]** stick-figure هو رسم بسيط للغاية لشخص أو لحيوان أو ما إلى ذلك، ويتألف من بضعة خطوط ومنحنيات ونِقَاط. يتم تمثيل الرأس بدائرة، مزينة أحيانًا بتفاصيل مثل العينين أو الفم أو الشعر. عادة ما يتم تمثيل الذراعين والساقين والجذع بخطوط مستقيمة. قد تكون التفاصيل مثل اليدين والقدمين والرقبة موجودة أو غير موجودة. تم العثور على رُسُوم stick-figure عبر التاريخ، وغالبًا ما يتم نحتها بأداة حادة على الأسطح الصلبة مثل الحجر أو الجدران الخرسانية. غالبًا ما تستخدم الأشكال اللاصقة في الرُّسُوم التخطيطية للقصص المصورة للأفلام.

تقوم الدالة التي تبحث عن أمر بإجراء بحث خطي للجدول، مع إرجاع الإدخال المطابق أو NULL.

lang/turtle.c
```
Command *findCommand(int cmd) {
    int i;
    for (i = 0; i < ARRAY_SIZE(cmds); i++) {
        if (cmds[i].cmd == cmd)
        return cmds + i;
    }
    fprintf(stderr, "Unknown command '%c'\n", cmd);
     return 0;
}
```

أخيرًا، قراءة المعادلة الرَقَمية بسيطة جدًا باستخدام sscanf.

lang/turtle.c
```
int getArg(const char *buff, int *result) {
   return sscanf(buff, "%d", result) == 1;
}
```

الإجابة 5 (من التمرين 5 في الصفحة 86)

في الواقع، لقد قمت فعلًا بحل هذه المشكلة في التمرين السابق، حيث كتبت مترجمًا للغة الخارجية، سيحتوي على مفسر (interpreter) داخلي. في عيّنة الشِفرة خاصتنا، إنها دالات doXxx.

الإجابة 6 (من التمرين 6 في الصفحة 87)

باستخدام BNF، يمكن أن يكون تحديد الوقت

time ::= *hour ampm* | *hour* : *minute ampm* | *hour* : *minute*
ampm ::= am | pm
hour ::= *digit* | *digit digit*
minute ::= *digit digit*
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

سيأخذ تحديد الساعة والدقيقة بشكل أفضل في الاعتبار أن الساعة يمكن أن تكون فقط من 00 إلى 23، ودقيقة من 00 إلى 59:

hour ::= h-tens digit | digit

minute ::= m-tens digit

h-tens ::= 0 | 1

m-tens ::= 0 | 1 | 2 | 3 | 4 | 5

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

الإجابة 7 (من التمرين 7 في الصفحة 87)

إليك المحلل اللغوي المكتوب باستخدام مكتبة Pegjs لجافا سكريبت:

lang/peg_parser/time_parser.pegjs
```
time
   = h:hour offset:ampm { return h + offset }
   / h:hour ":" m:minute offset:ampm { return h + m + offset }
   / h:hour ":" m:minute { return h + m }
ampm
   = "am" { return 0 }
   / "pm" { return 12*60 }
hour
   = h:two_hour_digits { return h*60 }
   / h:digit { return h*60 }
minute
   = d1:[0-5] d2:[0-9] { return parseInt(d1+d2, 10); }
digit
   = digit:[0-9] { return parseInt(digit, 10); }
two_hour_digits
   = d1:[01] d2:[0-9 ] { return parseInt(d1+d2, 10); }
   / d1:[2] d2:[0-3] { return parseInt(d1+d2, 10); }
```

تظهر الاختبارات أنه قيد الاستخدام:

lang/peg_parser/test_time_parser.js
```
let test = require('tape');
let time_parser = require('./time_parser.js');
// time ::= hour ampm |
// hour : minute ampm |
// hour : minute
//
// ampm ::= am | pm
//
// hour ::= digit | digit digit
//
// minute ::= digit digit
//
// digit ::= 0 |1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
const h = (val) => val*60;
const m = (val) => val;
const am = (val) => val;
const pm = (val) => val + h(12);
let tests = {
  "1am": h(1),
  "1pm": pm(h(1)),
  "2:30": h(2) + m(30),
  "14:30": pm(h(2)) + m(30),
  "2:30pm": pm(h(2)) + m(30),
}
test('time parsing', function (t) {
      for (const string in tests) {
          let result = time_parser.parse(string)
        t.equal(result, tests[string], string);
    }
t.end()
});
```

الإجابة 8 (من التمرين 8 في الصفحة 87)

إليك أحد الحلول الممكنة في روبي (Ruby):تم

lang/re_parser/time_parser.rb
```
TIME_RE = %r{
```

```
(?<digit>[0-9]){0}
(?<h_ten>[0-1]){0}
(?<m_ten>[0-6]){0}
(?<ampm> am | pm){0}
(?<hour> (\g<h_ten> \g<digit>) | \g<digit>){0}
(?<minute> \g<m_ten> \g<digit>){0}
\A(
( \g<hour> \g<ampm> )
| ( \g<hour> : \g<minute> \g<ampm> )
| ( \g<hour> : \g<minute> )
)\Z
}x
def parse_time(string)
  result = TIME_RE.match(string)
   if result
     result[:hour].to_i * 60 +
     (result[:minute] || "0").to_i +
     (result[:ampm] == "pm" ? 12*60 : 0)
   end
end
```

(تستخدم هذه الشِفرة خدعة تحديد الأنماط المسماة في بداية التعبير النمطي (regular expression)، ثم الإشارة إليها على أنها أنماط فرعية في المطابقة الفعلية.)

