Helion

Ciągłe dostarczanie oprogramowania

Automatyzacja kompilacji, testowania i wdrażania

Jez Humble
David Farley

Dostarczaj oprogramowanie na zawołanie!

Tytuł oryginału: Continuous Delivery: Reliable Software Releases
through Build, Test, and Deployment Automation

Tłumaczenie: Dorota Konowrocka-Sawa

ISBN: 978-83-246-9918-6

Authorized translation from the English language edition, entitled: CONTINUOUS DELIVERY: RELIABLE SOFTWARE RELEASES THROUGH BUILD, TEST, AND DEPLOYMENT AUTOMATION; ISBN 0321601912; by Jez Humble; and by David Farley; published by Pearson Education, Inc, publishing as Addison Wesley.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Spis treści

Rozdział 3

Ciągła integracja

Wstęp

Niezmiernie dziwną, ale powszechnie spotykaną cechą wielu projektów programistycznych jest to, że na przestrzeni długich okresów procesu rozwojowego aplikacja w ogóle nie działa. W gruncie rzeczy większość oprogramowania wytwarzanego przez duże zespoły nie nadaje się do użycia przez znaczną część czasu przeznaczonego na jego rozwój. Powody nietrudno zrozumieć: nikt nie jest zainteresowany podjęciem prób uruchomienia całej aplikacji, dopóki nie zostanie ukończona. Programiści ewidencjonują zmiany i być może nawet przeprowadzają zautomatyzowane testy jednostkowe, ale nikt nie próbuje tak naprawdę uruchomić całej aplikacji i posłużyć się nią w środowisku zbliżonym do produkcyjnego.

Dzieje się tak zwłaszcza w projektach, w których od dawna funkcjonują oddzielne gałęzie lub w których testy akceptacyjne odkładane są na sam koniec. W wielu takich projektach na koniec fazy rozwoju zaplanowano długotrwałą fazę integracji pozostawiającą zespołowi projektowemu czas na scalenie gałęzi i uruchomienie całości aplikacji, by mogła przejść testy akceptacyjne. Co gorsza, w niektórych projektach okazuje się, że kiedy dochodzi się już do tej fazy, oprogramowanie nie realizuje postawionych przed nim zadań. Integracja może się okazać niezwykle czasochłonna, a co najgorsze, nie sposób przewidzieć, ile to wszystko ostatecznie potrwa.

Z drugiej strony zetknęliśmy się też z projektami, w których aplikacja po wprowadzeniu najnowszych zmian nie działa zaledwie przez kilka minut. Różnica polega na zastosowaniu ciągłej integracji. Ciągła integracja wymaga, by za każdym razem, gdy ktoś ewidencjonuje jakąkolwiek zmianę, cała aplikacja została skompilowana i poddana wszechstronnym testom zautomatyzowanym. Zasadnicze znaczenie ma to, by w sytuacji, kiedy proces kompilacji lub testów nie powiedzie się, zespół projektowy przerwał to, czym w danym momencie się zajmuje, i natychmiast rozwiązał powstały problem. Celem ciągłej integracji jest stała dostępność działającego oprogramowania.

O ciągłej integracji po raz pierwszy napisał Kent Beck w książce *Extreme Programming Explained* (pierwsze wydanie w 1999 roku). Podobnie jak w przypadku innych praktyk programowania ekstremalnego, u podstaw ciągłej integracji leży następująca konstatacja: jeśli regularna integracja bazy kodu jest dobra, to dlaczego nie robić tego ciągle? W kontekście integracji „ciągle" oznacza „za każdym razem, gdy ktokolwiek zaewidencjonuje jakąkolwiek zmianę w systemie kontroli wersji". Jak mówi Mike Roberts, jeden z naszych współpracowników, „»ciągle« to częściej, niż myślisz" [aEu8Nu].

Ciągła integracja stanowi zmianę paradygmatu. Bez ciągłej integracji oprogramowanie nie działa, dopóki ktoś nie udowodni, że jest inaczej — zazwyczaj na etapie testów lub integracji. Przy ciągłej integracji oprogramowanie w sposób udowodniony działa (zakładając wystarczająco wszechstronny i kompleksowy zestaw zautomatyzowanych testów) po każdej nowej zmianie. Wiesz doskonale, w którym momencie przestaje działać, i możesz to naprawić. Zespoły, które w efektywny sposób posługują się ciągłą integracją, są w stanie dostarczać oprogramowanie znacznie szybciej i z mniejszą ilością błędów niż zespoły niestosujące ciągłej integracji. Błędy wychwytywane są na znacznie wcześniejszych etapach procesu projektowego, a tym samym ich poprawienie jest tańsze, co pociąga za sobą oszczędność czasu i pieniędzy. Z tego względu uznajemy ciągłą integrację za kluczową praktykę stosowaną w profesjonalnych zespołach, być może równie istotną jak stosowanie systemu kontroli wersji.

W pozostałej części tego rozdziału opisujemy metody wdrażania ciągłej integracji. Wyjaśnimy, jak rozwiązać powszechnie występujące problemy pojawiające się wraz ze wzrostem złożoności projektu, i wyliczymy praktyki skutecznie wspierające ciągłą integrację oraz ich wpływ na proces projektowania i wytwarzania oprogramowania. Omówimy również bardziej zaawansowane tematy, w tym metody wprowadzania ciągłej integracji w zespołach rozproszonych.

Ciągła integracja omówiona jest obszernie w książce Paula Duvalla *Continuous Integration* (Addison-Wesley, 2006), którą można uznać za komplementarną wobec niniejszej. Jeśli szukasz bardziej szczegółowego omówienia ciągłej integracji niż zaproponowane przez nas w tym rozdziale, znajdziesz je właśnie w książce Duvalla.

Rozdział ten skierowany jest głównie do programistów. Zawiera jednak również nieco informacji, które naszym zdaniem przydadzą się menedżerom projektów chcącym poszerzyć swoją wiedzę o *praktycznych aspektach* ciągłej integracji.

Wdrażanie ciągłej integracji

Praktyka ciągłej integracji zależy od spełnienia pewnych warunków wstępnych. Omówimy je, a następnie przyjrzymy się dostępnym narzędziom. Być może najbardziej istotne jest to, że ciągła integracja zależy od tego, czy zespół postępuje zgodnie z kilkoma kluczowymi procedurami, więc poświęcimy chwilę na ich omówienie.

Czego potrzebujesz na początek?

Przed rozpoczęciem ciągłej integracji potrzebujesz trzech elementów.

1. Kontrola wersji

Wszystkie elementy projektu muszą zostać zaewidencjonowane w jednym repozytorium kontroli wersji: kody, testy, skrypty baz danych, skrypty kompilacji i wdrożenia i cała reszta, której potrzebujesz, by stworzyć, zainstalować, uruchomić i przetestować aplikację. Może się to wydawać oczywiste, ale ku naszemu zaskoczeniu istnieją jeszcze projekty, w których nie jest stosowany żaden system kontroli wersji. Niektórzy sądzą, że rozmiar ich projektów nie uzasadnia wprowadzenia systemu kontroli wersji. My natomiast jesteśmy przekonani, że nie istnieją projekty na tyle małe, by móc się bez niego obejść. Nawet gdy samodzielnie piszemy kod — na własne potrzeby i na własnym komputerze — to i tak stosujemy kontrolę wersji. Istnieje kilka prostych, lekkich i darmowych, a jednocześnie posiadających duże możliwości systemów kontroli wersji.

Dostępne systemy kontroli zmian i ich zastosowanie opisujemy szczegółowo w sekcji „Stosowanie systemów kontroli wersji” w rozdziale 2., a także w rozdziale 14., „Zaawansowana kontrola wersji”.

2. Automatyczna kompilacja

Musisz być w stanie uruchomić kompilację z wiersza poleceń. Możesz zacząć od programu wiersza poleceń, który każe Twojemu IDE skompilować oprogramowanie, a następnie przeprowadzić testy, ale może to być również złożony zbiór wieloetapowych skryptów kompilacji wywołujących się po kolei. Niezależnie od tego, jak będzie wyglądał ostateczny mechanizm, człowiek lub komputer muszą mieć możliwość przeprowadzenia procesu kompilacji, testów i wdrożenia w sposób zautomatyzowany za pomocą wiersza poleceń.

Narzędzia IDE i narzędzia ciągłej integracji są współcześnie dość wyszukane i zazwyczaj możesz skompilować oprogramowanie i przeprowadzić testy, nie zbliżając się nawet do wiersza poleceń. Sądzimy jednak, że mimo to powinieneś mieć skrypty kompilacji możliwe do uruchomienia z wiersza poleceń bez sięgania po IDE. Może się to wydawać kontrowersyjne, ale wynika z kilku przyczyn:

- Musisz być w stanie uruchomić swój proces kompilacji w sposób zautomatyzowany z własnego środowiska ciągłej integracji, aby mógł zostać poddany audytowi, gdy coś pójdzie nie tak.
- Skrypty kompilacji powinny być traktowane jak baza kodu. Powinny być testowane i ustawicznie refaktoryzowane, aby były uporządkowane i czytelne. Nie da się tego zrobić, kiedy proces kompilacji przeprowadzany jest za pomocą narzędzi IDE. Im bardziej złożone stają się Twoje projekty, tym większego nabiera to znaczenia.
- Ułatwia to zrozumienie kompilacji, utrzymanie i debugowanie kompilacji; pozwala również na lepszą współpracę z pracownikami działu eksploatacji systemów IT.

3. Zgoda zespołu

Ciągła integracja jest procedurą, nie narzędziem. Wymaga od zespołu programistycznego pewnego zaangażowania i dyscypliny. Wszyscy muszą często ewidencjonować w głównej gałęzi projektu małe, inkrementalne zmiany i zgodzić się co do tego, że zadaniem o najwyższym priorytecie w projekcie jest poprawienie każdej zmiany psującej aplikację. Jeśli ludzie nie narzucą sobie koniecznej do osiągnięcia tego celu dyscypliny, próba wprowadzenia ciągłej integracji nie doprowadzi do oczekiwanego podniesienia jakości.

Podstawowy system ciągłej integracji

Nie potrzebujesz systemu ciągłej integracji, by prowadzić ciągłą integrację — jak już powiedzieliśmy, jest to procedura, a nie narzędzie. James Shore opisuje najprostszą metodę rozpoczęcia ciągłej integracji w artykule pod tytułem *Continuous Integration on a Dollar a Day* [bAJpjp] za pomocą jedynie niewykorzystywanej stacji roboczej, gumowego kurczaka i dzwonka stawianego zazwyczaj na kontuarze hotelowej recepcji. Warto przeczytać ten artykuł, bo cudownie pokazuje podstawy ciągłej integracji bez stosowania jakichkolwiek narzędzi z wyjątkiem systemu kontroli wersji.

W rzeczywistości jednak współczesne narzędzia ciągłej integracji niezwykle łatwo jest zainstalować i uruchomić. Istnieje kilka opcji open source, takich jak Hudson i czcigodna rodzina CruiseControl (CruiseControl, CruiseControl.NET i CruiseControl.rb). Hudsona, a już zwłaszcza CruiseControl.rb niezwykle łatwo jest postawić i uruchomić. CruiseControl.rb jest bardzo lekki i może być bez trudu rozszerzany przez każdego, kto dysponuje choćby niewielką wiedzą na temat Ruby. Hudson ma ogromny zestaw wtyczek pozwalających na integrację z dosłownie każdym narzędziem w ekosystemie kompilacji i wdrożenia.

W chwili pisania tej książki dwa komercyjne serwery ciągłej integracji mają wersje darmowe zaprojektowane dla małych zespołów: Go wydawany przez ThoughtWorks Studios i TeamCity autorstwa JetBrains. Inne popularne komercyjne serwery CI to między innymi Bamboo autorstwa Atlassian i Pulse stworzony przez Zutubi. Wysokiej klasy systemy zarządzania wydaniami i przyspieszania kompilacji, które można stosować również w celu zwykłej ciągłej integracji, to między innymi AntHillPro produkcji UrbanCode[1], ElectricCommander produkcji Electric Cloud i Rational BuildForge produkcji IBM. Istnieje wiele innych systemów; pełną listę znajdziesz w macierzy funkcjonalności ciągłej integracji [bHOgH4].

Kiedy już zainstalujesz wybrane narzędzie ciągłej integracji, to przy spełnieniu opisanych wyżej warunków powinieneś być w stanie w ciągu dosłownie kilku minut rozpocząć pracę, wskazując mu, gdzie znajdzie repozytorium kontroli wersji, jakie skrypty uruchomić, by przeprowadzić kompilację, jeśli to konieczne, i przeprowadzić zautomatyzowane testy towarzyszące przekazywaniu zmian do aplikacji, oraz jak zasygnalizować, że ostatni zestaw zmian zepsuł aplikację.

