Spis treści

1 Wprowadzenie

Oprogramowanie jest wszędzie. Właściwie każdy złożony wyrób produkowany obecnie jest sterowany przez oprogramowanie, a wiele usług komercyjnych opiera się na systemach informatycznych. Jakość wykorzystywanego oprogramowania jest więc istotnym czynnikiem dla bycia konkurencyjnym. Im szybciej firma będzie mogła zintegrować nowe oprogramowanie w swoich produktach lub dostarczyć produkty programowe na rynek, tym większą będzie miała okazję do pokonania konkurencji.

Metody programowania zwinnego (*agile*) zaprojektowano w celu ograniczenia czasu dostarczenia produktu na rynek i poprawienia jakości oprogramowania przy jednoczesnym zwiększeniu dopasowania produktów do potrzeb klientów. Nie jest zaskoczeniem, że zastosowanie metodyki zwinnej staje się coraz popularniejsze w dużych projektach międzynarodowych, a także w rozwijaniu produktów w różnego rodzaju wielkich korporacjach. W większości przypadków oznacza to przejście z wypróbowanego modelu V do testowania zwinnego przy użyciu metodyki Scrum.

Jednakże przejście na programowanie zwinne i nauczenie się korzystania z niego w sposób efektywny i wydajny nie zawsze jest łatwe, zwłaszcza gdy zaangażowanych jest wiele zespołów. Każdy członek zespołu, menedżer projektu i wszyscy członkowie zarządzający muszą dokonać znaczących zmian w swoim sposobie pracy. W szczególności skuteczność testowania oprogramowania i zarządzania jakością stanowi istotny czynnik sukcesu przy wprowadzaniu i wykorzystywaniu metodyki zwinnej w zespole, dziale lub całym przedsiębiorstwie oraz będzie wpływać na potencjalną możliwość zastosowania jej w praktyce.

Większość literatury na temat metodyki zwinnej (w tym wiele książek podanych w bibliografii na końcu tej książki) napisano z punktu widzenia programistów i inżynierów oprogramowania. Kładą one główny nacisk na techniki programistyczne i zwinne zarządzanie projektami – testowanie jest zwykle jedynie wspominane przy omawianiu testów jednostkowych i związanych z nimi narzędzi. Innymi słowy skupiają się na testowaniu z punktu widzenia programisty. Jednakże

same testy jednostkowe nie są wystarczające i szersze ujęcie testowania jest istotne dla sukcesu procesów programowania zwinnego.

Celem tej książki jest wypełnienie tej luki i opisanie procesu programowania zwinnego z punktu widzenia testowania i zarządzania jakością. Przedstawia ona, jak działa testowanie zwinne i pokazuje, gdzie tradycyjne techniki testowania są nadal konieczne w środowisku zwinnym oraz jak je wkomponować w podejście zwinne.

1.1 Grupa docelowa

Zrozumienie, jak działa testowanie w projektach zwinnych

Z jednej strony książka ta jest przeznaczona dla początkujących, którzy dopiero zaczynają pracę z metodyką zwinną, mają w planach pracę nad projektami zwinnymi lub zamierzają wprowadzić (lub właśnie wprowadzają) Scrum do swojego projektu lub zespołu.

- Zapewniamy wskazówki i porady, w jaki sposób menedżerowie zarządzający produktem, menedżerowie projektów, menedżerowie testów i menedżerowie jakości mogą pomóc w wykorzystaniu pełni potencjału metodyki zwinnej.
- Profesjonalni (certyfikowani) testerzy i eksperci od zarządzania jakością dowiedzą się, jak wydajnie pracować w zespołach zwinnych i optymalnie wykorzystywać swoje doświadczenie. Można się też dowiedzieć, jak dostosować swój styl pracy tak, aby dopasować go do środowiska zwinnego.

Wzbogacanie swojej wiedzy na temat (zautomatyzowanego) testowania i zwinnych technik zarządzania jakością

Z drugiej strony kierujemy tę książkę również do czytelników, którzy mają już doświadczenie w pracy w zespołach zwinnych i chcą poszerzyć swoją wiedzę na temat testowania i technik zarządzania jakością w celu zwiększenia wydajności i poprawienia jakości produktów.

- Menedżerowie produktów, mistrzowie Scrum (Scrum Master), pracownicy zarządzania jakością i członkowie zespołów zarządzających dowiedzą się, jak działa systematyczne, zautomatyzowane testowanie i jakie jest znaczenie zapewnienia stałych, niezawodnych i wyczerpujących informacji zwrotnych na temat jakości tworzonego oprogramowania.
- Programiści, testerzy i inni członkowie zespołu zwinnego dowiedzą się, jak stosować wysoce zautomatyzowane metody testowania do przeprowadzania testów jednostkowych, integracyjnych i systemowych.

Tekst tej książki zawiera wiele praktycznych przykładów i pytań testowych, co sprawia, że dobrze nadaje się ona na podręcznik lub samouczek.

1.2 Zawartość książki

Rozdział 2

Rozdział 2 zapewnia krótkie omówienie popularnych obecnie metodyk Scrum i Kanban oraz omówienie najważniejszych metod zarządzania zwinnego dla menedżerów zamierzających wdrażać metody zwinne w swoich departamentach. Metody te są porównywane z metodami tradycyjnymi, żeby pokazać zmiany wprowadzane przez metody zwinne. Czytelnicy, którzy są już zaznajomieni z metodyką Scrum i Kanban, mogą pominąć ten rozdział.

Rozdział 3

Rozdział 3 opisuje proste narzędzia planowania i sterowania wykorzystywane przez Scrum zamiast tradycyjnych narzędzi planowania projektów. Warto pamiętać, że praca w sposób zwinny (*agile*) nie oznacza pracy bez wyznaczonego celu! Rozdział 3 jest też przeznaczony głównie dla czytelników, którzy przechodzą na programowanie zwinne. W każdym razie wyjaśnienia i wskazówki dotyczące znaczenia narzędzi planowania dla konstruktywnego zarządzania jakością i zapobiegania błędom będą przydatne również dla zaawansowanych czytelników.

Rozdział 4

Rozdział 4 obejmuje testowanie jednostkowe i techniki programowania opartego na testach. Omówione zostanie między innymi, czego można oczekiwać od testów jednostkowych i jak je można zautomatyzować. Ten rozdział omawia podstawy technik i narzędzi testowania jednostkowego oraz pomoże testerom systemowym, specjalistom od testowania i członkom zespołów projektowych mającym niewielkie doświadczenie w testowaniu jednostkowym w bliższej i skuteczniejszej współpracy z programistami i testerami jednostkowymi. Zawiera też sporo przydatnych wskazówek, które pomogą zaawansowanym programistom i testerom w poprawieniu swoich technik testowania. Wyjaśnimy też oparte na testach podejście do programowania i jego znaczenie w projektach zwinnych.

Rozdział 5

Rozdział 5 omawia testowanie integracyjne i pojęcie ciągłego testowania integracyjnego. Testowanie integracyjne obejmuje przypadki testów, które są pomijane nawet przez najdokładniejszych programistów przy korzystaniu z testów jednostkowych. Ten rozdział obejmuje wszystkie podstawy integracji oprogramowania i testów integracyjnych oraz wprowadza techniki ciągłej integracji wyjaśniając, jak korzystać z nich w projektach.

Rozdział 6

Rozdział 6 omawia testowanie systemowe i pojęcie testowania non-stop. W oparciu o podstawy testowania systemowego rozdział ten wyjaśnia, jak korzystać z technik zwinnych do przeprowadzania testów ręcznych. Wyjaśnia też, jak automatyzować testy systemowe i umieszczać je w procesie ciągłej integracji. Ten rozdział jest przeznaczony nie tylko dla testerów systemowych i innych specjalistów od testowania, ale również dla programistów, którzy chcą lepiej zrozumieć techniki

testowania zwinnego, które znajdują się poza ich zwykłymi kompetencjami programistycznymi.

Rozdział 7

Rozdział 7 porównuje tradycyjne i zwykłe sposoby zarządzania jakością oraz wyjaśnia konstruktywne i profilaktyczne praktyki zarządzania jakością, które są wbudowane w Scrum. Ten rozdział zawiera wskazówki, jak przeprowadzać zwinne zarządzanie jakością i wyjaśnia, jak wszyscy specjaliści zarządzania jakością mogą wykorzystać swoją wiedzę w projektach zwinnych i przyczynić się do ich sukcesu.

Rozdział 8

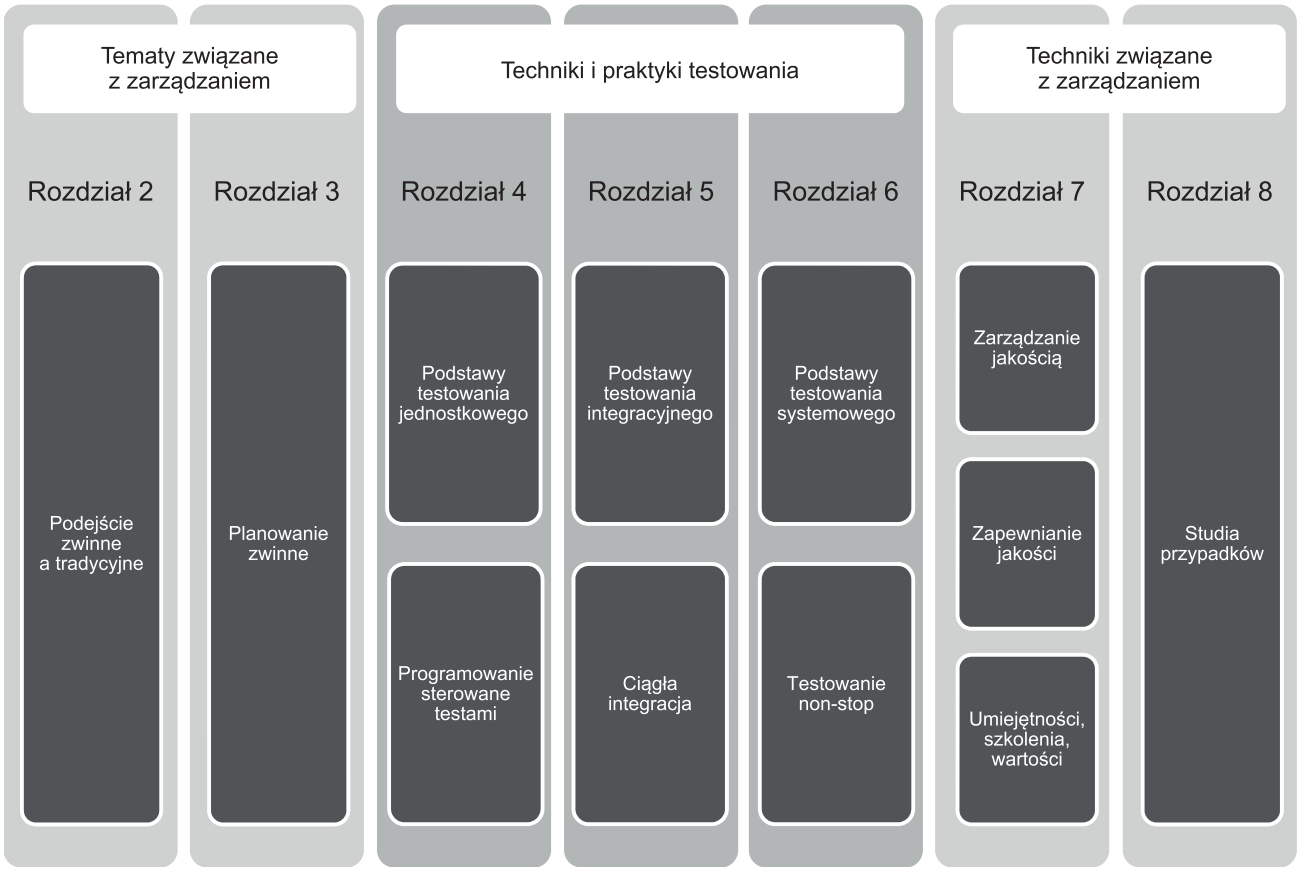
Rozdział 8 przedstawia sześć przemysłowych studiów przypadku z dziedziny programowania. Odzwierciedlają one nabyte doświadczenie i wnioski zebrane przez respondentów podczas wprowadzania i stosowania technik zwinnych.

Omówienie rozdziałów

Rozdziały 2, 3, 7 i 8 omawiają zagadnienia związane z procesami oraz zarządzaniem i są przeznaczone dla czytelników, którzy są bardziej zainteresowani zarządczymi aspektami tej tematyki. Rozdziały 4, 5 i 6 omawiają (zautomatyzowane) testowanie zwinne na różnych poziomach i są przeznaczone dla bardziej technicznie nastawionych czytelników. Ponieważ testowanie zwinne nie jest tym samym, co testowanie jednostkowe, wyjaśniamy też szczegółowo cele i różnice pomiędzy testowaniem jednostkowym, integracyjnym i systemowym.

Rys. 1–1
Struktura książki Studium przypadku, podsumowanie i ćwiczenia

Rysunek 1-1 przedstawia wizualny przegląd struktury tej książki:



Według najpopularniejszej literatury na ten temat wiele zagadnień, technik i praktyk programowania zwinnego jest prostych w implementacji. Zagadnienia, porady i wskazówki omówione w kolejnych rozdziałach również mogą na pierwszy rzut oka wydawać się proste, ale najważniejsze kwestie staną się jasne dopiero w trakcie implementacji. Żeby pomóc w zrozumieniu wyzwań z tym związanych, tekst zawiera:

- Fikcyjne studium przypadku ilustrujące omawiane metodyki i techniki.
- Testy i ćwiczenia pomagające zapamiętać treści omawiane w każdym rozdziale i przeanalizować swoją własną sytuację i zachowanie w ramach projektu.

1.3 Studium przypadku

Fikcyjne studium przypadku, do którego odwołuje się ta książka, korzysta z następującego scenariusza. Firma eHome Tools jest twórcą systemów automatyki domowej opartych na następujących elementach:

Studium przypadku: eHome Tools

- **Elementy wykonawcze:** Lampy i inne urządzenia elektryczne są przyłączane do systemu przy użyciu przełączników elektrycznych. Każdy element wykonawczy jest połączony przewodem lub bezprzewodowo z magistralą komunikacyjną, która może służyć do sterowania nim zdalnie.
- **Czujniki:** Czujniki temperatury, wiatru i wilgotności oraz proste czujniki styku (na przykład wskazujące na otwarte okno) również mogą być przyłączone do magistrali.
- **Magistrala:** Instrukcje przełączające i sygnały o stanie dla elementów wykonawczych i czujników są przesyłane przez magistralę do centralnego sterownika w formie „telegramów".
- **Sterownik:** Centralny sterownik wysyła sygnały przełączające do elementów wykonawczych (np. „włącz światło w kuchni") i odbiera sygnały o stanie z czujników (np. „temperatura w kuchni 20 stopni") i elementów wykonawczych (np. „światło w kuchni jest włączone"). Sterownik jest albo sterowany zdarzeniami (tzn. reaguje na nadchodzące sygnały, albo działa w oparciu o zaplanowane polecenia (np. „o 8 wieczorem zasuń żaluzje w kuchni").
- **Interfejs użytkownika:** Sterownik ma interfejs użytkownika, który wizualizuje aktualny stan wszystkich elementów elektronicznego domu i umożliwia mieszkańcom wysyłanie poleceń (np. „wyłącz światło w kuchni") dla systemów elektronicznego domu.

Firma eHome Tools ma wielu konkurentów i aby pozostać konkurencyjną zamierza opracować nowe oprogramowanie sterownika. Coraz większa liczba klientów prosi o oprogramowanie, którym można

sterować za pośrednictwem smartfonów i innych urządzeń mobilnych, od początku jest więc jasne, że projekt ten musi być ukończony szybko, jeśli ma się udać. Ważne jest, aby nowy system był łatwy do rozszerzania i otwarty na urządzenia firm trzecich. Jeśli nowy system będzie w stanie kierować urządzeniami innych producentów, to w odczuciu zarządu firma może zdobyć klientów, którzy chcą rozszerzać swoje istniejące systemy. Aby osiągnąć ten cel, nowy system musi obsługiwać możliwie szeroki zakres istniejącego sprzętu innych firm i musi być w stanie przystosowywać się do obsługi nowych urządzeń, jak tylko będą się one pojawiać na rynku.

W obliczu tych wyzwań podjęta zostaje decyzja o opracowaniu nowego systemu przy użyciu metodyki zwinnej i wypuszczaniu zaktualizowanej wersji oprogramowania sterownika (z obsługą nowych urządzeń i protokołów) co miesiąc.

1.4 Strona WWW

Przykłady kodu zawarte w tekście są dostępne do pobrania ze strony WWW tej książki [URL: SWT-knowledge][1]. Można z nich swobodnie korzystać do tworzenia swoich własnych przypadków testowych.

Pytania i ćwiczenia są również dostępne w sieci i można je komentować. Chętnie omówimy komentarze i sugestie czytelników.

Pomimo podjęcia wszelkich wysiłków ze strony autora i wydawcy, możliwe, że tekst zawiera jakieś błędy. Wszelkie niezbędne poprawki będą publikowane online.

1 Spis odnośników internetowych i innych źródeł bibliograficznych znajduje się w Dodatku B (przyp. red.).

2 Podejście zwinne a tradycyjne

Ten rozdział przedstawia platformę zarządzania zwinnym projektem Scrum oraz popularną metodykę zarządzania projektami Kanban. Kanban został rozwinięty na bazie teorii szczupłej produkcji i wykazuje pewne podobieństwa do Scrum. Porównamy obie te metody z tradycyjnymi modelami procesów. Osobom zaangażowanym w zarządzanie przejściem do metod zwinnych na poziomie projektu lub całego działu rozdział ten da ogólne wyobrażenie zmian potrzebnych na poziomie przedsiębiorstwa, działu i zespołu projektowego. Osoby zaznajomione z działaniem Scrum i/lub Kanban mogą pominąć ten rozdział.

2.1 Scrum

Scrum jest platformą zwinnego[1] zarządzania projektami wprowadzoną po raz pierwszy w 1999 roku przez Kena Schwabera w artykule *Scrum: A Pattern Language for Hyperproductive Software Development* [Beedle i in. 99]. Wydana w 2002 roku książka *Agile Software Development with Scrum* [Schwaber/Beedle 02] pomogła metodyce Scrum dotrzeć do szerszej publiczności.

Scrum nie określa, jakie konkretne techniki powinny być stosowane przez twórców oprogramowania (np. refaktoring), natomiast pozostawia takie decyzje samemu zespołowi programistów. Zwykle prowadzi to do wykorzystania technik programowania ekstremalnego[2] (XP),

1 Termin zwinne (agile) jest stosowany do opisywania lekkich procesów rozwijania oprogramowania, które znacznie różnią się od tradycyjnych, „ciężkich" metod. Termin ten został ukuty na konferencji programistycznej w Utah w 2001, gdzie powstał szkic manifestu programowania zwinnego (*Agile Manifesto*) (zobacz [URL: Agile Manifesto]).

2 Programowanie ekstremalne (Extreme Programming – XP) jest luźnym zbiorem wartości, zasad i praktyk programowania zwinnego wprowadzonym przez Kenta Becka w roku 1999. Większość procesów programowania zwinnego opiera się na elementach XP. Aktualna wersja XP jest wyjaśniona przez Becka w [Beck/Andres 04].

które są wprowadzane podczas przechodzenia na metodykę Scrum. Ponadto Scrum nie dyktuje, jakim rodzajom testów ma być poddawany zwinny projekt.

Zwinne praktyki w zarządzaniu

Platforma Scrum opisuje zestaw praktyk zarządzania projektami (programistycznymi), które radykalnie zmieniają wykorzystywane procesy i zastępują tradycyjne podejście planowania deterministycznego przez adaptacyjny, empiryczny system sterowania projektem [Schwaber/Beedle 02]. Głównym celem wykorzystania Scrum jest umożliwienie zespołowi projektowemu szybkiego, prostego i właściwego reagowania zamiast tracenia czasu i energii na tworzenie, implementowanie i poprawianie nieaktualnych planów.

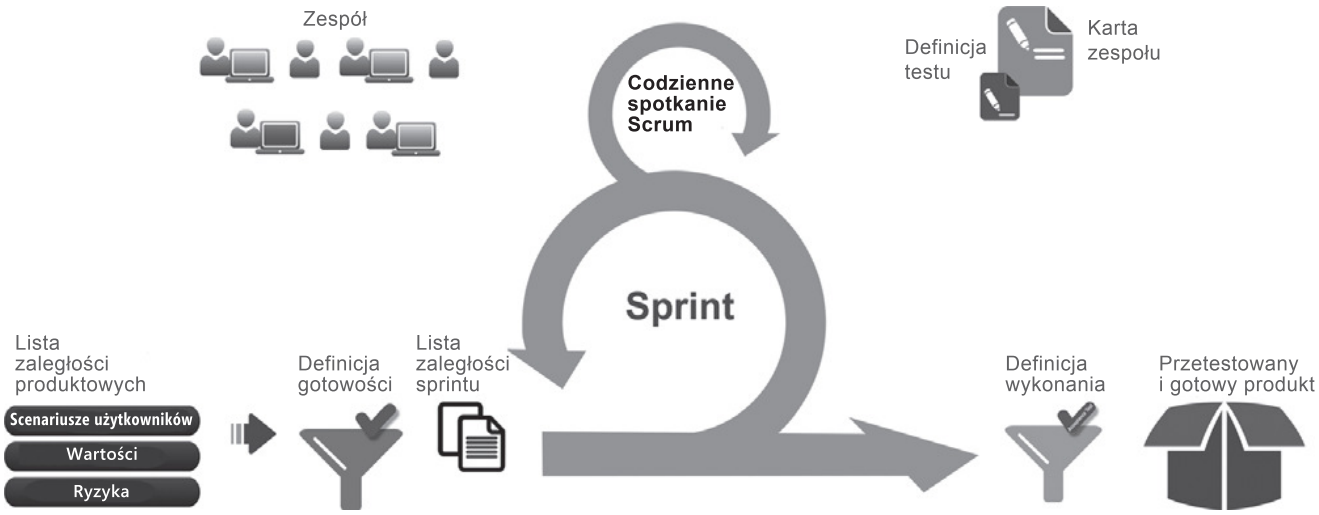
Podstawowymi instrumentami zarządzania projektami wykorzystywanymi przez Scrum są:

- **Sprint:** Scrum dzieli projekt na krótkie iteracje o ustalonej długości zwane sprintami[3]. Chodzi o to, że sprint trwający trzy lub cztery tygodnie można planować i zarządzać nim prościej i skutecznej niż tradycyjnymi cyklami wypuszczania oprogramowania trwającymi rok lub więcej (zobacz [Schwaber/Beedle 02, str. 52]).
- **Przyrost produktowy:** Produkt rośnie z każdą iteracją, więc każdy sprint daje w wyniku działający i potencjalnie nadający się do wypuszczenia/sprzedaży produkt (zwany przyrostem).
- **Zaległość produktowa:** Jeśli planowanie jest ograniczone do jednowymiarowej listy celów, projekt staje się znacznie mniej skomplikowany, a dokładnie o to chodzi w metodyce Scrum. Właściciel produktu (zobacz poniżej) utrzymuje listę tzw. zaległości produktowych, która według [Pichler 10] zawiera wszystkie znane wymagania i wyniki, które mają zostać zaimplementowane lub osiągnięte, aby udanie zrealizować projekt. Obejmuje to wymagania funkcjonalne i niefunkcjonalne, a także specyfikacje interfejsów. Zaległość produktowa może też zawierać elementy robocze, takie jak struktura środowiska testowego i programistycznego oraz debugowania. Elementy wymagań produktowych na tej liście mają nadane priorytety hierarchizujące je względem siebie, ale poza tym nie są zakładane żadne współzależności lub założenia czasowe. Lista zaległości produktowych nie jest ustalona raz na zawsze i stale się zmienia wraz z kończeniem każdego sprintu. Utrzymywanie zaległości na bieżąco jest zwane czyszczeniem zaległości. Właściciel produktu odpowiada za dodawanie nowych wymagań, a jeśli to konieczne,

3 Metafora „sprintu" odnosi się nie do wysokich prędkości, ale do krótkich dystansów.

rozbijanie ich na mniejsze jednostki tak, aby cały zespół dobrze rozumiał nowe wymaganie. Wymagania podlegają ciągłej analizie i przydzielaniu priorytetów na nowo. Przestarzałe wymagania są usuwane. Mówiąc zwięźle, Scrum upraszcza planowanie i czyni je bardziej niezawodnym, ponieważ unikamy wszystkich czynników źle wpływających na niezawodność planowania konwencjonalnego.

- **Zaległość sprintu:** Oczywiście w rzeczywistości sprawy nie są tak całkiem proste. Złożony projekt oprogramowania komputerowego nie może być zarządzany jedynie przy użyciu długiej listy zhierarchizowanych wymagań, a zespół Scrum musi ustalić, którzy członkowie zespołu mają do wykonania które zadania i kiedy. Członkowie zespołu i właściciel produktu podejmują te decyzje krok po kroku dla każdego kolejnego sprintu. Na początku sprintu zespół sięga po zadania ze szczytu listy zaległości produktowych (te o najwyższym priorytecie) i wykorzystuje je do utworzenia ściślejszej listy zaległości sprintu. W tym momencie istotne jest, aby wymagania na nowej liście zaległości były dobrze rozumiane i precyzyjnie określone. Wymagania, które nie spełniają tych kryteriów, nie są jeszcze gotowe na to, aby je włączyć do sprintu. W tym momencie podejmowane są decyzje dotyczące współzależności pomiędzy wymaganiami, potrzebnych zadań i zasobów oraz ram czasowych całego sprintu. Wszystkie zadania, które zespół uzna za konieczne do udanego zakończenia sprintu, są teraz wprowadzane do listy zaległości sprintu.
- **Definicja wykonania:** W celu zapewnienia, że wynikiem sprintu będzie działająca wersja produktu (potencjalnie gotowy produkt), zespół nakreśla kryteria, które umożliwiają mu ocenę, czy praca nad danym przyrostem produktu jest ukończona [URL: Scrum Guide]. Użyte kryteria są określane przez zespół jako „definicja wykonania" i mogą być sporządzone w postaci globalnej listy kontrolnej dla całego sprintu lub w odniesieniu do poszczególnych zadań w ramach tego sprintu. Dyskusja nad tymi kryteriami realizacji jest istotna dla jasnego zdefiniowania wymagań i zadań dla zespołu, choć odnoszą się one tylko do bieżącego sprintu. Horyzont planowania jest krótkoterminowy a zakres prac do wykonania jest niewielki w porównaniu z całym projektem. To podejście oznacza, że zespół jest skoncentrowany na bieżących zadaniach, a zaległości sprintu nie mogą być zmieniane podczas danego sprintu (zobacz [Pichler 10]). Ten typ planowania daje duże szanse na zakończenie projektu z powodzeniem.



Rys. 2–1
Podstawowe zasady procesu Scrum

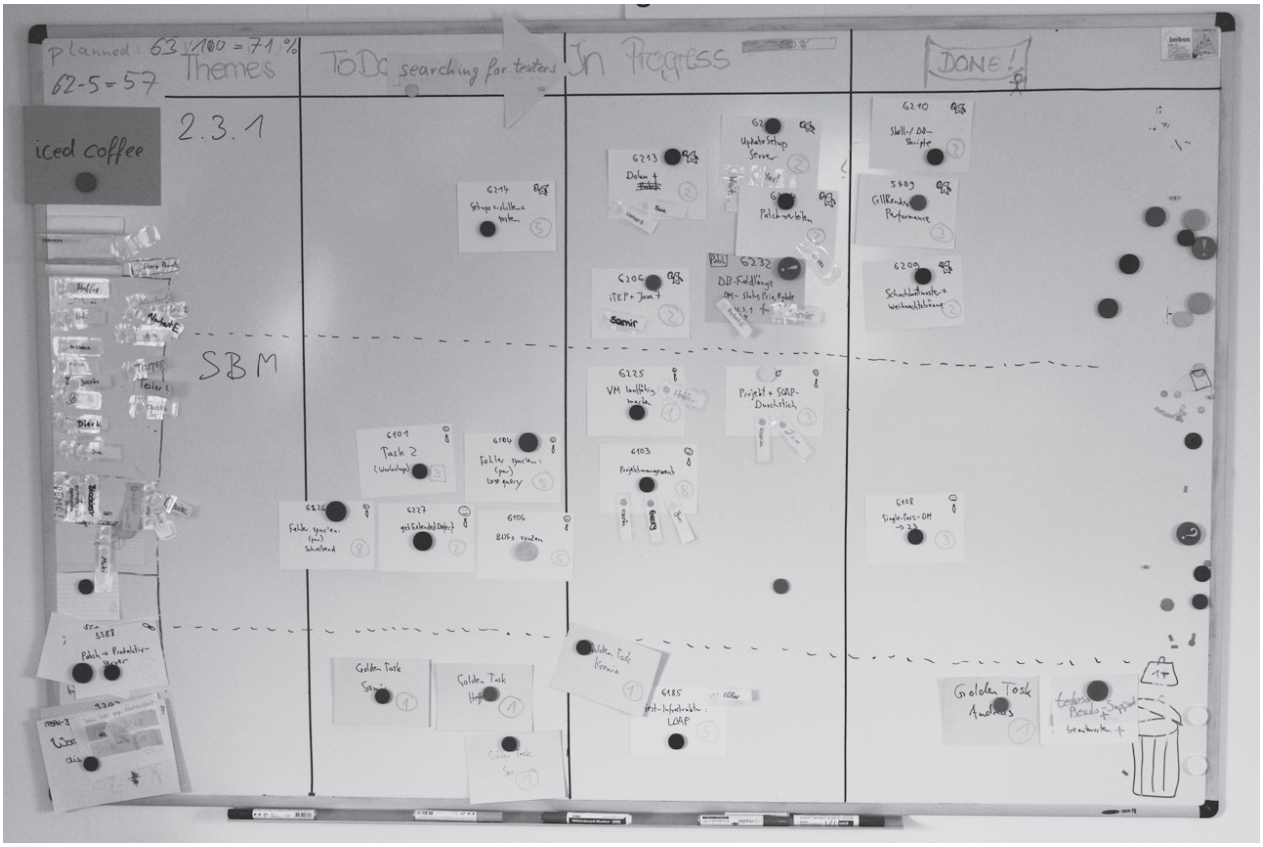
- **Ramy czasowe:** Wymaganiem względem każdego sprintu jest dostarczenie potencjalnie gotowego produktu[4]. To z kolei oznacza, że zaległości sprintu powinny zawierać tylko te zadania, które przyczyniają się do stworzenia gotowego produktu. Wszelkie elementy produktu, których nie można ukończyć podczas sprintu, należy odłożyć, a w tym celu na początku sprintu zespół musi wybrać te funkcje, które realistycznie da się ukończyć. W razie wątpliwości elementy możliwe do zrealizowania mają pierwszeństwo nad elementami funkcjonalnymi – zgodnie z tzw. zasadą ram czasowych. Zespół musi oszacować ilość pracy związanej z ukończeniem wszystkich funkcji, które zamierza dołączyć do danego sprintu. Funkcje, które wymagają zbyt dużo pracy, są odkładane, dzielone na mniejsze zadania lub zakres ich docelowej funkcjonalności jest ograniczany. Podobnie jak w przypadku tradycyjnych procesów planowania zespoły Scrum muszą podjąć wysiłek związany z szacowaniem, choć dwa istotne czynniki zapewniają, że szacunki oparte na Scrum są dokładniejsze od tradycyjnych:
  - Zespół musi jedynie brać pod uwagę pracę związaną z aktualnym sprintem. Analizowane zadania mają ograniczony rozmiar, a dzięki wcześniej zakończonym sprintom zespół jest zwykle dobrze przygotowany.
  - Szacunków dokonuje się przy pomocy pokera planistycznego będącego kolejną techniką XP (zobacz podrozdział 3.5). Szacunki dokonywane przez poszczególnych członków zespołu mogą się znacznie różnić, ale po uśrednieniu są dość dokładne.