الإجابة 9 (من التمرين 9 في الصفحة 92)

يجب صياغة إجابتنا في عدة افتراضات:

- يحتوي جهاز التخزين على المعلومات التي نحتاج إلى نقلها.
- نحن نعرف السرعة التي يمشي بها الشخص.
- نعرف المسافة بين جهازي الحاسوب.
- نحن لا نحسب الوقت المستغرق لنقل المعلومات من وإلى جهاز التخزين.
- النفقات العامة لتخزين البيانات تساوي تقريبًا النفقات العامة لإرسالها عبر خط اتصالات.

الإجابة 10 (من التمرين 10 في الصفحة 92)

مع مراعاة التحذيرات الواردة في الإجابة السابقة: يحتوي الشريط 1 تيرابايت على 2^{40} 8X، أو 2^{43} بت، لذلك سيتعين على خط 1 جيجابت في الثانية ضخ البيانات لحوالي 9000 ثانية، أو ما يقرب من ساعتين ونصف الساعة، لنقل المقدار المكافئ من المعلومات. إذا كان الشخص يسير بسرعة ثابتة 3 ميل في الساعة، فيجب أن يكون الجهازان لدينا على بعد 9 أميال تقريبًا عن بعضهم البعض حتى يتفوق خط الاتصالات على الناقل الخاص بنا. خلاف ذلك، يفوز الشخص.

الإجابة 14 (من التمرين 14 في الصفحة 132)

سنعرض تواقيع الدوال في جافا، مع الشروط المسبقة واللاحقة في التعليقات.

أولاً: الثابت (invariant) بالنسبة للصنف:

```
/**
  * @invariant getSpeed() > 0
  * implies isFull() // ليس فارغا
  *
  * @invariant getSpeed() >= 0 &&
  * getSpeed() < 10 // فحص المجال
  */
```

بعد ذلك، الشروط المسبقة واللاحقة:

```
/**
  * @pre Math.abs(getSpeed() - x) <= 1 // تغيير فقط بمقدار واحد
  * @pre x >= 0 && x < 10 //  فحص المجال
  * @post getSpeed() == x // السرعة المطلوبة
  */
public void setSpeed(final int x)
/**
  * @pre !isFull() // لا تملأه مرتين
  * @post isFull() // تأكد أنه تم
  */
void fill()
/**
  * @pre isFull() // لا تملأه مرتين
  * @post !isFull() // تاكد انه تم
  */
void empty()
```

الإجابة 15 (من التمرين 15 في الصفحة 132)

هناك 21 مفردة في هذه السلسلة. إذا قلت 20، فقد واجهت للتو خطأ السياج " fencepost error " (لا تعرف ما إذا كنت ستحسب نِقَاط السياج "الحدود" أو المجالات بينها).

الإجابة 16 (من التمرين 16 في الصفحة 137)

- شهر أيلول عام 1752 كان به 19 يومًا فقط. تم القيام بذلك لمزامنة التقويمات كجزء من الإصلاح الغريغوري.
- ربما أزيل الدليل بواسطة عملية أخرى، وقد لا يكون لديك إذن بقراءته، وقد لا يكون محرك الأقراص مثبتًا،...؛ أصبحت الصورة واضحة.
- لم نحدّد بشكل سري نوعي a و b. قد يكون التحميل الزائد على المشغل قد عرّف + أو = أو! = ليكون له سلوك غير متوقع. أيضًا، قد يكون a و b اسمين مستعارين لنفس المتغير، لذا فإن الإسناد الثاني سيحل محل القيمة المخزنة في الأول. أيضًا، إذا كان البرنامج متزامنًا ومكتوبًا بشكل سيئ، فربما يكون قد تم تحديثه بحلول وقت حدوث الإضافة.
- في الهندسة غير الأقليدية، لن يكون مجموع زوايا المثلث 180 درجة. فكّر في مثلث مرسوم على سطح كرة.

- قد تحتوي الدقائق الكبيسة على 61 أو 62 ثانية.
- اعتمادًا على اللغة، قد يترك التحميل العددي الزائد النتيجة a+1 سالبة.