Kiedy po raz pierwszy uruchomisz kompilację w narzędziu CI, odkryjesz zapewne, że maszynie, na której uruchomiłeś narzędzia CI, brakuje oprogramowania wspierającego i ustawień. To wyjątkowa okazja do nauczenia się czegoś — zapisuj wszystko, co zrobiłeś, by wszystko zaczęło działać, i umieść to w wiki projektu. Powinieneś poświęcić odpowiednio dużo czasu, by wprowadzić do systemu kontroli wersji wszystkie programy i ustawienia, od których uzależnione jest działanie systemu, i zautomatyzować proces dostarczania nowej maszyny.

W następnej fazie wszyscy powinni zacząć korzystać z serwera ciągłej integracji. Oto prosty proces, według którego należy postępować.

Kiedy już jesteś gotów do zaewidencjonowania ostatniej zmiany:

1. Sprawdź, czy kompilacja już się rozpoczęła. Jeśli tak, poczekaj, aż się zakończy. Jeżeli kompilacja się nie powiedzie, będziesz musiał popracować wraz z resztą zespołu nad zapewnieniem poprawności kompilacji, zanim zaewidencjonujesz zmianę.
2. Kiedy kompilacja dobiegnie końca, a testy przebiegną pomyślnie, zaktualizuj kod w środowisku programistycznym z aktualnej wersji z repozytorium kontroli wersji, by pobrać wszystkie aktualizacje.
3. Uruchom skrypt kompilacji i testy na własnej maszynie programistycznej, aby upewnić się, że wszystko nadal działa poprawnie na Twoim komputerze, ewentualnie zastosuj funkcję osobistej kompilacji narzędzia CI.
4. Jeśli lokalne kompilacje wykonywane są poprawnie, zaewidencjonuj kod w systemie kontroli wersji.
5. Poczekaj, aż narzędzie CI przeprowadzi kompilację z Twoimi zmianami.
6. Jeśli kompilacja się nie powiedzie, przerwij to, co akurat robisz, i natychmiast popraw błędy na maszynie programistycznej — przejdź do kroku 3.
7. Jeśli kompilacja przebiegnie poprawnie, ciesz się i bierz się za następne zadanie.

Jeśli wszyscy członkowie zespołu będą postępować według tej prostej procedury przy każdym wprowadzaniu jakiejkolwiek zmiany, będziesz wiedział, że Twoje oprogramowanie zawsze działa na każdej maszynie z taką samą konfiguracją jak serwer ciągłej integracji.

[1] Obecnie część firmy IBM — *przyp. red.*

Warunki wstępne ciągłej integracji

Sama z siebie ciągła integracja nie naprawi procesu kompilacji. Może się wręcz okazać bardzo bolesna, jeśli zaczniesz ją stosować w samym środku projektu. Aby okazała się efektywna, będziesz musiał przed jej wprowadzeniem wdrożyć opisane poniżej praktyki.

Ewidencjonuj regularnie

Najważniejszą praktyką zapewniającą poprawne działanie *ciągłej integracji* jest częste ewidencjonowanie zmian w głównej gałęzi projektu. Powinieneś ewidencjonować swój kod przynajmniej kilka razy dziennie.

Regularne ewidencjonowanie przynosi wiele innych korzyści. Zmiany stają się mniejsze, a tym samym zmniejsza się prawdopodobieństwo, że zepsują kompilację. Oznacza to, że masz ostatnią znaną dobrą wersję oprogramowania, do której możesz się cofnąć, kiedy popełnisz błąd albo obierzesz złą drogę. Pozwala to utrzymać większą dyscyplinę w kwestii refaktoryzacji i poprzez niewielkie zmiany utrzymać zachowanie aplikacji. Dzięki temu łatwiej jest osiągnąć stan, w którym modyfikacje zmieniające wiele plików nie będą stały w sprzeczności z pracą innych osób. Pozwala to programistom odważniej badać możliwe rozwiązania, wypróbowując kolejne koncepcje i odrzucając je poprzez powrót do ostatniej zaewidencjonowanej wersji. Zmusza to do robienia regularnych przerw i prostowania kości, dzięki czemu spada ryzyko zespołu kanału nadgarstka lub urazów będących skutkiem chronicznego przeciążenia mięśni i ścięgien. Oznacza to również, że jeśli dojdzie do jakiejś katastrofy (jak przypadkowe skasowanie czegoś istotnego), utracisz stosunkowo niewiele swojej pracy.

Celowo wspomnieliśmy o ewidencjonowaniu w gałęzi głównej. W wielu projektach w systemie kontroli wersji stosuje się gałęzie, traktując to jako metodę zarządzania dużymi zespołami. Prawdziwa ciągła integracja wyklucza jednak rozgałęzianie, bo z definicji jeśli pracujesz nad jakąś gałęzią, kod nie jest integrowany z kodem innych programistów. Zespoły, które stosują funkcjonujące od dawna w separacji gałęzie, stają dokładnie przed takimi samymi problemami z integracją jak te, które opisaliśmy na początku tego rozdziału. Nie możemy zalecić stosowania gałęzi z wyjątkiem niewielkiej liczby sytuacji wyjątkowych. Kwestie te omawiamy znacznie bardziej szczegółowo w rozdziale 14., „Zaawansowana kontrola wersji".

Stwórz obszerny i kompleksowy zestaw zautomatyzowanych testów

Jeśli nie masz obszernego zestawu zautomatyzowanych testów, poprawnie wykonująca się kompilacja oznacza tylko tyle, że aplikacja może zostać skompilowana i spakowana w pakiet instalacyjny. I chociaż dla niektórych zespołów jest to duży krok naprzód, kluczowe jest posiadanie zestawu zautomatyzowanych testów dającego pewność, że aplikacja naprawdę działa. Istnieje wiele rodzajów zautomatyzowanych testów; omawiamy je bardziej szczegółowo w następnym rozdziale. Istnieją jednak trzy rodzaje testów, które będziemy chcieli przeprowadzić na naszej aplikacji poddanej ciągłej integracji: testy jednostkowe, testy integracji modułów i testy akceptacyjne.

Testy jednostkowe pisane są w celu przetestowania zachowania niewielkich fragmentów aplikacji (np. metody lub funkcji, lub też interakcji zachodzących pomiędzy ich niewielkimi grupami) w izolacji od pozostałych. Zazwyczaj można je przeprowadzić bez uruchamiania całej aplikacji. Nie dotykają bazy danych (jeśli aplikacja z niej korzysta), systemu plików ani sieci.

Nie wymagają, by aplikacja działała w środowisku zbliżonym do środowiska produkcyjnego. Testy jednostkowe powinny przebiegać bardzo szybko — przeprowadzenie całego ich zestawu, nawet w przypadku dużych aplikacji, powinno zabierać nie więcej niż dziesięć minut.

Testy integracji modułów sprawdzają zachowanie kilku modułów aplikacji. Podobnie jak testy jednostkowe, nie zawsze wymagają uruchomienia całej aplikacji. Mogą jednak dotykać bazy danych, systemu plików lub innych systemów (które mogą zostać zaimitowane za pomocą obiektów zastępczych). Testy integracji modułów trwają zazwyczaj dłużej.

Testy akceptacyjne sprawdzają, czy aplikacja spełnia kryteria akceptacyjne ustalone przez biznes, zapewniając zarówno odpowiednią funkcjonalność, jak i cechy takie jak wydajność, dostępność, bezpieczeństwo itd. Testy akceptacyjne najlepiej pisać w taki sposób, by sprawdzały działanie całej aplikacji w środowisku produkcyjnym. Ich przeprowadzenie może zabierać sporo czasu — zdarzają się zestawy testów akceptacyjnych, których przeprowadzenie po kolei trwa ponad dobę.

Te trzy połączone zestawy testów powinny nam dać niemal stuprocentową pewność, że żadna z wprowadzonych zmian nie zniszczyła istniejącej funkcjonalności.

Niech proces kompilacji i testowania będzie możliwie krótki

Jeśli skompilowanie kodu i przeprowadzenie testów trwa zbyt długo, napotkasz opisane niżej problemy:

- Ludzie przestaną wykonywać pełną kompilację i przeprowadzać testy, zanim zaewidencjonują cokolwiek. Będziesz miał coraz więcej nieudanych kompilacji.
- Proces ciągłej integracji będzie zabierał tyle czasu, że zanim będziesz gotów do ponownego przeprowadzenia kompilacji, będziesz miał już szereg zatwierdzonych zmian, więc nie będziesz wiedział, które zaewidencjonowanie zepsuło kompilację.
- Ludzie będą ewidencjonowali rzadziej ze świadomością, że kompilacja i przeprowadzenie testów równają się trwającemu w nieskończoność czekaniu.

Najlepiej by było, żeby proces kompilacji i testowania, który przeprowadzasz przed zaewidencjonowaniem na swoim serwerze ciągłej integracji, trwał najwyżej kilka minut. Uważamy, że dziesięć minut to górna granica, pięć minut to dobry wynik, a dziewięćdziesiąt sekund to ideał. Ludziom przyzwyczajonym do pracy przy niewielkich projektach dziesięć minut będzie się dłużyć, a weteranom doświadczonym wielogodzinnymi kompilacjami minie jak z bicza trzasł. Tyle mniej więcej czasu można poświęcić na zrobienie sobie herbaty, krótką pogawędkę, sprawdzenie poczty i rozprostowanie kości.

Może się wydawać, że ten wymóg jest sprzeczny z wymienioną wcześniej koniecznością posiadania odpowiednio obszernego zestawu zautomatyzowanych testów. Istnieje jednak wiele technik, które można zastosować w celu skrócenia czasu kompilacji. Przede wszystkim należy się zastanowić, czy testy nie mogłyby być wykonywane szybciej. Narzędzia z rodziny XUnit, takie jak JUnit czy NUnit, udostępniają na wyjściu informację o faktycznej czasochłonności każdego testu. Sprawdź, które testy wykonują się szczególnie powoli, i zobacz, czy istnieje jakaś metoda ich przyspieszenia lub uzyskania tego samego pokrycia i pewności co do poprawności kodu przy mniejszej ilości obliczeń. Powinieneś to sprawdzać regularnie.

W pewnym momencie będziesz jednak musiał podzielić swój proces testowania na kilka etapów, jak opisaliśmy to szczegółowo w rozdziale 5., „Anatomia potoku wdrożeń”. Według jakiego kryterium podzielić zestaw testów? W pierwszej kolejności należy podzielić go na dwa etapy. Pierwszy etap powinien obejmować kompilacją oprogramowania, przeprowadzenie zestawu testów jednostkowych sprawdzających po kolei każdą z klas składających się na aplikację

i stworzenie nadającego się do wdrożenia pliku binarnego. Ten etap określamy mianem fazy przekazywania zmian. Ten etap kompilacji omawiamy bardzo szczegółowo w rozdziale 7.

W drugim etapie powinieneś pobrać pliki binarne z pierwszego etapu i przeprowadzić testy akceptacyjne oraz testy integracyjne i testy wydajnościowe, jeśli je przygotowałeś. Współczesne serwery ciągłej integracji znacznie ułatwiają stworzenie w ten sposób kompilacji tymczasowych, jednoczesne wykonanie szeregu zadań i zagregowanie wyników tak, byś jednym rzutem oka mógł ocenić stan kompilacji.

Fazę przekazywania zmian należy przeprowadzić na serwerze CI przed ostatecznym zaewidencjonowaniem każdej zmiany. Faza testów akceptacyjnych powinna się rozpocząć po tym, jak zmiana przejdzie wcześniejsze testy, i może zająć więcej czasu. Jeśli zauważysz, że druga kompilacja trwa dłużej niż pół godziny, powinieneś rozważyć możliwość przeprowadzenia tego zestawu testów równolegle na większej maszynie wieloprocesorowej, a może nawet stworzyć na potrzeby kompilacji sieć obliczeniową. Współczesne systemy CI znacznie to ułatwiają. Często przydatne okazuje się włączenie do fazy przekazywania zmian prostego zestawu testów dymnych. Testy dymne powinny obejmować kilka prostych testów akceptacyjnych i integracyjnych sprawdzających, czy nie popsuto najczęściej wykorzystywanej funkcjonalności, a jeśli do tego doszło, natychmiast o tym informujących.

Pożądane jest często powiązanie testów akceptacyjnych w grupy według kryterium funkcjonalności. Pozwala to przeprowadzić zestaw testów skupiających się na wybranych aspektach zachowania aplikacji po dokonaniu zmiany w konkretnym obszarze. Wiele środowisk testów jednostkowych pozwala kategoryzować testy w taki sposób.

Możesz dojść do etapu, w którym projekt musi zostać podzielony na kilka funkcjonalnie niezależnych modułów. Musisz się dobrze zastanowić, jak zorganizować te podprojekty zarówno w systemie kontroli wersji, jak i na swoim serwerze CI. Rozważymy to zagadnienie bardziej szczegółowo w rozdziale 13., „Zarządzanie modułami i zależnościami".