4 „Przyrostowe dostarczanie ‘gotowych’ produktów zapewnia, że potencjalnie użyteczna wersja działającego produktu jest zawsze dostępna” [URL: Scrum Guide].

Zasada ram czasowych nie dotyczy jedynie sprintów, ale też innych sytuacji, które wymagają realizacji konkretnego zadania od zespołu. Na przykład może być używana do zapewniania, aby zebrania rozpoczynały się punktualnie i trwały tylko przez ustalony okres czasu.

- **Przeźroczystość:** Przeźroczystość jest jednym z najpotężniejszych narzędzi Scrum. Zaległości sprintu są prowadzone publicznie na tablicy[5], co sprawia, że zawartość i postępy bieżącego sprintu są łatwo zrozumiałe dla zespołu, zarządu i innych zainteresowanych stron. Tablica zawiera wymagania i związane z nimi zadania ułożone w rzędach, natomiast kolumny reprezentują aktualny stan każdego zadania (do wykonania, w trakcie realizacji lub ukończone). Stan sprintu jest aktualizowany codziennie podczas spotkania Scrum, a poszczególne karty zadań są przestawiane na tablicy zgodnie z aktualnymi postępami każdego zadania. Rysunek 2-2 przedstawia tablicę zespołu Test*Bench* (zobacz studium przypadku 8.2).

Rys. 2–2
Przydzielanie ról w ramach zespołu Tablica zespołu TestBench



Przeźroczystość zapewniana przez aktualizowanie tablicy podczas codziennego spotkania Scrum oznacza, że każdy członek zespołu wie, co się dzieje wokół niego (co pomaga zapobiegać wchodzeniu

5 Ruchoma ścianka działowa jest dobrą alternatywą. Jeśli zespół korzysta z narzędzi informatycznych, to lista zaległości sprintu powinna być wyświetlana na czołowej pozycji na dużym monitorze w sali zespołu.

sobie w drogę), a każdy jest świadomy postępów zadań poszczególnych osób, co automatycznie wywiera nacisk na sukces wewnątrz zespołu. Zespół jest zachęcany do omawiania i rozwiązywania problemów, które się pojawiają[6]. Gdy trudności są jawnie widoczne, dużo łatwiej jest zaoferować lub przyjąć pomoc. Ponadto codzienne omawianie zadań i wyników zapewnia stały napływ informacji o elementach zrealizowanych przez poszczególnych członków zespołu i zespół jako całość.

W zgodzie z tymi technikami zarządzania projektami znaczenie zespołu i przydziału ról jest inaczej definiowane w środowisku Scrum. Model Scrum zakłada jedynie trzy główne role w ramach zespołu (za [Schwaber/Beedle 02, rozdz. 3]):

- **Mistrz Scrum** jest nowym typem roli zarządzającej. Jest osobą odpowiedzialną za zapewnianie, aby praktyki Scrum były implementowane i realizowane. Gdyby praktyki te stawały się zbyt przeciążone lub utrudnione, zadaniem mistrza Scrum jest zatrzymanie procesu lub znalezienie alternatywnego rozwiązania. Mistrz Scrum nie ma realnej władzy, ale działa raczej jak trener. Przeszkody związane z procesem mogą być często rozwiązywane przez wyjaśnianie zespołowi szczegółów procesu, przypominanie o odpowiednich procedurach lub przeprowadzanie warsztatów. Inne przeszkody (na przykład niedziałające środowisko budowania) czasami wymagają zaangażowania dodatkowych zasobów (takich jak szybszy serwer lub bardziej odpowiednie narzędzie) i konieczne może być przydzielenie lepiej wykwalifikowanych członków zespołu do procesu budowania. Mistrz Scrum nie powinien nigdy przekazywać nierozwiązanych problemów z powrotem do zespołu. Jeśli dzieje się tak regularnie, rozwiązaniem jest zwykle znalezienie nowego mistrza Scrum.
- **Właściciel produktu** odpowiada za aktualność listy zaległości produktowych i stanowi interfejs pomiędzy klientem a zespołem[7]. Właściciel produktu decyduje, które funkcje mają być implementowane – innymi słowy on lub ona odpowiada za określenie natury projektu. W praktyce rola właściciela produktu może być spełniana przez aktualnego menedżera projektu, lidera zespołu lub szefa projektu[8]. Właściciel projektu nie jest przywódcą zespołu

6 Nie mogą być ukrywane.

7 Jeśli dany projekt obejmuje specyfikacje klienckie, właściciel produktu może być członkiem własnego zespołu klienta.

8 Osoby o różnym doświadczeniu podchodzą do roli właściciela produktu w różny sposób i będą wykorzystywać zaległości produktowe zgodnie

i nie nadzoruje całego procesu Scrum, co jest obowiązkiem mistrza Scrum.

- **Zespół programistów**[9] zwykle składa się z trzech do dziewięciu członków, którzy wykonują zadania konieczne do ukończenia produktu sprint po sprincie. „Zespół (programistów) Scrum powinien składać się z osób mających wszystkie umiejętności potrzebne do osiągnięcia celów sprintu. Scrum wystrzega się pionowo zorganizowanych zespołów analityków, projektantów, programistów i inżynierów zarządzania jakością. Zespół Scrum samodzielnie organizuje się tak, aby każdy przyczyniał się do końcowego wyniku. Każdy członek zespołu wykorzystuje swoje doświadczenie do rozwiązywania wszystkich problemów. Synergia wynikająca z tego, że tester pomaga projektantowi skonstruować kod, poprawia jakość kodu i zwiększa wydajność. Niezależnie od składu zespołu, ważne jest realizowanie całej analizy, projektowania, kodowania, testowania i dokumentacji" ([Schwaber/Beedle 02]).

W związku z tym zespoły powinny być elastyczne[10]. Jest to trudne do osiągnięcia i zwykle oznacza, że zamiast budowania zespołu, w którym każdy ma porównywalne umiejętności, wszyscy członkowie zespołu chętnie pracują nad wszystkimi bieżącymi zadaniami korzystając ze swoich konkretnych umiejętności. Scrum skupia się na optymalizowaniu możliwości i wydajności zespołu jako całości.

Zespoły wielofunkcjonalne

Fakt, że Scrum zachęca do korzystania z elastycznych zespołów i płaskich hierarchii w zespole jest często opacznie rozumiany. Wielofunkcjonalność oznacza, że członkowie zespołu pracują na różnych podstawach funkcjonalnych. Na przykład architekt systemowy i programista tworzą wspólnie architekturę, a architekt pomaga programiście w fazie kodowania, ucząc się przy okazji, że teoretyczne projekty systemów mogą być dość trudne do praktycznego zaimplementowania. Analogicznie tester może pomagać programiście w definiowaniu odpowiednich testów jednostkowych, być może ucząc się przy tym, jak automatyzować testowanie w ramach procesu. Niemniej jednak każdy członek zespołu ma swoje własne, specyficzne umiejętności i wykorzystuje je do pracy nad bieżącymi zadaniami. Chodzi o to, że nikt nie może stwierdzić na przykład, że „jestem testerem i nie robię

---

z własnymi umiejętnościami i preferencjami. Wszyscy członkowie zespołu powinni być tego świadomi, zwłaszcza podczas pierwszej implementacji Scrum.

9 „Zespół Scrum składa się z właściciela produktu, zespołu programistów i mistrza Scrum". [URL: Scrum Guide]. [Schwaber/Beedle 02] nie rozróżnia pomiędzy zespołem a zespołem programistów, dlatego termin ten jest użyty powyżej w nawiasie.

10 W XP [Beck/Andres 04] to pojęcie znane jest jako zasada jednego zespołu, natomiast Scrum [Schwaber/Beedle 02] nazywa to zespołem wielofunkcjonalnym.

nic innego". Studia przypadków 8.1 i 8.4 w szczególności ilustrują niektóre przeszkody, które można napotkać podczas przechodzenia na metodykę Scrum.

Studium przypadku 2-1

Projekt sterownika eHome 2-1: przygotowanie projektu

Decyzja o implementacji projektu zawiera budżet obejmujący trzech programistów, jednego testera, właściciela produktu i mistrza Scrum.

Różne są koncepcje dotyczące tego, kto powinien należeć do zespołu. Niektóre głosy są za wybraniem najlepszych pracowników, inni są za najbardziej doświadczonymi, a jeszcze inni woleliby widzieć w zespole najnowszych pracowników z nowymi pomysłami. Członkowie istniejącego zespołu też mają różniące się opinie na temat wprowadzenia metodyki Scrum. Podczas gdy niektórzy sądzą, że powinno się ją wprowadzić już dawno, to inni uważają, że trudno tradycjonalistom będzie przejść na nowy sposób działania. Niektórzy uważają, że już pracują w sposób zwinny, natomiast bardziej technicznie nastawieni członkowie zespołu sądzą, że niemożliwe będzie zagwarantowanie ciągłej funkcjonalności dokładnie zdefiniowanych protokołów magistrali i predefiniowanych standardów bez użycia tradycyjnych specyfikacji i dokumentacji.

Ostatecznie konieczność zapełnienia ról wewnątrz zespołu pracownikami mającymi odpowiednią wiedzę uprościła znacznie decyzję. Co najmniej jeden z programistów musi wiedzieć wszystko na temat protokołów magistrali i urządzeń sprzętowych (dotyczy to zarówno własnych produktów firmy, jak i produktów konkurencji), a wiedza ta jest już dostępna w firmie. Kreatywny programista WWW powinien odpowiadać za interfejs użytkownika, więc to stanowisko wymaga zaangażowania nowego pracownika. Obecny szef zespołu programistycznego przejmuje rolę właściciela produktu.

Ponieważ zespół nie ma wcześniejszego doświadczenia z metodyką Scrum, wynajęty zostanie niezależny konsultant jako mistrz Scrum i będzie odpowiedzialny za nauczenie zespołu wszystkiego odnoście technik i praktyk Scrum oraz za kontakty z zarządem i zapewnienie, że zespół nie wróci do swoich dawnych praktyk.

Projekt dostaje zielone światło na okres 12 miesięcy, a zespół marketingu zażyczył sobie, aby w tym okresie próbnym zawarte były cztery wersje produktu. Innymi słowy, pierwsza wersja musi być gotowa za trzy miesiące!

Zespoły funkcyjne

Duże, skomplikowane projekty muszą być dzielone pomiędzy wiele zespołów Scrum. Niektóre projekty są dzielone zgodnie z podziałami architektury systemu, co daje w efekcie zespoły zorientowane na komponenty, natomiast inni korzystają z zespołów funkcyjnych, które pracują nad grupami wymagań w wielu sprintach obejmujących komponenty dla różnych systemów – podejście to jest znane jako globalna własność kodu. Studium przypadku 8.5, „Scrum w środowisku technologii medycznych" jest przykładem tego podejścia.

W teorii zaletą tego podejścia jest to, że zespół funkcyjny pracuje nad wszystkimi bieżącymi wymaganiami i dlatego ma bardziej

skupione na kliencie spojrzenie na projekt. Wadą tego podejścia jest to, że zespół funkcyjny ma (znowu w teorii) mniej dogłębną wiedzę na temat poszczególnych składników oprogramowania, co prowadzi do wolniejszego i bardziej podatnego na błędy projektowania, kodowania i testowania. Każdy zespół projektowy musi zdecydować sam za siebie, który model działa najlepiej w danych warunkach.

2.2 Kanban

Metodyki zarządzania projektem i zmianami

Kanban (dosłownie „szyld" lub „tablica", [URL: Kanban]) jest systemem planowania produkcji szczupłej i ma wiele podobieństw do Scrum. W kontekście tworzenia oprogramowania jest opisywany jako podejście do zarządzania projektem i zmianami w projektach informatycznych [Anderson 10]. Jego podstawowe zasady wywodzą się z teorii szczupłego zarządzania [URL: Lean] i są zaprojektowane tak, aby wizualizować i optymalizować przepływ zadań w łańcuchu produkcyjnym. Kanban opiera się na trzech kluczowych składnikach:

- **Tablica Kanban:** Proces jest wizualizowany przy użyciu kart na tablicy. Poszczególne kroki w danym procesie są reprezentowane przez kolumny, a zadania przez karty, które są przesuwane po tablicy od lewej do prawej w miarę postępów prac nad poszczególnymi zadaniami.
- **Limit pracy w toku:** Liczba bieżących zadań (praca w toku) jest ograniczona przez liczbę kart dopuszczalnych dla każdego kroku procesu (albo przez ograniczenie liczby kart mieszczących się na tablicy). Jeśli jakaś część linii produkcyjnej ma nadmiarowe możliwości, pobiera karty z wcześniejszego etapu procesu. To podejście odpowiada metodyce „dociągania" zamiast „wypychania".
- **Czas realizacji:** Kanban służy do optymalizowania ciągłego przepływu zadań przez minimalizowanie (średniego) czasu realizacji potrzebnego do ukończenia strumienia działań.

Scrum działa na podobnej zasadzie wykorzystując tablicę do zapewniania przeźroczystego, wizualnego modelu postępów dla poszczególnych zadań w ramach procesu. Zadania, dla których aktualnie nie są prowadzone żadne działania, są utrzymywane na liście zaległości i trafiają na tablicę, gdy tylko zwolni się jakieś miejsce.

Kanban nie wykorzystuje iteracji i sprintów.

W przeciwieństwie do Scrum, Kanban nie wykorzystuje iteracji i nie ma odpowiednika pojęcia sprintu[11]. Kanban zaprojektowano dla *ciągłego* optymalizowania procesu produkcji i zmniejszania całościowego

11 Iteracje nie są zabronione, Kanban można stosować iteratywnie i zarządzać iteracjami o różnej długości.

czasu wymaganego do wytworzenia gotowego produktu – innymi słowy zapewnia stały przepływ zadań. Proces Kanban pozwala na osobne dostarczanie poszczególnych elementów. Wyprodukowane elementy nie muszą od razu dawać całego, gotowego produktu. Szacowanie wysiłków związanych z ustalonymi ramami czasowymi nie jest konieczne jako część modelu Kanban. Z kolei Scrum zaprojektowano tak, aby dostarczać gotowy produkt we wstępnie zdefiniowanych okresach czasu tak, że koniec sprintu automatycznie synchronizuje wszystkie zadania bieżące i daje w wyniku potencjalnie gotowy do wykorzystania produkt.

Scrum kontra Kanban

Różnice zarysowane powyżej uzmysławiają, że Scrum jest dobrym wyborem do tworzenia pakietów oprogramowania – wersja po wersji – natomiast Kanban pasuje do efektywnego zarządzania produkcją bardziej lub mniej zależnych elementów.

Na przykład model Kanban może być stosowany w celu zwiększenia wydajności zespołu wsparcia informatycznego, gdzie każde zgłoszenie problemu przez klienta reprezentowane jest przez kartę Kanban. [Anderson 10] cytuje zespół utrzymania oprogramowania, który tworzy poprawki jako przykład udanego zastosowania Kanban. Celem zespołu jest tworzenie poszczególnych poprawek i udostępnianie ich klientom tak szybko, jak to możliwe. Każda poprawka reprezentuje niezależne, jednoosobowe zadanie programistyczne, co czyni z Kanban idealny model zarządzania projektem w tym wypadku.

Studium przypadku 2-2

Projekt sterownika eHome 2-2: wykorzystanie Kanban do tworzenia adapterów

Nowy mistrz Scrum zaprasza wszystkich członków zespołu Scrum, wszystkich pozostałych twórców sprzętu i oprogramowania oraz zespół wsparcia na spotkanie wyjaśniające nową metodykę. Spotkanie to wprowadza zespół w podstawy metod Scrum i Kanban.

W ramach dyskusji omawiane jest, czy Scrum, czy Kanban będzie odpowiedniejszym podejściem i czy Kanban może się nadawać do implementacji w innych zespołach. Staje się jasne, że oparty na kuponach system zespołu wsparcia już zawiera pewne elementy modelu Kanban, ale mógłby mimo to skorzystać na wprowadzeniu innych elementów, takich jak limity pracy w toku.

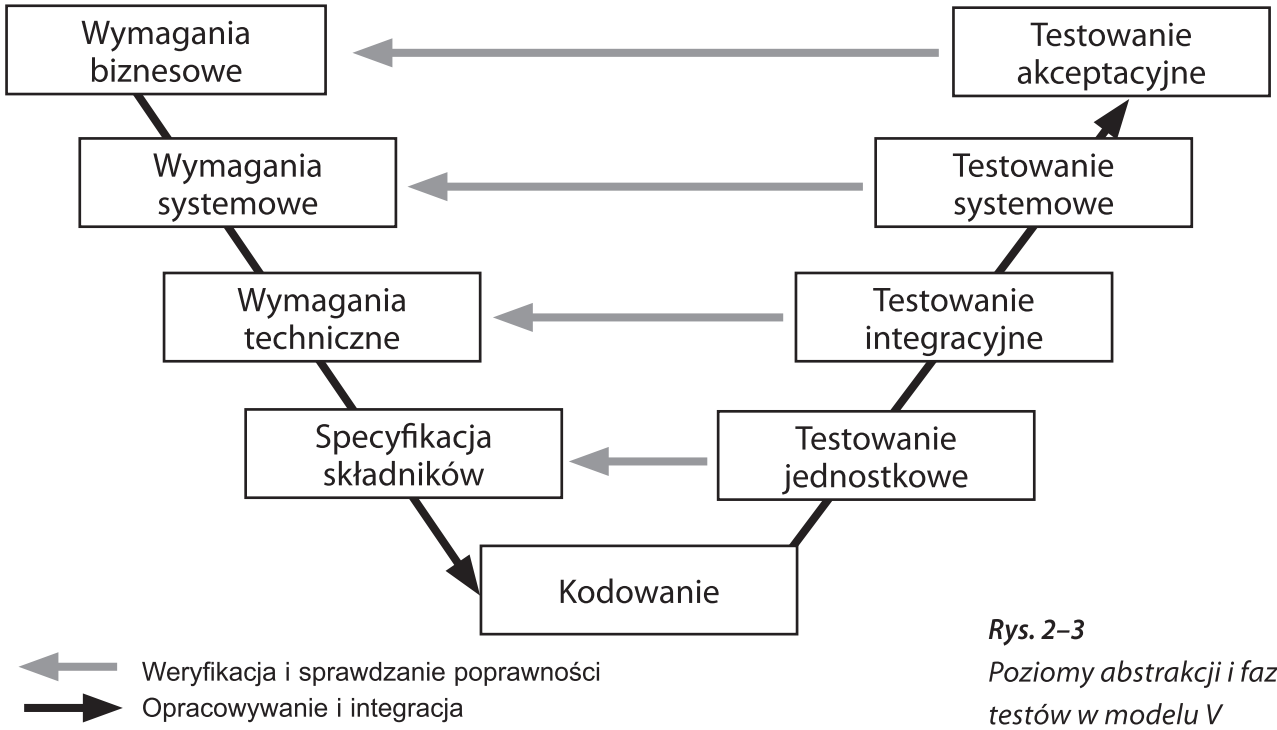
Ponieważ adaptery mogą być tworzone w izolacji, zespół rozważa, czy tworzenie nowych adapterów będzie lepiej zarządzane przy użyciu Scrum, czy Kanban. Poszczególne adaptery nie zależą od siebie i rzadko mają znaczący wpływ na system jako całość. W zasadzie nowy lub zaktualizowany adapter może być integrowany z systemem w dowolnym momencie, co sprawia, że ścisły cykl rozwojowy sterowany przez sprinty nie jest konieczny. Zespół postanawia przyjrzeć się bliżej metodom Kanban, gdy tylko zadanie tworzenia adapterów pojawi się na liście zaległości produktowych.

2.3 Tradycyjne modele procesów

Tradycyjne modele procesów dzielą projekt na wyraźne fazy mające osiągać wstępnie zdefiniowane kamienie milowe. Definiują też role przypisujące zadania do określonych członków zespołu projektowego.

Fazy projektu jako poziomy abstrakcji

Oprócz zarządzania czasem fazy tradycyjnego projektu definiują też różne poziomy abstrakcji, przez które postrzegany jest opracowywany system. Szeroko stosowany model V jest szczególnie dobrym przykładem tego podejścia:



Rys. 2–3
Poziomy abstrakcji i fazy testów w modelu V

Fazy kontra sprinty

„Lewa strona diagramu symbolizuje coraz bardziej skomplikowane kroki związane z projektowaniem, opracowywaniem i programowaniem żądanego systemu, natomiast prawa strona przedstawia kroki związane z integracją i testowaniem konieczne do złożenia poszczególnych elementów składowych w gotowy, działający system. Integracja i testowanie kończy się sprawdzeniem poprawności działania" [Spillner/Linz 14, podrozdział 3.1]. Ten krok obejmuje też tworzenie dokumentacji i/lub artefaktów w pełni opisujących system na aktualnym poziomie abstrakcji. To podejście do tworzenia oprogramowania jest w dużej mierze oparte na dokumentach.

Podejście Scrum jest inne. Obejmuje wszystkie poziomy abstrakcji jednocześnie podczas każdego sprintu i tworzy cały system w sposób przyrostowy. Wymagania są zmieniane w razie konieczności, kod jest przepisywany od nowa, a architektura systemu i odpowiednie testy są optymalizowane na każdym etapie. Rozmiar dokumentacji jest ograniczony do minimum dla każdej iteracji, a jej miejsce

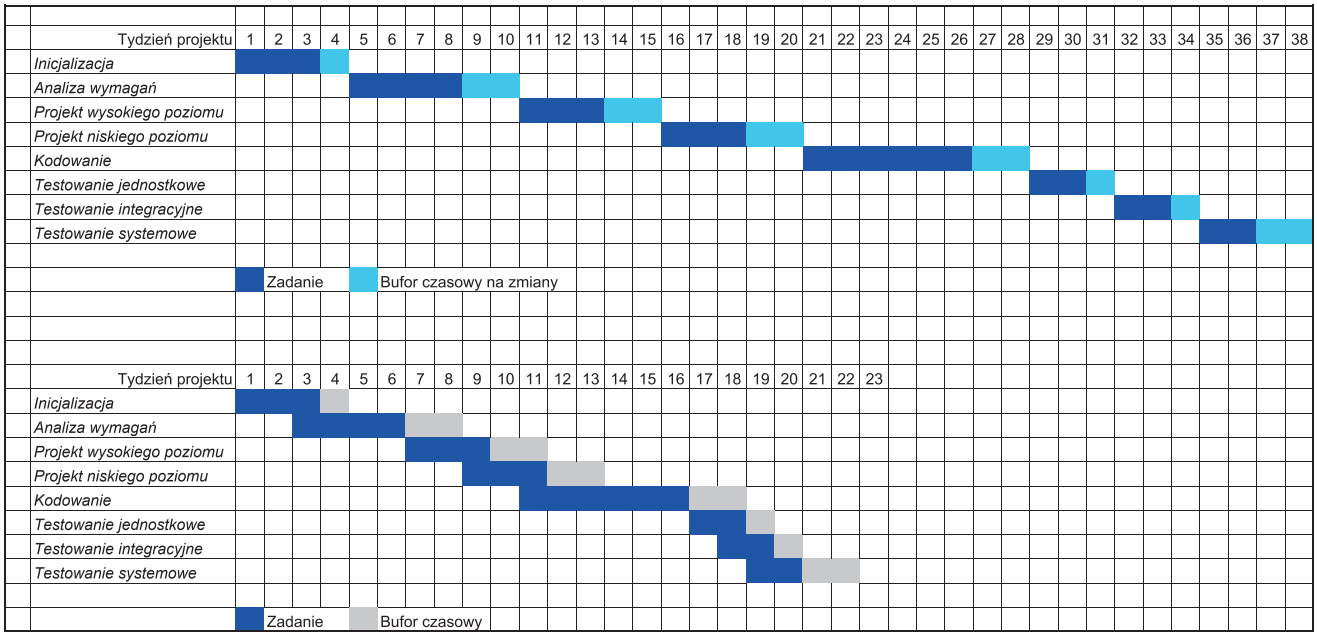
zajmuje bezpośrednia komunikacja i dyskusje pomiędzy wszystkimi zainteresowanymi.

Planowanie i zarządzanie projektami

Tradycyjne techniki zarządzania projektami zajmują się różnymi aspektami projektu (celami, ograniczeniami czasowymi, kosztami i zasobami) tak dokładnie, jak to możliwe z góry i prowadzą projekt w taki sposób, że dany plan jest realizowany od początku do końca.

Studium przypadku Sterownik eHome 2–3: *Tradycyjny plan dla projektu sterownika eHome*

Menedżer projektu wykorzystuje zaproponowaną strukturę projektu do utworzenia planu, który zawiera wszystkie zaplanowane zadania i współzależności między nimi oraz umieszcza je w odpowiednich ramach czasowych. Czas i zasoby wymagane do realizacji całego projektu są szacowane z góry. Gdybyśmy mieli zaplanować nasz projekt sterownika eHome przy użyciu metod tradycyjnych, to początkowy plan mógłby wyglądać, jak pokazano w studium przypadku 2-3:



Słabości tradycyjnych metod planowania projektów

Nieodłączne słabości tego podejścia są oczywiste:

- Wczesne zadania, takie jak inicjacja projektu są zaplanowane jasno, ale często zbyt szczegółowo w porównaniu z resztą planu. Wczesnym fazom projektu często przydzielone jest więcej zasobów, niż to konieczne.
- Ponieważ konieczne dane wejściowe (na przykład projekt interfejsu użytkownika lub wymagane protokoły magistrali) są często dostępne dopiero po rozpoczęciu projektu, szacowanie nakładów potrzebne do dokładnego przydzielenia czasu nie zawsze jest wystarczająco dokładne. Taki plan bierze więc pod uwagę czas przydzielony przez zarząd, a nie czas faktycznie potrzebny zespołowi. Oznacza to, że zaplanowane terminy są często niepewne.

- Aby więc sprostać celowi zdefiniowanemu jako „wersja 1 gotowa w trzy miesiące", menedżer projektu przygotował też plan zapasowy, który pozwoli na ukończenie pierwszej wersji po pięciu miesiącach pracy. Trzeba było sobie poradzić jakoś bez potrzebnego bufora czasowego.
- Zawartość poszczególnych faz nie jest jasno zdefiniowana. Menedżer projektu musi zapewnić, że on/ona i zespół będą dokładnie znać cel każdej fazy, zanim się ona rozpocznie. Z drugiej strony pewien stopień niedokładności daje zarządowi konieczne pole manewru na uwzględnienie błędów planowania i potencjalnych odstępstw od pierwotnego planu.
- Założenie, że zarządzanie jakością wykryje błędy i wady, jest zabezpieczone przez bufor czasowy przydzielony na dokonanie zmian. Na tym etapie nie jest jasne, czy dwa tygodnie czasu na dokonanie zmian to będzie za mało, czy za dużo. W każdym razie zmiany te wiążą się z lokalnymi poprawkami w danym kamieniu milowym. Nie ma budżetu czasowego na żadne zmiany w poprzednich fazach, które mogą być potrzebne.
- Może minąć wiele czasu, zanim projekt będzie mógł być dokładnie przetestowany i gotowy do dostarczenia. W naszym przykładzie 20 lub 38 tygodni upłynie od końca fazy planowania do pierwszego testu integracyjnego, co oznacza, że dostarczenie produktu może mieć miejsce najwcześniej za pięć lub dziewięć miesięcy.

Nawet jeśli początkowy plan jest realistyczny i pozbawiony błędów, to wszystkie aspekty tego planu (zadania, ramy czasowe, zasoby, itd.) mogą podlegać zmianom w czasie trwania projektu. Nawet najdokładniej przygotowany plan może szybko się zdezaktualizować, co będzie wymagać czasochłonnych regulacji i ponownego planowania. W najgorszym przypadku tradycyjne zarządzanie projektami może być traktowane jako ciągle przegrywana bitwa z nieprzewidzianymi ale też nieuniknionymi zmianami.

Projekty Scrum również wymagają planowania.

Scrum rozwiązuje ten problem radząc sobie bez całościowego planu projektu, a zamiast tego wykorzystuje wymienione wcześniej narzędzia zarządzania zwinnego. Niemniej jednak środowisko pracy może sprawić, że konieczne będzie też definiowanie planów dla projektów Scrum. Na przykład kamień milowy lub budżet czasowy obejmujący wiele sprintów może być przydatnym narzędziem pomocniczym do planowania długoterminowych zobowiązań umownych wobec klientów, dostawców, współpracowników lub innych równoległych projektów. Jeśli potrzebne są dodatkowe zasoby, to pomocna (lub nawet konieczna) może być możliwość uzasadnienia tego. W takich przypadkach pomocne może być posiadanie zapisanego przydziału zasobów w oparciu o systematyczne szacowanie nakładów.

Programowanie iteracyjne (przyrostowe) pomaga przeciwdziałać słabościom wymienionym powyżej i jest szeroko stosowanym aspektem tradycyjnych modelów procesów.

Programowanie iteracyjne

Modele iteracyjne też definiują różne fazy projektu, ale wprost[12] pozwalają na powtarzanie (iteracje) określonych faz lub kombinacji faz. Iteracja ma umożliwiać poprawienie niedociągnięć lub błędów przez powtórzenie fazy albo rozszerzenie jej celów (tzn. utworzenie nowego przyrostu produktu). Racjonalny proces zunifikowany utworzony przez Rational Software Corporation (dział IBM) [URL: RUP] jest szeroko stosowanym modelem procesu iteracyjnego.