الإجابة 17 (من التمرين 17 في الصفحة 145)

في معظم تطبيقات C و C++، لا توجد طريقة للتحقق من أن المؤشر يشير فعلًا إلى ذاكرة صالحة. من الأخطاء الشائعة إلغاء تخصيص كتلة من الذاكرة ثم الإشارة إلى تلك الذاكرة لاحقًا في البرنامج. بحلول ذلك الوقت، ربما تم إعادة تخصيص الذاكرة المشار إليها لغرض آخر. فبواسطة تعيين المؤشر إلى NULL، يأمل المبرمجون في منع هذه المراجع السائبة - في معظم الحالات، سيؤدي إلغاء الإشارة إلى مؤشر NULL إلى حدوث خطأ وقت التشغيل.

الإجابة 18 (من التمرين 18 في الصفحة 145)

بواسطة تعيين المرجع إلى NULL، يمكنك تقليل عدد المؤشرات إلى الكائن المشار إليه بمقدار واحد. بمجرد أن يصل هذا العدد إلى الصفر، يصبح الكائن مؤهلاً لعملية جمع البيانات المهملة. يمكن أن يكون تعيين المراجع إلى NULL أمرًا مهمًا للبرامج التي تعمل لمدة طويلة، حيث يحتاج المبرمجون إلى ضمان عدم زيادة استخدام الذاكرة بمرور الوقت.

الإجابة 19 (من التمرين 19 في الصفحة 167)

يمكن أن يكون التنفيذ البسيط لذلك هو:

event/strings_ex_1.rb

```
class FSM
  def initialize(transitions, initial_state)
    @transitions = transitions
    @state = initial_state
  end
  def accept(event)
    @state, action = TRANSITIONS[@state][event] || TRANSITIONS[@state][:default]
  end
end
```

(نزّل هذا المِلَفّ للحصول على الشِفرة المحدثة التي تستخدم صنف FSM الجديد.)

الإجابة 20 (من التمرين 20 في الصفحة 168)

- ... تعطل ثلاثة أحداث لواجهة الشبكة في غضون خمس دقائق

يمكن تنفيذ ذلك باستخدام آلة الحالة، ولكنه سيكون أكثر تعقيدًا مما قد يظهر أول مرة: إذا حصلت على أحداث في الدقائق 1 و 4 و 7 و 8، فيجب إليك تشغيل التحذير في الحدث الرابع، مما يعني أن آلة الحالة يجب أن تكون قادرة على التعامل مع إعادة ضبط نفسها.

لهذا السبب، يبدو أن تدفقات الأحداث هي التِقَانَة المفضلة. هناك دالة تفاعلية تسمى المخزن المؤقت (buffer) مع معاملات الحجم والإزاحة والتي تتيح لك إرجاع كل مجموعة من ثلاثة أحداث واردة. يمكنك بعد ذلك إلقاء نظرة على الطوابع الزمنية للحدث الأول والأخير في مجموعة لتحديد ما إذا كان يجب تشغيل الإنذار.

- **... الوقت بعد غروب الشمس، وتُكتشف هناك حركة في أسفل الدرج تليها حركة تُكتشف في أعلى الدرج ...**

من المحتمل أن يُنفّذ ذلك باستخدام مزيج من pubsub وآلات الحالة. يمكنك استخدام pubsub لنشر الأحداث على أي عدد من آلات الحالة، ومن ثم جعل آلات الحالة تحدد ما يجب القيام به.

- **... أخطر أنظمة التقارير المختلفة باكتمال الطلب.**

ربما يكون من الأفضل التعامل مع هذا باستخدام pubsub. قد ترغب في استخدام التدفقات، ولكن هذا يتطلب أن تكون الأنظمة التي يتم إخطارها تعتمد أيضًا على التدفق.

- **... ثلاث خِدْمَات خلفية وانتظار الردود.**

هذا مشابه لمثالنا الذي استخدم التدفقات لجلب بيانات المستخدم.

الإجابة 21 (من التمرين 21 في الصفحة 179)

1. تضاف ضريبة الشحن والمبيعات إلى الطلب:

   الطلب البدائي ← الطلب النهائي

   في الشِفرة التقليدية، من المحتمل أن يكون لديك دالة تحسب ثمن الشحن وأخرى تحسب الضريبة. لكننا نفكر في التحولات هنا، لذلك نحول الطلب الذي يحتوي على عناصر فقط إلى نوع جديد من الأشياء: طلب يمكن شحنه.

2. يقوم التطبيق بتحميل معلومات الضبط من مِلَفّ معيّن:

   اسم المِلَفّ ← بنية الضبط

3. يقوم شخص ما بتسجيل الدخول إلى تطبيق ويب:

   بيانات اعتماد المستخدم ← الجَلسة

الإجابة 22 (من التمرين 22 في الصفحة 179)

التحول عالي المستوى:

```
field contents as string
    → [validate & convert]
        → {:ok, value} | {:error, reason}
```

يمكن تقسيمه إلى:

```
field contents as string
    → [convert string to integer]
    → [check value >= 18]
    → [check value <= 150]
        → {:ok, value} | {:error, reason}
```

هذا يفترض أن لديك مسار تنفيذ (pipeline) لمعالجة الأخطاء.