Zarządzanie środowiskiem programistycznym

Aby zapewnić wydajność i ocalić zdrowie psychiczne programistów, należy uważnie zarządzać środowiskiem programistycznym. Programiści powinni zawsze rozpoczynać kolejny etap pracy od znanej dobrej wersji. Powinni być w stanie uruchomić kompilację, wykonać zautomatyzowane testy i wdrożyć aplikację w kontrolowanym przez nich środowisku. W zasadzie powinno się ono znajdować na ich własnej lokalnej maszynie. Tylko w wyjątkowych przypadkach usprawiedliwione jest wytwarzanie oprogramowania w środowisku współdzielonym. Uruchamianie aplikacji w lokalnym środowisku programistycznym powinno się odbywać zgodnie z tymi samymi zautomatyzowanymi procesami, które stosowane są w środowiskach ciągłej integracji i testowym, a ostatecznie w środowisku produkcyjnym.

Pierwszym warunkiem osiągnięcia tego stanu jest staranne zarządzanie konfiguracją, które obejmuje nie tylko kod źródłowy, ale i dane testowe, skrypty baz danych, skrypty kompilacji i skrypty wdrożenia. Wszystkie te elementy muszą być przechowywane w systemie kontroli wersji, a najbardziej aktualne znane dobre wersje powinny być punktem wyjścia przy rozpoczynaniu kolejnej sesji kodowania. W tym kontekście „znane dobre" oznacza, że wersja, od której rozpoczynasz pracę, przeszła wszystkie zautomatyzowane testy na serwerze ciągłej integracji.

Drugim krokiem jest zarządzanie konfiguracją zależności stron trzecich, bibliotek i modułów. To bardzo ważne, abyś miał poprawne wersje wszystkich bibliotek lub modułów, czyli te same wersje, które na pewno współdziałają prawidłowo z tą wersją kodu źródłowego, od której rozpoczynasz pracę. Istnieją narzędzia open source ułatwiające zarządzanie zależnościami firm trzecich, spośród których najpopularniejsze są Maven i Ivy. Pracując z tymi narzędziami, musisz jednak zachować ostrożność i upewnić się, że zostały poprawnie skonfigurowane i nie pobierasz zawsze do lokalnej kopii roboczej najnowszej dostępnej wersji jakiejś zależności.

W większości projektów biblioteki firm trzecich, od których projekty te są uzależnione, nie zmieniają się zbyt często, więc najprostszym rozwiązaniem jest zapisanie ich w systemie kontroli wersji razem z kodem źródłowym. Więcej informacji na ten temat znajdziesz w rozdziale 13., „Zarządzanie modułami i zależnościami".

Ostatnim krokiem jest upewnienie się, że zautomatyzowane testy, w tym testy dymne, mogą zostać wykonane na maszynach programistycznych. W dużym systemie może to obejmować konfigurację systemów middleware i uruchomienie baz danych w wersji jednodostępowej lub funkcjonującej w pamięci RAM systemu. Może to wymagać pewnego wysiłku, ale umożliwienie programistom przeprowadzenia testów dymnych działającego systemu na maszynie programistycznej przed każdym zaewidencjonowaniem zmiany może w znaczący sposób podnieść jakość aplikacji. Tak naprawdę jedną z cech dobrej architektury aplikacji jest możliwość uruchomienia aplikacji bez trudu na maszynie programistycznej.

Stosowanie systemów ciągłej integracji

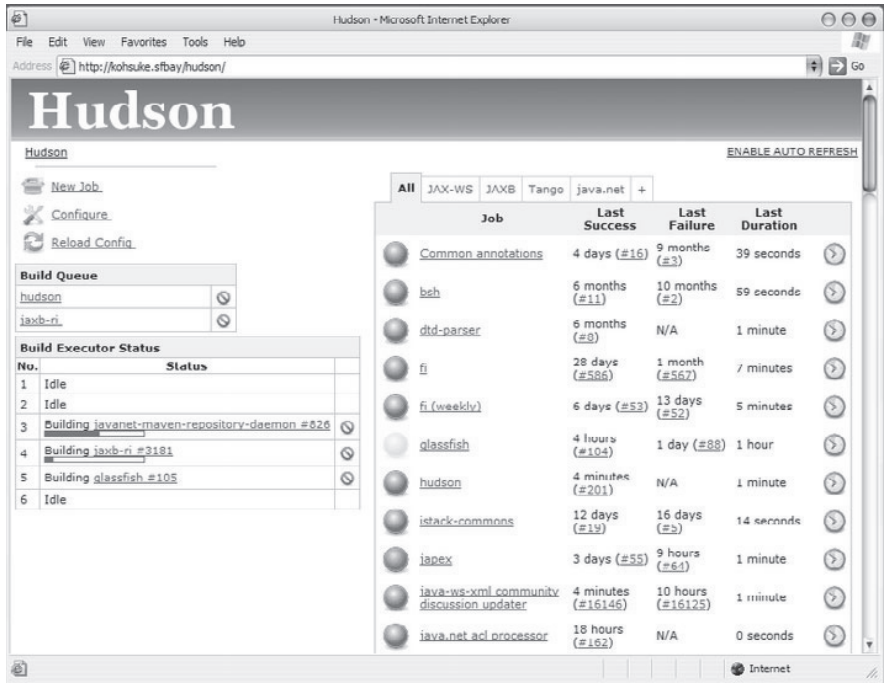
Na rynku istnieje wiele produktów pozwalających stworzyć infrastrukturę będącą fundamentem zautomatyzowanego procesu kompilacji i testowania. Najbardziej podstawową funkcjonalnością systemu ciągłej integracji jest odpytanie systemu kontroli wersji pod kątem zaewidencjonowanych ostatnio zmian, a jeśli takie zmiany zostaną wykryte, pobranie ostatniej wersji oprogramowania, uruchomienie skryptów kompilacji w celu jego skompilowania, przeprowadzenie testów i poinformowanie Cię o wynikach.

Podstawowa funkcjonalność

Oprogramowanie serwera ciągłej integracji ma dwie składowe. Pierwsza to długotrwały proces wykonujący w regularnych odstępach czasu prostą sekwencję zadań. Druga przedstawia wyniki przeprowadzonych procesów, informuje o powodzeniu lub niepowodzeniu kompilacji i przeprowadzonych testów oraz umożliwia dostęp do raportów z testów, instalatorów i tak dalej.

Typowy serwer CI odpytuje system kontroli wersji w regularnych odstępach czasu. Jeśli wykryje jakiekolwiek zmiany, wyewidencjonuje kopię projektu do katalogu na serwerze lub do katalogu agenta kompilacji. Następnie wykona określone przez Ciebie polecenia. Zazwyczaj polecenia te kompilują aplikację i wykonują odpowiednie zautomatyzowane testy.

Większość serwerów CI obejmuje serwer WWW, który pokazuje listę przeprowadzonych kompilacji (rysunek 3.1) i pozwala przejrzeć raporty informujące o sukcesie lub niepowodzeniu każdej kompilacji. Ta sekwencja instrukcji kompilacji powinna zaowocować wytworzeniem i zapisaniem wynikowych artefaktów takich jak pliki binarne lub pakiety instalacyjne, aby testerzy i klienci mogli bez trudu pobrać ostatnie dobre wersje oprogramowania. Większość serwerów CI da się konfigurować za pomocą interfejsu webowego lub prostych skryptów.



Rysunek 3.1. *Zrzut Hudsona, autor: Kohsuke Kawaguchi*

Wodotryski

Możesz wykorzystać wykraczające poza podstawową funkcjonalność możliwości pakietu CI w zakresie monitorowania przepływu pracy. Możesz przykładowo uzyskać status ostatniej kompilacji wysłanej na urządzenie zewnętrzne. Spotkaliśmy się z prezentowaniem statusu ostatniej kompilacji za pomocą czerwonych i zielonych lamp magmowych. Widzieliśmy też system CI, który wysyłał status do elektronicznego bezprzewodowego królika Nabaztag. Jeden ze znanych nam programistów, dysponujący pewnymi umiejętnościami w zakresie elektroniki, stworzył ekstrawagancką wieżę regularnie eksplodującą światłami i dźwiękami i sygnalizującą w ten sposób postęp rozmaitych kompilacji w złożonym projekcie. Kolejnym pomysłem było głośne wyczytywanie za pomocą syntetyzatora mowy nazwiska osoby, która popsuła kompilację. Niektóre serwery ciągłej integracji wyświetlają status kompilacji wraz z awatarami osób, które zaewidencjonowały zmiany — i można to wyświetlić na wielkim ekranie.

Tego rodzaju gadżety wykorzystuje się w projektach z prostego powodu: to świetny sposób, by wszyscy mogli jednym rzutem oka ocenić status kompilacji. Przejrzystość jest jedną z najważniejszych zalet stosowania serwera CI. Większość serwerów CI sprzedawanych jest z widgetem, który możesz zainstalować na maszynie programistycznej, a który w rogu pulpitu pokazuje status kompilacji. Takie narzędzia są szczególnie użyteczne dla zespołów rozproszonych, a w każdym razie nie pracujących w tym samym pokoju.

Jedyną wadą takiej przejrzystości jest to, że jeśli zespół programistyczny pracuje w tym samym kampusie, co jego klienci — jak zaleca się to w większości projektów zwinnych — niepowodzenia kompilacji będące naturalną częścią procesu mogą zostać uznane za objawy problemów z jakością aplikacji. Tak naprawdę jest zupełnie odwrotnie: za każdym razem, gdy kompilacja się nie udaje, sygnalizuje to wykrycie problemu, który w innym przypadku mógłby

prześliznąć się do wersji produkcyjnej. Czasem trudno to wytłumaczyć. Spotykaliśmy się z tym wielokrotnie i odbyliśmy nawet kilka nieprzyjemnych rozmów z klientem, gdy kompilacja była zepsuta dłużej, niż obie strony by sobie tego życzyły. Możemy tylko zalecać umieszczenie monitora kompilacji w widocznym miejscu i wytrwałe przekonywanie wszystkich osób zaangażowanych w projekt o płynących z tego realnych korzyściach. Najlepszym, oczywiście, rozwiązaniem jest ciężka praca nad utrzymaniem nieustającej poprawności kompilacji.

W proces kompilacji możesz również włączyć analizę kodu źródłowego. Zespoły zazwyczaj określają pokrycie testami, duplikację kodowania, przestrzeganie standardów kodowania, złożoność cyklomatyczną i inne wskaźniki kondycji kompilacji, po czym wyświetlają wyniki na stronie podsumowania każdej kompilacji. Możesz także uruchomić programy generujące grafy modelu obiektowego lub schematu bazy danych. We wszystkim tym chodzi o lepszą widoczność i czytelność statusu kompilacji.

Współczesne zaawansowane serwery CI mogą dystrybuować pracę w sieci obliczeniowej stworzonej na potrzeby kompilacji, zarządzać kompilacjami i zależnościami zbiorów współpracujących modułów, przekazywać informacje bezpośrednio do systemu zarządzania przebiegiem projektu i wykonywać wiele innych pożytecznych zadań.

Poprzednicy ciągłej integracji

Przed wprowadzeniem praktyki ciągłej integracji wiele zespołów programistycznych przeprowadzało kompilację nocą, po zakończeniu dnia pracy. Przez wiele lat praktyka ta obowiązywała powszechnie w Microsofcie. Osoba, która popsuła kompilację, musiała zostać i monitorować kolejne kompilacje, dopóki kompilacja nie została popsuta przez kolejną osobę.

W wielu projektach nadal przeprowadza się conocne kompilacje. Koncepcja polega na uruchomieniu procesu wsadowego pozwalającego na kompilację i integrację bazy kodu po wyjściu zespołu projektowego do domu. To krok we właściwym kierunku, ale wszystko bierze w łeb, gdy zespół po pojawieniu się rano w pracy odkryje, że kod nie skompilował się właściwie. Następnego dnia wprowadzane są kolejne zmiany, ale aż do wieczora nie można stwierdzić, czy system integruje się poprawnie. W ten sposób przez wiele dni kompilacja może nie wykonywać się poprawnie, aż w końcu — dobrze zgadujesz! — nadchodzi znów pora integracji. Do tego wszystkiego strategia ta naprawdę nie na wiele się zdaje, kiedy ma się geograficznie rozproszony zespół pracujący na wspólnej bazie kodu w różnych strefach czasowych.

Kolejnym rewolucyjnym krokiem było dodanie automatycznych testów. Po raz pierwszy podjęliśmy tę próbę wiele lat temu. Wspomniane testowanie było najbardziej podstawowym testem dymnym zapewniającym po prostu, że aplikacja wykona następną kompilację. W naszym ówczesnym procesie kompilacji był to wielki postęp i byliśmy z siebie bardzo zadowoleni. Dzisiaj oczekiwalibyśmy nieco więcej po najbardziej nawet podstawowych zautomatyzowanych kompilacjach. Testy jednostkowe ogromnie się rozwinęły i nawet najprostszy zestaw testów jednostkowych pozwala osiągnąć znacznie wyższy poziom pewności co do wynikowej kompilacji.