Procesy programowania iteracyjnego są często błędnie uważane za takie same jak procesy programowania zwinnego. Sprint procesu Scrum może zawierać dowolny typ zadania od poprawek architektury systemu po implementację i testowanie poszczególnych funkcji. Z kolei tradycyjne fazy projektów obejmują zadania określonego typu (na przykład szkic architektury systemu). Założenie Scrum, że każdy sprint (tzn. każda iteracja) powinien dawać potencjalnie gotowy produkt, nie jest w ogólności nieodłącznym elementem procesu iteracyjnego.

Innymi słowy iteracja w tradycyjnie planowanym projekcie nie jest równoważna ze sprintem procesu Scrum. Każdy programista iteracyjny, który chce przejść na metodykę Scrum, musi zrozumieć i uwzględnić tę fundamentalną różnicę. Iteracje w środowisku programowania ekstremalnego (XP) również różnią się od sprintów. XP daje gotowy produkt korzystając ze zmiennej liczby iteracji, co oznacza, że zestaw zadań obejmowanych przez pojedynczą iterację również może się różnić – od przeprojektowania systemu, przez refaktoring kodu, po projektowanie i implementację nowych funkcji.

Programowanie przyrostowe

Programowanie przyrostowe jest najszerzej używanym procesem tworzenia oprogramowania będącym obecnie w użyciu. Dostarcza produkt na różnych etapach rozwoju w regularnych odstępach czasu. Programowanie iteracyjne jest warunkiem wstępnym programowania przyrostowego, a wynikowy proces jest zwykle połączeniem obu.

Okresy czasowe, w których tworzone są przyrosty, różnią się dla różnych modeli procesów. Modele tradycyjne przewidują wypuszczanie wersji produktu w cyklach półrocznych lub rocznych (a w niektórych przypadkach nawet w dłuższych ostępach czasu). Procesy zwinne radykalnie skracają cykl wytwarzania oprogramowania. Scrum zakłada, że każdy sprint tworzy nową wersję, którą w razie konieczności można udostępnić. W praktyce proces rozróżnia pomiędzy wewnętrznymi wersjami wstępnymi (RC) i prawdziwymi wersjami zewnętrznymi.

12 Menedżer projektu może też powtarzać (iterować) fazę modelu liniowego. Istnieje wiele dobrych powodów, aby to zrobić, ale wszystkie one (ściśle mówiąc) reprezentują odchylenie od podejścia liniowego.

Wiele projektów Scrum ma na celu stworzenie nowej wersji co trzy miesiące przy zastosowaniu miesięcznych cykli Scrum. Zespoły, które opanowały do perfekcji sztukę ciągłej integracji (zobacz rozdział 5) i w efekcie stosują ciągłe dostarczanie produktu, często są w stanie wypuszczać gotowy produkt codziennie. Jednakże zasada ciągłego dostarczania zakłada, że rozwijanie produktu odbywa się w całkowicie zautomatyzowanym, kontrolowanym środowisku spotykanym w wewnętrznych systemach informatycznych niektórych firm oraz w wielu środowiskach WWW i eCommerce. Systemy takie mogą być aktualizowane poprzez wdrażanie nowych wersji co noc.

2.4 Porównanie modeli procesów

Rozróżnienie pomiędzy zarządzaniem projektami, zarządzaniem zasobami ludzkimi, technikami programistycznymi i zarządzaniem jakością jest przydatnym sposobem na zwizualizowanie różnic pomiędzy wyżej wymienionymi modelami procesów.

Rys. 2–4
Główne aspekty modeli procesów

Każdy model radzi sobie z tymi aspektami na różnych poziomach intensywności i wykorzystując różne filozofie i metody:

- **Scrum** jest empirycznym, adaptacyjnym podejściem do zarządzania projektami. Zastępuje kompleksowe planowanie przez możliwość szybkiego i elastycznego reagowania na zdarzenia projektowe. Scrum nie wymaga użycia określonych technik programistycznych, chociaż często zalecane jest XP (programowanie

ekstremalne). Jakość produktu jest zapewniana przez wiedzę zespołu i spójne wykorzystanie uzgodnionych technik programistycznych. Zespół odpowiada za swój własny sukces i najlepiej współpracuje z przywódcą zespołu, który nie ma władzy nadrzędnej. Wstępnie zdefiniowane procedury zapewniają, że ciągła optymalizacja odbywa się na poziomie projektu.

- **Kanban** jest podejściem do zarządzania zadaniami, które zostało zaprojektowane do wykorzystania w stałych grupach roboczych lub departamentach, a nie w indywidualnych projektach. Nie wymaga użycia określonych technik. Zespół działa w ramach wspólnego łańcucha wartości, w którym każdy członek zespołu ma swoją własną, określoną funkcję. Jakość produktu jest zapewniana przez umiejętności członków zespołu i natychmiastowe poprawianie błędów. Wstępnie zdefiniowane procedury zapewniają, że ma miejsce ciągła optymalizacja.
- **Modele tradycyjne** oparte są na wstępnie zaplanowanym, deterministycznym podejściu do zarządzania projektami. Zespół pracuje zgodnie z instrukcjami podawanymi przez menedżera projektu, który jest traktowany jako zwierzchnik członków zespołu. Menedżer projektu odpowiada za monitorowanie postępu oraz inicjowanie wszelkich wymaganych działań naprawczych. Stosowane techniki zależą od struktury bazowego planu i regulacji obowiązujących w danym przedsiębiorstwie (np. określonych protokołów lub własnych systemów zarządzania jakością). Ciągła optymalizacja jest zwykle częścią planu projektu i jest implementowana przy pomocy systemu zarządzania jakością i regularnych audytów wewnętrznych lub zewnętrznych.

Tabela 2-1 ilustruje najważniejsze różnice pomiędzy modelami procesów.

Tab. 2–1 Porównanie modeli procesów

		Scrum	Kanban	Tradycyjne
Zarządzanie projektami	**Planowanie produktu**	Wizja produktu, mapa drogowa, lista zaległości produktowych	–	Mapa drogowa
	Planowanie projektu	Zaległości produktowe	–	Plan projektu z kamieniami milowymi
	Planowanie zadań	Zaległości sprintu dla każdej iteracji	Zaległości	Określone fazy
	Iteracja	Wstępnie zdefiniowany czas trwania iteracji (sprinty)	Ciągłe	Zgodnie z planem projektu
	Czas trwania iteracji	Zwykle 1-4 tygodnie	–	Zgodnie z planem projektu

Tab. 2–1 Porównanie modeli procesów

		Scrum	Kanban	Tradycyjne
Zarządzanie projektami	**Monitorowanie stanu**	Codziennie, z całym zespołem, na tablicy	Codziennie, z całym zespołem, na tablicy	Przez menedżera projektu w oparciu o kamienie milowe
	Ramy czasowe	Tak	Nie	Decyduje menedżer projektu
	Dostarczanie	Po każdej iteracji (tzn. na końcu każdego sprintu)	Ciągle wraz z kończeniem każdego zadania	Z każdą wersją lub po ukończeniu projektu
	Zarządzanie zmianami	Poprzez aktualizację zaległości	Poprzez aktualizację zaległości	Poprzez istniejący system zarządzania zmianami
	Miary	Wykres spalania (burndown)	Praca w toku	Zgodność z terminami, kosztami, zasobami
Zarządzanie jakością	**Optymalizacja procesów**	Przegląd projektu, inspekcja i adaptacja	Kaizen	Audyty wewnętrzne i zewnętrzne (np. zgodnie z ISO 9001), programy optymalizacji procesów
	Weryfikacja/ testowanie	Ciągłe wewnątrz sprintu	Dla każdego zadania	Testowanie zgodności ze specyfikacjami projektowymi podczas faz testowania
	Sprawdzanie/ zatwierdzanie	Demonstracja dla użytkowników na końcu sprintu (spełnienie kryteriów przyjęcia)	Dla każdego zadania zgodnie z wstępnie zdefiniowanymi kryteriami	Test przyjęcia po zakończeniu projektu
Narzędzia programistyczne	**Techniki**	Typowe, ale nie obowiązkowe programowanie w parach[a], ciągła integracja (CI)[b], programowanie oparte na testach[c], projektowanie przyrostowe[d], czysty kod[e], refaktoring[f]	Brak konkretnych założeń	Zależy od stosowanego modelu. Niekiedy stosowane są określone zasady kodowania i narzędzia
Zasoby ludzkie	**Wartości i zasady**	Manifest programowania zwinnego (agile), wartości Scrum, zaangażowanie, otwartość, szacunek, odwaga	Kaizen, zarządzanie szczupłe, manifest programowania zwinnego	Zarządzanie projektami, model procesu, ciągła optymalizacja (ISO 9000)
	Organizacja	Zespół Scrum, właściciel produktu, mistrz Scrum		Lider zespołu, menedżer projektu, organizacja projektu
	Szkolenie	Z własnej inicjatywy, chęć poprawy, uczenie się z zespołem	Uczenie się z zespołem	Plan szkoleniowy przedsiębiorstwa, rozwijanie zasobów ludzkich
	Przydział pracy	Międzydyscyplinarny (wielofunkcjonalny)	Specjaliści pracują na każdym etapie procesu	Wysoce wyspecjalizowany

a. Z zasad XP: „Pisać cały kod produkcyjny z dwoma osobami siedzącymi przy tym samym komputerze" [Beck/Andres 04, str. 42].

b. Z zasad XP: „Integrować i testować zmiany po maksymalnie kilku godzinach" [Beck/Andres 04, str. 49].

c. Z zasad XP: „Pisać zautomatyzowane testy przed zmianą jakiegokolwiek kodu" [Beck/Andres 04, str. 50].
d. Z zasad XP: „Codziennie inwestować w projekt systemu" [Beck/Andres 04, str. 51].
e. „Będziemy wiedzieć, jak pisać dobry kod. Będziemy też wiedzieć, jak zmieniać zły kod w dobry kod" [Martin 08, str. 2].
f. „Dla każdych kilku dodawanych wierszy kodu zatrzymamy się i zastanowimy nad nowym projektem. Czy obniżyliśmy jego jakość? Jeśli tak, wyczyśćmy go..." [Martin 08, str. 172].

3 Planowanie projektu zwinnego

Ten rozdział opisuje lekkie narzędzia do zarządzania projektami, które Scrum wykorzystuje zamiast tradycyjnego planu projektu. Jest skierowany głównie do czytelników, którzy chcą przejść na programowanie zwinne, ale zawiera też wiele komentarzy i wskazówek dotyczących zapobiegania błędom dzięki korzystaniu z narzędzi planowania zwinnego.

Zwinne zarządzanie projektami jest oparte na założeniu, że zespół uczy się podczas każdego sprintu. Złe decyzje mogą zostać zbadane i poprawione w kolejnych sprintach, a zespół może szybko reagować na zmiany w środowisku projektu. Każdy nowy przyrost produktu generowany przez sprint daje zespołowi i klientowi nowy obraz prawdziwych wymagań produktu. Oczywiście każdy przyrost może też generować nowe i lepsze pomysły dotyczące funkcjonalności produktu i sposobów jego implementacji w elegancki sposób.

Zespół powinien stale oceniać te doświadczenia i dołączać wszystkie nowe pomysły, które mają jakąś wartość, do listy zaległości produktowych. Oznacza to, że najlepsze pomysły mogą być natychmiast wprowadzane w formie nowych zadań do planu na następny sprint.

Adaptacyjne planowanie empiryczne

Takie adaptacyjne, empiryczne podejście do planowania projektów sprawia, że proces tworzenia oprogramowania staje się niezwykle elastyczny, a ciągłe uczenie się jest automatyczną częścią tego procesu. Oznacza to też, że zespół jest zawsze w stanie szybko reagować na zmiany w wymaganiach klienta i ogólnym środowisku projektu.

Możliwość zmiany kierunku w każdym nowym sprincie nie oznacza, że zespół pracuje bez celu. Nawet w środowisku Scrum same praktyki reaktywne nie dadzą udanego projektu. Zespół Scrum potrzebuje też celów wykraczających poza bieżący sprint. Kolejne podrozdziały przedstawią instrumenty, które mogą służyć do wyznaczania kierunków zespołowi Scrum.

3.1 Wizja produktu

Wizja produktu opisuje w sposób zwięzły i dokładny, jak powinien wyglądać końcowy produkt i jakie powinny być jego możliwości. Im krótsza i bardziej zwięzła wizja produktu, tym większa szansa na stworzenie żądanego produktu. Może ona przyjmować formę szkicu na kartce, listy najważniejszych 10 funkcji wymaganych przez klienta lub rysunku produktu konkurencji, który zamierzamy pokonać.

Lista zaległości produktowych nie nadaje się do wykorzystania jako wizja produktu.

Zaległości produktowe (zobacz podrozdział 3.3) zwykle są zbyt szczegółowe, aby służyć jako wizja produktu, a zbyt duża liczba elementów na liście zaległości produktowych zaciemnia obraz celu projektu. Najlepiej, aby wizja produktu służyła pomocą w ustalaniu priorytetów na liście zaległości produktowych i oddzielała istotne elementy od mniej istotnych. Zespół eHome zakończył szkic następującej wizji:

Studium przypadku 3-1

Projekt sterownika eHome 3-1: wizja produktu

Właściciel produktu zaprasza zespół programistów na warsztaty w celu omówienia projektu i opracowania wizji w oparciu o cele zarządu i wymagania własne klienta.

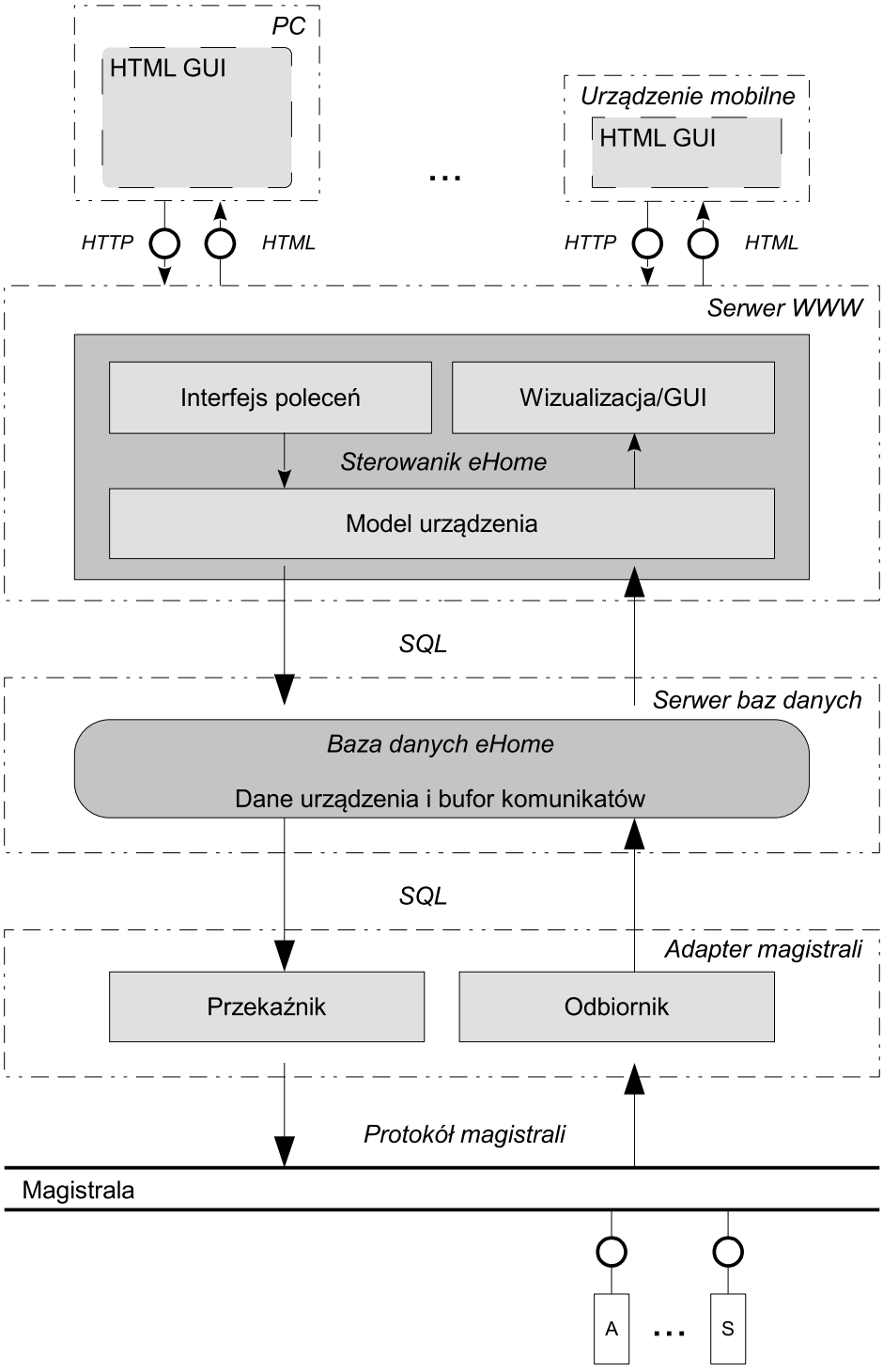
System musi działać na smartfonach i innych urządzeniach mobilnych oraz musi obsługiwać istniejące urządzenia firm trzecich, a także wszelkie nowe urządzenia, które w najbliższym czasie trafią na rynek. Właściciel produktu i zespół zarysowują następujące cele:

- Interfejs użytkownika oparty na przeglądarce: Oznacza to, że właściciel produktu eHome może sterować swoimi systemami poprzez komputer PC, smartfon lub tablet bez konieczności instalowania jakiegokolwiek dodatkowego oprogramowania.
- Jasna i prosta obsługa: Wszystkie pokoje i urządzenia w nich się znajdujące mają być reprezentowane przez osobne ikony lub zdjęcia załadowane przez użytkownika. Interfejs nie powinien wyglądać zbyt technicznie i musi być zrozumiały dla użytkowników, którzy nie mają doświadczenia w korzystaniu z systemów informatycznych.
- Niezależność od producentów: Sterownik musi obsługiwać elementy wykonawcze i czujniki od różnych producentów oraz odpowiednie protokoły magistrali.

3.2 Wizja architektury

Po zarysowaniu wizji produktu zespół może zacząć zastanawiać się, jak wprowadzić ją w życie. Pierwszym krokiem w tym procesie jest zarysowanie wizji architektury, która zawiera przegląd produktu i jego poszczególnych składników. W oparciu o wizję produktu zakreśloną powyżej, zespół eHome projektuje podstawową architekturę pokazaną na diagramie studium przypadku 3-2[1].

1 Notacja oparta na FMC [URL: FMC].



Studium przypadku sterownik eHome 3–2: *Architektura projektu sterownika eHome*

Podobnie jak wizja produktu, wizja architektury nie musi być zbyt szczegółowa i wystarczy prosty szkic głównych elementów. Schemat architektury nie musi przyjmować postaci diagramu UML ani żadnego innego typu formalnej notacji, o ile wszyscy członkowie zespołu (włączając właściciela produktu i mistrza Scrum) go rozumieją.

Wizja architektury jako wytyczne dla kodowania i testowania

Podobnie do wizji produktu wizja architektury działa jako przewodnik po liście zaległości produktowych i nadchodzących sprintach. Pomaga zapewnić zrównoważony obraz projektu, którego nie trzeba

przeorganizowywać przed każdym sprintem, a także pomaga naszkicować sensowne zestawy zadań, które można rozdzielić pomiędzy członków zespołu w nadchodzących sprintach.

Wizja architektury kładzie też ważne podstawy pod testowanie od bardzo wczesnego etapu definiując najważniejsze składniki produktu i ich interfejsy systemowe jako obiekty do testowania. W ten sposób testowanie i inne środki zarządzania jakością mogą być dopasowane do rozwijającego się systemu od pierwszego sprintu.

Wyłaniająca się architektura

Nawet jeśli zespół zarysuje architekturę systemu i towarzyszący jej schemat na wczesnym etapie, nie oznacza to, że nie można jej zmieniać później. Architektura systemu może i powinna być regularnie poprawiana i dostosowywana podczas kolejnych sprintów. Odkrycia dokonywane podczas ciągłej implementacji i pochodzące z informacji zwrotnych z demonstracji produktu mogą wpływać na ostateczną architekturę. Dostosowywanie logicznie zaprojektowanej architektury krok po kroku jest z pewnością dużo bardziej wydajnym podejściem, niż po prostu kodowanie przez programistów chaotycznego zestawu klas, które nie przystają do wyraźnego planu.

3.3 Zaległości produktowe

Właściciel produktu wykorzystuje listę zaległości produktowych do zbierania i ustalania priorytetów dla wszystkich wymagań, które należy zawrzeć w końcowym produkcie. Jednakże lista zaległości produktowych nie jest planem projektu i nie jest listą zadań do zrobienia. Służy natomiast jako rodzaj notatnika na pomysły, które pojawiają się w trakcie tworzenia oprogramowania. Poszczególne wpisy otrzymują priorytety zgodnie z ich znaczeniem i potencjalną wartością dla projektu, ale nawet te priorytety nie są wiążące, jeśli chodzi o to, czy i kiedy dany wpis zostanie faktycznie zaimplementowany. Właściciel produktu dla projektu sterownika eHome naszkicował listę zaległości produktowych i pogrupował ją zgodnie z szacunkową wartością biznesową[2] dla całego projektu (patrz Studium przypadku 3-3).

Podobnie jak wszelkie inne dokumenty planistyczne oparte na Scrum, lista zaległości produktowych zmienia się ze sprintu na sprint. Dodawane są nowe pomysły i wymagania, zmieniane są priorytety, usuwane są nieaktualne wymagania, pojedyncze elementy są grupowane, a ogólne pomysły są uściślane i dzielone na mniejsze, lepiej zdefiniowane zadania. Właściciel produktu jest odpowiedzialny za utrzymywanie listy zaległości produktowych, natomiast mistrz Scrum zapewnia, że

2 Priorytety nie definiują dokładnie ani nie determinują żądanej kolejności zakończenia. Zespół może wybierać wymagania z listy zgodnie z kolejnością priorytetów, a przy tym zachować elastyczność podczas planowania sprintu.

lista zaległości spełnia podstawowe reguły procesu Scrum. W naszym przykładzie mistrz Scrum zauważa, że ostatnie dwa elementy na liście zaległości produktowych nie mają kryteriów przyjęcia i prosi właściciela produktu o ich naniesienie (zobacz podrozdział 3.9.3).

Zagadnienie	Priorytet	Opis / kryteria przyjęcia
Sterowanie i monitorowanie		Jako właściciel systemu eHome chcę w prosty sposób i centralnie sterować wszystkimi podłączonymi urządzeniami i monitorować wszystkie dane zbierane przez czujniki.
	2	❒ *Graficzny interfejs użytkownika wyświetla bieżący stan wszystkich przyłączonych elementów wykonawczych i czujników*
	1	❒ *Element wykonawczy: kliknięcie jego ikony aktywuje funkcję przełączającą danego elementu*
	2	❒ *Czujnik: Kliknięcie jego ikony przesyła dane z czujnika i je wyświetla*
Adapter magistrali		Jako właściciel systemu eHome chcę, aby system obsługiwał urządzenia różnych producentów tak, abym nie był zależny od konkretnego producenta.
	1	❒ *Adapter magistrali tłumaczy sygnały sterownika na takie, które mogą być przetwarzane przez odpowiedni protokół urządzenia*
	1	❒ *Wszystkie urządzenia systemu eHome mogą być sterowane poprzez adapter eHome*
Sterowanie urządzeniami		Jako właściciel systemu eHome chcę sterować różnymi klasami urządzeń (np. lampami, ściemniaczami, żaluzjami), ale w celu uniknięcia wadliwego działania sterownik pozwala dla każdej klasy urządzeń na tylko te polecenia, które mogą być wykonywane przez to urządzenie. *Jako użytkownik systemu eHome mogę:*
	1	❒ *Włączać i wyłączać przyłączone lampy*
	3	❒ *Dostosowywać jasność każdej lampy w zakresie od 0-100% przy pomocy ściemniacza*
	2	❒ *Odsuwać i zasuwać żaluzje lub dopasowywać je do określonego położenia*
Graficzny interfejs użytkownika		Chcę sterować systemem przy użyciu interfejsu opartego na przeglądarce, aby nie było wymagane dodatkowe oprogramowanie. *Graficzny interfejs użytkownika działa w następującej przeglądarce na komputerze PC:*
	1	❒ *Firefox wersja 15.0 i późniejsze*

Studium przypadku sterownik eHome 3–3: *Początkowa lista zaległości produktowych dla projektu sterownika eHome*

ciąg dalszy na następnej stronie

Zagadnienie	Priorytet	Opis / kryteria przyjęcia
Ikony interfejsu użytkownika	2	Każda klasa urządzeń jest reprezentowana przez swoją ikonę. Upraszcza to identyfikowanie określonych urządzeń i ładnie wygląda.
Programowanie przełączników	3	Wszystkie urządzenia mogą być oprogramowane tak, aby zautomatyzować regularne procedury. Na przykład, aby uchylić wszystkie żaluzje o 7 rano w dni powszednie i wszystkie żaluzje skierowane na południe o 8.30 rano oraz zamknąć wszystkie żaluzje o 8.30 wieczorem.

Testerzy pomagają nakreślić kryteria przyjęcia

Aby naszkicować właściwe kryteria przyjęcia, testerzy zespołu muszą zostać zaangażowani w tym procesie. Ułatwi to też testerom zaprojektowanie odpowiednich testów akceptacyjnych później.

3.4 Mapa scenariuszy

W oparciu o wizje produktu i architektury oraz listę zaległości produktowych właściciel produktu wraz z zespołem może stworzyć mapę scenariuszy przedstawiającą cele nadchodzących sprintów. Celem sprintu może być ważne wymaganie lub zestaw wymagań, które tworzą dominujący temat sprintu. Cele zaprojektowane dla poprawy samego procesu również mogą służyć jako cele sprintu. Zespół eHome utworzył poniższą mapę scenariuszy dla swoich pierwszych trzech sprintów:

Studium przypadku sterownik eHome 3–4: *Początkowa mapa scenariuszy zespołu eHome*

Sprint 1

Produkt:
Symulacja/demonstracja dla sprzedaży i marketingu

- Działające prototypy interfejsów dla komputera osobistego, smartfona i tabletu.
- Szkicowa struktura bazy danych dla urządzeń i bufora komunikatów.
- Brak adaptera magistrali (na razie).

Proces:

- Platforma testowania jednostkowego jest zainstalowana i w użyciu.

Sprint 2

Produkt:
Można sterować urządzeniami „eHome Tools", które mogą być włączane i wyłączane

- Kliknięcie ikony Przeglądarka inicjuje funkcjonalność przełączania.
- Adapter magistrali przetwarza komunikaty włączenia/wyłączenia (przesyłanie) i potwierdzenia (odbieranie)

Proces:

- Środowisko testowania jednostkowego działa na centralnym serwerze ciągłej integracji
- Środowisko testowania systemowego z urządzeniami eHome Tools

Sprint 3

Produkt:
Można sterować wszystkimi urządzeniami „eHome Tools"

- Interfejs użytkownika oparty na przeglądarce z wizualizacją stanu urządzania (włączone/wyłączone)
- Adapter magistrali dla wszystkich urządzeń eHome Tools (przesyłanie i odbieranie)

Proces:

- Analizator kodu działający w ramach ciągłej integracji
- Mierzone jest pokrycie kodu
- Zautomatyzowane przypadki testów systemowych

Mapa scenariuszy nie jest obowiązkowa.

Jednakże mapa scenariuszy jest zawsze jedynie deklaracją intencji. Zespół nie gwarantuje, że zawarte w niej cele będą osiągnięte. Cele i sprinty, do których są przydzielone, mogą się zmieniać z powodu informacji zwrotnych od klienta lub rosnącego doświadczenia zespołu.

Gdy właściciel produktu omawia mapę scenariuszy z klientem, trzeba jasno powiedzieć, że nie oznacza to określonej deklaracji dołączenia danej funkcji w określonej wersji produktu. Takie deklaracje w jakiejkolwiek formie podważają podejście zwinne i szybko zmuszą zespół do powrotu do tradycyjnego, przyrostowego procesu tworzenia oprogramowania.

Zalety mapy scenariuszy w porównaniu z tradycyjną mapą drogową

Chociaż właściciel produktu nie gwarantuje, kiedy produkt osiągnie określony etap w swoim rozwoju, to zwinna mapa scenariuszy ma jednak pewne zalety względem tradycyjnej mapy drogowej projektu:

- Zespół sprzedaży i właściciel produktu nie obiecuje terminów, których zespół programistyczny mógłby nie być w stanie dotrzymać. Zespoły programistyczne w tradycyjnych projektach często i tak traktują takie terminy jako nierealistyczne. Jeśli nie będą ściśle zarządzane, projekty szybko nabiorą opóźnień względem harmonogramu, a zespół sprzedaży będzie musiał odpowiednio poprawić mapę drogową. Zwinna mapa scenariuszy jest bezstresową alternatywą, a ostatecznie osiąga te same cele, co tradycyjna mapa drogowa.
- Oprócz rozważenia priorytetów wewnątrz zespołu, klient otrzymuje realistyczny przegląd zaplanowanych funkcji i ramy czasowe, w których zostaną one ukończone. To podejście umożliwia też klientowi zobaczenie swoich ulubionych funkcji w kontekście całego projektu i może zapewnić przydatne informacje zwrotne odnośnie tego, w której wersji mapy powinny się pojawić.
- Klient może aktywnie wpływać na priorytety funkcji. Zespół może też analizować wymagania i priorytety różnych grup klientów i użytkowników oraz uwzględniać je w swoim własnym procesie nadawania priorytetów.
- Producent ma większe pole manewru, żeby łagodzić konflikty interesów, które mogą powstawać na gruncie zmian priorytetów i przydzielania funkcji do wcześniejszych lub późniejszych sprintów. Projekty tradycyjne są często mniej elastyczne, ponieważ określona mapa drogowa została obiecana klientowi, a takie obietnice są często trudne lub niemożliwe do zmiany lub poprawienia.

3.5 Zaległość sprintu

Podczas spotkania w sprawie planowania sprintu (na początku każdego sprintu) zespół wybiera do listy zaległości sprintu te elementy z listy zaległości produktowych, które mają być ukończone w trakcie trwania najbliższego sprintu. Należą do nich wszystkie zadania i wymagania konieczne do osiągnięcia celu sprintu wynikającego z mapy scenariuszy.