الإجابة 23 (من التمرين 23 في الصفحة 179)

دعنا نجيب عن الجزء الثاني أولاً: نحن نفضل الشِفرة الأولى.

في الشِفرة الثانية، تقوم كل خطوة بإرجاع كائن ينفذ الدالة التالية التي نسميها: يجب أن يقوم الكائن المُرجع بواسطة content_of بتنفيذ find_matching_lines، وهكذا.

هذا يعني أن الكائن المُرجع بواسطة content_of مقترن بشفرتنا. تخيل أن المتطلبات تغيرت، وعلينا تجاهل الأسطر التي تبدأ بحرف . في أسلوب التحويل، سيكون ذلك سهلاً:

```
const content = File.read(file_name);
const no_comments = remove_comments(content)
const lines = find_matching_lines(no_comments, pattern)
const result = truncate_lines(lines)
```

يمكننا حتى تبديل ترتيب remove_comments و find_matching_lines وسيظل يعمل.

لكن في النمط المتسلسل، سيكون هذا أكثر صعوبة. أين يجب أن تستقر الطريقة remove_comments الخاصة بنا: في الكائن المُرجع بواسطة content_of أو الكائن المُرجع بواسطة find_matching_lines؟ وما هي الشِفرة الأخرى التي سنكسرها إذا غيرنا هذا الكائن؟ هذا الاقتران هو سبب تسمية نمط طريقة التسلسل أحيانًا بحطام القطار.

الإجابة 24 (من التمرين 24 في الصفحة 214)

معالجة الصورة.

لجدولة بسيطة لحمل العمل بين العمليات المتوازية، قد تكون رتل انتظار العمل المشترك (queue) أكثر من كافية. قد ترغب في التفكير في نظام السَّبُّورَة (blackboard)ـ إذا كان هناك ملاحظات مُضمنة - أي إذا كانت نتائج جزء تمت معالجته تؤثر على أجزاء أخرى، كما هو الحال في تطبيقات رؤية الآلة، أو تحويلات التفاف الصورة ثلاثية الأبعاد المعقدة.

تقويم المجموعة

قد يكون هذا مناسبا. يمكنك نشر الاجتماعات المجدولة وإمكانية التوافر على السَّبُّورَة. لديك كيانات تعمل بشكل مستقل، والتغذية الراجعة من القرارات مهمة، وقد يأتي المشاركون ويذهبون.

قد ترغب في التفكير في تقسيم هذا النوع من نظام السَّبُّورَة اعتمادًا على الشخص الذي يبحث: قد يهتم الموظفون المبتدئون بالمكتب المباشر فقط، وقد ترغب الموارد البشرية فقط في المكاتب الناطقة باللغة الإنجليزية في جميع أنحاء العالم، وقد يرغب المدير التنفيذي في كل شيء.

هناك أيضًا بعض المرونة في تنسيقات البيانات: فنحن أحرار في تجاهل التنسيقات أو اللغات التي لا نفهمها. علينا أن نفهم الصيغ المختلفة فقط لتلك المكاتب التي لها اجتماعات مع بعضها البعض، ولسنا بحاجة إلى تعريض جميع المشاركين لجميع التنسيقات الممكنة. يقلل هذا من الاقتران إلى المكان الضروري فقط، ولا يقيدنا بشكل مصطنع.

أداة مراقبة الشبكة

هذا مشابه جدًا لبرنامَج طلب الرهن العقاري / القرض في الصفحة 213. لقد تلقيت تقارير المشكلات من قبل المستخدمين والإحصاءات التي يتم الإبلاغ عنها تلقائيًا، وكلها تُنشر على السَّبُّورَة. يمكن لعامل بشري أو وكيل برمجيات تحليل السَّبُّورَة لتشخيص أعطال الشبكة: قد يكون الخطأان الموجودين على السطر مجرد شيء طبيعي، ولكن اذا كانوا 0 ألف خطأ فستواجه مشكلة في الأجهزة. تمامًا مثلما يحل المحققون لغز القتل، يمكن أن يكون لديك كيانات متعددة تحلل وتساهم بأفكار لحل مشكلات الشبكة.

الإجابة 25 (من التمرين 25 في الصفحة 227)

الافتراض مع قائمة أزواج *القيمة والمفتاح* هو أن المفتاح فريد من نوعه، وعادةً ما تفرض مكتبات التجزئة ذلك إما بسلوك التجزئة نفسها أو برسائل خطأ صريحة للمفاتيح المكررة. ومع ذلك، لا تحتوي المصفوفة عادةً على هذه القيود، وسوف تخزن بسعادة مفاتيح مكرّرة ما لم تكتب شِفرة خاصة لمنع ذلك. لذلك في هذه الحالة، تم العثور على المفتاح الأول الذي يطابق DepositAccount يُقبل، وأية إدخالات مطابقة أخرى سيتم تجاهلها. ولأن ترتيب الإدخالات غير مضمون، لذا فهو يعمل أحيانًا وأحيانًا لا يعمل.