Następnym poziomem wyrafinowania stosowanym w niektórych projektach (choć przyznajemy, że ostatnio się z takimi nie spotkaliśmy) był proces „toczących się kompilacji", w którym proces wsadowy uruchamiany zgodnie z określonym wcześniej harmonogramem tworzenia kompilacji w ciągu nocy zastąpiono ciągłą kompilacją. Po każdym zakończeniu kompilacji z systemu kontroli wersji pobierana jest ostatnia wersja aplikacji i proces uruchamia się od nowa. Dave praktykował to z powodzeniem na początku lat 90. — było to rozwiązanie znacznie lepsze niż nocne kompilacje. Problem z tym podejściem polega na tym, że nie ma bezpośredniego połączenia między konkretnym zaewidencjonowaniem i kompilacją. Więc chociaż programiście dawało to pożyteczną informację zwrotną, niewystarczająca możliwość wyśledzenia przyczyn niepoprawnej kompilacji sprawiała, że trudno było mówić o zastosowaniu tej strategii w przypadku większych zespołów.

Kluczowe praktyki

Do tej pory skupialiśmy się na automatyzacji kompilacji i wdrożenia, ale projekt realizowany jest przede wszystkim przez ludzi. Ciągła integracja jest nawykiem, a nie narzędziem, jej efektywność zaś zależy od dyscypliny członków zespołu. Utrzymanie systemu ciągłej integracji, zwłaszcza gdy masz do czynienia z dużymi i złożonymi systemami CI, wymaga rygorystycznej dyscypliny od wszystkich członków zespołu programistycznego.

Celem naszego systemu CI jest w istocie zapewnienie nieprzerwanego działania naszego oprogramowania. Poniżej przedstawiamy praktyki, które narzucamy swoim zespołom, aby to osiągnąć. Praktyki te są niezbywalnym warunkiem poprawności działania ciągłej integracji. W następnej kolejności omówimy praktyki zalecane, lecz nieobowiązkowe.

Nie ewidencjonuj niczego w popsutej kompilacji

Grzechem głównym ciągłej integracji jest ewidencjonowanie czegokolwiek w popsutej kompilacji. Jeśli kompilacja nie działa, odpowiedzialni za to programiści czekają już w pogotowiu, by ją poprawić. Jeśli przyjmiemy to podejście, będziemy zawsze znakomicie przygotowani do wykrycia przyczyn niepowodzenia i natychmiastowego rozwiązania problemów. Jeśli jeden z naszych współpracowników zaewidencjonował zmianę i wskutek tego popsuł kompilację, będzie potrzebował czystego pola, by to możliwie szybko naprawić. Jeśli zaewidencjonujemy kolejne zmiany i uruchomimy kolejne kompilacje, spotęgujemy powstałe problemy.

Kiedy zasada ta jest łamana, naprawienie kompilacji zajmuje nieuchronnie więcej czasu. Ludzie przyzwyczajają się do widoku popsutych kompilacji i bardzo szybko dochodzi do sytuacji, w której kompilacja popsuta jest bez przerwy. Sytuacja ta utrzymuje się do momentu, gdy któryś członek zespołu uzna, że ma dość, zmuszając wszystkich do podjęcia herkulesowego wysiłku naprawienia kompilacji, a później cały proces zaczyna się od nowa. Uwieńczenie pracy sukcesem to idealny moment, by zgromadzić wszystkich i przypomnieć im, że przestrzeganie tej zasady zapewni nieprzerwanie poprawną kompilację, a tym samym nieprzerwanie działające oprogramowanie.

Zawsze testuj lokalnie wszystkie zmiany przed ich zatwierdzeniem albo zleć to serwerowi CI

Jak już ustaliliśmy, zaewidencjonowanie zmiany powoduje stworzenie kandydata do wydania. To forma publikacji. Większość ludzi sprawdzi swoją pracę, zanim zdecyduje się upublicznić ją w jakiejkolwiek formie, i przy zatwierdzaniu zmian w systemie kontroli wersji wygląda to tak samo.

Kolejne ewidencjonowania powinny być na tyle lekkie, żebyśmy mogli bez problemu ewidencjonować regularnie co mniej więcej dwadzieścia minut, ale jednocześnie na tyle formalne, żebyśmy poświęcili chwilę na refleksję, zanim zdecydujemy się je zatwierdzić. Lokalne przeprowadzenie testów towarzyszących przekazywaniu zmian to moment na dokonanie trzeźwego osądu przed podjęciem działań. To również metoda sprawdzenia, czy to, co sądzimy, że działa, naprawdę działa.

Kiedy programiści przerywają na chwilę pracę, gotowi do zaewidencjonowania zmiany, powinni odświeżyć swoją lokalną kopię projektu, pobierając aktualizację z systemu kontroli wersji. Następnie powinni uruchomić lokalną kompilację i przeprowadzić testy towarzyszące

przekazywaniu zmian. Dopiero po zakończeniu ich pomyślnego przebiegu programista jest gotowy do zatwierdzenia zmian w systemie kontroli wersji.

Jeśli nie spotkałeś się wcześniej z takim podejściem, możesz się zastanawiać, dlaczego przeprowadzamy testy towarzyszące przekazywaniu zmian lokalnie przed zaewidencjonowaniem, jeśli pierwszą rzeczą po zaewidencjonowaniu jest kompilacja kodu i powtórne przeprowadzenie testów towarzyszących przekazywaniu zmian. Istnieją ku temu dwa powody:

1. Inni mogli coś zaewidencjonować, zanim pobrałeś ostatnią aktualizację z systemu kontroli wersji, a połączenie Waszych zmian może zaowocować niepowodzeniem testów. Jeśli wyewidencjonujesz i przeprowadzisz testy towarzyszące przekazywaniu zmian lokalnie, zidentyfikujesz problem bez psucia kompilacji.
2. Częstym źródłem problemów występujących po zaewidencjonowaniu jest zapomnienie o dodaniu do repozytorium jakichś nowych artefaktów. Jeśli trzymasz się tej procedury i lokalna kompilacja się powiedzie, a następnie system zarządzania CI nie przeprowadzi pomyślnie *fazy przekazywania zmian*, będziesz wiedział, że stało się tak dlatego, iż albo ktoś zaewidencjonował coś w międzyczasie, albo zapomniałeś dodać do systemu kontroli wersji nową klasę lub plik konfiguracji, nad którym właśnie pracowałeś.

   Przestrzeganie tej procedury zapewnia nieprzerwaną poprawność kompilacji.

Wiele współczesnych serwerów CI udostępnia funkcjonalność znaną jako *pretested commit*, *preflight build* lub osobista kompilacja. Korzystając z niej, nie ewidencjonujesz sam, lecz cedujesz na swoją sieć obliczeniową ciągłej integracji odpowiedzialność za pobranie lokalnych zmian i przeprowadzenie kompilacji. Jeśli kompilacja się powiedzie, serwer CI zaewidencjonuje zmiany za Ciebie. Jeśli kompilacja się nie powiedzie, zasygnalizuje, co poszło nie tak. To znakomita metoda realizowania tej procedury bez konieczności oczekiwania, aż testy towarzyszące przekazywaniu zmian się powiodą, by móc rozpocząć pracę nad kolejną funkcją lub usuwaniem kolejnego błędu.

Na etapie pisania tej książki funkcjonalność tę oferowały serwery ciągłej integracji Pulse (aktualna nazwa tego produktu to Secure Delivery Center), TeamCity i ElectricCommander. Funkcjonalność tę najlepiej połączyć z rozproszonym systemem kontroli wersji pozwalającym przechowywać zaewidencjonowane zmiany lokalnie, bez eksportowania ich na centralny serwer. W ten sposób bardzo łatwo jest zagregować swoje zmiany poprzez stworzenie łaty, a jeśli Twoja osobista kompilacja się nie powiedzie, wrócić do wersji kodu, którą wysłałeś na serwer CI.

Zanim podejmiesz pracę, poczekaj na powodzenie testów towarzyszących przekazywaniu zmian

System CI jest zasobem współdzielonym przez cały zespół. Kiedy zespół efektywnie posługuje się serwerem CI, postępując zgodnie z naszymi zaleceniami i często ewidencjonując zmiany, każde popsucie kompilacji jest dla zespołu i dla całego projektu tylko niewielką niedogodnością.

Psucie się kompilacji jest normalną i spodziewaną częścią procesu. Naszym celem jest wykrycie błędów i wyeliminowanie ich najszybciej jak to możliwe — nie oczekujemy perfekcji i bezbłędności.

W chwili ewidencjonowania programista zatwierdzający zmianę odpowiedzialny jest za obserwowanie postępów kompilacji. Dopóki zaewidencjonowana zmiana się nie skompiluje i nie przejdzie *testów towarzyszących przekazywaniu zmian*, programista nie powinien podejmować żadnych nowych zadań. Nie powinien wychodzić na lunch ani rozpoczynać zebrania. Powinien obserwować kompilację na tyle uważnie, by znać jej wynik w ciągu kilku sekund od zakończenia *fazy przekazywania zmian*.

Tylko wówczas, gdy ewidencjonowanie zmiany się powiedzie, programiści mogą się zabrać za kolejne zadanie. Jeśli się nie powiedzie, są na miejscu, by określić naturę problemu i go rozwiązać — za pomocą kolejnego zaewidencjonowania lub powrotu do poprzedniej wersji z systemu kontroli wersji, co oznacza wycofanie zmian do czasu, aż zrozumieją, jak zapewnić ich poprawne działanie.

Nigdy nie idź do domu, dopóki kompilacja nie działa poprawnie

Jest 17.30 w piątek, wszyscy Twoi współpracownicy opuszczają biuro, a Ty właśnie wprowadziłeś swoje zmiany. Kompilacja nie działa. Masz trzy możliwości. Możesz się pogodzić z faktem, że wyjdziesz później, i spróbować ją naprawić. Możesz wycofać zmiany i podjąć kolejną próbę zaewidencjonowania w przyszłym tygodniu. Możesz też wyjść, zostawiając niedziałającą kompilację.

Jeśli zostawisz popsutą kompilację, to do poniedziałku wspomnienie o wprowadzonych zmianach zblednie i zrozumienie problemu i usunięcie jego przyczyn zajmie Ci znacznie więcej czasu. Jeśli w poniedziałek rano nie będziesz pierwszą osobą próbującą naprawić kompilację, reszta zespołu zmiesza Cię z błotem, bo gdy przyjdą do pracy odkryją, że kompilacja nie działa, a oni nie mogą pracować. Jeśli podczas weekendu złoży Cię choroba i nie dotrzesz do pracy w poniedziałek, bądź pewien, że co najmniej kilka razy ktoś zadzwoni z pytaniem, co robiłeś, że kompilacja nie działa, i jak ją naprawić, albo też Twoje zmiany zostaną bezceremonialnie usunięte przez któregoś ze współpracowników. Tak czy inaczej, Twoja reputacja zostanie zszargana.

Skutki popsutej kompilacji, a zwłaszcza kompilacji pozostawionej w tym stanie pod koniec dnia pracy, ulegają spotęgowaniu, kiedy pracujesz w zespole rozproszonym geograficznie i funkcjonującym w różnych strefach czasowych. W tych okolicznościach powrót do domu po zepsuciu kompilacji jest prawdopodobnie jednym z najskuteczniejszych sposobów, by na długo zrazić sobie swoich zdalnych współpracowników.

Aby to było całkowicie jasne: *nie zalecamy siedzenia po godzinach w celu naprawienia kompilacji*. Zalecamy natomiast, żebyś ewidencjonował zmiany na tyle regularnie i na tyle wcześnie, by pozostawić sobie czas na rozwiązanie ewentualnych problemów. Możesz przecież zostawić sobie ich zaewidencjonowanie na następny dzień; wielu doświadczonych programistów celowo nie wprowadza żadnych zmian na godzinę przed końcem pracy, zostawiając to sobie na następny poranek. Jeśli wszystko inne zawiedzie, po prostu wycofaj zmiany z systemu kontroli wersji i pozostaw je tylko w swojej lokalnie działającej kopii. Niektóre systemy kontroli wersji, w tym wszystkie systemy rozproszone, ułatwią Ci to, pozwalając gromadzić ewidencjonowania w ramach lokalnego repozytorium bez publikowania ich dla pozostałych użytkowników.

Dyscyplina kompilacji w projektach rozproszonych

Pracowaliśmy kiedyś przy bodajże największym wówczas zwinnym projekcie na świecie. Był to geograficznie rozproszony projekt, w ramach którego programiści pracowali na wspólnej bazie kodu. Zespół jako całość pracował jednocześnie, na różnych etapach życia projektu, w San Francisco, Chicago, Londynie i Bangalore. Tylko przez mniej więcej trzy godziny w ciągu doby nikt, w żadnym z tych miejsc, nie pracował nad projektem. Przez resztę czasu do systemu kontroli wersji płynął nieprzerwany strumień zmian, któremu towarzyszył nieprzerwany ciąg wyzwalanych kompilacji.