Zasada „dociągania"

Zespół upewnia się, że nie alokuje więcej zadań, niż jest w stanie wykonać w trakcie sprintu. Innymi słowy zespół sam wykorzystuje zasadę „dociągania", aby ustalić, co może osiągnąć. Aby zapewnić sprawne działanie tego procesu, zespół musi dokładnie szacować wysiłek związany z każdym zadaniem – proces ten jest też przeprowadzany wewnątrz zespołu jako część pokera planowania.

Poker planowania

Każdy członek zespołu szacuje, ile wysiłku będzie wymagać każde z zaplanowanych zadań. Gdy już wszyscy członkowie zespołu podadzą swoje szacunki, wyliczana jest wartość średnia dla każdego zadania[3]. Jeśli szacunki różnią się znacząco, zespół musi rozważyć, czy zadanie jest prawidłowo interpretowane przez każdego zainteresowanego i musi naprawić wszelkie anomalie przed ponownym oszacowaniem wymaganego nakładu pracy. To podejście zapewnia, że wynikowe szacunki są dokładne.

W odróżnieniu od tradycyjnego planu projektu, w którym szef projektu ustala z góry, które zadania mają być wykonane w jakich terminach, zaległości sprintu reprezentują deklarację zespołu dla właściciela produktu, że ukończony zostanie sprint i przypisane do niego zadania w zaplanowanym okresie czasu[4]. Ta deklaracja jest kompromisem pomiędzy możliwością wyboru zadania i szacowaniem czasu potrzebnego do jego wykonania. Zespół może złożyć taką deklarację, tylko jeśli będzie miał konieczną swobodę planowania.

Potencjalnie gotowy produkt

Udany sprint daje w wyniku potencjalnie nadający się do użycia produkt. Wczesne sprinty powinny dostarczać przynajmniej prototyp do pokazania klientowi, natomiast późniejsze sprinty powinny generować coraz lepsze wersje, które mogą być używane w środowisku produkcyjnym. Deklaracja dostarczenia użytecznego produktu ma różne konsekwencje, zwłaszcza dla aspektów testowania i zarządzania jakością w projekcie Scrum.

- Funkcje implementowane w trakcie sprintu muszą być testowane, co oznacza, że cała praca związana z zarządzaniem jakością i harmonogramy testów muszą być zaplanowane jako część sprintu.

3 Aby zapobiec nadmiernemu wpływowi skrajnych szacunków na wyniki, często używana jest wartość mediany zamiast średniej arytmetycznej.

4 Poszczególne zadania mogą być opracowywane ponownie podczas sprintu, jeśli zespół uważa, że dzięki temu cel sprintu będzie łatwiejszy do osiągnięcia.

Nie ma oddzielnej fazy testowania, która miałaby miejsce po zakończeniu sprintu.

- Obiecane funkcje muszą działać! To oznacza, że wszystkie inne wymagane składniki, takie jak sterowniki i interfejs użytkownika muszą być obecne i muszą się ściśle integrować z nową funkcją. Sam produkt jest budowany „pionowo", a nie poziomymi warstwami, które odpowiadają warstwom architektury systemu (co często dotyczy tradycyjnie zarządzanych projektów).
- Zespół ryzykuje kodowaniem zbyt wielu ogólnych funkcji. Oczywiście każda architektura projektu zawiera funkcje ogólne i biblioteki klas, które ograniczają rozmiar wysiłków związanych z implementacją. Staje się to problemem tylko wtedy, gdy generowanych jest zbyt wiele klas mających wspierać przyszłe przypadki użycia. Wiele projektów ostatecznie wykorzystuje jedynie 10-30% dostępnych funkcji ogólnych, co oznacza, że czas poświęcony na kodowanie pozostałych 70-90% jest stracony. Dodatkowo brak jest przydatnych informacji zwrotnych dla nieużywanego kodu. Dlatego bardzo ważne jest zapewnienie, aby biblioteki klas były tworzone tylko w celu wspierania części systemu związanych z bieżącym sprintem.
- Zadanie jest gotowe tylko wtedy, gdy jego kryteria definicji gotowości zostały spełnione. To oznacza, że po każdym zadaniu następują kroki dodatkowe (takie jak przegląd kodu, szkicowanie nowych testów, automatyzacja testów, aktualizowanie dokumentacji, itd.), które często są pomijane lub odkładane na później w tradycyjnie zarządzanych projektach. Wszystkie te zadania dodatkowe muszą być brane pod uwagę (bezpośrednio lub pośrednio jako kryteria gotowości) podczas planowania sprintu. Zwiększa to z kolei nakład pracy wymaganej do ukończenia zadania i ogranicza liczbę funkcji, które mogą być zrealizowane w trakcie danego sprintu. Zapewnia to jednak, że opracowywane funkcje faktycznie będą gotowe na zakończenie sprintu.

3.6 Karta zespołu

Tradycyjnie zarządzane projekty obecnie uwzględniają tworzenie planu zarządzania jakością zgodnie z IEEE 730 i ogólnego planu testowania (ewentualnie z dodatkowymi planami testów) zgodnie z IEEE 829. Plany te są używane jako podstawa dalszych prac. Zarówno plany zarządzania jakością, jak i testowania mogą być ciężkimi, wielostronicowymi dokumentami, a ich zawartość często różni się od faktycznie powstającego projektu. Podstawowe składniki (takie jak organizacja projektu, harmonogram, procesy i konwencje), które są wymaganymi

częściami wspominanych wyżej norm[5], nie wymagają szczególnej uwagi w projektach opartych na Scrum, ponieważ są już integralną częścią platformy Scrum. Jednakże inne aspekty planowania, takie jak zarządzanie dostawcami, strategie testowania i infrastruktura testowa nie stają się automatycznie bezużyteczne, kiedy zespół projektowy korzysta z procesu Scrum.

Korzystanie z IEEE 730 oraz IEEE 829 jako list kontrolnych

O ile projekt nie podlega regulacjom, które wymagają planowania zgodnego z normami (zobacz podrozdział 7.3), projekt Scrum może łatwo obejść się bez tych planów i może korzystać z założeń standardów jako list kontrolnych. Mistrz Scrum wykorzystuje je jako część retrospektyw do określenia, czy wymagania pochodzące z tych standardów są istotne dla danego projektu. Tam gdzie dana regulacja jest konieczna, zespół nakreśla własne reguły, które są następnie umieszczane w karcie zespołu. Karta ta zawiera też reguły dotyczące testowania i zarządzania jakością (zobacz rozdział 7). Główną różnicą pomiędzy kartą zespołu a tradycyjną dokumentacją zarządzania jakością jest to, że nie zawiera ona żadnych reguł nakreślonych na zewnątrz, które byłyby wmuszane zespołowi. Podobnie do zaległości sprintu reprezentuje natomiast dobrowolną deklarację ze strony zespołu. Ponieważ reguły te są dobrowolne i są często oparte na znanych praktykach zwinnych, są zwykle proste i zwarte. Zespół eHome nakreślił następujące reguły:

Studium przypadku 3-5 *Karta zespołu projektu sterownika eHome*

Karta zespołu

– Jak będziemy implementować Scrum w projekcie sterownika eHome –

Sprint

- Czas trwania: 4 tygodnie
- Każdy sprint jest poprzedzany fazą planowania sprintu i kończy się przeglądem

Spotkania

- Codzienny Scrum: O 11 przed południem, maksymalny czas trwania 15 minut, właściciel produktu bierze udział
- Planowanie sprintu: Środa o 9 rano (na początku sprintu), maksymalny czas trwania 1 dzień
- Przegląd: Wtorek o 11 przed południem (na końcu sprintu)

Praktyki

- Poker planowania:
  - Ocena w punktach (Story Points – SP), minimum 1, maksimum 13 punktów na zadanie
  - Rozbijanie złożonych zadań na mniejsze jednostki
  - Kryteria przyjęcia uzgadniane i notowane na piśmie

5 Rozdział 7 dokładniej zajmuje się zawartością i strukturą tego typu planów.

3.7 Planowanie testów i zarządzanie testami

3.7.1 Tradycyjne zarządzanie testami

W tradycyjnych projektach zarządzanych przy użyciu modelu V menedżer testów ma zwykle następujące obowiązki:

- Organizowanie testu: Planowanie i zapewnianie wymaganych zasobów (pracowników, infrastruktury, narzędzi), organizacja zespołu i zapewnianie dostępności odpowiednich umiejętności.
- Definiowanie strategii testów: Cele, obiekty, wybór odpowiednich metod, definiowanie kryteriów rozpoczęcia i zakończenia, szacowanie nakładów i kosztów, ocena ryzyka i ciągłe rozwijanie strategii testowania zgodnie z wynikami i postępami projektu.
- Definiowanie zawartości testów: Zawartość, zakres i priorytety.
- Zarządzanie testami: Przydzielanie zadań związanych z testowaniem i koordynowanie ich na bieżąco, monitorowanie postępów przy użyciu odpowiednich miar, analizowanie i ogłaszanie wyników.
- Konsultacje: Wspieranie zarządzania projektem przez sprawy związane z zarządzaniem jakością i pomaganie w podjęciu decyzji o wypuszczeniu wersji produktu.

Tradycyjny menedżer testów spełnia więc dwie główne role:

- Organizuje i prowadzi zespół testowy jako szef projektu odpowiedzialny za podprojekt „testowanie".
- Planuje, co zostanie przetestowane (i jak) w roli eksperta od testowania oprogramowania.

3.7.2 Zarządzanie testami w Scrum

Zespół Scrum jest samodzielnie organizującym się, wielofunkcyjnym zespołem, który nie pracuje pod nadzorem menedżera projektu. Scrum przekazuje odpowiedzialność za całą pracę całemu zespołowi. Kodowanie i testowanie odbywa się w tym samym zespole i nie jest traktowane oddzielnie.

Zespół Scrum nie ma wyznaczonego menedżera testów.

W konsekwencji zespół Scrum nie ma wyznaczonego menedżera testów i rozdziela odpowiedzialność związaną z tą rolą wewnątrz zespołu.

Organizowanie testów jest obowiązkiem mistrza Scrum. Jeśli narzędzia lub infrastruktura testowa wymagają poprawienia lub zastąpienia albo jeśli potrzebne umiejętności nie są dostępne wewnątrz zespołu,

mistrz Scrum odpowiada za zapewnienie, że te przeszkody zostaną usunięte.

Ogólna rola menedżera jest obejmowana przez podstawowe praktyki Scrum, a zadania testowe są albo planowane bezpośrednio jako zadania wewnątrz sprintu, albo pośrednio jako część kryteriów realizacji innych zadań. Monitorowanie i analizowanie postępów testowania oraz jego wyników stanowi część wysoce zautomatyzowanego aspektu Scrum – ciągłej integracji (zobacz podrozdział 5.5), co sprawia, że ręczne monitorowanie testów jest niemal całkowicie niepotrzebne, a tradycyjna rola menedżera testów jest zbędna.

W teorii, Scrum oczekuje od całego zespołu odpowiedzialności za definiowanie strategii i zawartości testów. Jednakże testowanie oprogramowania wymaga umiejętności specjalistycznych, a doświadczenie w testowaniu jest konieczne, jeśli zespół ma podjąć odpowiednią decyzję.

Zespół potrzebuje osoby mającej doświadczenie w testowaniu.

Dlatego zespół powinien zawierać co najmniej jednego odpowiednio wykwalifikowanego członka poświęcającego się testowaniu[6]. Osoba ta jest następnie odpowiedzialna za projektowanie odpowiednich testów i implementowanie ich we wszystkich sprintach. Rola ta obejmuje też doradzanie właścicielowi produktu w sprawach związanych z zarządzaniem jakością i wypuszczaniem wersji produktu. Nie ma powodu, żeby nie nazywać tej roli menedżerem testów w zespole Scrum, a niezależni specjaliści od testowania oferujący metodyczne wsparcie są często dobrymi kandydatami do tej roli.

Zarządzanie usterkami

Zarządzanie usterkami odgrywa jedynie niewielką rolę w scenariuszach zarządzania projektami zwinnymi. Dzieje się tak dlatego, że większość usterek można odtworzyć na żądanie przy użyciu zautomatyzowanych procedur testowych, wszyscy członkowie zespołu stale blisko ze sobą współpracują[7] i zwykle są już świadomi wszelkich usterek lub problemów z kodem, a większość usterek jest naprawianych od razu po ich odkryciu. Kombinacja tych czynników oznacza, że wprowadzenie usterki do systemu zarządzania usterkami zwykle nie zwiększa wartości procesu. Osoba odpowiedzialna za naprawienie usterki wie już o tym, więc nie ma potrzeby rejestrowania tego na później. Jednakże nawet Scrum nie zawsze daje taki idealny scenariusz, co sprawia, że konieczne jest implementowanie systemu zarządzania usterkami opartego na narzędziach[8]. Usterka jest wprowadzana do systemu, jeśli:

---

6 Powszechnie przyjętym standardem międzynarodowym jest program ISTQB® Certified Tester. Więcej szczegółów można znaleźć w podrozdziale 7.

7 W oparciu o podstawowe zasady XP, takie jak programowanie w parach (zobacz [Beck/Andres 04]).

8 Wymagania dla zarządzania usterkami i struktura raportów dotyczących usterek są zarysowane w IEEE 829, IEEE 1044, ISO/IEC/IEEE 29119-2.

- Zostaje odkryta podczas ręcznego testowania (lub innej procedury), a nie przez zwykłe, zautomatyzowane systemy testowe. W tym przypadku zarejestrowanie usterki jest konieczne, jeśli trzeba będzie ją odtworzyć.
- Wymaga dalszej analizy lub obsługi obejmujących dodatkowe (wewnętrzne lub zewnętrzne) zasoby. W tym przypadku udokumentowanie usterki zapewni, że wszystkie zainteresowane osoby zostaną poinformowane (ma to często miejsce podczas testów integracyjnych i systemowych).
- Nie może lub nie powinna być usuwana podczas bieżącego sprintu. To zapewnia, że dany błąd nie zostanie zapomniany.

3.7.3 Poziomy testowania w Scrum

W projektach opartych na Scrum konieczne jest przeprowadzanie testów na wszystkich poziomach, jak w tradycyjnym projekcie opartym na modelu V (zobacz rys. 2-3). Różne poziomy testów są oparte na różnych wymaganiach technicznych i dążą do różnych celów, wymagają więc zastosowania różnych metod testowania i wiedzy specjalistycznej.

Zasady testowania ugruntowane w modelu V nadają się również do wykorzystania w projektach zwinnych:

Zasady modelu V nadal obowiązują.

- Działania programistyczne i testowe (reprezentowane przez lewą i prawą stronę V) są równie ważne dla powodzenia projektu.
- Testy przeprowadzane na określonym poziomie służą sprawdzaniu produktu na określonym poziomie abstrakcji.
- Testy mogą służyć albo do sprawdzania poprawności (tzn. czy opracowaliśmy prawidłowy produkt?), albo do weryfikacji (tzn. czy opracowaliśmy produkt poprawnie?).

Jednakże w projekcie opartym na Scrum te fazy testowania nie są wykonywane sekwencyjnie, ale odbywają się równolegle do siebie w ramach każdego sprintu, najlepiej codziennie! Rozdziały 4, 5 i 6 przedstawiają szczegółowo, jak działa to podejście.

Fazy testowania odbywają się równolegle, a nie sekwencyjnie.

3.8 Wprowadzenie do planowania zwinnego

Jeśli wprowadzamy Scrum w zespole po raz pierwszy albo rozpoczynamy nowy projekt Scrum, to mistrz Scrum musi upewnić się, że właściciel produktu zacznie od opracowania razem z zespołem wizji produktu, wizji architektury i mapy scenariuszy. We wszystkich trzech przypadkach zwięzłość jest znacznie ważniejsza od poziomu szczegółowości:

- Lista dziesięciu najważniejszych funkcji produktu jest bardziej przydatna niż długi arkusz specyfikacji.
- Jasny szkic planowanej architektury systemu jest bardziej przydatny niż skomplikowany diagram UML.
- Przydzielenie zadań o najwyższych priorytetach do następnych trzech sprintów jest bardziej przydatne niż próba przydzielenia całej zawartości listy zaległości do całego ciągu przyszłych sprintów.

Istotne jest, aby osoba, która utrzymuje listę zaległości, regularnie pilnowała, żeby wszystkie zadania istotne z punktu widzenia produktu i pracy zespołu były zawarte w liście zaległości.

Zapobieganie tworzeniu dodatkowych list zaległości

W przeciwnym razie członkowie zespołu szybko zaczną tworzyć dodatkowe listy zaległości składające się z prywatnych list zadań do zrobienia. Jeśli jacyś członkowie zespołu pracują równocześnie nad innymi projektami, plan sprintów musi brać pod uwagę wynikający z tego niedobór zasobów. Na przykład członek zespołu, który jest też odpowiedzialny za odbieranie telefonów z zapytaniami od klienta, nie może być w 100% zaangażowany w sprint. Zespół nie może zakładać, że mniej zapytań od klienta zostanie zgłoszonych podczas danego sprintu.

Na koniec mistrz Scrum musi zapewnić, że praca nad testami nie zostanie zaniedbana lub zapomniana.

3.9 Pytania i ćwiczenia

3.9.1 Samoocena

Pytania i ćwiczenia pomagające w ocenie, jak zwinny jest naprawdę projekt lub zespół.

1. Czy wizja produktu i wizja architektury dla mojego produktu/projektu jest zdefiniowana? Jakie są te wizje?
2. Kto opracował te dokumenty? Kto się nimi zajmuje?

3. Czy lista zaległości produktowych jest dobrze zorganizowana? Jaka jest jej struktura? Kto ją obsługuje?
4. Jakie kryteria są używane do nadawania priorytetów elementom listy zaległości?
5. Czy mamy ogólny plan obejmujący więcej niż tylko następne trzy sprinty? Czy korzystamy z mapy scenariuszy? W jakim stopniu istniejące plany są wiążące?
6. Czy lista zaległości sprintu jest dobrze zorganizowana? Czy ma dobrą strukturę?
7. Kto obsługuje listę zaległości sprintu? Kto może edytować elementy listy zaległości? Czy aktywny sprint jest chroniony przed zmianami?
8. Jaka jest nasza definicja gotowości?
9. Czy istnieje karta zespołu? Jeśli nie, kto definiuje zasady działania zespołu i w jaki sposób?
10. Jak radzimy sobie z zarządzaniem testami? Czy mamy wyznaczonego menedżera testów? Kto spełnia rolę zarządzania testami?
11. Jakie różne poziomy testów zdefiniowaliśmy i z których korzystamy?

3.9.2 Metody i techniki

Te pytania pomogą w podsumowaniu treści bieżącego rozdziału.

1. Jaki jest cel wizji architektury?
2. Jaka jest różnica pomiędzy listą zaległości produktowych a listą zaległości sprintu?
3. Jaki jest cel mapy scenariuszy?
4. Dlaczego musimy dokładnie szacować nakłady związane ze wszystkimi planowanymi zadaniami podczas planowania sprintu?
5. Co oznacza, kiedy mówimy, że sprint jest chroniony?
6. W jaki sposób zadania testowania i zarządzania jakością są reprezentowane na liście zaległości sprintu?
7. Jaki jest cel karty zespołu?

3.9.3 Inne ćwiczenia

Te ćwiczenia pomogą zagłębić się w zagadnienia poruszone w trakcie tego rozdziału.

1. Wyjaśnić, które aspekty wizji architektury oraz architektury planowanego produktu wpływają na planowanie sprintu.

2. Naszkicować dodatkowe kryteria przyjęcia brakujące w studium przypadku 3-3.
3. Zaprojektować testy, których można by użyć do ustalenia, czy produkt spełnia te kryteria.
4. Do którego poziomu modelu V można by przypisać te testy?

4 Testy jednostkowe i programowanie sterowane testami

Ten rozdział objaśnia, czym są testy jednostkowe i jak można je zautomatyzować. Jest skierowany do testerów systemowych oraz specjalistów od testowania, a także członków zespołu mających niewielkie doświadczenie w programowaniu i oferuje podstawowe informacje na temat technik i narzędzi testowych. Materiał ten ma za zadanie pomóc w bliższej współpracy z programistami i testerami. Zawiera też wiele wskazówek, które pomogą zaawansowanym testerom poprawić swoje metody pracy. W oparciu o te założenia wprowadzamy też pojęcie programowania sterowanego testami i wyjaśniamy jego znaczenie w kontekście zwinnego zarządzania projektami.

4.1 Testowanie jednostkowe

Termin „testowanie jednostkowe" jest często traktowany jako synonim „testowanie przez programistę" i służy do opisania wszystkich testów pisanych przez programistów dla swojego oprogramowania. Różnica pomiędzy tymi dwoma podejściami ma pewien sens w projektach tradycyjnych, w których wydzielone są osobne zespoły do testowania integracyjnego i testowania systemowego, ale nie ma zastosowania w projektach Scrum, w których wszystkie zadania obsługiwane są przez cały zespół. W środowisku Scrum większy sens ma rozróżnianie pomiędzy różnymi obiektami, które wymagają testowania.

Testy jednostkowe sprawdzają wewnętrzne działanie pojedynczego, niezależnego składnika.

Termin „test jednostkowy"[1] opisuje wszystkie dynamiczne[2] testy, które służą do sprawdzania funkcjonalności pojedynczego, niezależnego

1 Słownik ISTQB [URL: ISTQB Glossary] wykorzystuje pojęcie „testów składnikowych". Jednakże „testy jednostkowe" są szerzej stosowane w środowisku programistów.

2 Test dynamiczny uruchamia obiekt jako część testu. Natomiast sprawdzenie statyczne analizuje strukturę obiektu, ale nie wykonuje jego kodu (zobacz [Spillner/Linz 14]).

składnika oprogramowania. Według [Spillner/Linz 14] wyróżniającą cechą testów jednostkowych jest to, że służą do sprawdzania składników niezależnie od innych elementów systemu. Ta izolacja służy wyeliminowaniu wpływów zewnętrznych na dany składnik. Jeśli test jednostkowy wykryje usterkę, jej źródło z pewnością będzie znajdować się w testowanym składniku.

W środowiskach kodowania, które nie są zorientowane obiektowo, funkcje, moduły i skrypty działają jako samodzielne składniki, natomiast programowanie zorientowane obiektowo traktuje klasy i związane z nimi metody jako najmniejsze możliwe elementy programu, które mogą być wykonywane (i testowane) w izolacji. Ponieważ programowanie zorientowane obiektowo jest obecnie najpowszechniej stosowanym podejściem, kolejne podrozdziały wyjaśniają testowanie jednostkowe stosowane wobec klas[3].

4.1.1 Klasy i obiekty

Klasa składa się z zestawu zmiennych (znanych też jako atrybuty lub właściwości) oraz zestawu metod (znanych też jako funkcje w niektórych językach programowania). W zależności od ich typu zmienne te mogą mieć przypisane różne wartości. Metody działają na zmiennych i manipulują ich wartościami.

Jednakże sama klasa jest konstrukcją abstrakcyjną i tylko podczas wykonywania tworzy obiekty, które istnieją w pamięci komputera, na którym uruchomiono kod. Obiekty te są wystąpieniami zdefiniowanej klasy, a ich zachowanie jest określane przez metody związane z daną klasą. Wartości zmiennych wewnątrz obiektu określają jego stan w danym momencie, a obiekt może zmieniać swój stan w czasie, jeśli wartości zmiennych się zmienią. W ten sposób wiele obiektów utworzonych z tej samej klasy może mieć różne stany. Klasa `Device`[4] w oprogramowaniu sterownika eHome jest idealnym przykładem:

3 Jednostka przeznaczona do przetestowania może się składać z wielu podstawowych składników. Wszystkie odniesienia do testowania w tym rozdziale można równie dobrze zastosować względem jednostek prostych, jak i złożonych.

4 Podobnie jak wszystkie przykłady kodu w tej książce, ta klasa przykładowa jest napisana w PHP. Język PHP definiuje metody przy użyciu słowa `function`. Identyfikatory zmiennych zaczynają się od znaku dolara `$`. Operatorem dostępu do obiektów, zmiennych i metod jest `->`. [Gutmans i in. 05] oferuje dobre wprowadzenie do PHP, a [Schlossnagle 04] przedstawia dokładniej bardziej szczegółowe pojęcia. Zobacz też [URL: PHP]. Testowanie (jednostkowe) dla PHP objaśniono w [Bergmann/Priebsch 11].

Studium przypadku 4-1a

Studium przypadku 4-1a dla sterownika eHome: Klasa Device

Interfejs użytkownika sterownika eHome jest zaprojektowany tak, aby wyświetlać aktualny stan włączenia każdego z urządzeń do niego podłączonych (zobacz element listy zaległości produktowych „Sterowanie i monitorowanie" w Studium przypadku 3-3).

Aby spełnić ten cel, zespół programuje klasę o nazwie `Device`, a każdy obiekt tej klasy reprezentuje określone urządzenie (na przykład określoną lampę). Klasa zawiera zmienną `$status`, która opisuje obecny stan danego urządzenia. Metody `set_status()` i `get_status()` służą do zmieniania i odczytywania bieżącego stanu:

```
class Device {                // wersja 1
   public $name ='';
   public $status ='';
   public function __construct($my_name) {
      $this->name = $my_name;
   }
   public function set_status ($new_status) {
      $this->status = $new_status;
   }
   public function get_status () {
      return $this->status;
   }
}
```

Ta klasa istnieje jako element kodu w systemie eHome, ale może służyć do generowania dowolnej liczby wystąpień urządzeń podczas działania programu – na przykład „światło w kuchni" albo „gniazdko w przedpokoju". Wszystkie te obiekty zachowują się w taki sam sposób określony przez klasę `Device`. Każde urządzenie, czyli każdy obiekt urządzenia może przy tym mieć swój własny stan w danym momencie. Na przykład światło w kuchni jest włączone, natomiast gniazdko w przedpokoju jest wyłączone.

Te podstawowe cechy klas i obiektów pomogą nam zdefiniować zadania wymagane do ich przetestowania. Musimy mieć pewność, że klasa działa poprawnie, dla wszystkich potencjalnych stanów obiektów. W kolejnych dwóch podrozdziałach wyjaśniamy odpowiednie metody testowania.

4.1.2 Testowanie metod klasy

Jakie podejście musimy zastosować do testów jednostkowych klasy? Nasuwa się oczywista odpowiedź „Każda metoda zawarta w klasie wymaga przetestowania". Zespół eHome pisze klasę testową o nazwie `DeviceTest` do testowania klasy `Device`.

Studium przypadku 4-1b

Studium przypadku 4-1b dla sterownika eHome: Klasa testowa DeviceTest

```
include 'Device.php';           // testowana klasa
class DeviceTest {              // wersja 1
   public function test_KitchenLightOn() {
   $device = new Device('kitchen light');
                                // przygotowanie
   $device->set_status('on');   // procedura testowa
   if ($device->status == 'on') // sprawdzenie
      $myResult = TRUE;
   else   $myResult = FALSE;
   unset($device);              // zakończenie
   return $myResult;
   }
}
$myTestSuite = new DeviceTest();
$myTestResult = $myTestSuite->test_KitchenLightOn();
                          // wykonaj przypadek testowy
if ($myTestResult == TRUE)
   echo 'passed';
else
   echo 'failed';
```

Pokazana powyżej klasa `DeviceTest` zawiera tylko jeden prosty test o nazwie `test_KitchenLightOn`, który służy do pokazania zasad związanych z tworzeniem testu jednostkowego.

- Przypadki testowe są budowane zgodnie ze ścisłym wzorcem:
  - **Przygotowanie:** Następuje utworzenie i zainicjowanie obiektu testowego. W naszym przykładzie jest to obiekt o nazwie `kitchen light`.
  - **Procedura testowa:** Przeprowadzane jest właściwe testowanie. W naszym przypadku wykorzystujemy metodę `set_status()` do zmiany stanu obiektu `kitchen light` na on i odczytujemy zmieniony stan.
  - **Sprawdzenie**: Wykorzystujemy porównanie, aby zobaczyć, czy test się powiódł, czy nie. W tym przypadku test ma sprawdzić, czy metoda `set_status()` działa poprawnie, więc po prostu sprawdzamy, czy wartość zmiennej `status` jest taka sama jak wartość ustawiona przy użyciu metody `set_status()`.
  - **Zakończenie:** Następnie podejmowane są odpowiednie kroki, aby przywrócić system do pierwotnego stanu. To zapewnia, że test pozostawi testowany obiekt w pierwotnej postaci. Jeśli będziemy ściśle trzymać się tego podejścia, przypadki testowe pozostaną niezależne od siebie i będzie można przeprowadzać wszystkie testy w dowolnej kolejności. W naszym przykładzie

obiekt `kitchen light` utworzony na potrzeby przeprowadzenia testu jest usuwany na końcu procedury[5].

- Przypadek testowy służy do przetestowania tylko jednej metody.
  - Jeśli części **procedura testowa** i **sprawdzenie** zostaną rozszerzone o poniższe polecenia, to druga metoda `get_status()` może zostać przetestowana podczas tego samego przebiegu:

```
...
   $device = new Device('kitchen light');
                                    // przygotowanie
   $device->set_status('on');
                          // procedura testowa krok 1
   $status_read = $device->get_status();
                          // procedura testowa krok 2
   if ($status_read == 'on')        // sprawdzenie
      $myResult = TRUE;
   else   $myResult = FALSE;
   unset($device);                  // zakończenie
...
```

  - Jeśli jednak test się nie powiedzie i zwróci wartość `failed`, to tester nie będzie wiedział, czy to metoda `set_status()`, czy też `get_status()` spowodowała błąd. Utrudnia to analizę usterek. Za dobry styl kodowania uważa się ograniczanie testów do tylko jednego aspektu funkcjonalności programu.

Podczas działania testu `test_KitchenLightOn` wykonywanych jest sześć z ośmiu wierszy kodu składających się na klasę `Device`, co odpowiada pokryciu 6/8=75% wierszy kodu. Wersja zawierająca test `get_status()` (wykorzystująca `$status_read = $device->get_status()`) daje 100% pokrycia linii kodu! Wynika to jedynie z prostoty klasy `Device`, a uznanie, że testowany obiekt jest poprawny, byłoby błędem. Klasa `Device` ma w istocie kilka poważnych wad:

- Specyfikacja zmiennej `$status` mówi, że powinna wyświetlać aktualny stan włączenia urządzenia korzystając z wartości `on` i `off`. Jak na razie wszystko w porządku. Niestety w swojej obecnej formie zmienna `$status` może też przyjmować dowolne inne wartości – na przykład 30%, co nie ma sensu dla gniazdka elektrycznego. Klasa `Device` musi więc zapewniać, żeby zmiennej `$status` można było przypisywać tylko wartości `on` lub `off`.