وماذا عن اختلاف الآلية من التطوير إلى الإنتاج؟

إنها مجرد مصادفة.

الإجابة 26 (من التمرين 26 في الصفحة 227)

الحقيقة أن حقل رَقَمي بحت يعمل في الولايات المتحدة وكندا ومنطقة البحر الكاريبي هي مصادفة. وفقًا لمواصفات الاتحاد الدُّوَليّ للاتصالات (ITU)، يبدأ تنسيق المكالمة الدولية بعلامة + حرفية. يتم استخدام المحرف أيضًا في بعض المناطق، وبشكل أكثر شيوعًا، يمكن أن تكون الأصفار البادئة جزءًا من الرَقَم. لا تقم أبدًا بتخزين رَقَم هاتف في حقل رَقَمي.

الإجابة 27 (من التمرين 27 في الصفحة 315)

استنادا إلى مكان وجودك. في الولايات المتحدة، تعتمد مقاييس الحجم على الجالون، وهو حجم أسطوانة يبلغ ارتفاعها 6 بوصات وقطرها 7 بوصات، مقربًا إلى أقرب بوصة مكعبة.

في كندا، "كوب واحد" في الوصفة يمكن أن يعني أيًا من

- 1/5 كوارت إمبراطوري، أو 227 مل
- 1/4 كوارت أمريكي، أو 236 مل
- 16 ملعقة طعام مترية، أو 240 مل
- ربع لتر أو 250 مل

ما لم تكن تتحدث عن قدر طهي الأرز، في هذه الحالة "كوب واحد" يساوي 180 مل. هذا مشتق من koku، وهو الحجم المقدر للأرز الجاف المطلوب لإطعام شخص واحد لمدة عام واحد: على ما يبدو، حوالي 180 لترًا. أكواب طبخ الأرز هي 1 gō، أي 1/1000 من koku. وهي، كَمِّيَّة الأرز التي يتناولها الشخص تقريبًا في الوجبة الواحدة.[160]

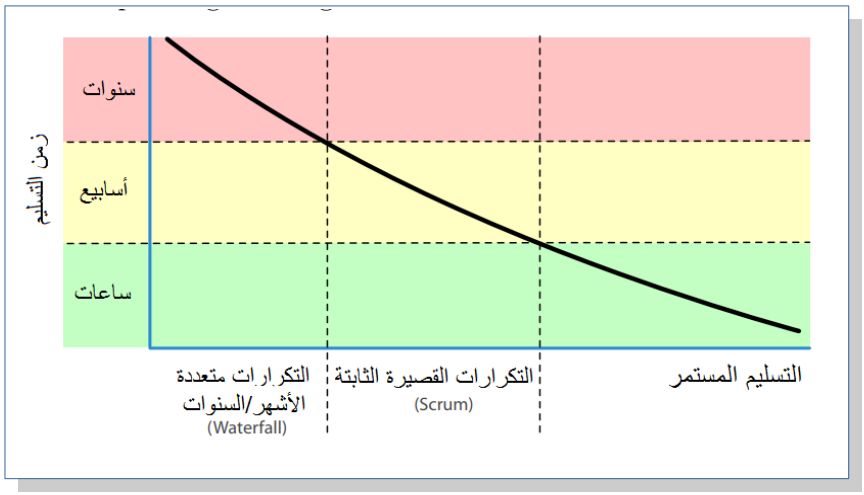
الإجابة 28 (من التمرين 28 في الصفحة 315)

من الواضح أنه لا يمكننا إعطاء أي إجابات مطلقة على هذا التمرين. ومع ذلك، يمكننا أن نقدم لك بعض المؤشرات.

إذا وجدت أن نتائجك لا تتبع منحنىً سلسًا، فقد ترغب في التحقق لمعرفة ما إذا كان هناك نشاط آخر يستخدم بعضًا من قوة معالجك. ربما لن تحصل أرقام جيدة إذا كانت العمليات التي تتم في الخلفية تنتزع دورات معالجة من برامجك بشكل دوري. قد ترغب أيضًا في التحقق من الذاكرة: إذا بدأ التطبيق في استخدام مساحة التبديل (swap)، فسيتراجع الأداء.

فيما يلي رسم بياني لنتائج تشغيل الشِفرة على أحد أجهزتنا:

160 شكرًا على هذا الجزء من المعلومات العامة يذهب إليه آفي براينت (avibryant)



الإجابة 29 (من التمرين 29 في الصفحة 315)

هناك طريقتان للوصول إلى هناك. الأول هو قلب المشكلة رأسًا على عقب. إذا كانت المصفوفة تحتوي على عنصر واحد فقط، فإننا لا نكررها حول الحلقة. كل تكرار إضافي يضاعف حجم المصفوفة التي يمكننا البحث عنها. إذن، الصيغة العامة لحجم المصـفوفة هي $n = 2^m$، حيث m هو عدد التكرارات. إذا أخذت اللوغاريتم إلى القاعدة 2 لكل جانب، فستحصل على $\lg n = \lg 2^m$، والتي تصبح وفقًا لتعريف اللوغاريتم $\lg n = m$.