Gdyby zespół w Indiach popsuł kompilację i poszedł do domu, dramatycznie zaburzyłoby to pracę zespołu londyńskiego. Analogicznie, gdyby londyński zespół poszedł do domu, popsuwszy kompilację, amerykańscy koledzy przez następne osiem godzin klęliby pod nosem.

Ścisła dyscyplina kompilacji była kluczowa do tego stopnia, że mieliśmy wyznaczonego pracownika odpowiedzialnego za kompilację, który nie tylko ją utrzymywał, ale też czasem egzekwował przestrzeganie ustalonych zasad, dopilnowując, by osoba, która popsuła kompilację, poświęciła czas na jej naprawienie. W przeciwnym razie inżynier kompilacji wycofywał jej zaewidencjonowanie.

Zawsze bądź przygotowany na powrót do poprzednich wersji

Jak opisaliśmy wcześniej, chociaż bardzo się staramy pracować starannie, wszystkim nam zdarza się popełniać błędy, więc nie jest dla nas żadnym zaskoczeniem, że od czasu do czasu każdemu zdarzy się popsuć kompilację. Przy większych projektach często dochodzi do tego codziennie, chociaż wcześniejsze testowanie ewidencjonowań znacznie zmniejsza skalę tego zjawiska. W tych okolicznościach błędy wykrywa się zazwyczaj szybko, a poprawki są przeważnie proste i polegają na wprowadzeniu zmiany dosłownie w jednej linii kodu. Czasem jednak sprawy przybierają gorszy obrót i albo nie możemy znaleźć źródła problemu, albo też po niepowodzeniu kolejnego zaewidencjonowania dochodzimy do wniosku, że umknęło nam coś istotnego, co ma związek z naturą zmiany, której właśnie dokonaliśmy.

Jaka by nie była nasza reakcja na niepowodzenie *fazy przekazywania zmian*, ważne, żebyśmy przywrócili poprawne działanie całości. Jeśli z jakiegokolwiek powodu nie jesteśmy w stanie szybko rozwiązać problemu, powinniśmy powrócić do wcześniejszego zestawu zmian przechowywanego w systemie kontroli wersji i rozwiązać problem w naszym środowisku lokalnym. W końcu jednym z powodów, dla których w ogóle utrzymujemy system kontroli wersji, jest zapewnienie sobie tej właśnie swobody powrotu do wcześniejszej wersji.

Pilotów samolotów uczy się, że przy każdym lądowaniu powinni zakładać, iż coś pójdzie nie tak, a zatem w każdej chwili powinni być gotowi do przerwania podejścia do lądowania i do odejścia na drugi krąg, by podjąć kolejną próbę lądowania. Ewidencjonuj zmiany z tym samym nastawieniem. Przyjmij, że możesz zepsuć coś, czego naprawa potrwa dłużej niż kilka minut, i miej świadomość, co należy zrobić, by cofnąć zmiany i wrócić do znanej dobrej wersji z systemu kontroli wersji. Wiesz, że wcześniejsza wersja była dobra, ponieważ *nie ewidencjonuje się zmian w popsutej kompilacji*.

Ustaw sobie limit czasu na poprawki przed cofnięciem zmian

Przyjmij tę zasadę w zespole: jeśli zmiana popsuje kompilację, postaraj się rozwiązać problem w ciągu dziesięciu minut. Jeśli po dziesięciu minutach nie masz gotowego rozwiązania, powróć do wcześniejszej wersji z systemu kontroli wersji. O ile jesteśmy w szczególnie pobłażliwym nastroju, możemy Ci okazjonalnie dać nieco dodatkowego czasu. Jeśli jesteś w trakcie dokonywanej lokalnie kompilacji i przygotowujesz się do zatwierdzenia zmian, pozwolimy Ci skończyć, żeby sprawdzić, czy to działa. Jeśli zadziała, możesz zaewidencjonować i mamy nadzieję, że poprawka rozwiąże problem; jeśli zawiedzie w środowisku lokalnym albo po zaewidencjonowaniu, powróć do ostatniej znanej dobrej wersji.

Doświadczeni programiści będą egzekwować tę zasadę w każdych okolicznościach, z przyjemnością wycofując wprowadzone przez innych zmiany, które popsuły kompilację na co najmniej dziesięć minut.

Nie wyłączaj testów, które zakończyły się niepowodzeniem

Kiedy zaczniesz egzekwować poprzednią zasadę, programiści mogą zacząć wyłączać testy zakończone niepowodzeniem, żeby mimo wszystko zaewidencjonować zmiany. Ten odruch jest zrozumiały, ale nieprawidłowy. Gdy testy, które przez dłuższy czas dawały poprawne rezultaty, zaczynają się kończyć niepowodzeniem, może być trudno dociec, co poszło nie tak. Czy naprawdę wykryto regresję? Może jedno z założeń testu przestało obowiązywać lub też aplikacja naprawdę zmieniła testowaną funkcjonalność z istotnego powodu. Dojście, który z tych warunków ma zastosowanie, może wymagać przeprowadzenia szeregu rozmów i pochłonąć masę czasu, ale włożenie tego wysiłku, by dojść, co się dzieje i albo poprawić kod (jeśli znaleziono regresję), albo zmodyfikować test (jeśli zmieniło się jedno z założeń), albo skasować test (jeśli poddawana mu funkcjonalność już nie istnieje), ma kluczowe znaczenie.

Wyłączanie testów, które się nie powiodły, powinno być wyjściem ostatecznym, stosowanym bardzo rzadko i niechętnie, jeśli nie starcza Ci dyscypliny, żeby natychmiast rozwiązać powstały problem. W porządku, jeśli w zupełnie wyjątkowych okolicznościach wyłączysz test do czasu zakończenia jakichś poważnych prac rozwojowych, które muszą zostać rozplanowane w czasie, lub do czasu rozstrzygnięcia jakiejś ważkiej, obejmującej wiele kwestii dyskusji z klientem. W ten sposób możesz się jednak zacząć zsuwać po równi pochyłej. Spotkaliśmy się z kodem, w którym połowa testów została wyłączona. Zalecane jest śledzenie liczby wyłączonych testów i prezentowanie jej na dużym, czytelnym wykresie lub na ekranie. Można nawet nie zatwierdzać kompilacji, jeśli liczba wyłączonych testów przekroczy określoną wartość progową ustaloną na poziomie na przykład 2% całości.

Weź odpowiedzialność za wszystkie szkody powstałe w wyniku zmian

Jeśli wprowadzisz zmianę i wszystkie napisane przez Ciebie testy zostaną przeprowadzone pomyślnie, lecz inne zakończą się niepowodzeniem, kompilacja jest zepsuta tak czy inaczej. Zazwyczaj oznacza to, że w wyniku Twoich zmian w aplikacji pojawiła się regresja. Do Twoich obowiązków — ponieważ to Ty wprowadziłeś zmianę — należy poprawienie wszystkich testów, które nie przebiegają pomyślnie w wyniku wprowadzonych przez Ciebie zmian. W kontekście ciągłej integracji wydaje się to oczywiste, ale tak naprawdę w wielu projektach nie jest to wcale praktyka oczywista.

Przyjęcie tego zwyczaju ma szereg konsekwencji. Oznacza to, że musisz mieć dostęp do każdego kodu, który możesz zepsuć w wyniku wprowadzonych przez siebie zmian, żebyś mógł go poprawić, kiedy zostanie zepsuty. Oznacza to, że nie możesz pozwolić sobie, by programiści posiadali na własność fragmenty kodu, nad którymi tylko oni mogą pracować. Aby można było skutecznie prowadzić ciągłą integrację, wszyscy muszą mieć dostęp do całości bazy kodu. Jeśli z jakiegoś powodu z konieczności znalazłeś się w sytuacji, w której dostęp do kodu nie może być współdzielony w całym zespole, możesz to obejść dzięki dobrej współpracy z ludźmi dysponującymi potrzebnym dostępem. Zdecydowanie nie jest to jednak najlepsze rozwiązanie i powinieneś dołożyć starań, by usunąć tego rodzaju ograniczenia.

Programowanie sterowane testami

Posiadanie obszernego i wszechstronnego zestawu testów jest kluczowym warunkiem ciągłej integracji. Omawiamy obszernie strategie automatycznego testowania w następnym rozdziale, ale warto podkreślić, że szybka informacja zwrotna będąca podstawowym rezultatem ciągłem integracji możliwa jest tylko przy doskonałym pokryciu testami jednostkowymi (doskonałe pokrycie testami akceptacyjnymi również ma zasadnicze znaczenie, ale ich przeprowadzenie

zabiera więcej czasu). Z naszego doświadczenia wynika, że jedynym sposobem uzyskania doskonałego pokrycia testami jednostkowymi jest projektowanie i programowanie sterowane testami (ang. *test-driven development*). W niniejszej książce próbowaliśmy uniknąć dogmatyzmu, jeśli chodzi o praktyki zwinnego wytwarzania oprogramowania, ale sądzimy, że programowanie sterowane testami odgrywa kluczowe rolę, jeśli chodzi o umożliwienie praktyki ciągłego dostarczania oprogramowania.

Zgodnie z koncepcją programowania sterowanego testami programiści opracowujący nowy fragment funkcjonalności lub poprawiający błąd najpierw tworzą testy będące wykonywalną specyfikacją oczekiwanego zachowania kodu, który ma zostać napisany. Testy te nie tylko określają projekt oprogramowania, ale służą jednocześnie jako testy regresji oraz dokumentacja kodu i oczekiwanego zachowania aplikacji.

Omówienie programowania sterowanego testami wykracza poza zakres tematyczny tej książki. Warto jednak odnotować, że podobnie jak w przypadku wszystkich tego rodzaju praktyk, w programowaniu sterowanym testami ważny jest pragmatyzm i dyscyplina. Osobom szczególnie zainteresowanym tym tematem polecamy dwie książki: *Growing Object-Oriented Software, Guided by Tests* Steve'a Freemana i Nata Pryce'a oraz *xUnit Test Patterns: Refactoring Test Code* Gerarda Meszarosa.

Zalecane praktyki

Poniższe praktyki nie są wymogiem, ale stwierdziliśmy ich przydatność, a czytelnik powinien przynajmniej rozważyć ich zastosowanie w kontekście własnych projektów.

Praktyki programowania ekstremalnego (XP)

Ciągła integracja jest jedną z dwunastu kluczowych praktyk XP opisanych w książce Kenta Becka i jako taka jest komplementarna wobec pozostałych praktyk XP. Ciągła integracja może znacząco wpłynąć na pracę każdego zespołu, nawet jeśli nie stosuje on pozostałych praktyk, ale największą efektywność pozwala osiągnąć w połączeniu z innymi praktykami. W szczególności — jako uzupełnienie programowania sterowanego testami i współwłasności kodu, które opisaliśmy na poprzednich stronach — powinieneś rozważyć refaktoryzację jako kamień węgielny efektywnego rozwoju oprogramowania.

Refaktoryzacja oznacza dokonywanie serii niewielkich, przyrostowych zmian poprawiających kod bez zmiany zachowania aplikacji. Ciągła integracja i programowanie sterowane testami umożliwiają refaktoryzację, upewniając Cię, że Twoje zmiany nie zmieniają dotychczasowego zachowania aplikacji. Tym samym Twój zespół może swobodnie dokonywać zmian obejmujących znaczne obszary kodu, nie przejmując się, że zepsują całą aplikację. Praktyka ta umożliwia również częste zaewidencjonowania — programiści ewidencjonują każdą niewielką, przyrostową zmianę.

Odrzucanie kompilacji ze względu na naruszenie architektury

Istnieją pewne aspekty architektury systemu, o których programistom zbyt łatwo jest zapomnieć. Jedną ze stosowanych przez nas technik jest umieszczanie testów towarzyszących przekazywaniu zmian, które dowodzą, że zasady rządzące architekturą nie zostały złamane.

Tak naprawdę jest to technika jedynie taktyczna i trudna do opisania inaczej niż na przykładach.

Wymuszanie zdalnych wywołań w czasie kompilacji

Najlepszy przykład, jaki nam się nasuwa, pochodzi z projektu, który został wdrożony jako zbiór usług rozproszonych. Był to system rzeczywiście rozproszony w tym sensie, że znacząca część logiki biznesowej została umieszczona na stacjach klienckich, a jednocześnie znacząca część logiki biznesowej wykonywana była na serwerze — wynikało to z realnych wymogów biznesu, a nie z kiepskiego programowania.

Nasz zespół programistyczny wdrożył całość kodu na platformie zarówno klienckiej, jak i serwerowej w środowiskach programistycznych. Programiście aż za łatwo było zrobić lokalne wywołanie z klienta do serwera albo z serwera do klienta, przy czym mógł nie zdawać sobie nawet sprawy, że jeśli chce rzeczywiście sprawdzić poprawność danego zachowania aplikacji, musi wykonać zdalne wywołanie.