5 Część „zakończenie" nie jest tutaj niezbędnie konieczna, ponieważ PHP automatycznie usuwa obiekt testowy przy wyjściu z metody `test_KitchenLightOn()`. Ręczne kodowanie takich procedur czyszczących jest konieczne jedynie w językach takich, jak C lub C++, które nie robią tego automatycznie.

Możemy to osiągnąć dodając odpowiednie sprawdzenie do metody `set_status()`.

- Po wbudowaniu tego sprawdzenia musimy też zapewnić, żeby nie dało się bezpośrednio manipulować zmienną `$status` (tzn. z pominięciem `set_status()`).Wszystkie zmienne w tej klasie mają aktualnie zasięg `public`, co oznacza, że nawet jeśli poprawiona metoda `set_status()` gwarantuje, że `$status` może przyjmować tylko wartości `on` lub `off`, to nadal możliwe jest użycie instrukcji `$device->status='30%'` w celu przypisania nieprawidłowej wartości do zmiennej. To oznacza, że klasa może przyjmować stany, które nie mogą być przetwarzane przez jej metody i są niezgodne z zaplanowaną funkcjonalnością. Musimy więc zapewnić, żeby nie było bezpośredniego dostępu do zmiennej `$status` deklarując ją jako `private`.
- Zmienna `$name` również musi zostać zadeklarowana jako `private`. Inną słabością zmiennej `$name` jest to, że jest ustawiana podczas tworzenia obiektu przy użyciu konstruktora klasy `__construct()`, ale nie można jej później odczytywać. W tym celu musimy dodać metodę o nazwie `get_name()`. Gdy tylko zadeklarujemy zmienną `$name` jako prywatną, pojawi się kwestia metody `set_name()`.Specyfikacja nie jest jasna w tym zakresie. Czy metoda `set_name()` jest konieczna, czy nie, zależy od natury urządzeń elektrycznych, którymi chcemy sterować. Jeśli założymy, że te urządzenia są mobilne, ich nazwy muszą być elastyczne (na przykład `kitchen light` może zmienić się w `hall light`). Jeśli jednak użyjemy nazwy `kitchen light` do zdefiniowania lampy podłączonej kablem na środku sufitu w kuchni, to stała nazwa może być lepszym odzwierciedleniem rzeczywistości.

Żadna z tych wad przykładowej klasy `Device` nie jest uwzględniana w teście jednostkowym pokazanym powyżej. Warto wyciągnąć z tego następujące wnioski:

- Pokrycie wierszy kodu w stopniu mniejszym niż 100% dowodzi, że pewne przypadki zostały pominięte. Jednakże argument przeciwny nie jest prawdziwy – pokrycie 100% wierszy kodu nie oznacza automatycznie, że przypadki testowe obejmują wszystkie ewentualności.
- Nawet test jednostkowy potrzebuje przypadków testowych opartych na wymaganiach lub specyfikacji. Jeśli testy jednostkowe będą zaprojektowane jako testy strukturalne (testy białej skrzynki), to niemożliwe będzie określenie, czy konkretne wymagania zostały błędnie zinterpretowane, czy po prostu zignorowane.

- Do sprawdzania konkretnych atrybutów kodu (na przykład „wszystkie zmienne są prywatne") lepiej nadają się techniki przeglądu kodu lub analizy statycznej niż testy dynamiczne.
- Ogólna reguła mówiąca, że „każda metoda musi zostać przetestowana" jest nic nie warta, a jeden przypadek testowy na metodę nie wystarcza (za wyjątkiem metod bezparametrowych). Nawet nasza dość trywialna metoda `set_status()` wymaga co najmniej czterech przypadków testowych, aby uwzględnić każdą z wartości `'on'`, `'off'`, `'30%'` oraz `''`[6]. Musimy określić wystarczającą liczbę przypadków testowych dla każdej metody korzystając z partycjonowania równoważnościowego i analizy wartości granicznych[7].

Projektowanie kodu a nakłady na testowanie

Dzięki tym spostrzeżeniom jasno widać, że liczba wymaganych przypadków testowych i potrzebnego zaangażowania w ich utworzenie zależy w znacznym stopniu od projektu kodu źródłowego. Duża liczba parametrów i skomplikowane, zagnieżdżone warunki zwiększają wymagane nakłady pracy. Podczas przeglądu kodu zespół musi sprawdzać, czy każda metoda może być poddana refaktoringowi tak, aby zawierała mniej parametrów i/lub upraszczała warunki sprawdzające.

Po zidentyfikowaniu tych problemów zespół eHome stworzył następującą poprawioną wersję klasy `Device`:

Studium przypadku 4-2a

Studium przypadku 4-2a dla sterownika eHome: Poprawiona klasa Device

```
class Device {        // wersja 2
   private $name;
   private $status;
   public function __construct($my_name) {
   $this->name = $my_name;
   $this->status = 'unknown';
   }
   public function get_name () {
   return $this->name;
   }
   public function set_status ($new_status) {
   if (is_validStatus($new_status)){
      $this->status = $new_status;
      return TRUE;
      }
   else return FALSE;
   }
  public function get_status () {
   return $this->status;
   }
```

ciąg dalszy na następnej stronie

6 Reprezentują one dwie poprawne wartości, wartość nieprawidłową i parametr, który nie został ustawiony (tzn. pusty łańcuch znaków).

7 Te techniki projektowania testów opisano w [Spillner/Linz 14].

```
   private function is_validStatus ($status) {
   if ($status=='on' OR $status=='off')
      return TRUE;
   else    return FALSE;
   }
}
```

Metody `public` w nowej wersji klasy reprezentują jej interfejs API i dlatego muszą być objęte wystarczającą liczbą przypadków testowych. Jako ćwiczenie sprawdzimy pokrycie wierszy kodu wynikające z wykonania przypadku `test_KitchenLightOn` dla nowej wersji klasy `Device` (zobacz też podrozdział 4.6.2). Zespół również omówił tę kwestię i zakodował nową wersję klasy `DeviceTest`:

Studium przypadku 4-2b

Studium przypadku 4-2b dla sterownika eHome: Poprawiona klasa testowa DeviceTest

Programista odpowiedzialny za klasę testową dodał przypadek testowy obejmujący wartość stanu `'30%'`. Ponieważ zmienna `$status` (w klasie `Device`) jest teraz zadeklarowana jako prywatna, przypadek testowy nie może teraz sprawdzać wartości `$status` bezpośrednio i musi korzystać z metody `getStatus()` obiektu testowego.

Wersja 2.1 klasy `DeviceTest` zawiera teraz dwa podobne przypadki testowe, które różnią się tylko zawartością kodu procedury testowej. Programista poprawił też ogólną jakość kodu przenosząc powtarzający się kod do klasy wyższego poziomu o nazwie `TestFrame`[a]. Dołączono też dodatkowe przypadki testowe dla wartości `'off'` oraz `''`. Wersja 2.2 klasy `DeviceTest` wygląda następująco:

```
include 'Device.php';      // klasa do przetestowania
include 'TestFrame.php';   // prosta platforma testów
                           // jednostkowych
class DeviceTest extends TestFrame {    // wersja 2.2
   public function test_KitchenLightOn() {
   $device = new Device('kitchen light');
                                  // przygotowanie
   $device->set_status('on');   // procedura testowa
   $this->assertEquals('on', $device->get_status(),
      'KitchenLightOn');          // sprawdzenie
   unset($device);                // zakończenie
   }
   public function test_KitchenLightOff() {
   $device = new Device('kitchen light');
                                  // przygotowanie
   $device->set_status('off');   // procedura testowa
   $this->assertEquals('off', $device->get_status(),
      'KitchenLightOff');        // sprawdzenie
```

a Kod źródłowy dla tej i innych klas przykładowych jest dostępny do pobrania ze strony WWW książki [URL: SWT-knowledge].

```
    unset($device);                // zakończenie
    }
    public function test_setStatusInvalid30() {
    $device = new Device('kitchen light');
                                    // przygotowanie
    $device->set_status('30%');  // procedura testowa
    $this->assertEquals('unknown', $device->get_
status(), 'setStatusInvalid30');    // sprawdzenie
    unset($device);                 // zakończenie
    }
    public function test_setStatusInvalidEmpty() {
    $device = new Device('kitchen light');
                                     // przygotowanie
    $device->set_status('');      // procedura testowa
    $this->assertEquals('unknown', $device->get_
status(),
       'setStatusInvalidEmpty');      // sprawdzenie
    unset($device);                   // zakończenie
    }
}
$myTestSuite = new DeviceTest();
$myTestSuite->test_KitchenLightOn();
                      // wykonaj przypadek testowy 1
$myTestSuite->test_KitchenLightOff();
                      // wykonaj przypadek testowy 2
$myTestSuite->test_setStatusInvalid30();
                      // wykonaj przypadek testowy 3
$myTestSuite->test_setStatusInvalidEmpty();
                      // wykonaj przypadek testowy 4
$myTestSuite->printResult();
```

Wykonanie testu daje 100% pokrycia wierszy kodu. Jednakże w przedstawionym dalej przeglądzie przypadków testowych tester zauważa, że istniejące przypadki nadal niewystarczająco testują całą klasę. Wynika to z trzech zmian w kodzie w porównaniu do wersji 1 klasy `Device`:

- Konstruktor inicjuje stan nową wartością `unknown`
- `set_status()` pozwala tylko na prawidłowe wartości stanu
- Sprawdzanie poprawności wartości stanu jest teraz przeprowadzane przez metodę `is_validStatus()`

Najważniejszą nową funkcjonalnością w klasie `Device` jest nowa metoda `is_validStatus()` decydująca, które wartości stanu są poprawne. Metoda `is_validStatus()` musi zostać przetestowana, aby zapewnić, że właściwie rozpoznaje prawidłowe wartości i odrzuca nieprawidłowe.

Aktualne przypadki testowe sprawdzają jedynie zachowanie metody `set_status()`. Wywołuje ona metodę `is_validStatus()` pośrednio, ale jedynie powierzchownie sprawdza jej funkcjonalność. Obecnie metoda `is_validStatus()` klasyfikuje wartość stanu `'unknown'` jako nieprawidłową. Nie jest jasne, jak powinien zachować się program, ale jest wiele powodów, dla których wywołanie `is_validStatus('unknown')` powinno zwracać wartość `TRUE`.

Ćwiczenia 4.6.2 – 3 pomogą nam w pracy nad tym przykładem.

Czy reguła „każda metoda musi zostać przetestowana" odnosi się również do metod prywatnych?

- **W teorii, tak:** W przeciwnym razie metody typu `private` będą testowane tylko pośrednio poprzez wywołania z metod typu `public` – taka praktyka nie zapewnia, że metody `private` będą odpowiednio przetestowane, nawet jeśli wszystkie przypadki testowe zakończą się powodzeniem. Metoda typu `private` może zawierać (wadliwy) kod, który nie będzie uruchamiany przez wywołującą ją metodę typu `public`, albo może zawierać kod, który działa poprawnie tylko w kontekście wywołującej ją metody typu `public`, ale w istocie zawiera usterki, które mogą się objawiać tylko wtedy, gdy klasa zostanie przebudowana albo będzie wykorzystywana w innym kontekście przez inne klasy.
- **Praktycznie mówiąc, nie:** W praktyce, wielu testerów zadowala się testowaniem tylko klas interfejsu API klasy (tzn. jej metod i zmiennych typu `public`)[8]. Problem przy tym podejściu stanowią dwa potencjalnie niebezpieczne założenia:

  a) Metody `public` są testowane wystarczająco dokładnie tak, że wszystkie pośrednio wywoływane metody `private` są jednocześnie automatycznie testowane (wystarczająco dokładnie).
  b) Gdy edytowana jest metoda `private`, wszelkie nowe usterki będą wykrywane przez niepowodzenie co najmniej jednego przypadku testowego dla metod `public`.

Nawet jeśli przypadki testowe dla metod `public` są dokładne, nie gwarantuje to (choć jest to możliwe), że metody `private` będą odpowiednio przetestowane. W tym przypadku „dokładnie" oznacza:

- Każda metoda `public` wymaga przypadków testowych definiowanych przy użyciu partycjonowania równoważnościowego i analizy wartości granicznych dla wszystkich parametrów metody.
- Przypadki testowe dla każdej metody `private` powinny być przeglądane w celu sprawdzenia, że wszystkie klasy równoważnościowe i wartości graniczne są uwzględniane dla wszystkich parametrów[9]. Jeśli nie jest to prawdą i nie ma dodatkowych

8 [Meszaros 07, p. 40] nazywa to zasadą „korzystania najpierw z drzwi frontowych".

9 100% pokrycie wierszy kodu nie jest samo w sobie wystarczającym kryterium. Prywatna metoda `p(x)` mogłaby mieć parametr `x`, który przykładowo jest przetwarzany tylko dla wartości `x<100`. Odgałęzienie `x>=100` jest ignorowane. Jeśli testy API wywołują ten parametr tylko z wartościami `<100`, usterka ta pozostanie niewykryta, choć wynikowe pokrycie wierszy kodu wynosi 100%.

przypadków testowych dla interfejsu API, jest to znakiem wadliwego projektu. Może tak być, ponieważ metoda `private` została zakodowana w sposób zbyt ogólny albo ponieważ powinna być przeniesiona do innej (lub nowej) klasy.

Testowanie metod prywatnych

Jeśli zamierzamy testować metody `private` nie tylko pośrednio, ale też bezpośrednio, to staniemy przed problemem niemożności wywoływania metod prywatnych obiektu[10]. Istnieją różne sposoby poradzenia sobie z tą kwestią. Możemy wbudować kod testowy w klasę, którą chcemy przetestować lub możemy go wstrzykiwać podczas testu. Jednakże to podejście zmienia kod produkcyjny, aby był możliwy do testowania i stwarza niebezpieczeństwo, że kod testowy zostanie przypadkowo aktywowany w środowisku produkcyjnym, co potencjalnie może być szkodliwe. Paradoks polega na tym, że choć testowanie ma ograniczać ryzyko, to wstrzykiwanie kodu testowego zwiększa ryzyko błędnego działania produktu.

Nie modyfikować testowanego systemu.

Należy więc unikać rozbudowywania kodu produkcyjnego przez kod testowy. [Meszaros 07] poświęca temu dwie ze swoich 13 zasad dobrego automatyzowania testów: „nie modyfikować testowanego systemu" i „utrzymywać logikę testową poza kodem produkcyjnym".

4.1.3 Testowanie stanu obiektów

Poprzedni podrozdział dotyczył tego, jak zdefiniować wystarczającą liczbę znaczących testów dla metod publicznych klasy. Przypadki testowe zdefiniowane w ten sposób były następnie wykorzystywane do indywidualnego testowania każdej metody.

Ponieważ niestety klasa definiuje zmienne dla swoich metod, takie izolowane testowanie poszczególnych metod nadal nie jest wystarczające dla zapewnienia, że klasa jako całość będzie funkcjonować poprawnie. W dowolnym momencie każdy obiekt tworzony przez klasę ma określony stan definiowany przez wartości przypisane do jego zmiennych.

Akcja lub reakcja metody zależy więc nie tylko od wartości parametrów użytych do jej wywołania, ale też od wartości przypisywanych do zmiennych obiektu. Zachowanie obiektu i jego reakcja na określony przypadek testowy zależy więc od ostatniej historii obiektu i stanu, w jakim znajduje się podczas uruchamiania testu. Modele stanów są

10 Ten problem można obejść na różne sposoby w zależności od używanego języka programowania. W C++ klasa testowa lub obiekt testowy mogą być zadeklarowane jako zaprzyjaźnione (friend). PHP wykorzystuje metodę `setAccessible` w interfejsie Reflection API, aby uzyskać dostęp do zmiennych typu `private`.

używane do definiowania i wizualizowania pożądanego zachowania obiektu.

Studium przypadku 4-3a

Sterownik eHome 4-3a: Diagram stanów dla urządzeń eHome

Zespół eHome utworzył następujący diagram stanów dla klasy `Device`:

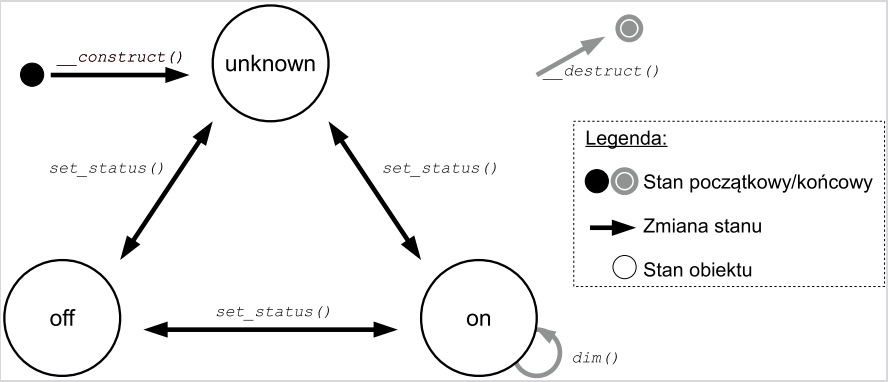


Diagram ten pokazuje trzy możliwe stany włączenia urządzenia (zakodowane w zmiennej `$status` klasy).

Gdy tworzony jest obiekt `Device`, początkowo ma stan '`unknown`'. Może być on następnie zmieniany przy użyciu metody `set_status`. Metodę `get_status` można wywoływać przy dowolnym stanie, ale ona sama nie zmienia stanu. Dla większej jasności diagramu pominęliśmy ją więc, jak również zmienną klasy `$name`.

Teoretycznie zakres wartości wszystkich zmiennych klasy definiuje jej przestrzeń stanów. W naszym przykładzie są to wartości zmiennych `$status` i `$name`. Ponieważ między nimi nie występuje współzależność funkcjonalna, klasa `Device` ma dwie całkiem niezależne przestrzenie stanów. W kontekście bieżącego sprintu interesuje nas przestrzeń określana przez zmienną `$status` i została ona zamodelowana na diagramie.

Testowanie oparte na stanach

Obiekt (albo ogólniej system), którego zachowanie zależy od swojej wcześniejszej historii, nazywa się opartym na stanach i analogicznie powinien być testowany. Następujące informacje są wymagane, aby w pełni zdefiniować przypadek testowy oparty na stanach (za [Spillner/Linz 14, podrozdział 5.1.3]):

1. Stan początkowy testowanego obiektu
2. Wartości parametrów używane do wywoływania testowanej metody
3. Spodziewana reakcja metody lub zwracana przez nią wartość (włączając w to wszelkie potencjalne efekty uboczne)
4. Oczekiwany następny stan testowanego obiektu

Model stanów pokazuje testerowi, których stanów obiektu można oczekiwać podczas jego czasu trwania. Celem testu jest sprawdzenie,

czy obiekt zachowuje się zgodnie ze swoją specyfikacją dla każdego z tych stanów. Według [Vigenschow 10] następujące aspekty każdego stanu muszą być testowane przy użyciu odpowiednich przypadków testowych opartych na stanach:

- Wszystkie prawidłowe wywołania metod, które powinny być dopuszczalne w każdym stanie.
- Wszystkie wywołania metod, które nie są dopuszczalne w każdym stanie i które powinny być odrzucane.

To podejście zapewnia, że każdy stan jest testowany przy użyciu pozytywnych i negatywnych przypadków testowych. Przypadki pozytywne sprawdzają, czy obiekt działa zgodnie ze swoją specyfikacją, natomiast negatywne testują, czy obiekt reaguje prawidłowo na użycie, które jest niezgodne z założeniami. Do usterek, które zwykle ujawniane są przy użyciu tej techniki (zobacz [Vigenschow 10, podrozdział 9.4.1]), należą:

- Brakujące przejścia między stanami: Metoda jest odrzucana, choć powinna być dopuszczalna.
- Niedozwolone przejścia między stanami: Metoda jest przyjmowana, choć powinna być odrzucana.
- Błędne działanie: Metoda daje błędne wyniki.
- Błędny stan następny: Metoda ustawia nieprawidłową wartość zmiennej, co powoduje wygenerowanie nieprawidłowego stanu następnego.

Studium przypadku 4-3b

Studium przypadku 4-3b dla sterownika eHome: Testowanie klasy Device oparte na stanach

Zespół eHome przegląda istniejące przypadki testowe, aby sprawdzić, czy klasa `Device` podlega już testowaniu opartemu na stanach. Dochodzą do następujących wniosków:

Przypadki testowe dla metody `set_status` wymienione w podrozdziale 4.1.2 „Testowanie metod klasy" pasują do punktów 1, 2 i 4 definicji, która określa, że każdy przypadek testowy `set_status` wykorzystuje część inicjującą do ustawienia określonego stanu dla obiektu testowego, a część sprawdzającą do potwierdzenia, że `$status` odpowiada żądanemu stanowi wynikowemu.

Zespół całkiem zapomniał o punkcie 3, a wartości zwracane przez metodę `set_status` (`TRUE/FALSE`) nie są sprawdzane. Porównanie przypadków testowych z diagramem stanów ujawnia też, że brakuje przypadku testowego sprawdzającego przełączanie stanów z włączonego na wyłączony i odwrotnie. Dla użytkownika systemu eHome jest to z pewnością najważniejszy przypadek użycia obiektu `Device`.

Ponadto przypadki testowe nie sprawdzają, czy uwzględniana jest cała przestrzeń stanów, ani czy stany (lub wywołania metod) muszą występować w określonej kolejności.

4.1.4 Kryteria pokrycia kodu w testowaniu opartym na stanach

Jeśli każdy stan w modelu jest objęty przez przypadki testowe, to najprostsze kryterium pokrycia dla testowania opartego na stanach jest całkowicie wypełnione. Jak jednak nasz przykład pokazuje, stuprocentowe pokrycie nie oznacza, że wszystkie przejścia pomiędzy stanami (tzn. wszystkie przypadki graniczne) są sprawdzane. Pokrycie przejść między stanami jest ściślejszym kryterium[11].

Pokrycie stanów i przejść między stanami

Co się stanie, jeśli wykorzystamy strategię testowania opisaną powyżej do sprawdzania wszystkich prawidłowych i nieprawidłowych wywołań metod dla każdego stanu? W naszym przykładzie gwarantuje to, że wszystkie przejścia zostaną przetestowane. W przypadku naszej klasy przykładowej każde przejście odpowiada wywołaniu metody `(set_status)`. Ponieważ wszystkie zmienne klasy są prywatne, nie istnieją stany, które można osiągnąć bez wywołania metody. Zwykle jednak tester nie może w pełni polegać na tych teoriach. Jeśli klasa ma zmienne publiczne, bezpośredni dostęp do tych zmiennych musi być zawarty w procedurze testowej tak, jakby była to osobna metoda klasy. Jeśli model stanów opisuje jedynie fragmenty teoretycznej przestrzeni stanów, które są istotne dla praktycznego wykorzystania klasy, to nie musi istnieć bezpośrednie połączenie pomiędzy wywołaniami metod klasy a przejściami pomiędzy jej stanami. W takich przypadkach strategia testowania opisana powyżej nie obejmuje wszystkich przejść.

Pokrycie ścieżek

Trzecim kryterium pokrycia związanym z testowaniem opartym na stanach jest pokrycie ścieżek analizujące, które z możliwych ścieżek w modelu stanów są objęte istniejącymi przypadkami testowymi. Jak w naszym przykładzie sterownika eHome, jeśli przejścia pomiędzy stanami (aplikacji) są związane z określonymi metodami klasy, odpowiednie sekwencje metod będą tworzyć odpowiadające im ścieżki przez model aplikacji. Ponieważ modele stanów mogą wykazywać zachowania cykliczne, ścieżki mogą być zmiennej długości i występować w nieskończonej liczbie, choć istnieje skończona liczba warunków brzegowych. Jeśli chcemy osiągnąć założony stopień pokrycia ścieżek, musimy zdefiniować maksymalną możliwą długość ścieżki.

11 Pokrycie przejść obejmuje pokrycie stanów, jeśli diagram stanów jest ciągły (tzn. każdy stan można osiągnąć przez co najmniej jedno przejście). Tester musi założyć, że implementacja produktu pomija pewne przypadki przejść i powinien zawsze porównywać uzyskane pokrycie z (funkcjonalnym) modelem stanów.

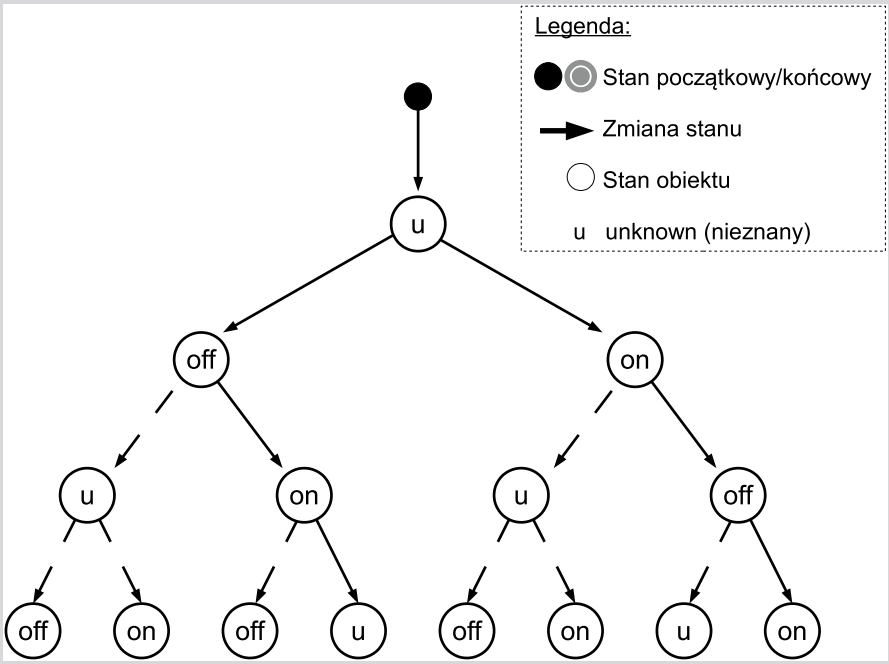
Studium przypadku 4-3c

Studium przypadku 4-3c dla sterownika eHome: Pokrycie ścieżek dla klasy Device

W naszym przykładzie sterownika eHome możemy osiągnąć 25% pokrycie ścieżek korzystając z dwóch ścieżek o długości czterech węzłów (tzn., dwóch z ośmiu możliwych ścieżek czterowęzłowych):

```
__construct() → set_status('off') →
   set_status('on') → set_status('unknown');
__construct() → set_status('on') →
   set_status('off') → set_status('on');
```

Wynikowe sekwencje testowe będą zawierały 100% pokrycie stanów i przypadków brzegowych. Poniższa ilustracja przedstawia diagram drzewa dla urządzeń eHome:



Jeśli wszystkie testy negatywne (tzn. te, które obejmują niedopuszczalne wywołania metod) będą wywoływane z góry lub kolejno dla wszystkich możliwych stanów, to wystarczy budować sekwencje testowe tylko z testów pozytywnych. Innymi słowy samo pokrycie stanów wystarcza za testy negatywne.

Diagramy drzew

Najbardziej skutecznym sposobem systematycznego identyfikowania wszystkich możliwych ścieżek przez model stanów jest narysowanie diagramu drzewa dla modelu podobnego do pokazanego wcześniej. Odpowiednie procedury są wyczerpująco opisane w [Spillner/Linz 14], [Vigenschow 10] i innych. Diagram drzewa może być następnie wykorzystywany do śledzenia wszystkich możliwych ścieżek aż do określonej wstępnie długości maksymalnej.

Wszelkie dodatkowe ograniczanie liczby ścieżek testowych powinno być oparte na szacowaniu ryzyka i zorientowane wokół potencjalnych przypadków użycia. W wielu przypadkach ścieżki odzwierciedlają faktyczne scenariusze użycia klasy i mogą być skutecznie sortowane według ich znaczenia i częstości użycia.

4.1.5 Testowanie permutacji metod

Poprzedni podrozdział opisał, jak wywieść przypadki testowe z modelu stanu, który oferuje abstrakcję kodu programu opartą na technicznej rzeczywistości. Dla naszego przykładu sterownika eHome modelowaliśmy tylko część dostępnej przestrzeni stanów. Zmienna klasy `$name` została uznana za technicznie nieistotną i nie była uwzględniana. Testy oparte na stanach, które opisaliśmy, są więc w istocie oparte na specyfikacji.

Co więc możemy zrobić, jeśli nie mamy modelu stanów określonego w postaci specyfikacji albo jeśli klasa może być testowana jedynie na poziomie kodu? [Bashir/Goel 99] opisuje „testowanie interakcji między metodami", które dobrze się sprawdza w takich przypadkach, a przynajmniej daje użyteczne dane heurystyczne dotyczące liczby i typów testów, które należy wybrać.

Punkt wyjścia jest taki sam, jak opisany w podrozdziale 4.1.3, czyli klasa, którą chcemy przetestować, ma kilka metod publicznych i zmiennych prywatnych. Obiekty tej klasy mają więc określony stan i muszą być testowane przy użyciu technik opartych na stanach. Ponieważ wszystkie zmienne są prywatne, stan obiektu może być zmieniany tylko przy użyciu sekwencji wywołań metod publicznych.

Na przykład klasa z 10 metodami publicznymi daje sumę (S) równą 10! (= 3 628 800) sekwencji wywołań (zobacz [Bashir/Goel 99, podrozdział 6.2]), przy czym każda metoda jest wywoływana tylko raz, a jej parametry nie mają być zmieniane.

Dzielenie klasy

Potrzebne nakłady zwiększają się gwałtownie wraz z liczbą metod, ale [Bashir/Goel 99] opisuje sposób na znaczące ograniczenie liczby łączonych ze sobą metod. Ta technika wiąże się z dzieleniem klasy. Każda część składa się z pojedynczej zmiennej klasy i wszystkich metod, które dokonują na niej odczytu lub zapisu. Metoda może pojawiać się w wielu częściach, natomiast zmienna może się pojawić tylko w jednej, a zamiast ciągu wszystkich metod łączone są tylko metody z pojedynczej części.

Studium przypadku 4-3d

Studium przypadku 4-3d dla sterownika eHome: Dzielenie klasy Device

Wersja 2 klasy Device oferuje w sumie sześć sekwencji wywołań (jeśli zignorujemy wywołanie konstruktora):

```
S = 3! = 3*2*1 = 6;
```

Klasę `Device` można podzielić na dwie podklasy: `Sname={get_name()};` `Sstatus={set_status(),get_status()}`.