الإجابة 30 (من التمرين 30 في الصفحة 315)

من المحتمل أن يكون هذا كثيرًا من الرجوع (flashback) إلى رياضيات المدرسة الثانويـة، لكن الصـيغة لتحويـل لوغـاريتم في الأساس a إلى واحد في الأساس b هي:

$$\log_b x = \frac{\log_a x}{\log_a b}$$

ونظرًا لأن $\log_a b$ ثابت، فيمكننا تجاهله داخل نتيجة Big-O.

الإجابة 31 (من التمرين 31 في الصفحة 315)

إحدى الخصائص التي يمكننا اختبارها هي نجـاح الطلب إذا كـان المسـتودع يحتـوي على عناصـر كافيـة في متنـاول اليـد. يمكننـا إنشاء طلبات لكميات عشوائية من العناصر، والتحقق من إرجاع قائمة "موافق" إذا كان المستودع يحتوي على مخزون.

الإجابة 32 (من التمرين 32 في الصفحة 315)

هذا استخدام جيد للاختبار القائم على الخاصية. يمكن أن تركز اختبارات الوحدة على الحالات الفردية التي توصلت فيها إلى النتيجة ببعض الوسائل الأخرى، ويمكن أن تركز اختبارات الخصائص على أشياء مثل:

- هل يتداخل أي صندوقين؟
- هل يتجاوز أي جزء من أي صندوق عرض أو طول الشاحنة؟
- هل كثافة التعبئة (المساحة المستخدمة من قبل الصناديق مقسومة على مساحة صندوق الشاحنة) أقل من أو تساوي 1؟
- إذا كان ذلك جزءًا من المطلب، فهل تتجاوز كثافة التعبئة الحد الأدنى للكثافة المقبولة؟

الإجابة 33 (من التمرين 33 في الصفحة 275)

1. هذا البيان يبدو وكأنه مطلب حقيقي: قد تكون هناك قيود على التطبيق من قبل بيئته.
2. هذا البيان في حد ذاته ليس متطلبا حقيقيًا. لكن لمعرفة ما المطلوب حقًا، عليك أن تسأل السؤال السحري، "لماذا؟" قد يكون هذا معيارًا مشتركًا، وفي هذه الحالة يجب أن تكون المتطلبات الفعلية شيئًا مثل "يجب أن تتوافق جميع عناصر واجهة المستخدم مع معايير واجهة المستخدم MegaCorp- V12.76 ،.

قد يكون هذا هو اللون الذي يحبه فريق التصميم. في هذه الحالة، يجب أن تفكر في الطريقة التي يحب بها فريق التصميم أيضًا تغيير رأيهم، وصياغة المتطلبات على أنها "يجب أن يكون لون الخلفية لجميع النوافذ المشروطة[161] قابلاً للضبط. عند الشحن، سيكون اللون رماديًا ". والأفضل من ذلك هو العبارة الأوسع نطاقًا "يجب أن تكون جميع العناصر المرئية للتطبيق (الألوان والخطوط واللغات) قابلة للضبط".

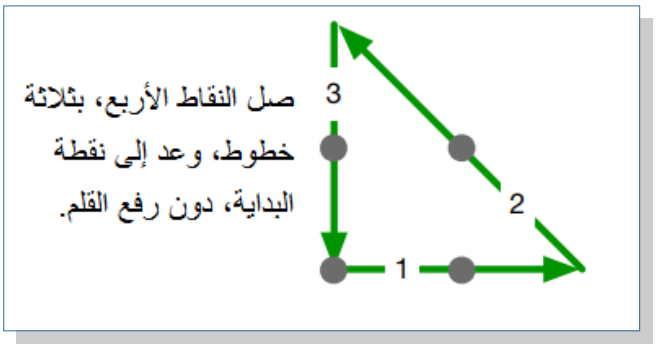
أو قد يعني ذلك ببساطة أن المستخدم يحتاج إلى أن يكون قادرًا على التمييز بين النوافذ المشروطة وغير المشروطة. إذا كان الحال كذلك، فسيكون هناك حاجة إلى مزيد من المناقشات.

3. هذا البيان ليس متطلبًا، إنه هيكل. عندما تواجه شيئًا كهذا، عليك أن تبحث بعمق لمعرفة ما يفكر فيه المستخدم. هل هذه قضية توسع؟ أم أداء؟ كلفة؟ أمان؟ الإجابات ستعلم عن التصميم الخاص بك.