Uporządkowaliśmy nasz kod w pakiety odzwierciedlające strategię warstwową, by było nam łatwiej przeprowadzić wdrożenie. Wykorzystaliśmy tę informację oraz oprogramowanie open source pozwalające ocenić zależności w kodzie, po czym zastosowaliśmy grep (narzędzie do wyszukiwania ciągów), by przeszukać wyniki podane przez narzędzie badania zależności i sprawdzić, czy wystąpiły jakieś zależności pomiędzy pakietami, które łamałyby nasze zasady architektoniczne.

Zapobiegało to niepotrzebnym niepowodzeniom na etapie testów funkcjonalnych i pomagało umacniać architekturę systemu, przypominając programistom o znaczeniu rozgraniczania procesów pomiędzy dwa systemy.

Technika ta może wydawać się nieco przyciężka i nie zastępuje jasnego zrozumienia architektury rozwijanego systemu przez zespół programistyczny. Może być jednak bardzo użyteczna, kiedy należy bronić ważnych zasad architektonicznych, których naruszenie trudno byłoby wcześnie wychwycić w inny sposób.

Odrzucanie kompilacji ze względu na powolność testów

Jak powiedzieliśmy wcześniej, ciągła integracja działa najlepiej przy częstym ewidencjonowaniu niewielkich zmian. Jeśli testy towarzyszące przekazywaniu zmian trwają zbyt długo, może to poważnie zachwiać produktywnością zespołu ze względu na czas spędzony na oczekiwaniu na ukończenie procesu kompilacji i testów. To z kolei zniechęci do częstego ewidencjonowania. W efekcie zespół będzie miał tendencję do gromadzenia zmian w większe, bardziej złożone pakiety, co zwiększy prawdopodobieństwo konfliktów przy scalaniu oraz wprowadzenia błędów, a tym samym niepowodzenia testów. Wszystko to jeszcze bardziej spowolni cały proces.

Aby zespół programistyczny nie stracił przekonania o znaczeniu utrzymywania zestawu szybkich testów, możesz odrzucić kompilację na etapie przekazywania zmian, jeśli znajdziesz test zabierający więcej czasu, niż zostało to wcześniej określone. Kiedy ostatnio stosowaliśmy to podejście, odrzucaliśmy kompilację, jeśli tylko wykryliśmy test, którego przeprowadzenie trwało dłużej niż dwie sekundy.

Mamy upodobanie do praktyk, w których niewielka zmiana może mieć znaczący wpływ na funkcjonowanie całego projektu. To jest właśnie tego rodzaju praktyka. Jeśli programista napisze test towarzyszący przekazywaniu zmian, którego przeprowadzenie zabiera zbyt dużo czasu, kompilacja się nie powiedzie, kiedy już będzie gotów zaewidencjonować zmianę. Zachęci go to do uważnego przemyślenia strategii zapewniających szybkie przeprowadzanie testów. Jeśli testy wykonywane są szybko, programiści będą częściej ewidencjonować zmiany. Jeśli programiści

będą częściej ewidencjonować zmiany, zmniejszy się szansa wystąpienia problemów podczas scalania, a problemy, które ewentualnie mogłyby się pojawić, będą raczej niewielkie i szybko poddające się rozwiązaniu, więc programiści będą bardziej wydajni.

W tym miejscu należy jednak zrobić pewne zastrzeżenie: tego rodzaju praktyka może się okazać mieczem obosiecznym. Musisz uważać, by nie stworzyć nieciągłych, fragmentarycznych testów, które zawiodą, gdy środowisko CI zostanie z jakiegoś powodu poddane nietypowym obciążeniom. Stwierdziliśmy, że najbardziej wydajną metodą zastosowania tego podejścia jest przyjęcie go jako strategii koncentrowania uwagi dużego zespołu na określonym problemie, a nie jako czegoś, co stosujemy przy każdej kompilacji. Jeśli kompilacje zaczynają przebiegać zbyt wolno, możesz zastosować to podejście do skoncentrowania uwagi zespołu przez jakiś czas, by przyspieszyć bieg spraw.

Zauważ, że mówimy tutaj o wydajności testów, a nie o testach wydajnościowych. Testy wydajnościowe omówimy w rozdziale 9., „Testowanie wymagań niefunkcjonalnych".

Odrzucanie kompilacji ze względu na ostrzeżenia i niewłaściwe formatowania kodu

Ostrzeżenia kompilatora zazwyczaj nie pojawiają się bez powodu. Strategia, którą stosujemy z powodzeniem, chociaż nasze zespoły programistyczne określają ją czasem mianem „koderskiego nazizmu", polega na odrzucaniu kompilacji opatrzonych ostrzeżeniami. W niektórych okolicznościach można to uznać za drakońskie środki, ale jest to skuteczne jako metoda egzekwowania dobrych praktyk.

Możesz dowolnie wzmocnić tę technikę, dodając testy analizujące kod pod kątem konkretnych lub bardziej ogólnych programistycznych potknięć. Udało nam się zastosować z powodzeniem jedno z wielu narzędzi open source sprawdzających jakość kodu.

- Simian jest narzędziem, które wykrywa duplikację w większości popularnych języków (również w zwykłym tekście).
- JDepend dla Javy i jego komercyjny kuzyn .Net o nazwie NDepend pozwala uzyskać mnóstwo użytecznych (oraz kilka mniej użytecznych) wskaźników jakości projektowania.
- CheckStyle wykrywa złe praktyki programistyczne, takie jak konstruktory publiczne w klasach narzędzi, zagnieżdżone bloki i długie linie. Potrafi również wychwycić powszechne źródła błędów i luki bezpieczeństwa. Może być bez trudu rozszerzany. Jego kuzyn dla .Net nosi nazwę FxCop.
- FindBugs to oparty na Javie system będący alternatywą dla CheckStyle, obejmujący podobny zestaw wskaźników potwierdzających jakość kodu.

Jak już powiedzieliśmy, w niektórych projektach odrzucanie kompilacji ze względu na występujące ostrzeżenia może się wydawać zbyt drakońskie. Jednym z podejść, które stosowaliśmy w celu stopniowego wprowadzenia tej praktyki, jest swoisty „mechanizm zapadkowy". Chodzi o porównywanie liczby takich elementow jak ostrzeżenia czy listy zadań z liczbą wcześniejszych zaewidencjonowań. Jeśli liczba ta się zwiększa, odrzucamy kompilację. Stosując to podejście, możesz z łatwością wymusić przestrzeganie zasady, że każda zaewidencjonowana zmiana powinna zmniejszyć liczbę ostrzeżeń lub list zadań do wykonania co najmniej o jedno.

CheckStyle — w ostatecznym rozrachunku warto się czepiać

W jednym z naszych projektów, w którym dodaliśmy test CheckStyle do naszego zestawu testów towarzyszących przekazywaniu zmian, wszyscy włącznie z nami po jakimś czasie byli już zmęczeni jego czepialstwem. Byliśmy jednak zespołem doświadczonych programistów i zgodziliśmy się gremialnie, że warto przez jakiś czas znosić to czepialstwo, abyśmy nabrali dobrych nawyków i rozwijali projekt na solidnych podstawach.

Po kilku tygodniach projektu usunęliśmy test CheckStyle. Przyspieszyliśmy w ten sposób rozwój systemu i pozbyliśmy się źródła ciągłego strofowania. Po jakimś czasie w zespole pojawiły się nowe osoby i kilka tygodni później zaczęliśmy wykrywać w kodzie coraz więcej „zgniłych jabłek", coraz więcej też czasu spędzaliśmy na prostym porządkowaniu kodu.

W końcu zdaliśmy sobie sprawę, że chociaż korzystanie z CheckStyle miało swoją cenę, pozwalał nam utrzymywać pod kontrolą elementy niemal całkowicie pozbawione znaczenia, które w sumie składały się jednak na różnicę między kodem wysokiej jakości a po prostu kodem. Przywróciliśmy CheckStyle i spędziliśmy nieco czasu na poprawianiu wskazanych przez niego potknięć, ale warto było. Przynajmniej w przypadku tego projektu nauczyliśmy się nie narzekać na to, że ciągle jesteśmy strofowani.

Zespoły rozproszone

Stosowanie ciągłej integracji w zespołach rozproszonych jest — w kategoriach procesu i technologii — mniej więcej tym samym, co w każdym innym środowisku, jednak fakt, że zespół nie siedzi razem w jednym pomieszczeniu, a może nawet nie przebywa w jednej strefie czasowej, rzeczywiście wywiera wpływ na niektóre obszary.

Podejściem najprostszym z perspektywy technicznej — i najbardziej efektywnym z perspektywy procesowej — jest zachowanie współdzielonego systemu kontroli wersji i systemu ciągłej integracji. Jeśli w Twoim projekcie wykorzystywane są potoki wdrożeń, jak zostało to opisane w kolejnych rozdziałach, one również powinny zostać w łatwy sposób udostępnione na równych zasadach wszystkim członkom zespołu.

Kiedy mówimy, że podejście to jest najbardziej efektywne, powinniśmy podkreślić, że jest znacznie bardziej efektywne niż wszystkie pozostałe. Warto ciężko pracować, by osiągnąć ten ideał. Wszystkie inne opisane tu podejścia są znacznie gorsze.

Wpływ na proces

W przypadku zespołów rozproszonych pracujących w tej samej strefie czasowej ciągła integracja przebiega tak samo. Nie możesz oczywiście używać fizycznych „bonów na zaewidencjonowanie" — chociaż niektóre serwery CI obsługują tokeny wirtualne — i jest to nieco bardziej bezosobowe, więc łatwiej jest kogoś obrazić, gdy przypominasz mu o konieczności poprawienia kompilacji. Bardziej przydatne stają się funkcjonalności takie jak osobiste kompilacje. Ogólnie rzecz biorąc, proces wygląda jednak tak samo.

W przypadku zespołów rozproszonych pracujących w różnych strefach czasowych rozwiązania wymaga więcej kwestii. Jeśli zespół pracujący w San Francisco zepsuje kompilację i pójdzie do domu, może to poważnie utrudnić pracę zespołowi pracującemu w Pekinie, który zaczyna pracę właśnie wtedy, gdy San Francisco wychodzi z biura. Proces nie ulega zmianie, ale wzrasta waga przestrzegania ustalonych zasad.

W dużych projektach obsługiwanych przez zespoły rozproszone ogromnego znaczenia nabierają narzędzia takie jak VoIP (np. Skype) i komunikatory, gdyż umożliwiają precyzyjną komunikację niezbędną do tego, by sprawy toczyły się gładko. Wszystkie osoby związane z rozwojem oprogramowania — menedżerowie projektów, analitycy, programiści, testerzy — powinny mieć dostęp do wszystkich pozostałych — i być dostępne dla nich — za pośrednictwem komunikatorów i VoIP. Aby proces dostarczania oprogramowania przebiegał gładko, bardzo ważne jest, by członkowie zespołu od czasu do czasu spotykali się twarzą w twarz. W ten sposób członkowie lokalnego zespołu mają szansę nawiązania osobistego kontaktu z członkami pozostałych zespołów. To ważne, by zbudować zaufanie między członkami zespołów, gdyż najczęściej w zespołach rozproszonych brakuje przede wszystkim zaufania. Za pomocą systemów wideokonferencyjnych można przeprowadzać retrospektywy, prezentacje i inne regularne spotkania. Inną świetną techniką jest poproszenie każdego zespołu programistycznego o nagranie z pomocą oprogramowania pozwalającego zrzucać zawartość ekranu krótkiego materiału wideo, w którym omawiają funkcjonalność opracowaną danego dnia.

Oczywiście jest to temat znacznie wykraczający poza ciągłą integrację. Chcieliśmy tylko podkreślić, że należy zachować zasadniczo ten sam proces, natomiast jeszcze staranniej dbać o jego rygorystyczną realizację.

Scentralizowana ciągła integracja

Niektóre posiadające większe możliwości serwery ciągłej integracji mają takie funkcje jak centralnie zarządzane farmy kompilacji i wyszukane schematy autoryzacji pozwalające udostępnić ciągłą integrację jako scentralizowaną usługę dla dużych, rozproszonych zespołów. Systemy te ułatwiają zespołom samoobsługową ciągłą integrację bez konieczności pozyskiwania własnego sprzętu. Pozwalają również zespołom eksploatacji systemów IT na konsolidację zasobów serwerowych oraz kontrolowanie konfiguracji środowisk ciągłej integracji i środowisk testowych, by zapewnić spójność i podobieństwo ich wszystkich do środowiska produkcyjnego oraz wymusić stosowanie dobrych praktyk, takich jak zarządzanie konfiguracją bibliotek firm trzecich i dostarczanie preinstalowanych narzędzi służących gromadzeniu spójnych wskaźników pokrycia i jakości kodu. Pozwalają wreszcie na gromadzenie i monitorowanie standardowych wskaźników w całym projekcie, umożliwiając menedżerom i zespołowi projektowemu tworzenie tablic wskaźników służących monitorowaniu jakości kodu na poziomie programu.