Liczba wymaganych przypadków testowych wynosi wtedy:

```
S = 1! + 2! = 3;
```

[Bashir/Goel 99] zaleca następującą strategię testowania[12] dla podzielonej klasy:

1. Testowanie konstruktorów i metod get przy użyciu sekwencji w postaci: konstruktor → reporter[13]
   Reporter odczytuje wartość odpowiedniej zmiennej i przekazuje ją do klasy testowej. Klasa testowa porównuje faktyczną wartość ze wstępnie zdefiniowanymi wartościami docelowymi, aby sprawdzić, że konstruktor prawidłowo zainicjował obiekt.
2. Testowanie metod get przy użyciu losowo generowanych sekwencji w postaci: konstruktor → transformer, …, transformer → reporter
   Transformery manipulują zmiennymi obiektu, a reporter musi być w stanie dostarczyć wynik na końcu każdej sekwencji. Jest on następnie porównywany z zawartością prywatnych zmiennych klasy. Klasa testowa musi być zadeklarowana jako zaprzyjaźniona, aby uniknąć konieczności ręcznego wstawiania wartości docelowych dla setek lub tysięcy istniejących sekwencji.
3. Testowanie metod set przy użyciu losowo generowanych sekwencji w postaci: konstruktor → transformer, …, transformer
   Transformery manipulują zmiennymi obiektu i tym samym również stanem obiektu. Po każdej zmianie stanu następuje test sprawdzający, czy nastąpiła zmiana do prawidłowego stanu następnego. Tester musi być w stanie wykonywać lokalne sprawdzenia, żeby uniknąć konieczności ręcznego wstawiania wartości docelowych dla setek lub tysięcy istniejących sekwencji.

Jeśli sekwencja się nie powiedzie, nie jest jasne, która metoda spowodowała błąd. Aby zlokalizować błąd, sekwencja jest skracana krok po kroku, aż test się powiedzie.

12 Uprościliśmy ten opis przedstawiając go tutaj. Szczegóły można sprawdzić w [Bashir/Goel 99].

13 [Bashir/Goel 99] nazywa metody get reporterami, a metody set transformerami.

Ta strategia ma też swoje wady, do których należy konieczność deklarowania klasy testowej jako zaprzyjaźnionej. Jednakże największym jej ograniczeniem jest to, że proces nie definiuje, które parametry są wykorzystywane do wywoływania transformerów. Aby zapewnić, że test będzie mimo to działał przy pełnej automatyzacji, musimy wykorzystać permutacje, aby wygenerować dużą liczbę sekwencji wywołań. Problemem tutaj jest to, że im więcej sekwencji wygenerujemy, tym trudniejsze stanie się ręczne osadzanie parametrów wywołań w kodzie testowym. Co więcej ręcznie wprowadzane parametry nie mogą być zróżnicowane.

To co sprawia, że to podejście jest interesujące (i o nim tutaj wspominam), to fakt, że demonstruje, jak można korzystać z heurystyki do generowania sekwencji wywołań bez użycia funkcjonalnego modelu stanów. W przypadkach, w których metody wymagają prostych parametrów wywołań (lub nie wymagają ich w ogóle), jest to elegancki sposób generowania w pełni zautomatyzowanych testów.

4.2 Programowanie sterowane testami

Podrozdział 4.1 przedstawił ważne podstawy dotyczące testowania jednostkowego. Kod programu (w naszym przypadku klasa `Device`) był napisany i poddany testom zaprojektowanym przy wykorzystaniu informacji pochodzących ze specyfikacji systemu a także wiedzy o strukturze kodu. Testy zostały następnie zakodowane i mogły być uruchamiane automatycznie. Na koniec programista musiał poprawić wszystkie usterki w kodzie, które zostały ujawnione przez testy. Po tym wszystkim procedura testowa zaczęła się od nowa przy wykorzystaniu poprawionego kodu. Jeśli to podejście do testowania jednostkowego jest używane konsekwentnie wewnątrz projektu (tzn. jest stosowane wobec każdej jednostki kodu i po każdej zmianie), jest to skuteczny sposób ujawniania i radzenia sobie z błędami i problemami w specyfikacjach oraz usterkami implementacyjnymi na wczesnym etapie procesu produkcyjnego.

Zanim zmienisz swój kod, napisz zautomatyzowany test, który zwraca błąd.

Programowanie sterowane testami jest jeszcze bardziej skutecznym podejściem do testowania. Podobnie do wielu praktyk zwinnych, sterowanie testami wywodzi się ze świata XP. Jest to jedna z najbardziej fundamentalnych praktyk zwinnych i odwraca tradycyjne podejście „programuj najpierw, testuj potem" do góry nogami. Sterowanie testami oznacza wzięcie pod uwagę, które testy będą potrzebne do pokazania, że oprogramowanie faktycznie spełnia nowe specyfikacje, zanim przeprowadzone zostaną jakiekolwiek zmiany w samym kodzie. Takie testy są projektowane i automatyzowane przed ich uruchomieniem. Ponieważ nowy (lub zmieniony) kod produkcyjny jeszcze nie istnieje,

testy oczywiście dadzą wynik negatywny. Zwięźle ujmując, programowanie sterowane testami oznacza: „przed zmianą jakiegokolwiek kodu należy napisać zautomatyzowany test dający wynik negatywny [Beck/Andres 04, str. 50].

Programista zaczyna pisanie kodu produktu dopiero, gdy istnieją zautomatyzowane testy. Jeśli wszystkie testy dadzą wynik pozytywny, zadanie kodowania jest ukończone. Jeśli którykolwiek z testów daje wynik negatywny, programista musi pracować nad kodem, aż wszystkie testy będą poprawne.

Programowanie sterowane testami

Ponieważ to testy ukierunkowują działania programisty, mówimy o programowaniu sterowanym testami albo o wczesnym testowaniu (zobacz też [Link 03]). Częstotliwość, z jaką programiści stosują cykl „napisz test → uruchom test → zmień kod", różni się dla różnych zespołów i jest to też kwestia osobistego stylu programowania. Jeśli kod programisty jest używany jako część ogólnego zarządzania konfiguracją przez zespół, może to się odbywać od kilku razy dziennie (za każdym razem, gdy programista przesyła swój kod do systemu zarządzania konfiguracją zespołu) aż po ciągłe testowanie jednostkowe, które uruchamia testy dla każdego pisanego wiersza kodu produkcyjnego.

Zespół, który ściśle stosuje metodykę programowania sterowanego testami, zwiększa skuteczność swojego testowania jednostkowego jako narzędzia zarządzania jakością. Dzieje się tak dlatego, ponieważ:

- **Testowanie zastępuje próby:** Zwykle programiści regularnie testują pisany kod, aby sprawdzać, czy wykonuje to, co miał. Jeśli kod wymaga danych wejściowych, programista zwykle wymyśla na miejscu odpowiednie wartości. Odpowiednio sparametryzowane wywołanie metody jest wstawiane we właściwym miejscu kodu, a wartości wynikowe są przeglądane pobieżnie (jeśli w ogóle). Zamiast korzystać z wbudowanych sprawdzeń weryfikujących, czy zmienna ma oczekiwaną wartość, to wartość danej zmiennej jest zwykle wypisywana w debugerze lub przy użyciu polecenia wypisującego ją na ekranie. Jeśli program nie zachowuje się zgodnie z oczekiwaniami, programista próbuje zastosować inne dane wejściowe, a dokonywanie poprawek na podstawie takiej metody prób i błędów staje się nieodłączną częścią procedury debugowania. Takie podejście jest niezwykle mało wydajne. Dodawanie i dopasowywanie nowych wywołań testowych i procedur wypisujących wyniki zabiera czas i ujawnia tylko najbardziej oczywiste usterki. To podejście zwykle nie uwzględnia przypadków specjalnych, wariacji lub kombinacji wartości wejściowych, ani też nieprawidłowych danych wejściowych. Jeśli natomiast programista poświęci czas na napisanie listy istotnych przypadków testowych (wraz z danymi wejściowymi i oczekiwanymi danymi

wyjściowymi), skuteczne testowanie wkrótce zastąpi nieefektywną, bezcelową metodę prób i błędów. Prosta czynność wypisania przypadków testowych pomoże programiście przemyśleć przypadki specjalne i ciekawe kombinacje danych wejściowych. Wypisanie ich często pomoże programiście w odkryciu, czego brakuje w kodzie produkcyjnym nawet bez samego uruchamiania testów. Aby skutecznie stosować podejście sterowane testami, trzeba przy pisaniu swoich przypadków testowych przejść od podejścia „kiedyś" poprzez „podczas programowania" do „przed napisaniem pierwszego wiersza kodu".

- **Przypadki testowe zapewniają obiektywną informację zwrotną na temat postępów:** Jeśli działające testy sprawdzające skuteczność kodu będą dostępne od początku, każdy test zakończony powodzeniem da programiście obiektywny, jednoznaczny dowód, że projekt postępuje naprzód. Od pierwszego wiersza kodu programista może stosować testy jako obiektywne kryteria gotowości kodu. To podejście ogranicza ryzyko poświęcania wielu godzin na programowaniu w złym kierunku i pomaga w pisaniu nowych, bardziej istotnych testów[14].
- **Testy zastępują pisemne specyfikacje:** Test sprawdza, czy obiekt, któremu dostarczymy określone dane wejściowe, reaguje poprawnie (albo zwraca poprawne dane wyjściowe). Dzięki temu przypadek testowy działa nie tylko jako zbiór instrukcji testowych, ale też jako definicja oczekiwanego zachowania obiektu. Każdy test określa wynik, który powinniśmy uzyskać poprzez użycie określonych danych wejściowych. Sterowanie testami oznacza definiowanie zachowania programu w formie testów, zanim program zostanie napisany. Automatyzowanie tych testów tworzy też dokładną, czytelną dla komputerów specyfikację funkcjonalną kodu, który trzeba napisać. Innymi słowy specyfikacje i instrukcje ich testowania są jednym i tym samym, a oddzielne kroki związane z ich pisaniem zlewają się w jedno. Wadą tego jest to, że mniej zaawansowani programiści mają mniejszą szansę na właściwe zrozumienie tworzonych w ten sposób specyfikacji. Można złagodzić tę sytuację, dołączając uwagi i komentarze w kodzie testowym wyjaśniające powód stworzenia danego testu i użytych wartości wejściowych. Innym sposobem na tworzenie testów bardziej czytelnych dla ludzi jest pisanie ich w odpowiednim języku poleceń technicznych, gdzie każde polecenie reprezentuje określoną funkcję. Ten typ notacji jest szczególnie skuteczny podczas testowania systemowego i jest wyjaśniony dokładniej w podrozdziale 6.4.

14 „To jasne, co należy zrobić dalej: albo napisać kolejny test, albo naprawić zły test" [Beck/Andres 04, str. 51].

- **Programowanie sterowane testami zwiększa jakość interfejsów publicznych (API):** Jak wyjaśniono w podrozdziale 4.1.2, testy jednostkowe korzystają z wywoływania publicznych metod klasy. Jeśli testy jednostkowe są pisane, zanim klasa w ogóle istnieje, to pisanie ich definiuje nazwy metod publicznych, wykorzystywanych przez nie parametrów i sposobów użycia tych metod. Projektowanie testów i interfejsów API łączy się w jedną całość. Autor testów patrzy na obiekt testowy z punktu widzenia przyszłego użytkownika – czyli z zewnątrz. Ponieważ elementy niewymagane w danym teście nie są w nim sprawdzane, to pisane testy i sekwencje testów będą najprawdopodobniej zawierać tylko te wywołania metod, które będą potrzebne przyszłym użytkownikom. Wynikiem jest jasny, lekki interfejs API zorientowany na aplikację.
- **Programowanie sterowane testami poprawia podatność kodu na testowanie:** Po napisaniu testów programista musi napisać kod i tam, gdzie to konieczne, dostroić go, aż wszystkie testy będą działać bezbłędnie. Oznacza to, że kod musi zawierać interfejsy wymagane przez testy. Zamiast pisać test pracujący na istniejącym kodzie, musimy teraz pisać kod, który da się przetestować przy użyciu istniejących testów. Jeśli kodu nie da się całkiem przetestować, to nie jest gotowy. Przy ścisłym stosowaniu metod programowania sterowanego testami (tzn. przez każdego programistę dla każdej zmiany w każdej jednostce kodu) testy będą docierać do najdrobniejszego poziomu kodu. Kod wynikowy będzie miał odpowiednie interfejsy testowe na każdym poziomie swojej architektury, a całość będzie można testować automatycznie. Ułatwia to testowanie i daje w efekcie projekt o jasnej strukturze, który jest łatwiejszy w utrzymaniu i rozbudowywaniu.

4.2.1 Programowanie sterowane testami a Scrum

Sterowanie testami jest jedną z 13 podstawowych praktyk XP[15]. Sam proces Scrum nie wymaga użycia programowania sterowanego testami i działa równie dobrze z tradycyjnymi testami jednostkowymi. Jednakże zespół Scrum, który wykorzystuje techniki programowania sterowanego testami, może odnieść znaczne korzyści na poziomie technicznym (zobacz poprzedni podrozdział) i dzięki przyspieszeniu pętli zwrotnej.

15 [Beck/Andres 04] wymienia następujących 13 głównych praktyk: wspólna praca, cały zespół, informacyjna przestrzeń robocza, energiczna praca, programowanie w parach, scenariusze, cykl tygodniowy, cykl kwartalny, luz, dziesięciominutowe budowanie, ciągła integracja, programowanie sterowane testami, projektowanie przyrostowe.

Mikro pętla zwrotna

Programowanie sterowane testami zapewnia każdemu programiście dodatkową mikro pętlę zwrotną dla każdego zadania programistycznego. Dla każdej zmiany w kodzie programista otrzymuje natychmiastową informację zwrotną o jej sukcesie lub porażce. Przy założeniu, że zautomatyzowane testy jednostkowe są skutecznie wbudowane w proces ciągłej integracji (zobacz rozdział 5) stosowany przez zespół, pętla zwrotna może działać w obrębie sekund lub minut.

Normalne testy jednostkowe, które są pisane po napisaniu kodu, mogą tylko sprawdzać, czy zmiany w kodzie nie uszkodziły istniejącej funkcjonalności. Nie mogą zapewniać informacji zwrotnych dotyczących nowego kodu, który jest właśnie pisany. Konieczne testy jeszcze nie istnieją, a w przypadku normalnych testów zostaną dopiero napisane po dostarczeniu przez programistę gotowego kodu.

Dla odróżnienia programowanie sterowane testami wyposaża programistę we wszystkie testy potrzebne do sprawdzania nowej funkcjonalności, a testy jednostkowe zespołu automatycznie obejmują funkcjonalność właśnie rozwijanego kodu.

4.2.2 Implementowanie sterowania testami

Pojęcie sterowania testami jest proste, ale oferuje niezmierzone korzyści. Jednak korzystanie z niego wymaga utrzymywania dyscypliny i ciężko wprowadzić je do zespołu Scrum bez pomocy.

Zmiana sposobu myślenia

Nie jest wcale łatwo projektować i pisać przypadki testowe dla obiektów, które nie istnieją. Aby skutecznie korzystać z tej metody, programiści i testerzy muszą zmienić sposób myślenia i działania. Sterowanie testami wymusza na nas myślenie w bardziej abstrakcyjny sposób niż przy konwencjonalnych testach jednostkowych, a wszyscy programiści w zespole muszą nauczyć się myśleć w kategoriach interfejsów API. Pisanie testów z góry może być łatwiejsze dla projektantów oprogramowania, którzy są przyzwyczajeni do projektowania interfejsów API, ale będzie to trudniejsze dla tych, którzy są przyzwyczajeni do implementowania gotowych interfejsów API. Niezależnie od posiadanego poziomu umiejętności, zawsze pojawią się przypadki, w których pisanie testów najpierw okaże się trudniejsze, niż można oczekiwać, a pokusa rozpoczęcia kodowania będzie trudna do powstrzymania. Zespół musi nauczyć się ściśle trzymać pojęcia sterowania testami.

Branie pod uwagę sterowania testami podczas planowania zadań

To oznacza, że sterowanie testami musi być aktywnie wspierane przez planowanie osobnych zadań projektowania i automatyzowania testów przed planowaniem zadań implementowania kodu. Jest to najlepsze podejście zwłaszcza w zespołach, które mają niewielkie doświadczenie w podejściu sterowanym testami. Gdy zespół zdobędzie nieco doświadczenia, może zacząć wykorzystywać testy jako kryteria

gotowości dla przyszłych zadań. Pomocne jest też przeprojektowanie tablicy z zadaniami zgodnie z sekwencją „napisz test → uruchom test → zmień kod". W ten sposób sterowanie testami stanie się widoczne dla całego zespołu i zintegruje się automatycznie z codziennym spotkaniem Scrum.

Szkolenie

Mistrz Scrum będzie też musiał wesprzeć przejście na sterowanie testami przy użyciu dodatkowych środków, takich jak szkolenia dotyczące technik testów jednostkowych omówionych w podrozdziale 4.1. Programowanie sterowane testami nie tylko oznacza zmianę sekwencji kodowania/testowania, ale również przerzuca na programistów nowe zadanie pisania testów dla nowej funkcjonalności. Umiejętności wymagane do skutecznego przeprowadzania odpowiednich testów stają się przez to bardziej cenione. Wcześniej trudniejsze testy można było odłożyć na później, natomiast sterowanie testami wymusza na każdym kroku pisanie testów najpierw. Warto jednak pamiętać, że bezużyteczne dla zespołu będzie, jeśli jakość i funkcjonalność tych testów będzie poniżej standardów. Źle napisane testy mogą nawet dawać zespołowi złudne poczucie bezpieczeństwa. Przykłady wymienione w podrozdziale 4.1 ilustrują, jak trudne może być pisanie skutecznych testów nawet w najprostszych przypadkach. Siła zautomatyzowanych testów jednostkowych nie leży w liczbie testów, ale raczej w elegancji i dopasowaniu odpowiednich przypadków testowych, a pisanie dobrych testów wymaga porządnej wiedzy na temat związanych z tym technik.

Programowanie w parach

Przejście na programowanie sterowane testami jest łatwiejsze, jeśli zespół przejdzie na podejście programowania w parach. Wymaga to od zespołu pracy w parach tester/programista. Nie oznacza to, że poszczególne role są zawsze spełniane przez tę samą osobą, a zwykle wymieniają się one rolami przy kolejnych zadaniach lub po określonych okresach czasu (na przykład co dwie godziny). Osoby przypisane do poszczególnych par też mogą się zmieniać, a zespół odpowiada za ustalenie stosowanych reguł. Mistrz Scrum jest odpowiedzialny za zainicjowanie procesu podziału na pary i musi interweniować w razie problemów. Przeglądy kodu testowego są również ważnym narzędziem i powinny być przeprowadzane na poziomie zespołu, a nie tylko w poszczególnych parach zwłaszcza, jeśli zespół dopiero przyzwyczaja się do nowego sposobu pracy. Pomoże to w stymulowaniu przekazywania wiedzy na temat projektowania i automatyzowania testów jednostkowych wewnątrz zespołu.

Sterowanie testami wprowadza dużą zmianę do procesu tworzenia oprogramowania i sposobu pracy zespołu. Projektanci oprogramowania, którzy są przyzwyczajeni do dostarczania własnych rozwiązań, będą początkowo mieli problem z zaakceptowaniem tego, że inni członkowie zespołu mają wpływ na całościowy projekt programu.

Korzystanie ze sterowania testami oznacza, że podstawowe decyzje projektowe są dokonywane w oparciu o testy. Nawet najbardziej doświadczony architekt oprogramowania będzie musiał uznać, że inni członkowie zespołu mają swoje własne pomysły związane z tym, jak tworzyć projekt przyjazny dla testów, co może przyczynić się do zmian testowanego kodu. Kwestie architektoniczne – na przykład projekt interfejsu API lub klasy albo sposobu przekazywania zewnętrznych obiektów do klasy – są teraz determinowane przez testy. Na decyzje i pomysły alternatywne patrzy się teraz krytyczniej niż poprzednio, a projekt oprogramowania staje się wspólnym wysiłkiem zespołu, a nie wynikiem pracy pojedynczego projektanta lub decyzji podejmowanych przez poszczególnych programistów. Sterowanie testami wymaga przekształcenia starego zespołu w zespół wielofunkcyjny i przyspiesza ten proces. Nie każdy członek zespołu będzie zadowolony z tej zmiany, a tych, którzy nie będą w stanie się przystosować na dłuższą metę, trzeba będzie wymienić.

Sterowanie testami jako kluczowa kompetencja

Sterowanie testami jest kluczową kompetencją w zespole Scrum i musi być ściśle przestrzegane. Mistrz Scrum jest odpowiedzialny za zapewnienie, że zostanie wprowadzone w odpowiedni sposób.

4.2.3 Korzystanie z programowania sterowanego testami

Zespół eHome postanawia skorzystać z zasad programowania sterowanego testami i wszystkie zadania implementacyjne otrzymują teraz odniesienie do zadania specyfikacji testowej i automatyzacji:

Studium przypadku 4-4a

Studium przypadku 4-4a dla sterownika eHome: Programowanie klasy Dimmer sterowane testami

Po codziennym spotkaniu Scrum jeden z programistów pobiera nowe zadanie implementacyjne z tablicy. Zespół postanowił pracować zgodnie z zasadami programowania sterowanego testami i programowania w parach, więc programista wybiera teraz testera, z którym będzie pracował przez cały dzień. Wybrali następujące zadanie:

Należy opracować nową klasę `Dimmer` (ściemniacz) wywodzącą się z klasy Device. Ściemniacz jest elektronicznym urządzeniem, które służy do zmieniania jasności lamp w zakresie od 0 do 100%. Obiekt `Dimmer` reprezentuje takie urządzenie w systemie sterownika eHome.

Kryteria gotowości są zdefiniowane następująco:

- Testy jednostkowe zautomatyzowane w klasie `DimmerTest`
- PHPUnit jako platforma testowa
- Zakodowana klasa `Dimmer`
- Zaliczone w 100% testy `DimmerTest` przy 100% pokryciu wierszy kodu klasy `Dimmer`

Z przyzwyczajenia programista zaczyna wpisywać kilka pierwszych wierszy nowej klasy `Dimmer`, ale jego partner powstrzymuje go i przypomina, że najpierw trzeba utworzyć klasę `DimmerTest`. Następnie piszą wspólnie następujący kod testu jednostkowego:

```
include 'Dimmer.php';      // klasa do przetestowania
include 'TestFrame.php';   // nasza prosta platforma
                           // do testów jednostkowych
   class DimmerTest extends TestFrame {  // wersja 1
   public function test_createDimmer() {
                           // sprawdzenie, czy
                           // początkowy poziom
                           // wynosi 0%
   $dimmer = new Dimmer('lounge chair light');
                           // przygotowanie
   $dimLevel = $dimmer->get_dimLevel();
                           // procedura testowa
   $this->assertEquals($dimLevel,0,'createDimmer');
                           // sprawdzenie
   unset($dimmer);         // zakończenie
   }
}

$myTestSuite = new DimmerTest();
$myTestSuite->test_createDimmer();           //
wykonaj przypadek testowy 1
$myTestSuite->printResult();
```

Oczywiście `test_createDimmer()` nie działa teraz poprawnie, ponieważ kod testowy nie może znaleźć klasy `Dimmer`[16], którą dopiero trzeba oprogramować. Zespół ściśle zastosował się do zasad metodyki programowania sterowanego testami i utworzył test, który na początku kończy się niepowodzeniem. Kod testowy jest prosty, ale mimo to dokładnie określa, jak powinna zachowywać się nowa klasa `Dimmer`:

- Tak jak w przypadku klasy `Device`, konstruktor klasy jest wykorzystywany do nadania nazwy obiektowi `Dimmer`.
- Metoda interfejsu API `get_dimLevel()` służy do pobierania bieżącego ustawienia jasności.
- Nowo utworzony obiekt `Dimmer` ma jasność ustawioną na zero[17].

Odpowiadająca karta zadania definiuje planowaną funkcjonalność na dużo prostszym poziomie. Programista pracujący bez użycia sterowania testami musiałby zdecydować podczas kodowania, które metody API są wymagane przez klasę i jak powinny się zachowywać.

16 Interpreter PHP zgłasza komunikat „Krytyczny błąd PHP: nie znaleziono klasy `'Dimmer'`".

17 Kod testowy wykorzystuje metodę platformy testowej `assertEquals()`, aby sprawdzać ten atrybut. Więcej szczegółów można znaleźć w następnym rozdziale.

W projekcie Scrum prowadzonym bez sterowania testami ten rodzaj decyzji opartych na projekcie nie byłby wprost udokumentowany, ale istniałby pośrednio jako część kodu źródłowego. W tradycyjnym środowisku projektowym (np. modelu V) decyzje te byłyby podejmowane na etapie specyfikacji, a jeśli projekt jest dobrze zarządzany, byłyby udokumentowane w szczegółowej specyfikacji.

Kod testowy jest specyfikacją.

W środowisku sterowanym testami sam kod testowy jest specyfikacją. Sterowanie testami nie tylko zapewnia, że zespół zaprojektuje i zautomatyzuje odpowiednie testy, ale też zasypuje lukę abstrakcyjną, która w przeciwnym razie istniałaby pomiędzy ogólnie zdefiniowanym zadaniem (należącym koncepcyjnie do poziomu architektury systemu) a kodem źródłowym.

Bez sterowania testami zespół musiałby zdefiniować osobne zadanie „tworzenie dokładnej specyfikacji" jako warunek realizacji danego zadania funkcyjnego. Sterowanie testami automatycznie zapewnia dokładną specyfikację w formie zautomatyzowanych testów.

Studium przypadku 4-4b

Studium przypadku 4-4b dla sterownika eHome: Programowanie klasy Dimmer sterowane testami

Partnerzy omawiają kod testu jednostkowego, który właśnie napisali. Tester proponuje napisanie dodatkowych przypadków testowych, natomiast programista wolałby już implementować klasę `Dimmer`, żeby test, który już napisali, był poprawny. Nie ma powodu, żeby nie skorzystać z tego podejścia, a wynikiem jest następujący kod:

```
include 'Device.php';
class Dimmer extends Device {
// wersja 1
private $dimLevel=0;
   public function get_dimLevel() {
   return $this->dimLevel;
   }
}
```

Uruchomienie testu daje następujące wyniki:

```
Assertions: 1 / passed: 1 / failed: 0
```

Napędzony tym sukcesem programista chciałby rozszerzyć klasę `Dimmer`, ale tester powstrzymuje go i przypomina o uzgodnionej zasadzie[a].

Zgodnie z regułami rozszerzają kod testu jednostkowego, zamiast pracować nad kodem programu. Zamiast swojej własnej platformy TestFrame zaczynają też korzystać z platformy testowania jednostkowego PHPUnit i odpowiednio przepisują swoje testy:

```
include 'Dimmer.php';                // testowana klasa
class DimmerTest extends PHPUnit_Framework_TestCase
```

a „Napisz zautomatyzowany test zwracający błąd, zanim zmienisz jakikolwiek kod" [Beck/Andres 04, str. 50].

```
   {                                  // wersja 2
   private $myDimmer;   public function setUp() {
// kroki
// przygotowawcze dla wszystkich przypadków testowych
   $this->myDimmer = new Dimmer('lounge chair
light');
   }

   public function tearDown() {
      // kroki końcowe dla wszystkich przypadków
      // testowych
    unset($this->myDimmer);
   }

   public function test_createDimmer() {
      // sprawdzenie, czy początkowy poziom wynosi 0%
   $dimLevel = $this->myDimmer->get_dimLevel();
   $this->assertEquals(0, $dimLevel);
   }

   public function test_set_dimLevel_to_default() {
   $this->myDimmer->set_dimLevel();
   $dimLevel = $this->myDimmer->get_dimLevel();
   $this->assertEquals(50, $dimLevel);  // domyślnie
                                  // powinno być 50%
   }

   public function test_set_dimLevel_to_min() {
   $this->myDimmer->set_dimLevel(0);
   $dimLevel = $this->myDimmer->get_dimLevel();
   $this->assertEquals(0, $dimLevel);
                           // minimum powinno być 0%
   }

   public function test_set_dimLevel_to_max() {
   $this->myDimmer->set_dimLevel(100);
   $dimLevel = $this->myDimmer->get_dimLevel();
   $this->assertEquals(100, $dimLevel);
                        // maksimum powinno być 100%
   }

   public function test_set_dimLevel_below_min() {
   $this->myDimmer->set_dimLevel(-1);
   $dimLevel = $this->myDimmer->get_dimLevel();
   $this->assertEquals(0, $dimLevel);
                           // minimum powinno być 0%
   }

   public function test_set_dimLevel_above_max() {
   $this->myDimmer->set_dimLevel(101);
   $dimLevel = $this->myDimmer->get_dimLevel();
   $this->assertEquals(100, $dimLevel);
                        // maksimum powinno być 100%
   }
}
```

ciąg dalszy na następnej stronie

Uruchamiają teraz testy jednostkowe, ponownie tym razem wywoływane przez PHPUnit. Jednakże kod testowy nie może znaleźć metody API `set_dimLevel()`[b] i zatrzymuje test. Brakującą metodę można następnie łatwo dodać do kodu `Dimmer`:

```
public function set_dimLevel($dimTo=0) {
if (($dimTo >0) AND ($dimTo <100))
   $this->dimLevel = $dimTo;
}
```

Testy jednostkowe są uruchamiane ponownie:

```
> phpunit DimmerTest.v2.php
PHPUnit 3.7.1 by Sebastian Bergmann.
Time: 0 seconds, Memory: 1.50Mb
There were 3 failures:
1) DimmerTest::test_set_dimLevel_to_default
Failed asserting that 0 matches expected 50.
...
2) DimmerTest::test_set_dimLevel_to_max
Failed asserting that 0 matches expected 100.
...
3) DimmerTest::test_set_dimLevel_above_max
Failed asserting that 0 matches expected 100.
...
Tests: 6, Assertions: 6, Failures: 3.
```

Programista był przekonany, że napisał tę trywialną metodę poprawnie, ale szybko poprawia kod w duchu przyznając, że sterowanie testami nie jest wcale takim złym pomysłem.

b Krytyczny błąd PHP: wywołanie niezdefiniowanej metody `Dimmer::set_dimLevel()`

Można samodzielnie dokonać tej poprawki w ćwiczeniu 4.6.3-1 albo sprawdzić poprawioną wersję na naszej stronie WWW [URL: SWT-knowledge].