161 [المترجم] في تصميم واجهة المستخدم لتطبيقات الحاسوب، إن النافذة المشروطة هي عنصر تحكم رسومي تابع للنافذة الرئيسة للتطبيق. ينشئ وضع يعطل النافذة الرئيسة ولكنه يبقيها مرئية، مع وجود نافذة مشروطة كنافذة فرعية أمامها. يجب على المستخدمين التفاعل مع النافذة المشروطة قبل أن يتمكنوا من العودة إلى التطبيق الأصلي. هذا يتجنب مقاطعة سير العمل على النافذة الرئيسة. تسمى الإطارات المشروطة أحيانًا النوافذ الثقيلة أو مربعات الحوار المشروطة لأنها تعرض غالبًا مربع حوار.

4. من المحتمل أن يكون المتطلب الأساسي أقرب إلى "سيمنع النظام المستخدم من إدخال إدخالات غير صالحة في الحقول، وسيحذّر المستخدم عند إجراء هذه الإدخالات".
5. من المحتمل أن يكون هذا البيان متطلبًا صعبًا، استنادًا إلى بعض قيود الأجهزة.

وإليك حل لمسألة النِقَاط الأربع:



مسرد المصطلحات

مسرد المصطلحات

تسلسل	المصطلح بالإنجليزيّة	تَرْجَمَة أولى	تَرْجَمَة ثانية
.1	Action	إجراء	
.2	Activity Diagram	مخطط النشاط	
.3	Actor Model	نموذج الممثل	نموذج الفاعل
.4	Anonymity	إخفاء الهوية	
.5	Approaches	نهج	
.6	Architecture	هيكلية	
.7	Argument	وسيط، وسطاء	مُعطى
.8	Asynchronous	غير متزامن	
.9	Attribute	سِمة	خاصيّة
.10	Blackboard	سَبُّورَة	لوح
.11	Chains	سلسلة	
.12	Characters	محرف	
.13	Class	صنف	
.14	Client	عميل	زَّبُون
.15	Code	شِفرة	تعليمات برمجيّة
.16	Compiler	مترجم	مصنّف
.17	Concurrency	تساير	
.18	Configurations	ضبط	تكوين
.19	Consequences	عواقب	
.20	Construction	بناء	
.21	Crash	تحطيم	تعطيل
.22	Customer	عميل	زَّبُون
.23	Data	بيانات	
.24	Debug	تصحيح	تنقيح
.25	Dependencies	تبعيات	

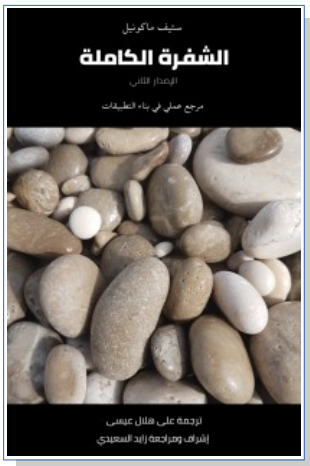
.26	Domain	نطاق	
.27	End-To-End Connection	اتصال نهاية لنهاية	اتصال كامل
.28	Entropy	اعتلاج، اضطراب	انتروبيا
.29	Event	حدث	
.30	Extensions	امتدادات	ملحقات
.31	Extreme Programming	البرمجة الفائقة	البرمجة القصوى
.32	Factor	عامل، عوامل	
.33	Feedback	انطباعـــات، ردود أفعـــال، ملاحظات	تغذية راجعة
.34	Fibers	ألياف	
.35	Framework	إطار عمل	
.36	Function	دالّة	
.37	Global	عام، عالمي	عموميّ
.38	Hardware	كيان مادي	
.39	Implementation	التنفيذ	التطبيق
.40	Information	معلومات	
.41	Instance	حالة	مثيل
.42	Integrity	تكامل	دمج
.43	Leverage	فاعليّة	
.44	Logging	تسجيل	أرشفة
.45	Macro	ماكرو	مُسجّل
.46	Markup	ترميز	تأشير
.47	Method	طريقة، طرائق	
.48	Mob Programming	البرمجة الجماعية	البرمجة الغوغائية
.49	Model	نموذج	
.50	Modularity	قابلية التركيب	
.51	Module	وحدة نمطيّة	وحدة
.52	Notation	تدوين	

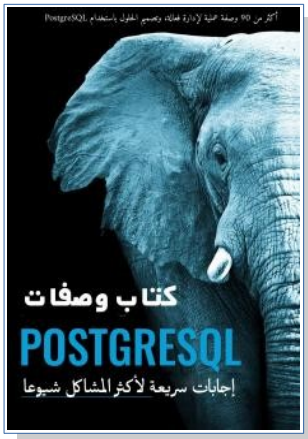
.53	Operator	معامل تشغيل	مُشّغل
.54	Pair Programming	البرمجة الزوجية	البرمجة بالأزواج
.55	Parallelism	التوازي	
.56	Parameters	مُعامِلات	محدّدات
.57	Parser	محلّل لغوي	
.58	Patterns	أنماط	
.59	Pipeline	مسار تنفيذ	خط أنابيب
.60	Plain	واضح	بسيط
.61	Platform	منصّة	
.62	Plugin	إضافة	مكوّن إضافي
.63	Pragmatic	عملي	واقعي
.64	Prerequisites	الشروط المُسبقة	
.65	Procedure	إجراء، إجراءات	
.66	Property	خاصيّة	
.67	Prototype	نموذج أولي	نسخة أوليّة
.68	Protocol	مراسم	بروتوكول
.69	Regular Expression	التعبير النظامي	التعبير النمطي
.70	Routine	روتين	إجرائية
.71	Scope	مجال	
.72	Semaphores	إشارات	
.73	Shared State	الحالة المشتركة	
.74	Statement	تعليمة، بيان	عبارة
.75	Stream	دفق	تدفق
.76	String	سلسلة محرفيّة	سلسلة نصيّة
.77	Structure	بنية	
.78	Style	أسلوب	نمط
.79	Synchronization	التزامن	
.80	Syntax	بناء الجملة	صياغة نحويّة
.81	Technical	تقني	