Przydatna może być również wirtualizacja w połączeniu ze scentralizowanymi usługami ciągłej integracji, umożliwiając postawienie nowych maszyn wirtualnych z zapisanych obrazów stanu odniesienia przyciśnięciem jednego guzika. Możesz się posłużyć wirtualizacją, by całkowicie zautomatyzować proces dostarczania nowych środowisk, który dzięki temu może być w pełni obsługiwany samodzielnie przez zespół projektowy. Dzięki temu kompilacje i wdrożenia działają zawsze na spójnej wersji odniesienia tych środowisk. Szczęśliwie prowadzi to do pozbycia się środowisk ciągłej integracji będących „dziełami sztuki", w których na przestrzeni wielu miesięcy narosły oprogramowanie, biblioteki i ustawienia konfiguracji niemające już nic wspólnego z zawartością środowisk testowych i produkcyjnych.

Scentralizowana ciągła integracja może przysłużyć się wszystkim uczestnikom projektu. Aby to jednak osiągnąć, ważne jest zapewnienie zespołom programistycznym możliwości zautomatyzowanej samoobsługi w odniesieniu do nowych środowisk, konfiguracji, kompilacji i wdrożeń. Jeśli zespół musi wysłać kilka maili i poczekać kilka dni, by otrzymać nowe środowisko ciągłej integracji dla swojej ostatniej opublikowanej wersji, będzie sabotować cały ten proces, powracając do zwyczaju wykorzystywania zbędnych komputerów zainstalowanych pod biurkami. Tam właśnie będą prowadzić prawdziwą ciągłą integrację albo — co gorsza — nie będą prowadzić jej w ogóle.

Problemy techniczne

Zależnie od wyboru systemu kontroli wersji, w przypadku działającego w skali globalnej zespołu rozproszonego, którego poszczególne podzespoły połączone są wolnymi łączami, współdzielenie dostępu do systemu kontroli wersji oraz zasobów umożliwiających kompilację i testowanie może być dość bolesne.

Kiedy ciągła integracja działa dobrze, cały zespół regularnie wprowadza zmiany. To oznacza, że integracja z systemem kontroli wersji będzie utrzymywana na dość wysokim poziomie. Chociaż każda interakcja jest zazwyczaj stosunkowo niewielka w kategoriach liczby wymienionych bitów, to z powodu częstotliwości wprowadzania zmian i aktualizacji kiepska komunikacja zaczyna się negatywnie odbijać na produktywności. Warto zainwestować w zapewniające wystarczająco dużą przepustowość linie telekomunikacyjne łączące centra programistyczne. Warto również rozważyć przejście na rozproszone systemy kontroli wersji w rodzaju Git czy Mercurial, które pozwalają ludziom ewidencjonować nawet wówczas, gdy nie ma połączenia z tradycyjnie zaprojektowanym serwerem centralnym.

Rozproszona kontrola wersji: kiedy nic innego nie zadziała

Kilka lat temu pracowaliśmy przy projekcie, w którym to właśnie było problemem. Infrastruktura telekomunikacyjna łącząca nas ze współpracownikami w Indiach była tak powolna i zawodna, że zdarzały się dni, kiedy nie mogli w ogóle ewidencjonować zmian, co w kolejnych dniach powodowało efekt domina. W końcu przeprowadziliśmy analizę traconego czasu i stwierdziliśmy, że koszt poprawienia telekomunikacji zwróci się w ciągu kilku dni. W innym projekcie uzyskanie wystarczająco szybkiego i niezawodnego połączenia było po prostu niemożliwe. Zespół przesiadł się z Subversion, scentralizowanego systemu kontroli wersji, na Mercurial, rozproszony system kontroli wersji, co zaowocowało znacznym zwiększeniem produktywności.

Uruchomienie systemu kontroli wersji w pobliżu infrastruktury kompilacyjnej, na której przeprowadzane są zautomatyzowane testy, ma dużo sensu. Jeśli testy te są przeprowadzane po każdym zaewidencjonowaniu, pomiędzy oboma systemami za pośrednictwem sieci przesyłana będzie znacząca ilość danych.

Fizyczne maszyny, na których zainstalowany jest system kontroli wersji, system ciągłej integracji oraz rozmaite środowiska testowe w potoku wdrożeń, muszą być na równi dostępne z każdego centrum rozwoju. Zespół programistyczny w Londynie będzie miał znacznie utrudnione zadanie, jeśli system kontroli wersji pracujący w Indiach przestanie działać wskutek tego, że dysk się zapełnił i wszyscy pracownicy hinduskiego biura poszli już do domu, a londyńczycy nie mają dostępu do systemu. Zapewnij zespołom we wszystkich lokalizacjach uprawnienia administracyjne dla każdego z wymienionych wcześniej systemów. Upewnij się, że zespoły w każdej lokalizacji mają nie tylko dostęp, ale i wiedzę konieczną do rozwiązania problemów, które mogą się pojawić na ich zmianie.

Podejścia alternatywne

Jeśli pojawi się jakiś problem nie do pokonania uniemożliwiający wydanie nieco większej sumy na łącze o większej przepustowości łączące Twoje centra rozwojowe, to wówczas możliwe jest — choć nie jest to sytuacja idealna — uruchomienie lokalnie systemów ciągłej integracji i systemów testowania, a w sytuacji skrajnej nawet lokalnych systemów kontroli wersji. Jak pewnie się spodziewasz, nie zalecamy tak naprawdę tego podejścia. Zrób wszystko, co w Twojej mocy, by

tego uniknąć. Jest to czaso- i pracochłonne i nie pozwala uzyskać rezultatów choćby zbliżonych do efektów współdzielenia dostępu.

Najprostsza jest sprawa z systemem ciągłej integracji. Całkiem możliwe jest uruchomienie lokalnych serwerów ciągłej integracji i środowisk testowych, a nawet pełnowymiarowego lokalnego potoku wdrożeń. Może być to cenne, gdy na miejscu wykonanych jest sporo ręcznych testów. Oczywiście środowiska te muszą być starannie zarządzane, by zapewnić ich spójność w ramach regionu. Należałoby jedynie zastrzec, że idealnie by było, gdyby pliki binarne lub instalatory były kompilowane jednokrotnie, a następnie wysyłane do wszystkich lokalizacji na całym świecie, w których są potrzebne. Często jest to jednak niepraktyczne ze względu na samą wielkość większości instalatorów. Jeśli musisz skompilować pliki binarne lub instalatory lokalnie, jeszcze większego znaczenia nabiera zapewnienie rygorystycznego zarządzania konfiguracją Twojego zestawu narzędzi, by zapewnić tworzenie wszędzie identycznych plików binarnych. Jednym z podejść umożliwiających wyegzekwowanie tego jest automatyczne generowanie skrótów za pomocą MD5 lub podobnego algorytmu i polecenie serwerowi CI automatycznego porównywania ich ze skrótami „oryginalnych" plików binarnych, by upewnić się, że są identyczne.

W pewnych sytuacjach skrajnych, na przykład wówczas, gdy system kontroli wersji działa zdalnie za pośrednictwem wolnego, zawodnego łącza, wartość uruchomienia lokalnie systemu ciągłej integracji jest poważnie nadszarpnięta. Często przywoływanym przez nas celem zastosowania ciągłej integracji jest możliwość wykrycia problemów przy pierwszej nadarzającej się okazji. Jeśli system kontroli wersji jest podzielony w jakikolwiek sposób, wtedy w mniejszym lub większym stopniu odbieramy sobie tę możliwość. Jeśli okoliczności nas do tego zmuszają, naszym celem przy okazji rozdzielania systemu kontroli wersji musi być minimalizacja czasu upływającego od wprowadzenia błędu do jego wykrycia.

Przede wszystkim istnieją dwie możliwości zapewnienia lokalnego dostępu do systemu kontroli wersji dla zespołów rozproszonych: podział aplikacji na moduły i zastosowanie systemów kontroli wersji, które są albo rozproszone, albo obsługują topologie z wieloma serwerami głównymi.

Przy podejściu opartym na modułach zarówno repozytoria kontroli wersji, jak i zespoły podzielone są albo według modułów, albo według rozgraniczenia funkcjonalnego. Podejście to omówimy znacznie bardziej szczegółowo w rozdziale 13., „Zarządzanie modułami i zależnościami".

Inną techniką, z którą się spotkaliśmy, jest posiadanie repozytoriów zespołów lokalnych i systemów kompilacji ze współdzielonym globalnym repozytorium nadrzędnym. Podzielone funkcjonalnie zespoły wprowadzają swoje zmiany do swoich repozytoriów lokalnych w ciągu całego dnia pracy. Każdego dnia o oznaczonej porze, zazwyczaj gdy jeden z zespołów rozproszonych działający w innej strefie czasowej kończy pracę, jeden z członków lokalnego zespołu bierze na siebie odpowiedzialność za wprowadzenie wszystkich zmian w imieniu całego zespołu i mierzy się z problemami związanymi ze scaleniem całego zestawu zmian. Oczywiście jest to znacznie łatwiejsze, jeśli korzysta się z rozproszonego systemu kontroli wersji zaprojektowanego właśnie z myślą o tego rodzaju zadaniach. Rozwiązanie to nie jest jednak w żadnym razie idealne i widzieliśmy, jak żałosne potrafi przynieść efekty, kiedy dojdzie do pojawienia się konfliktów na etapie scalania.

Podsumowując: skuteczność wszystkich technik opisanych w tej książce została dowiedziona w zespołach rozproszonych pracujących przy wielu projektach. Tak naprawdę uważamy zastosowanie ciągłej integracji za jeden z dwóch lub trzech najważniejszych czynników warunkujących zdolność zespołów rozproszonych geograficznie do efektywnej współpracy. W pojęciu ciągłej integracji ważny jest przymiotnik „ciągła". Jeśli naprawdę nie ma innego wyjścia, możliwe są pewne strategie alternatywne, ale zalecamy zainwestowanie w łącze o odpowiedniej przepustowości — w średnim i długim okresie wychodzi taniej.

Rozproszone systemy kontroli wersji

Upowszechnianie się rozproszonych systemów kontroli wersji (DVCS) rewolucjonizuje sposób działania zespołów. W projektach open source łaty wysyłano kiedyś e-mailem lub publikowano na forach. Dziś narzędzia w rodzaju Git czy Mercurial niesamowicie ułatwiają wymianę łat pomiędzy zespołami i programistami oraz rozgałęzianie i scalanie owoców ich pracy. DVCS-y pozwalają Ci z łatwością pracować offline, lokalnie wprowadzać zmiany i scalać zmiany poprzez zmianę bazy lub agregować je, zanim zostaną przekazane pozostałym użytkownikom. Kluczową cechą DVCS-ów jest to, że każde repozytorium zawiera całą historię projektu, co oznacza, że żadne repozytorium nie jest uprzywilejowane (chyba że tradycją). A zatem w porównaniu z systemami scentralizowanymi systemy rozproszone mają dodatkową warstwę pośrednią: zmiany wprowadzane w lokalnej kopii roboczej muszą zostać zaewidencjonowane w lokalnym repozytorium, zanim będą mogły zostać przekazane do innych repozytoriów, a aktualizacje z innych repozytoriów muszą zostać uzgodnione z lokalnym repozytorium, zanim będziesz mógł zaktualizować swoją kopię roboczą.

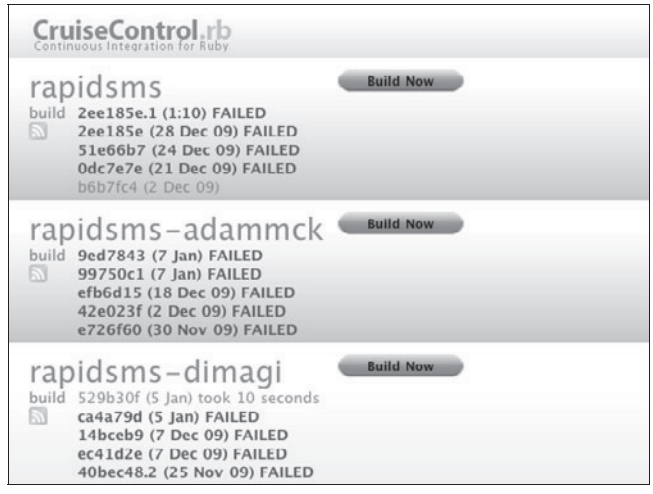
DVCS-y stwarzają nowe, wszechstronne możliwości współpracy. GitHub przykładowo jako pierwszy wprowadził nowy model współpracy w projektach open source. W tradycyjnym modelu osoby wprowadzające zmiany pełniły funkcję strażników określonego repozytorium projektu, akceptując lub odrzucając poprawki proponowane przez innych. Do rozgałęzienia projektu dochodziło tylko w sytuacjach skrajnych, kiedy pomiędzy współpracownikami pojawiały się konflikty nie do rozwiązania. W modelu GitHub postawiono to wszystko na głowie. Zmiany wprowadzane są w taki sposób, że najpierw rozgałęzia się repozytorium projektu, do którego chce się wprowadzić zmiany, wprowadza się zmiany, a następnie prosi się właściciela oryginalnego repozytorium o pobranie świeżo wprowadzonych zmian. W czynnie rozwijanych projektach dochodzi do gwałtownego przyrostu rozgałęzień, a każda gałąź zawiera odmienny zestaw nowych funkcji. Co jakiś czas rozgałęzienia się rozchodzą. Model ten jest znacznie bardziej dynamiczny niż model tradycyjny, w którym ignorowane poprawki marnieją w archiwach list mailingowych. W rezultacie tempo rozwoju jest zazwyczaj znacznie szybsze na GitHubie, a jednocześnie znacznie więcej osób bierze udział w projekcie.