4.3 Platformy testowania jednostkowego

Zautomatyzowanie testu jednostkowego jest właściwie bardzo proste. Przykłady, z których dotąd korzystaliśmy, przebiegają według następującego wzorca:

- Dla klasy `'xyz'`, którą należy przetestować (obiektu testowanego), należy utworzyć klasę testową `'xyzTest'`.
- Przypadki testowe są automatyzowane jako metody klasy testowej `'xyzTest'`. Każdy przypadek testowy jest implementowany przez jedną, odrębną metodę.

- Nazwy metod testowych są wybierane tak, aby odzwierciedlały zawartość danego przypadku testowego. Pomaga to rozpoznawać i utrzymywać testy jednostkowe.
- Każdy przypadek testowy (albo metoda przypadku testowego) jest dzielony na cztery części zwane `'przygotowanie'`, `'procedura testowa'`, `'sprawdzenie'` i `'zakończenie'`. Każdy test jest zaprojektowany tak, aby sprawdzał pojedynczy aspekt testowanego obiektu, co z kolei oznacza, że metoda testowa powinna zawsze zawierać tylko jedną sekcję `'sprawdzenie'`.
- Sekcja `'sprawdzenie'` porównuje faktyczny wynik z oczekiwanym. Innymi słowy sprawdza, czy stan określonego atrybutu obiektu (zdefiniowany przez jedną lub kilka zmiennych klasy) zgadza się z żądaną wartością.
- Jeśli oczekiwana i faktyczna wartość się zgadzają, test jest `'zdany'` (passed). Jeśli się różnią, test jest `'oblany'` (failed).
- Przypadki testowe mogą być rozpoczynane albo pojedynczo przy użyciu odpowiednich metod, albo zbiorowo przy wykorzystaniu metody `'run'` zapewnianej przez platformę testową. Zwykle odbywa się to w kolejności, w jakiej pojawiają się w klasie.

Wszystkie przypadki testowe i klasy testowe są programowane w ten sam sposób; dlatego wynikiem jest duża liczba podobnych wierszy kodu. Powtarzające się kroki są kodowane w osobnej klasie platformy testowej. Przykłady do 4-4a wykorzystują prostą klasę platformy testowej `TestFrame` zapewniającą centralną funkcję sprawdzającą `assertEquals`, która zbiera poszczególne wyniki testów. Zapewnia ona też funkcję `printResult` do wypisywania wyników. Wszystkie klasy testowe dziedziczą te funkcje (zobacz przykład 4-4a: `DeviceTest extends TestFrame`).

Klasę `TestFrame` można dalej rozbudowywać o funkcje, które rejestrują wyniki testów albo importują dane testowe z tabeli. Platforma powinna być też w stanie rozpoznawać, które przypadki testowe zawarte są w klasie testowej i powinna być w stanie wykonywać je zbiorowo bez konieczności wywoływania każdego z nich przez testera w osobnym wierszu kodu. Przydatne jest też dołączenie kroków `setUp` i `teardown` do platformy, aby można je było automatycznie aktywować przed i po każdym przypadku testowym.

Platformy xUnit

Taką funkcjonalność dodatkową można dodać korzystając z gotowych platform testów jednostkowych. Są one tworzone jako projekty open source i są dostępne za darmo w sieci WWW dla większości popularnych języków programowania. Wersja dla języka Java nazywa się JUnit, dla C++ CppUnit i dla .Net NUnit. Wszystkie trzy są oparte na platformie SUnit zaprojektowanej i wprowadzonej przez Kenta Becka w roku 1998 (zobacz [URL: SUnit]), w efekcie mają podobną

strukturę. To podobieństwo przyczyniło się do powstania szeroko używanego terminu „platformy xUnit”[18].

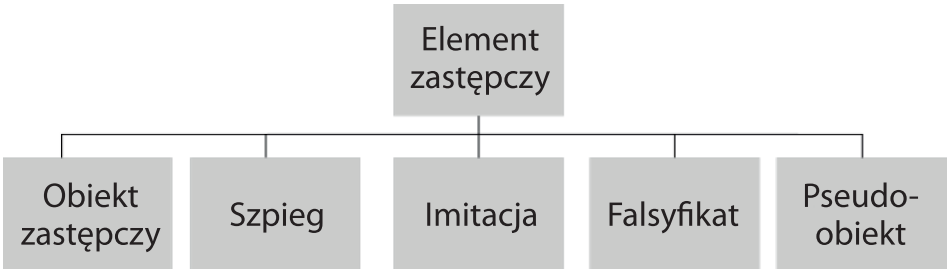
W naszym studium przypadku dla sterownika eHome korzystamy z platformy PHPUnit [URL: PHPUnit]. Jeśli porównamy kod testowy w studiach przypadków 4-2 i 4-4, zobaczymy, jak korzystanie z platformy testowej może uprościć strukturę kodu testowego.

4.4 Obiekty zastępcze

Wersje klasy `Device`, które pokazaliśmy do tej pory, nie mają zależności od innych składników systemu eHome. Klasa może być testowana niezależnie przy użyciu testów jednostkowych. W praktyce jednak testy jednostkowe są zwykle wykonywane na obiektach, które są składnikami większych systemów i są zależne od innych składników, bez których nie mogą działać.

Aby więc przetestować obiekt, wszystkie zależne składniki muszą być zainstalowane w środowisku testowym. Problem polega tutaj na tym, że podejście sterowane testami jest zaprojektowane specjalnie w celu unikania tego typu sytuacji poprzez indywidualne testowanie każdego obiektu. Dodatkowo, jeśli dane elementy zależne mają być implementowane w późniejszych sprintach, niektóre składniki dodatkowe mogą nie być dostępne w czasie testowania.

Rozwiązaniem tego dylematu jest zastąpienie wszelkich składników zależnych elementami zastępczymi zwanymi dublerami testowymi. [Meszaros 07] definiuje następujące typy dublerów:



Rys. 4–1 Typy elementów zastępczych

- **Obiekt zastępczy (stub):** Zastępuje składnik zależny elementem, który ma identyczny interfejs i daje określone reakcje lub zwraca pewne wartości zgodnie z wstępnie zdefiniowanym wzorcem. Z punktu widzenia testu obiekt zastępczy działa jako dodatkowy, pośredni parametr wejściowy.
- **Szpieg (spy):** Obiekt zastępczy, który zawiera mechanizm dodatkowy rejestrujący wywołania i dane przekazane przez obiekt testowy. Rejestrowane dane mogą być następnie używane do

18 [Meszaros 07] opisuje szczegółowo platformy xUnit.

tworzenia kopii zapasowej wyników testu lub pomagają w diagnozowaniu i debugowaniu.

- **Imitacja (mock):** „Inteligentny" obiekt zastępczy, który analizuje wywołania i poprawność danych otrzymywanych z obiektu testowego, a także niezależnie zwraca reakcję lub wynik do obiektu. Z punktu widzenia testu imitacja działa jak dodatkowy krok weryfikacyjny dla pośrednich danych wyjściowych obiektu testowego.
- **Falsyfikat (fake):** Wykorzystuje wysoce wyspecjalizowaną implementację w celu zastąpienia składnika zależnego, który jest wymagany do testowania, ale który nie ma wpływu na wyniki testów.
- **Pseudo-obiekt (dummy):** „Pseudo-obiekt", „obiekt pusty" lub „wskaźnik null", który zastępuje obiekt danych wymagany składniowo przez obiekt testowany. Pseudo-obiekt nie jest interpretowany i dlatego nie stanowi części danych testowych.

Szacując potrzebny nakład pracy zespół Scrum powinien mieć świadomość, że takie obiekty zastępcze odgrywają znacznie ważniejszą rolę niż w tradycyjnie zarządzanych projektach.

- Programowanie przyrostowe zwykle oznacza, że niektóre z wymaganych składników albo nie są gotowe, albo jeszcze nie istnieją. Żeby zapobiec zablokowaniu testu jednostkowego i zapewnić przepływ odpowiedniej informacji zwrotnej, trzeba korzystać z elementów zastępczych w miejsce brakujących składników.
- Sterowanie testami automatycznie wykonuje istniejące testy jednostkowe za każdym razem, gdy dany kod zostanie zmieniony. Może się to zdarzyć wiele razy w ciągu kilku minut, ważne jest więc, aby długość trwania każdego uruchomienia testu ograniczać do minimum. Ta konieczność jest kolejnym dobry powodem na wykorzystanie imitacji i innych elementów zastępczych.

Kodowanie wymaganych elementów zastępczych jest częścią procesu automatyzacji testów i może wymagać znaczących nakładów pracy. Trzeba to brać pod uwagę podczas planowania sprintu.

4.5 Zarządzanie testami jednostkowymi

Zarządzanie testami jest często zaniedbywane niezależnie od tego, czy zespół pracuje nad projektem tradycyjnym, czy zwinnym. Często zakłada się po prostu, że programiści tradycyjni albo zespół Scrum w jakiś sposób automatycznie przeprowadzą wymagane testy jednostkowe bez użycia konkretnych instrukcji lub postanowień. W rzeczywistości istnieje wiele aspektów procesu zarządzania testami, które muszą być uzgodnione:

- **Platforma testowania jednostkowego:** Mistrz Scrum (albo menedżer testów[19]) musi zapewnić, że wszyscy członkowie zespołu będą pisać testy o podobnej strukturze. Istotne jest unikanie sytuacji, w których każdy programista zarządza swoimi własnymi testami jednostkowymi na oddzielnym komputerze, a zespół nie wie, które testy istnieją, gdzie je można znaleźć albo kiedy były uruchamiane. Testy muszą być przechowywane centralnie w sieciowych zasobach zespołu w systemie plików o uzgodnionej strukturze. Aby uprościć instalację, dobrze jest przechowywać kod testowy oddzielnie od kodu programu przy wykorzystaniu folderów równoległych[20]. Trzeba też ustalić reguły nazewnicze dla klas i przypadków testowych, a także dla struktury każdego przypadku testowego (zobacz podrozdział 4.1.2). Menedżer testów powinien nakłonić zespół do wykorzystania gotowej platformy xUnit, a własne platformy powinny być używane tylko w wyjątkowych przypadkach. Z pewnością lepiej, aby zespół inwestował swój czas i energię w projektowanie i automatyzowanie testów zamiast w tworzenie platformy.
- **Mierzenie pokrycia:** Mistrz Scrum (lub menedżer testów) musi zapewnić, żeby pokrycie kodu testami było niezawodnie mierzone a jego rozwój pomiędzy sprintami regularnie analizowany. To wymaga integracji odpowiedniego narzędzia do mierzenia pokrycia w środowisku ciągłej integracji (zobacz rozdział 5). Powinno ono być w stanie zmierzyć pokrycie klas, pokrycie metod w klasie oraz pokrycie wierszy kodu w każdej metodzie. Oczywiście takie miary mają sens tylko wtedy, gdy zespół uzgodni z góry odpowiednie limity pokrycia. Nawet w projekcie Scrum niemożliwe jest testowanie wszystkich części kodu programu równie dokładnie, a limity pokryć mogą się różnić dla różnych podsystemów i jednostek. Wymagane limity pokryć powinny być ustanowione w zgodzie z analizą ryzyka przeprowadzoną dla każdej jednostki.
- **Statyczna analiza kodu:** Dynamiczne testy jednostkowe powinny być uzupełnione odpowiednią statyczną analizą kodu. Najlepsze narzędzie do wykorzystania będzie zależało od stosowanego języka programowania, a po jego wybraniu narzędzie musi zostać sparametryzowane i zintegrowane ze środowiskiem ciągłej integracji (zobacz rozdział 5). Ostatecznie narzędzie do analizy kodu powinno automatycznie sprawdzać, że zespół stosuje się do

19 Podrozdział 3.7.2 wyjaśnia, jak ta rola jest przydzielana w zespole Scrum.

20 Aby lepiej zilustrować interakcję pomiędzy kodem programu i kodem testowym, nasz przypadek testowy przechowuje oba w tym samym katalogu.

wszystkich narzuconych sobie reguł. Wszelkie reguły kodowania stosowane przez zespół (lub których planuje użyć), które nie mogą być automatycznie sprawdzane, powinny zostać odrzucone.

- **Przeglądy kodu programu:** Mistrz Scrum (albo menedżer testów) powinien zapewnić, że będą miały miejsce regularne przeglądy kodu programu[21]. Przegląd kodu programu powinien pytać, czy aktualna hierarchia klas odpowiednio reprezentuje planowaną architekturę systemu, czy interfejsy API są odpowiednie dla klas i czy restrukturyzacja kodu mogłaby poprawić jakość kodu. Wzorce czystego kodu opisane w [Martin 08][22] zapewniają mnóstwo przydatnych wskazówek i porad dotyczących rozpoznawania, kiedy warto skorzystać z restrukturyzacji.
- **Przeglądy kodu testowego:** Kod testowy powinien również podlegać regularnym przeglądom. Głównym powodem tego jest sprawdzanie, co ma robić dany test i jak skutecznie wykonuje przydzielone mu zadanie. Przeglądy służą do sprawdzenia, czy zespół wykorzystuje odpowiednie techniki projektowania testów, takie jak partycjonowanie równoważnościowe, analiza wartości granicznych, czy testowanie oparte na stanach. Jeśli analiza ta wykaże jakieś braki, zespół powinien rozważyć wdrożenie odpowiednich szkoleń. Przegląd pomoże też w ujawnieniu, które przypadki testowe trzeba przepisać, aby pokryć całą specyfikację funkcji lub klasy.

Karta zespołu dokumentuje reguły ustalone przez zespół.

Każdy zespół Scrum powinien mieć cele i uzgodnienia dla wymienionych wyżej punktów. Jeśli jeszcze nie istnieją, mistrz Scrum musi podjąć inicjatywę i pomóc zespołowi w wyborze menedżera testów. Wszystkie uzgodnienia są dokumentowane w karcie zespołu (zobacz podrozdział 3.6). Podobnie do wszelkich zobowiązań podjętych podczas projektu Scrum, również te mogą i powinny być modyfikowane pomiędzy sprintami, w miarę jak zespół stale się uczy. Jest to podejście przyjęte przez zespół eHome:

21 W tym kontekście termin „przegląd" obejmuje inspekcję, wykonywanie krok po kroku, przeglądanie w parach i inne praktyki. Najbardziej odpowiedni typ przeglądu będzie zależał od oszacowania ryzyka dla każdej jednostki.

22 Należą do nich zasady nazywania identyfikatorów i zmiennych, tworzenia struktur klas i metod, formatowania i komentowania kodu źródłowego oraz wskazówki dotyczące nazywania i tworzenia testów zautomatyzowanych.

Studium przypadku 4-5

Studium przypadku 4-5 sterownika eHome: rozszerzanie karty zespołu

Podczas podsumowania następującego po pierwszym sprincie zespół omawia stan swojego zaangażowana w testowanie jednostkowe. Wszyscy dochodzą do wniosku, że wszystko działa dobrze, a żeby tak było dalej, mistrz Scrum zgadza się z zespołem, aby dodać następujące wskazówki dotyczące testów jednostkowych do karty zespołu:

Karta zespołu

– Jak będziemy implementować Scrum w projekcie sterownika eHome –

…

Praktyki

- …
- Testowanie jednostkowe:
  - Programowanie sterowane testami! Napisz test wykrywający błędy przed modyfikacją dowolnego kodu
  - Istnieje co najmniej jeden test dla każdej klasy `public`
  - Każda klasa testowa jest weryfikowana przez innego członka zespołu

Najbardziej zaawansowany tester w zespole otrzymuje rolę menedżera testów i jest odpowiedzialny za nadzorowanie wszystkich zadań związanych z testowaniem – ma to pomóc w odciążeniu mistrza Scrum, który współpracuje z zespołem w niepełnym wymiarze godzin.

4.5.1 Planowanie testów jednostkowych

Główną ideą testowania jednostkowego jest sprawdzanie funkcjonalności i solidności jednostki oprogramowania, gdy jest ona poddawana nieprawidłowemu użyciu (na przykład jest wywoływana z nieprawidłowymi parametrami), a rolą menedżera testów jest zapewnienie, aby testowanie jednostkowe pozostawało skupione na tych dwóch czynnikach. Wprowadziliśmy już i opisaliśmy odpowiednie metody projektowania testów na początku tego rozdziału.

Planowanie programowania elementów zastępczych

Testy jednostkowe są zaprojektowane do testowania obiektu w izolacji od innych składników programu, a platforma testowania jednostkowego powinna wykonywać te testy tak szybko, jak to możliwe. Aby efektywnie spełniać oba te kryteria, ważne jest korzystanie z obiektów zastępczych (zobacz powyżej). Programowanie elementów zastępczych jest częścią procesu automatyzacji testów i może wymagać

sporo wysiłku, co trzeba wziąć pod uwagę podczas planowania sprintu. Menedżer testów w zespole Scrum musi mieć pewność, że elementy zastępcze faktycznie zostaną oprogramowane. Jeśli to zadanie zostanie zapomniane lub zignorowane (być może dla zaoszczędzenia czasu), nie będzie dało się przeprowadzić powiązanego z nim testu jednostkowego, dopóki wszystkie składniki wchodzące w interakcję z testowanym obiektem nie zostaną ukończone, co znacząco zwiększy czas potrzebny do przeprowadzenia testowania jednostkowego.

Dodawanie niefunkcjonalnych testów jednostkowych

Gdy już kilka sprintów spowoduje powstanie wystarczającej liczby zautomatyzowanych testów wysokiej jakości, możemy rozszerzyć pakiet testów jednostkowych o testy niefunkcjonalne – na przykład sprawdzające atrybuty wydajnościowe na poziomie klasy. Takie testy niefunkcjonalne nie gwarantują jednak niefunkcjonalnych aspektów gotowego produktu, które muszą być sprawdzone przy użyciu testów niefunkcjonalnych na poziomie systemowym Z drugiej strony pomagają zidentyfikować istniejące słabości na wczesnym etapie, więc dobrze jest korzystać z testów jednostkowych do sprawdzania niefunkcjonalnych aspektów jednostek, których cel lub pozycja w architekturze systemu wpływa na ogólne zachowanie produktu. To podejście pomaga wcześnie rozpoznać problemy wydajnościowe, które w przeciwnym razie stałyby się widoczne dopiero podczas testowania systemowego.

4.6 Pytania i ćwiczenia

4.6.1 Samoocena

Pytania i ćwiczenia pomagające w ocenie, jak zwinny jest naprawdę projekt lub zespół.

1. Czy zespół napisał już zautomatyzowane testy jednostkowe? Ile? Dla których klas?
2. Czy nasi programiści zarządzają swoimi własnymi testami, czy też wszystkie testy jednostkowe są zarządzane centralnie? Czy korzystamy z platformy testowania jednostkowego?
3. W jakim środowisku przeprowadzane są testy? Na własnym komputerze programisty, czy na serwerze ciągłej integracji? Czy środowisko testowe jest odpowiednio zdefiniowane i czy jest powtarzalne?
4. Kiedy wykonywane są testy jednostkowe? Czy jest to decyzja programisty? Podczas budowania programu? Kiedy kod jest rejestrowany w systemie zarządzania konfiguracją? Automatycznie wewnątrz środowiska ciągłej integracji (zobacz podrozdział 5.5)?

5. Z jakich kryteriów pokrycia korzystamy? Pokrycie klas (tzn. każda klasa wymaga swojej własnej klasy testowej)? Pokrycie metod (tzn. przynajmniej jeden przypadek testowy na metodę publiczną)? Pokrycie wierszy kodu? Pokrycie odgałęzień? Pokrycie ścieżek? Pokrycie oparte na stanach?
6. Z jakich docelowych wartości pokrycia korzystamy? Jak często wartości te są mierzone? Które wartości pokrycia faktycznie uzyskujemy?
7. Które wartości uzyskaliśmy teraz/dzisiaj? Gdzie zespół przechowuje te wartości? Gdzie można je znaleźć, jeśli są potrzebne w danym momencie?
8. Czy przeprowadzamy przeglądy przypadków testowych? Regularnie? Do jakich wniosków dochodzimy? Jakie środki zostały wprowadzone, żeby poprawić pokrycie testami i/lub jakość naszych testów jednostkowych?
9. Jakie techniki projektowania testów stosujemy? Partycjonowanie równoważnościowe? Analizę wartości granicznych? Testowanie oparte na stanach? Czy zespół został przeszkolony w korzystaniu z tych technik?
10. Kiedy testy są projektowane? Przed rozpoczęciem pisania kodu (sterowanie testami)? Po napisaniu kodu wykonywalnego dla danej jednostki? Na końcu sprintu?
11. Kto analizuje i rejestruje wyniki testów oraz sprawdza, czy dana funkcja jest gotowa?
12. Jakie reguły stosujemy, aby zdecydować, którymi usterkami trzeba zarządzać przy pomocy sytemu zarządzania usterkami? Jak zapewniamy, że wszystkie usterki zostaną poprawione?
13. Oprócz testowania jednostkowego, które miary zarządzania jakością są wykorzystywane przez zespół? Zautomatyzowana analiza kodu? Przeglądy kodu?
14. Czy zarządzanie jakością i testowanie (oraz ich wyniki) są omawiane podczas codziennego spotkania Scrum i retrospektyw? Jakie nowe wnioski zostały wyciągnięte? Jakie konkretne akcje usprawniające zostały uzgodnione? Które z nich są obecnie wdrażane?

4.6.2 Metody i techniki

Te pytania pomogą w podsumowaniu treści bieżącego rozdziału.

1. Jakie sekcje składają się na test jednostkowy? Wyjaśnić znaczenie każdej z nich.

2. Jaki jest wynik pokrycia wierszy kodu dla uruchomienia `test_KitchenLightOn` w wersji 2 klasy `Device` pokazanej w studium przypadku 4-2a?
3. Dodać przypadek testowy `test_setStatusUnknown`, który sprawdza, czy wartość stanu `unknown` jest traktowana jako prawidłowa przez `is_validStatus()`. Dlaczego ten dodatkowy krok testowania jest konieczny tutaj, a nie w innych przypadkach dla tej klasy?
4. Przyjrzyjmy się diagramowi stanów w studium przypadku 4-3a. Jeśli testy jednostkowe dla klasy uzyskają 100% pokrycia wierszy kodu na poziomie kodu, czy wszystkie metody klas są wywoływane co najmniej raz? Czy możemy bezpiecznie stwierdzić, że testy w modelu stanów osiągają 100% pokrycie? Jeśli tak, to dlaczego?
5. Wyjaśnić, dlaczego użycie programowania sterowanego testami daje lepsze interfejsy API i lepsze ogólne możliwości testowania.
6. Jakie problemy mogą wystąpić podczas wprowadzania sterowania testami i jak mistrz Scrum może im przeciwdziałać?

4.6.3 Inne ćwiczenia

Te ćwiczenia pomogą zagłębić się w zagadnienia poruszone w trakcie tego rozdziału.

1. Przekształcić metodę `set_dimLevel()` z klasy `Dimmer` tak, aby przypadki testowe wymienione w studium przypadku 4-4b się powiodły.
2. Użyć metod opisanych w [Bashir/Goel 99] do podzielenia klasy `Dimmer`. Jakie części powstaną? Które sekwencje przypadków testowych trzeba następnie uruchamiać dla klasy `Dimmer`?
3. Wywołanie `dim('0%')` ma przełączyć urządzenie do stanu `'off'`. Naszkicować przypadek testowy opisujący to zachowanie.
4. Polecenie `dim` nie powinno mieć wpływu na urządzenie w stanach `unknown` i `off`. Naszkicować przypadki testowe opisujące to zachowanie.
5. Kod klasy `Dimmer` jest rozszerzony tak, aby obejmował te nowe wymagania, co powoduje powstanie nowej wersji 3. Czy ta wersja zawiera po podziale części inne niż te, które już wymieniliśmy? Jeśli tak, to jaki wpływ ma to na wymagane sekwencje testowe?
6. System eHome ma być rozszerzony tak, aby zawierał zaplanowane polecenia przełączenia przy pomocy nowej klasy o nazwie `Timer`. Ponieważ zespół wykorzystuje sterowanie testami, musi naszkicować odpowiednie przypadki testowe dla nowej klasy. Naszkicować

przypadki testowania jednostkowego wymagane do zdefiniowania następującego zachowania klasy `Timer`:

- `set_interval()` określa liczbę sekund działania zegara
- `start()` uruchamia zegar, `stop()` go zatrzymuje
- `is_finished()`[23] zwraca `TRUE`, gdy upłynie ustawiony czas, a `FALSE` w przeciwnym razie.

Jakie pułapki można znaleźć w tych specyfikacjach?

23 Dla celów tego ćwiczenia zignorujemy fakt, że zegar czasu rzeczywistego musiałby wywołać przerwanie lub procedurę obsługi przerwania.

5 Testowanie integracyjne i ciągła integracja

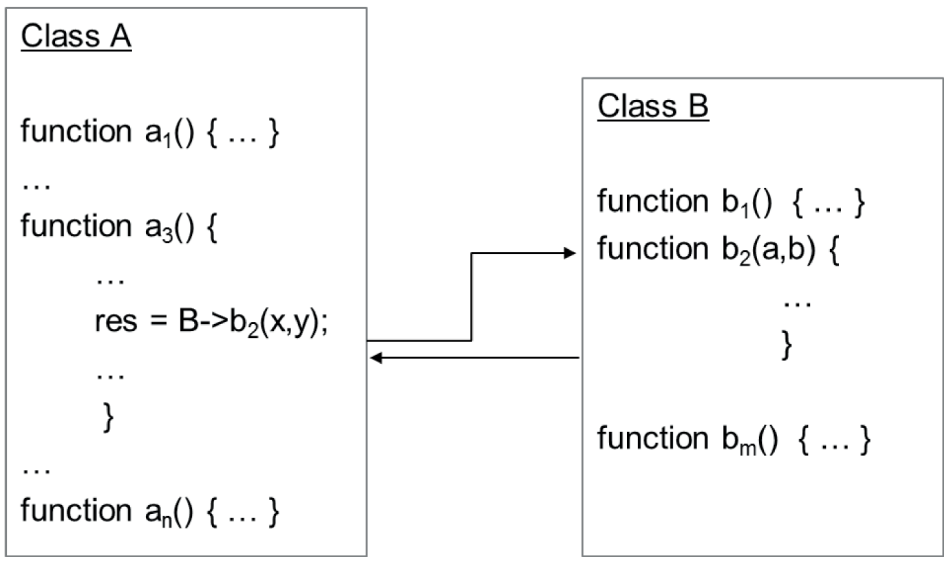
Ten rozdział wyjaśnia różnice pomiędzy testowaniem jednostkowym, a testowaniem integracyjnym, jak projektować przypadki testów integracyjnych i jak je osadzać w całkowicie zautomatyzowanym środowisku ciągłej integracji wraz z już napisanymi testami jednostkowymi.

5.1 Testowanie integracyjne

System informatyczny składa się z wielu składników. Aby system działał prawidłowo, każdy składnik musi samodzielnie funkcjonować niezawodnie, a co ważniejsze, wszystkie składniki muszą działać razem zgodnie z planem. Testowanie integracyjne sprawdza, czy tak jest.

Testy integracyjne są projektowane w celu odkrycia potencjalnych usterek w interakcji pomiędzy poszczególnymi składnikami i ich interfejsami. Aby przeprowadzić test integracyjny, dwa składniki są łączone (czyli integrowane) i aktywowane przez odpowiednie integracyjne przypadki testowe. Rys. 5-1 pokazuje schemat przykładowego testu integracyjnego obejmującego dwie klasy.

Testy integracyjne sprawdzają interakcję pomiędzy poszczególnymi składnikami.



Rys. 5–1
Test integracyjny obejmujący dwie klasy

Interfejs klasy A składa się z metod a_1, a_2, … a_n, natomiast klasa B zawiera metody b_1, b_2, … b_m. Obie klasy przeszły wszystkie swoje testy jednostkowe. Testy integracyjne, które teraz zdefiniujemy, sprawdzą, czy A i B prawidłowo współpracują ze sobą. W naszym przykładzie integracja ma miejsce pomiędzy metodą b_2 i jej wywołaniem z metody a_3[1]. Musimy więc sprawdzić, czy metoda b_2 jest wywoływana prawidłowo i czy następnie zwraca żądany wynik do klasy A.

5.1.1 Typowe błędy integracyjne i ich przyczyny

Choć A i B same funkcjonują prawidłowo, to różne wadliwe zachowania mogą pojawić się, gdy będą współpracować razem. Najważniejszymi z nich są błędy interfejsów:

- A wywołuje niewłaściwą metodę z klasy B:
  Jeśli założymy, że A jest klasą w interfejsie użytkownika sterownika eHome, a B jest klasą reprezentującą poszczególne urządzenia, to w celu wyłączenia lampy kuchennej klasa A musiałaby wywołać metodę `switch('kitchen', 'lamp', 'off')` w klasie B. Jednakże w klasie A błędnie zakodowano wywołanie `dim('kitchen', 'lamp', '50')`.
- A wywołuje prawidłową metodę z B korzystając z nieprawidłowych wartości parametrów:
  Przykładowo A ma włączyć lampę w kuchni, ale wywołuje `switch('living_room', 'lamp')`. Pierwszy parametr jest semantycznie niepoprawny, natomiast trzeciego parametru w ogóle brakuje.
- Połączone składniki kodują zwracaną wartość w inny sposób:
  W klasie A wywołanie `dim('kitchen', 'lamp', '50')` oznacza, że lampa jest ustawiona na 50% jasności korzystając z wartości zakodowanej w przedziale od 0 do 100. Jednakże metoda `dim()` w klasie B oczekuje, aby wartość jasności była zakodowana jako liczba zmiennoprzecinkowa pomiędzy `0` a `1` (w tym przypadku 0,5).
- A wywołuje żądaną metodę prawidłowo, ale w złym czasie lub w nieprawidłowej kolejności (zobacz podrozdział 4.1.3).
  Jako przykład, żądany obiekt w klasie B nie istnieje jeszcze w momencie, gdy A korzysta z B i powoduje błąd czasu wykonania w A. Gdyby nawet obiekt w B już istniał, mogłoby być tak, że jedna z jego zmiennych nie została zainicjowana lub została zainicjowana nieprawidłową wartością. To powoduje albo błąd czasu wykonania w B, albo zwrócenie nieprawidłowej wartości z B do A.

1 Klasa A wymaga obiektu klasy B. Innymi słowy A jest zależna od B.

Gdy A będzie pracować z nieprawidłową wartością, kolejne błędy są nieuniknione.