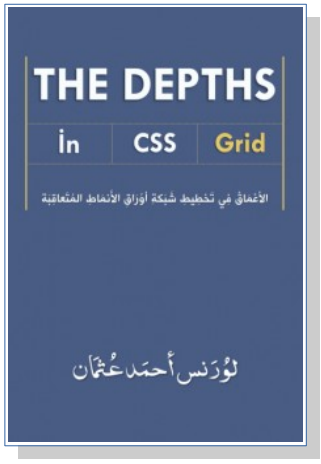
.82	Temporal coupling	الاقتران الزمني	
.83	Terminal	طرفيّة	جهاز طرفي
.84	Third-Party Tools	أدوات طرف ثالث	أدوات خارجية
.85	Thread	مسار	تشعّب
.86	Tip	نصيحة	تلميح
.87	Topic	موضوع	باب
.88	Traffic	حركة مرور البيانات	حِمل
.89	Transaction	مناقلات	مُعامَلات
.90	Transformation	تحويل	
.91	Tuples	قوائم	
.92	Unit	وحدة	
.93	Validation	التأكد الصَّلاحِيَة	
.94	Verification	التحقّق	

كتب وادي التقنية

تفضل بزيارة مكتبة وادي التقنية للاطلاع على الكثير من الكتب التقنية المجانية

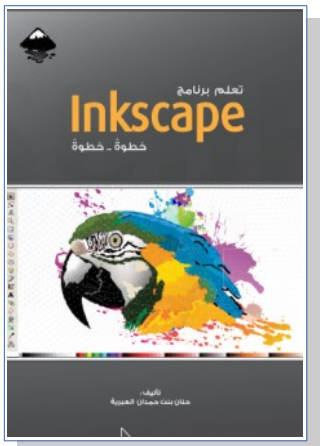




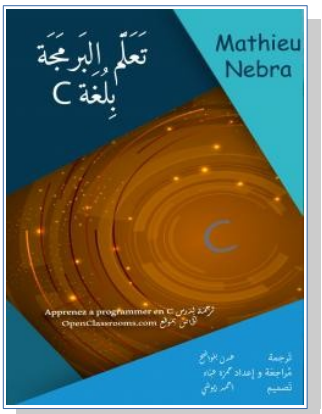

THE DEPTHS
In CSS Grid
الأعماق في تخطيط شبكة أوراق الأنماط المتعاقبة
لورنس أحمد عثمان

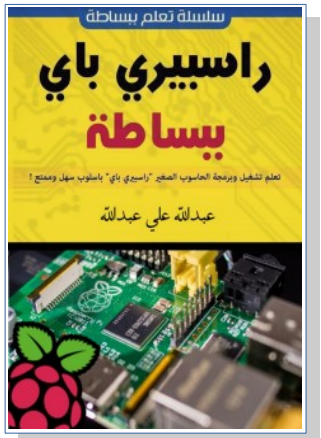

القرية الهندسية
ENGINEERINGVILLAGE
احترف الأردوينو
في 10 أيام
كتاب عملي ، تطبيقي ، مبسط


تعلم
استخدام JavaScript للتعامل مع HTML
DOM
عبد اللطيف ايمش


تعلم برنامج
Inkscape
خطوة ـ خطوة


php
تعلم البرمجة بلغة php
شرح بسيط و مفصل لمختلف نواحي اللغة
مع عدد من الأمثلة العملية
عبد اللطيف ايمش
أحمد ابو السعود
الطبعة الثانية


تعلم البرمجة بلغة C
Mathieu Nebra


سلسلة تعلم ببساطة
راسبيري باي
ببساطة
عبدالله علي عبدالله


القرية الهندسية
ENGINEERINGVILLAGE
احترف
سكراتش
في 10 أيام
SCRATCH


وادي التقنية
تعلم لغة Go بسهولة
كل شيء تحتاج لمعرفته لتبدء البرمجة مع لغة جو
آلان دونوفان
بريان كيرنيغان


مدخل إلى:
علم تصميم البرمجيات
المبادرة العربية للترجمة العلمية


GNU
PSPP
http://www.gnu.org


خطوة
بخطوة
لنحترف
GIMP