Model ten jednak podważa fundamentalne założenie praktyki ciągłej integracji, zgodnie z którym istnieje jedna kanoniczna wersja kodu (określana zazwyczaj mianem gałęzi głównej projektu), do której wprowadzane są wszystkie zmiany. Należy zaznaczyć, że możesz zastosować ten bazowy model kontroli wersji i jednocześnie z powodzeniem prowadzić ciągłą integrację, korzystając z DVCS-a. Określasz po prostu jedno repozytorium jako główne i polecasz swojemu serwerowi CI wydać komunikat za każdym razem, gdy w tym repozytorium wprowadzona zostanie zmiana, a jednocześnie każesz wszystkim przekazywać wszystkie zmiany do tego repozytorium, aby je współdzielić. Jest to absolutnie rozsądne podejście, które z powodzeniem stosowaliśmy przy wielu projektach. Pozwala zachować wiele korzyści płynących z zastosowania DVCS-a, takich jak możliwość częstego ewidencjonowania zmian bez konieczności udostępniania ich innym (przypomina to zapisywanie gry), co okazuje się bardzo użyteczne, kiedy bada się możliwości zastosowania nowej koncepcji lub przeprowadza złożoną serię zabiegów refaktoryzacyjnych. Istnieją jednak pewne schematy zastosowania DVCS-a, które uniemożliwiają ciągłą integrację. Model GitHub dla przykładu narusza model współdzielenia kodu gałęzi głównej, a przez to uniemożliwia prawdziwą ciągłą integrację.

W GitHubie zestaw zmian każdego użytkownika istnieje w oddzielnym repozytorium i nie ma sposobu łatwego określenia, który zestaw pochodzący od którego użytkownika przejdzie z powodzeniem integrację. Możesz przyjąć podejście polegające na tworzeniu repozytorium

obserwującego wszystkie pozostałe repozytoria i starającego się scalić je wszystkie za każdym razem, gdy w którymkolwiek z nich wykryta zostanie zmiana. Podejście to zawodzi jednak prawie zawsze na etapie scalania, nie wspominając już o etapie zautomatyzowanych testów. Wraz ze wzrostem liczby programistów, a tym samym liczby repozytoriów, problem pogłębia się w postępie geometrycznym. Nikt nie będzie zwracał już uwagi na komunikaty serwera CI, a tym samym zawiedzie ciągła integracja rozumiana jako metoda komunikowania, że aplikacja działa poprawnie (a jeśli nie, to kto i co ponosi za to winę).

Można zdać się na prostszy model pozwalający na osiągnięcie niektórych korzyści ciągłej integracji. W tym modelu tworzysz kompilację CI dla każdego repozytorium. Po każdej zmianie próbujesz scalić zmiany z oznaczonego repozytorium pierwotnego i przeprowadzić kompilację. Rysunek 3.2 przedstawia CruiseControl.rb kompilujący główne repozytorium projektu Rapidsms oraz jego dwóch odgałęzień.



Rysunek 3.2. *Integracja gałęzi*

Aby stworzyć ten system, do każdego z repozytoriów Git CC.rb dodano gałąź wskazującą na główne repozytorium projektu, za pomocą polecenia `git remote add core git://github.com/rapidsms/rapidsms.git`. Za każdym razem, gdy uruchamiana jest kompilacja, CC.rb próbuje scalić i przeprowadzić kompilację:

```
git fetch core
git merge --no-commit core/master
[polecenie uruchamiające kompilację]
```

Po kompilacji CC.rb uruchamia `git reset --hard`, by zresetować lokalne repozytorium do nagłówka repozytorium, na które wskazuje. System ten nie zapewnia prawdziwej ciągłej integracji. *Rzeczywiście* mówi jednak osobom odpowiedzialnym za utrzymanie gałęzi — i osobom odpowiedzialnym za utrzymanie głównego repozytorium — czy zasadniczo ich gałąź może zostać scalona z głównym repozytorium i czy w rezultacie powstanie działająca wersja aplikacji. Co ciekawe, na rysunku 3.2 pokazano, że kompilacja głównego repozytorium jest obecnie popsuta, ale gałąź Dimagi nie tylko z powodzeniem można z nią scalić, ale również naprawia ona zepsute testy (i prawdopodobnie dodaje własną dodatkową funkcjonalność).

Tylko jeden krok dzieli od ciągłej integracji coś, co Martin Fowler określa mianem „rozwiązłej integracji" [bBjxbS]. W tym modelu programiści przekazują zmiany nie tylko pomiędzy repozytorium głównym a gałęziami, ale też między gałęziami. Wzorzec ten występuje często w większych projektach stosujących GitHub, kiedy to niektórzy programiści pracują nad utrzymywanymi przez długi czas gałęziami obejmującymi oddzielną funkcjonalność i pobierają zmiany z innych repozytoriów będących odgałęzieniem gałęzi tej funkcjonalności. W tym modelu tak naprawdę nie musi istnieć nawet jedno uprzywilejowane repozytorium. Konkretna wersja oprogramowania może pochodzić z dowolnej gałęzi, przyjmując, że przeszła wszystkie testy i została zaakceptowana przez liderów projektu. Model ten jest logicznym rezultatem pełnego wykorzystania możliwości stwarzanych przez DVCS.

Alternatywne podejścia do ciągłej integracji mogą zaowocować działającym oprogramowaniem wysokiej jakości. Możliwe jest to jednak tylko po spełnieniu poniższych warunków:

- Niewielki i bardzo doświadczony zespół programistów, którzy zarządzają pobieraniem łat, pilnują zautomatyzowanych testów i zapewniają wysoką jakość oprogramowania.
- Regularne pobieranie zmian z gałęzi mające na celu uniknięcie akumulacji zmian, które później trudno będzie scalić. Warunek ten ma szczególne znaczenie, jeśli przyjęliśmy rygorystyczny harmonogram realizacji projektu, ponieważ istnieje wówczas pokusa odwlekania scalenia aż do daty bliższej terminowi zaplanowanego udostępnienia oprogramowania, kiedy staje się to skrajnie kłopotliwe i bolesne — dokładnie ten problem ma z założenia rozwiązywać ciągła integracja.
- Stosunkowo nieliczny rdzeń zespołu programistycznego, którego działanie uzupełnia liczniejsza społeczność wnosząca swoje poprawki stosunkowo wolno. Pozwala to stosunkowo łatwo opanować scalanie.

Warunki te obowiązują w przypadku większości projektów open source oraz z zasady w przypadku niewielkich zespołów. Bardzo rzadko mają jednak zastosowanie w przypadku średnich i dużych zespołów programistów pracujących na pełen etat.

Podsumowując: z zasady rozproszone systemy kontroli wersji oznaczają duży postęp i dostarczają skutecznych narzędzi wspierających współpracę, niezależnie od tego, czy pracujesz w projekcie rozproszonym czy nie. DVCS-y mogą być bardzo efektywne jako część tradycyjnego systemu ciągłej integracji, w którym znajduje się określone repozytorium centralne, do którego wszyscy regularnie przesyłają swoje zmiany (przynajmniej raz dziennie). Mogą być również stosowane w innych schematach, które nie pozwalają na ciągłą integrację, ale mogą być mimo to skutecznymi modelami dostarczania oprogramowania. Nie zalecamy jednak stosowania tych modeli, o ile nie są spełnione odpowiednie, wymienione wyżej warunki. W rozdziale 14., „Zaawansowana kontrola wersji", zamieściliśmy pełne omówienie tych i innych modeli oraz warunków, których spełnienie zapewnia ich efektywność.

Podsumowanie

Gdybyś miał wybrać z tej książki tylko jedną praktykę i wdrożyć ją w zespole programistycznym, naszym zdaniem powinieneś zdecydować się na ciągłą integrację. Wielokrotnie obserwowaliśmy, jak radykalnie podniosło to produktywność zespołów programistycznych.

Zastosowanie ciągłej integracji w praktyce równa się zmianie paradygmatu Twojego zespołu. Bez ciągłej integracji aplikacja nie działa, dopóki nie dowiedziesz, że jest inaczej. Przy ciągłej integracji poprawne działanie jest domyślnym statusem aplikacji, chociaż pewność co do tego ograniczona jest zakresem pokrycia zautomatyzowanych testów. CI dostarcza ustawicznej

informacji zwrotnej, która pozwala wykryć problemy niemal natychmiast po ich wprowadzeniu, kiedy ich poprawienie nie pociąga za sobą wysokich kosztów.

Wdrożenie ciągłej integracji zmusza Cię do stosowania dwóch innych ważnych praktyk: dobrego zarządzania konfiguracją oraz stworzenia i utrzymywania zautomatyzowanego procesu kompilacji i testowania. Niektórym zespołom będzie się to wydawać porywaniem się z motyką na słońce, ale zmiany można wprowadzać stopniowo. W poprzednim rozdziale rozważaliśmy kroki prowadzące do dobrego zarządzania konfiguracją. W rozdziale 6., „Skrypty kompilacji i wdrożenia", znajdziesz więcej na temat automatyzacji kompilacji. Testowanie omawiamy bardziej szczegółowo w następnym rozdziale.

Powinno być jasne, że ciągła integracja wymaga dyscypliny całego zespołu, ale ostatecznie wymaga tego każdy proces. Różnica w przypadku ciągłej integracji polega na tym, że dysponujesz prostym wskaźnikiem pokazującym, czy dyscyplina jest zachowywana czy nie: kompilacja jest cały czas poprawna. Jeśli odkryjesz, że kompilacja jest poprawna, ale brakuje dyscypliny (przykładowo pokrycie testami jednostkowymi wydaje się niewystarczające), możesz z łatwością dodać kolejne testy do swojego systemu CI, by wymusić lepsze zachowanie.

W ten sposób doszliśmy do ostatniego punktu. Ustabilizowany system CI jest podstawą, na której możesz budować kolejne elementy infrastruktury:

- duże widoczne ekrany gromadzące informacje z systemu kompilacji w celu dostarczenia wysokiej jakości informacji zwrotnej;
- system informacji źródłowej będącej źródłem danych dla raportów i instalatorów dla zespołu testowego;
- system dostarczania danych na temat jakości aplikacji dla menedżerów projektu;
- system, który można rozszerzyć na produkcję, wykorzystujący potok wdrożeń, który umożliwia testerom i zespołowi eksploatacji systemów IT wdrożenia za naciśnięciem jednego przycisku.

Skorowidz

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

Z

WYDAWANIE APLIKACJI JESZCZE NIGDY NIE BYŁO TAK PROSTE!

Jeśli Twoja praca wymaga, byś dostarczał oprogramowanie w sposób niemalże ciągły, a Ty chciałbyś uniknąć niepowodzeń, pomogą Ci w tym: automatyczna kompilacja, testowanie i wdrażanie. Dzięki nim możesz zaoferować użytkownikom aplikację najwyższej jakości w dowolnym czasie!

W tej książce znajdziesz instrukcje, jak zrealizować ten cel.Na samym początku poznasz typowe problemy z wdrażaniem oprogramowania, a w kolejnych rozdziałach zobaczysz, jak je rozwiązać. Zaczniesz od najlepszych technik zarządzania konfiguracją aplikacji, a następnie przejdziesz do zagadnień związanych z ciągłą integracją. Po tym wstępie czeka Cię niezwykle pasjonująca lektura dotycząca potoku wdrożeń oraz tworzenia skryptów automatyzujących proces tworzenia i budowania projektu. Ponadto zapoznasz się z detalami automatycznych testów akceptacyjnych i testów wymagań niefunkcjonalnych oraz zrozumiesz, jak stworzyć strategię udostępniania oprogramowania. Dla powodzenia całego przedsięwzięcia kluczowe jest zbudowanie ekosystemu wydawania oprogramowania. Ten temat został obszernie omówiony w trzeciej części książki. Jeżeli chcesz zmienić sposób wydawania Twojego oprogramowania, przeczytaj ten podręcznik!

Dzięki tej książce:

- poznasz najczęstsze problemy występujące podczas wdrażania oprogramowania
- zobaczysz, jak automatyzować poszczególne etapy tworzenia oprogramowania
- zbudujesz doskonały ekosystem wydawania oprogramowania
- wydasz Twoją aplikację

Addison-Wesley
Pearson Education

Helion

28614 numer katalogowy
księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-246-9918-6
9 788324 699186

cena: 69,00 zł