Jeśli składniki komunikują się asynchronicznie[2], mogą wystąpić również następujące dodatkowe błędy:

- **Błędy czasowe**
  - A przekazuje dane, gdy klasa B nie jest gotowa na ich odbiór.
  - B zwraca wynik do A albo za wcześnie, albo za późno. A nie może odebrać lub przetworzyć zwróconych danych.
  - A nie reaguje na upływ limitu czasu lub reaguje błędnie (na przykład powtarzając wywołanie).
- **Błędy przepustowości**
  - A przekazuje więcej danych, niż B może przetworzyć w określonym czasie, co powoduje utratę danych lub przepełnienie danymi w B.
  - B zwraca więcej danych, niż A może przetworzyć.
- **Problemy wydajnościowe**
  - A przekazuje dane szybciej lub częściej, niż B może przetworzyć, co powoduje przestoje lub powtórne przesyłanie danych. Wynikiem jest spadek szybkości przetwarzania pomiędzy tymi dwoma składnikami poniżej określonego minimum.

Jeśli dwa składniki są rozmieszczone w dwóch oddzielnych systemach sprzętowych, mogą również nastąpić **błędy transmisji**:

- Połączenie (sieć lokalna na przykład) jest uszkodzone lub wyłączone, co powoduje błąd lub przerwy w transmisji albo brak danych.

Powody błędów

Jeśli dane składniki były tworzone przez różnych programistów lub zespoły, to zwiększa się ryzyko usterek integracyjnych. W takich przypadkach główną przyczyną usterki jest zwykle niezrozumienie lub odmienna interpretacja specyfikacji albo po prostu brak komunikacji pomiędzy programistami.

Zmiany w kodzie i aktualizacje oprogramowania mogą również powodować usterki – na przykład, jeśli jeden składnik zostanie zmieniony, ale inny, zależny od niego, nie. Kompilatory w językach programowania opartych na kompilatorach i typach statycznych zwykle wykrywają

2 W odróżnieniu od synchronicznej pętli komunikacyjnej, w której składniki oczekują na odpowiedź z drugiej strony, zanim prześlą lub otrzymają dane, składniki w pętli asynchronicznej wysyłają i odbierają dane bez oczekiwania na odpowiedź, odczytują i zapisują dane ze wspólnego bufora danych.

te typy składniowych błędów, natomiast wiele języków opartych na interpreterach (na przykład PHP) wykrywa te błędy jedynie w czasie uruchomienia programu.

Studium przypadku 5-1

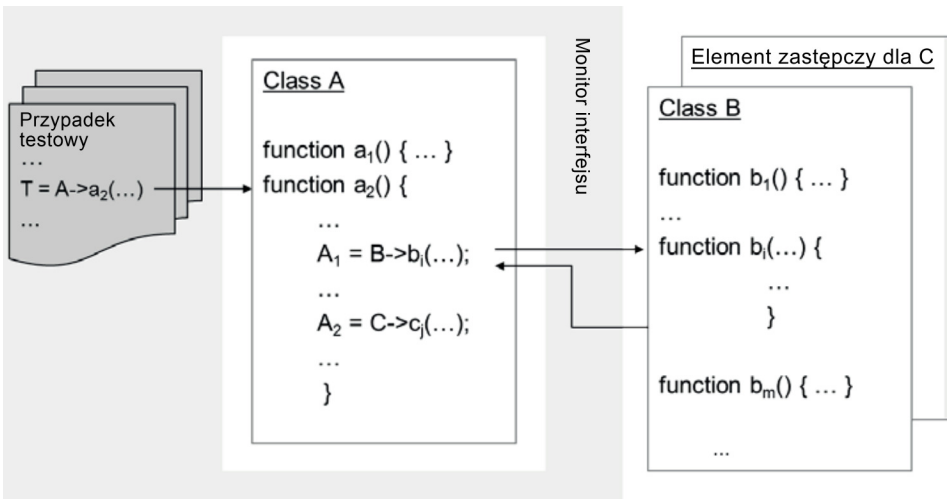
Studium przypadku 5-1 dla sterownika eHome: Usterki integracyjne spowodowane niekompletnym refaktoringiem

Metoda `set_status()` w klasie `Device` została rozszerzona o wywołanie `write()` zapisujące odpowiedni tekst w buforze bazodanowym, gdy następuje prawidłowa zmiana stanu. Jednakże osoba programująca interfejs bazodanowy zmieniła nazwę metody na `writeMsg()`, ale zapomniała odpowiednio zmienić wywołanie metody w klasie Device. Ponieważ programista wyłączył komunikaty o błędach interpretera PHP, usterka ta pozostała początkowo nieodkryta i wyszła na światło dzienne dopiero podczas uruchomienia testu integracyjnego, gdy wywołanie `write()` spowodowało błąd czasu wykonania.

Ze względu na stałe zmiany i poprawianie kodu związane z procesami zwinnymi ryzyko spowodowania usterek integracyjnych przez zmiany lub aktualizacje jest szczególnie częste w środowisku zwinnym. Dlatego nawet w przypadku zmiany pojedynczego składnika należy zawsze ponownie uruchomić nie tylko jego test jednostkowy, ale też wszystkie testy integracyjne, które obejmują składniki z nim współpracujące.

5.1.2 Projektowanie przypadków testów integracyjnych

Przypadki testów integracyjnych są projektowane do wykrywania usterek wymienionych wyżej rodzajów. Jeśli składniki A i B są zintegrowane w taki sposób, że A wykorzystuje B, to A wymaga przypadków testowych, które wywołują wszystkie istotne przypadki użycia B. Odpowiednia konfiguracja testów wygląda wtedy następująco:



Rys. 5–2 Konfiguracja testów integracyjnych i interfejsów

Systematyczne ustalanie przypadków testów integracyjnych

Aby systematycznie ustalać wymagane przypadki testów integracyjnych, należy wykonać następujące kroki:

1. Analiza interakcji: Zidentyfikować i wypisać, które usługi i metody w B są używane przez A. Na przykład zidentyfikować przy użyciu diagramu architektury, które interakcje z innymi składnikami istnieją i ustawić odpowiednie obiekty zastępcze w środowisku testowym.
2. Partycjonowanie równoważnościowe: Dla każdej zidentyfikowanej usługi lub metody przeanalizować, które zestawy parametrów i/lub komunikatów są przekazywane i wykorzystać partycjonowanie równoważnościowe do ustalenia, które z nich są istotne dla zachowania danych składników.
3. Ustalenie danych wejściowych dla przypadków testowych: Ustalić wywołania lub sekwencje wywołań w interfejsie API klasy A, które są konieczne do wywoływania B przy użyciu zestawów parametrów i komunikatów określonych w kroku 2 powyżej, co obejmie wszystkie zidentyfikowane równoważności.
4. Zdefiniowanie porównania stanu oczekiwanego z faktycznym: Zdefiniować oczekiwane zachowanie dla A i B dla każdego przypadku testowego i ustalić, gdzie i jak można przeanalizować oczekiwane reakcje. Będzie to interfejs API klasy A i/lub B, pośrednie dane wyjściowe z A i/lub B albo zarejestrowany ruch danych obsługiwany przez interfejs transmisyjny.
5. Testowanie solidności komunikacji asynchronicznej: Dla wywołań asynchronicznych w klasie B potrzebne będą dodatkowe testy sprawdzające reakcje czasowe, przepustowość, pojemność i błędy wydajnościowe. Składniki rozprowadzane pomiędzy wieloma środowiskami sprzętowymi również muszą być testowane w celu zapewnienia, że transfer danych pomiędzy nimi jest stabilny i niezawodny.

Krok 5 jest konieczny tylko wtedy, jeśli dane składniki działają równolegle i są połączone asynchronicznie. Jeśli tak jest, testy solidności, które celowo prowokują sytuacje krytyczne (np. limit czasu), są istotne dla ustalenia, czy integrowany system ma odpowiednią, wbudowaną odporność na błędy.

To podejście może być też stosowane do testowania złożonych składników zbudowanych z wielu jednostek – innymi słowy do projektowania przypadków testów integracyjnych dla złożonych podsystemów lub testowania integracji całego systemu (zobacz podrozdział 5.3).

Studium przypadku 5-2

Studium przypadku 5-2 sterownika eHome: odpowiednia reakcja na przerwanie połączenia

Obiekt `Device` sterownika eHome może zmienić swój stan włączenia (np. z włączonego na wyłączony), gdy dane urządzenie fizycznie lub wirtualnie przeprowadziło operację przełączenia i zwróciło komunikat z potwierdzeniem. Jeśli to potwierdzenie dotrze w zadanym limicie czasowym, obiekt `Device` przełączy swój stan. Jeśli potwierdzenie nie nadejdzie na czas, obiekt `Device` powtórzy operację przełączania.

Architektura systemu eHome (zobacz Studium przypadku 3-2 na stronie 27) wyznacza trzy miejsca, w których połączenie pomiędzy sterownikiem eHome a urządzeniem może być przerwane:

a) Sterownik eHome ↔ Bufor komunikatów (baza danych)

b) Bufor komunikatów (baza danych) ↔ adapter magistrali

c) Adapter magistrali ↔ System magistrali

Test integracyjny musi sprawdzać, czy sterownik eHome poprawnie reaguje bez konieczności podłączenia systemu do rzeczywistej, fizycznej magistrali tak jak podczas testowania systemowego.

Poniższa symulacja jest używana do testowania sytuacji a):

Po przesłaniu komunikatu przełączającego test zamyka połączenie z bazą danych. W tym stanie komunikat nie powinien być powtarzany, ponieważ mógł już zostać podany dalej. Gdy połączenie z bazą danych zostanie nawiązane ponownie, uruchamiana jest naprawa, która ponownie sprawdza stan wszystkich dołączonych urządzeń i importuje wyniki do bazy danych. Jednakże funkcjonalność ta jest zaplanowana do implementacji w późniejszym sprincie.

Aby przetestować sytuację b), instalacja testowa opróżnia bufor komunikatów w celu zapewnienia, że żadne komunikaty nie będą eksportowane. Limit czasowy jest ustawiany na minimalną wartość w celu zapewnienia, że test integracyjny nie zostanie niepotrzebnie spowolniony. Aby przetestować, czy obiekt `Device` powtórzył komunikat przełączający, test sprawdza, czy baza danych zawiera teraz dwa identyczne komunikaty przełączające.

Aby przetestować sytuację c), adapter magistrali jest podmieniany przez obiekt zastępczy, który można parametryzować i który przesyła (lub nie) komunikat potwierdzający, gdy osiągnięto limit czasowy po każdym poleceniu przełączającym.

5.1.3 Różnice pomiędzy testami jednostkowymi a testami integracyjnymi

Z punktu widzenia programisty testy integracyjne są bardzo podobne do testów jednostkowych. Oba typy przypadków testowych uzyskują dostęp do obiektu testowego poprzez jego interfejs API i w większości projektów wykorzystują tę samą platformę do automatyzacji testów (zespół eHome wykorzystuje w obu przypadkach platformę PHPUnit).

Z powodu tych podobieństw technicznych testy jednostkowe i integracyjne często nie są wyraźnie rozróżniane, choć są używane do testowania bardzo różnych atrybutów testowanego oprogramowania.

- **Przypadki testów jednostkowych** mają za zadanie sprawdzać, czy poszczególne składniki oprogramowania (np. pojedyncza klasa i jej metody) działają poprawnie. Test sprawdza wewnętrzne działanie jednostki i jej solidność w konfrontacji z błędnym użyciem (np. wywołanie wykorzystujące nieprawidłowe parametry). Przypadki testowe muszą obejmować metody jednostki i ich parametry, o ile to możliwe. Dany składnik jest testowany w izolacji, a inne składniki wymagane do uruchomienia testu są reprezentowane przez obiekty zastępcze. To podejście oznacza, że wszelkie odkrywane usterki są powodowane przez błędy w testowanym właśnie składniku.
- **Przypadki testów integracyjnych** są projektowane w celu sprawdzania, czy dwa niezależne składniki oprogramowania (np. dwie klasy) będą poprawnie współdziałać ze sobą. Test analizuje i sprawdza tak dużo różnych wariantów przekazywania danych pomiędzy składnikami, jak to możliwe. Składniki są łączone podczas testu, a wszelkie usterki, które zostają odkryte, można łatwo przypisać jednemu lub drugiemu składnikowi, albo też kanałowi komunikacyjnemu.
- **Pokrycie testów:** W przypadku konfiguracji pokazanej wyżej, jeśli uruchomione zostaną testy jednostkowe dla składnika A, możliwe jest, że przypadkowo wywoła to interakcję pomiędzy A i B, którą obejmują istniejące testy integracyjne. Z tych samych powodów, jeśli zostaną też uruchomione testy jednostkowe dla B, to niekoniecznie wywoła to zadania dokładnie odzwierciedlające żądane pokrycie testami integracyjnymi. Jeśli natomiast testy integracyjne są używane jako testy jednostkowe, to jest wielce prawdopodobne, że zawierają redundantne testy dla metod należących do A, które są wywoływane przez interakcję pomiędzy oboma składnikami i najprawdopodobniej ujawnią luki w testowaniu metod, które nie wymagają żadnej interakcji.
- **Narzędzia do testowania:** Oprócz platform wykorzystywanych przez testy jednostkowe i integracyjne, testy integracyjne wykorzystują też monitory do diagnozowania i zwracania (w formie czytelnej dla ludzi) ruchu danych pomiędzy interfejsami, magistralami i sieciami należącymi do środowiska testowego. Narzędzia monitorujące dla typowych protokołów, takich jak TCP/IP, są swobodnie dostępne i są wbudowane w niektóre systemy operacyjne (np. polecenie „tcpdump” w systemie Linux). Do monitorowania interfejsów i protokołów specyficznych dla danego projektu

zespoły będą musiały albo utworzyć narzędzia monitorujące specjalnego przeznaczenia, albo przeprojektować istniejące.

Gdy składniki oprogramowania są zintegrowane, nie wystarcza po prostu powtórzyć testy jednostkowe dla poszczególnych jednostek, aby sprawdzić interakcję pomiędzy nimi. Interakcja pomiędzy składnikami prawie zawsze powoduje nowe sytuacje, które muszą być objęte dodatkowymi, specyficznymi testami integracyjnymi. Pomimo podobieństw technicznych, ważne jest wyraźne rozróżnienie pomiędzy tymi dwoma typami testów i opracowywanie każdego zestawu testów przy użyciu wskazówek projektowych nakreślonych tutaj i w rozdziale 4.

5.2 Rola odgrywana przez architekturę systemową

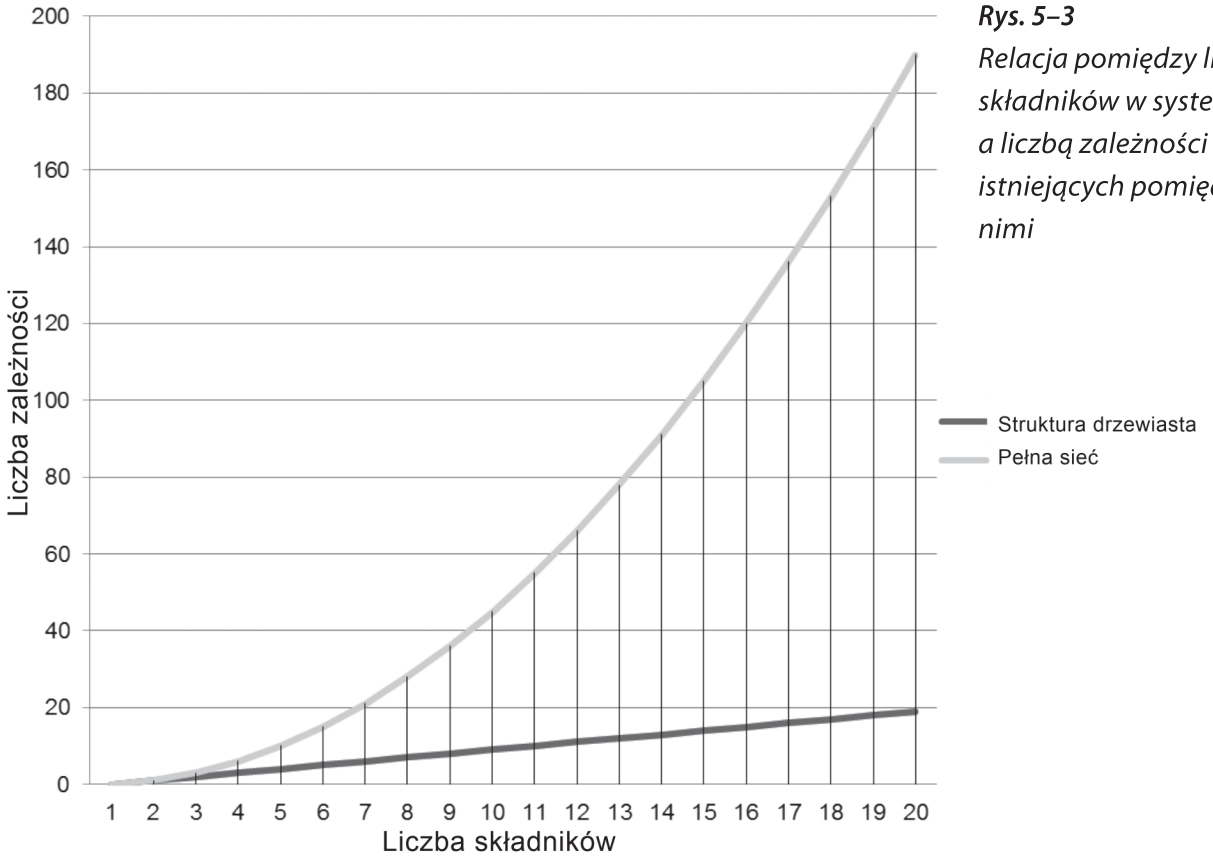
Rozdział 4 wprowadził kilka podstawowych zasad dotyczących minimalnej liczby testów jednostkowych wymaganej do przetestowania klasy. Następne podrozdziały zbadają możliwość wykorzystania podobnych zasad do szacowania nakładu pracy związanego z testowaniem integracyjnym.

Intuicja mówi nam, że im większy i bardziej złożony system, tym większa liczba testów integracyjnych będzie potrzebna do przetestowania go dokładnie. Dla uproszczenia rozmiar systemu może być wyrażony w postaci liczby składników, które wymagają integracji, natomiast złożoność może być wyrażona przy użyciu liczby zależności istniejących pomiędzy nimi. Podsystem pokazany na rys. 5-1 składa się z dwóch klas (A i B), gdzie A zależy od B. Korzystając z notacji opisanej powyżej, daje nam to system o rozmiarze 2 i złożoności 1.

Ponieważ każdy składnik może być też zależny od innych części systemu, liczba zależności jest często nieproporcjonalna do liczby składników, a nawet może rosnąć wykładniczo. Wielkość tych proporcji zależy od architektury testowanego systemu, jak pokazano na rys. 5-3.

Rysunek 5-3 pokazuje wzrost liczby zależności (k) w relacji do liczby składników w systemie. Jeśli składniki są połączone w strukturę drzewiastą, wzrost ten jest liniowy i może być wyrażony przy użyciu wzoru k=(n-1), natomiast połączenia typu sieciowego dają wzrost kwadratowy wyrażony wzorem k=n(n-1)/2.

Podczas zliczania zależności każdy kierunek musi być liczony oddzielnie. Jeśli składniki A i B komunikują się dwustronnie, to obie zależności (A względem B i B względem A) muszą być testowane. Szczegółowe omówienie tematów interfejsów, zależności i ich wpływów na testowanie integracyjne można znaleźć w [Winter et al. 12].



Rys. 5–3
Relacja pomiędzy liczbą składników w systemie a liczbą zależności istniejących pomiędzy nimi

Studium przypadku 5-3

Studium przypadku 5-3 sterownika eHome: Zależności pomiędzy składnikiem sterownika a urządzeniem magistrali

Kliknięcie w przeglądarce przycisku „Włącz lampę w kuchni" wywołuje komunikat `switch( 'kitchen', 'lamp', 'on')` i wysyła go do magistrali. Przełącznik elektroniczny, który jest używany do włączania lampy, musi zareagować na ten komunikat i potwierdzić działanie komunikatem „lampa jest włączona".

Odpowiedni test integracyjny musi obejmować dwa przypadki:

- Przesłanie `switch( 'kitchen', 'lamp', 'on')` i sprawdzenie, czy przełącznik wykonuje żądane działanie.
- Sprawdzenie w przeglądarce, czy komunikat potwierdzający przychodzi w zadanym okresie czasu.

W tym przypadku potrzebne są dwa kroki testowe, ponieważ istnieją dwie oddzielne zależności pomiędzy przeglądarką a przełącznikiem.

5.2.1 Zależności i interfejsy

Liczenie zależności w systemie jest trudniejsze niż szacowanie jego rozmiaru (na przykład przez ustalanie liczby zawartych w nim klas). Takie zależności nie zawsze są jasne i często wynikają z pośrednich połączeń wewnątrz systemu.

- Bezpośrednia zależność istnieje, jeśli (jak pokazano na rys. 5-1) składnik A wywołuje składnik B bezpośrednio poprzez interfejs API. Fakt, że A zależy od składnika B, jest oczywisty po przeczytaniu kodu. Bezpośrednie zależności mogą być więc identyfikowane albo przez kompilator, albo przy użyciu statycznej analizy kodu (tzn. bez faktycznego uruchamiania danych składników).
- Pośrednia zależność istnieje, jeśli wiele składników wspólnie korzysta z pojedynczego zasobu. Jeśli na przykład dwa lub więcej składników ma dostęp do tej samej zmiennej globalnej, korzysta z tego samego pliku lub ma dostęp do tej samej bazy danych.

Zależność pośrednia poprzez wspólnie wykorzystywane zasoby

Zależność pośrednia jest powodowana przez bliźniacze fakty, iż składnik A zapisuje dane do wspólnego zasobu (w ten sposób zmieniając zawartość obiektu danych), natomiast zachowanie składnika B (lub innych) różni się w zależności od tego samego zasobu. Choć oba składniki nie wywołują się nawzajem bezpośrednio, to składniki odczytujące zasoby są pośrednio pod wpływem działań składników, które zapisują do wspólnego obiektu danych. Ponadto jeśli dwa składniki odczytują i zapisują dane we wspólnym obiekcie, powstaje interakcja dwustronna. Jednakże sam kod ujawnia tylko to, że każdy składnik uzyskuje dostęp do danego zasobu – a nie, czy lub kiedy inne składniki też z niego korzystają.

Pośrednie zależności jest bardzo łatwo przeoczyć, a związane z nimi składniki są dlatego często pomijane podczas testów integracyjnych. Jeśli jednak zależności te zostaną w porę dostrzeżone, potrzebne im testy mogą zostać zdefiniowane w taki sam sposób, jak używane do testowania bezpośrednio zależnych obiektów:

- Testy integracyjne dla bezpośrednio zależnych obiektów muszą obejmować wszystkie istotne warianty wywołań metod (tzn. wszystkie istotne kombinacje parametrów), a w przypadku komunikacji asynchronicznej również transfer danych pomiędzy składnikami.
- W przypadku pośrednio zależnych składników testy integracyjne muszą obejmować wszystkie istotne kombinacje sekwencji odczyt/zapis, które wpływają na wspólny zasób i/lub istotną, możliwą zawartość wspólnego zasobu. To drugie może wymagać utworzenia dużej liczby danych testowych.

5.2.2 Łatwość testowania i nakłady pracy na testowanie

Podczas gdy liczba testów jednostkowych zależy od liczby składników[3], wymagana liczba testów integracyjnych zależy od liczby zależności istniejących pomiędzy nimi. Liczba zależności zależy od natury architektury systemu.

Nowoczesne systemy budowane przy użyciu obiektowo zorientowanych technik projektowania i programowania mają zwykle silnie powiązane architektury składające się z dużej liczby małych klas. Ten typ struktury sprawia, że testowanie integracyjne jest dużo bardziej istotne, niż w przypadku innych architektur systemowych. [Winter et al. 12] stwierdza, że „…metody zorientowane obiektowo przesunęły główne źródło usterek z modułów (lub klas) do interakcji pomiędzy klasami. Większość usterek i niemal wszystkie usterki krytyczne w systemie zorientowanym obiektowo mogą więc być odkryte jedynie poprzez testowanie integracyjne."

Testowanie integracyjne może wymagać wielkich nakładów, zwłaszcza gdy architektura systemu zawiera znaczną liczbę zależności w porównaniu z liczbą tworzących je składników. Z kolei nakłady potrzebne na testowanie integracyjne mogą zostać znacząco ograniczone, jeśli całościowa architektura systemowa zostanie odpowiednio uproszczona.

Oprócz technik zorientowanych obiektowo, przyrostowa natura projektów Scrum również zwiększa wymagane nakłady na testowanie. Jeśli sprint dodaje nowe funkcje do systemu, zwykle wiąże się to z dodawaniem nowych klas lub innych składników oprogramowania. Oznacza to nie tylko, że istniejące testy integracyjne muszą zostać powtórzone dla każdej iteracji, ale też nowe testy muszą zostać opracowane, zautomatyzowane i wykonane.

Żeby nakłady na testy integracyjne nie rosły zbyt szybko, ważne jest śledzenie architektury systemu i przeprowadzanie regularnego refaktoringu w celu zachowania jej prostoty.

Nakłady na testowanie integracyjne

Nakłady związane z testowaniem integracyjnym zależą też od łatwości testowania poszczególnych interfejsów. Według [Spillner/Linz 14] łatwość testowania oznacza łatwość i szybkość, z jaką można regularnie testować funkcjonalność i wydajność systemu. System o gorszej łatwości testowania wymaga więcej nakładów na testowanie.

Jeśli chodzi o zależności i interfejsy, które testowanie integracyjne ma sprawdzać, łatwe do testowania zależności można w prosty sposób zidentyfikować w kodzie źródłowym, uzyskać do nich dostęp poprzez platformę testową i obserwować (na przykład korzystając z udokumentowanego wywołania API lub monitora interfejsu). Interfejs jest

3 Jest to uproszczona ocena, która zakłada, że wszystkie klasy mają podobną liczbę metod z podobną liczbą wewnętrznych interakcji.

też łatwy do testowania, jeśli jest „szczupły" (tzn. liczba wariacji danych jest niewielka). Nakłady potrzebne na testowanie integracyjne zależą więc bezpośrednio od liczby istniejących zależności i łatwości ich testowania.

Są to atrybuty architektury systemu, więc dobra architektura będzie składać się z łatwo odróżnialnych podsystemów, które wchodzą ze sobą w interakcję poprzez niewielką liczbę bezpośrednich, szczupłych interfejsów.

Architektura systemowa może zostać poprawiona przez przekształcenie pośrednich zależności w bezpośrednie – na przykład poprzez umieszczenie wspólnych zasobów globalnych w osobnych klasach, które następnie zapewniają dostęp do zasobu poprzez usługę opartą na interfejsie API. To sprawia, że wywołania API są proste do zidentyfikowania wewnątrz kodu i oznacza, że identyfikacja oraz obsługa nieprawidłowych danych może odbywać się centralnie. To z kolei oznacza, że wymagane testy nie muszą być wykonywane osobno dla każdego składnika wykorzystującego zasób. Dodatkowo złożoność składników korzystających z usługi jest ograniczona, a nakłady na testowanie przechodzą z testów integracyjnych dla składników wykorzystujących usługę na testy jednostkowe obejmujące klasę usługową.

5.3 Poziomy integracji

W poprzednich podrozdziałach opisałem, jak tworzyć testy integracyjne dla poszczególnych klas. Powtarzana integracja łączy poszczególne składniki w pakiety i podsystemy, które mogą następnie być traktowane jako niezależne elementy. Składniki, które mają być integrowane, wykazują różne poziomy szczegółowości w zależności od poziomu abstrakcji, na którym odbywa się proces integracji.

5.3.1 Integracja klas

Integracja klas jest najniższym poziomem, na którym odbywa się testowanie integracyjne. W świecie programowania zorientowanego obiektowo mniejsze jednostki (na przykład poszczególne metody) są łączone w klasy i są testowane jako część testu jednostkowego. Jednakże podczas integrowania klas[4] można wyróżnić różne typy integracji:

- **Integracja pionowa** opisuje klasy, które integrują się wzdłuż linii swoich hierarchii dziedziczenia. Jeśli klasa B dziedziczy swoje

4 [Vigenschow 10] oraz [Winter et al. 12] zawierają rozbudowane omówienie odpowiednich sekwencji testowych i wyzwań prezentowanych przez problemy z dziedziczeniem.

atrybuty po klasie A, to wszystkie nieprywatne metody i zmienne w A będą dostępne w B i mogą być wywoływane lub używane przez B. Jeśli klasa A zostanie zmieniona (na przykład, jeśli implementacja funkcji zostanie zmieniona w danym sprincie), istnieje prawdopodobieństwo, że metody A będą działać inaczej. To oznacza również, że wywołania klasy A z poziomu klasy B mogą się nie udać lub zachowywać inaczej. Dlatego test integracyjny dla interakcji pomiędzy A i B musi być powtórzony po każdej zmianie w klasie A. Metody w klasie B (np. m_B) mogą też nadpisywać metody należące do A (np. m_A), więc po zmianach w B również muszą następować powtórzone testy. Przykładowo, jeśli nazwa m_B: zostanie zmieniona (być może przypadkiem), metoda m_A będzie wywoływana zamiast m_B, co może prowadzić do całkiem innego zachowania programu. Takie testy mogą być traktowane jako testy integracyjne pomiędzy obiektami, które należą do jednej hierarchii dziedziczenia. Jednakże w większości przypadków w czasie wykonywania będzie istniał tylko jeden obiekt, który będzie wykonywał kod w metodach klasy (fakt, że część tego kodu jest dziedziczona, nie ma znaczenia dla obiektu). Patrząc na to w ten sposób, testy śledzące hierarchię dziedziczenia mogą być też postrzegane jako testy jednostkowe. Sposób, w jaki zespół Scrum obsługuje takie testy, ma drugorzędne znaczenie dla konieczności dostępności odpowiednich testów do ponownego wykorzystania po zmianach w kodzie.

- **Integracja pozioma** opisuje relacje pomiędzy dwoma klasami bez dziedziczenia i z co najmniej jednym obiektem każdej klasy istniejącym w czasie działania programu – tzn. jedna klasa wykorzystuje drugą. Jeśli A wywołuje metodę z B, istnieje zależność, która musi być odpowiednio przetestowana.
- **Klasy i obiekty złożone:** Specjalny rodzaj zależności „A wykorzystuje B" istnieje, jeśli klasa A zawiera klasę B w postaci struktury danych. Jeśli A wykorzystuje B, to korzysta z metod i/lub zmiennych w klasie B. W takich przypadkach bezpośredni test integracyjny pomiędzy A i B jest często pomijany, ponieważ B jest uważane za część A. Testowanie jednostkowe klasy A testuje pośrednio B, ale tylko przypadkowo. Jeśli kod został dobrze napisany, naszkicowanie testu bezpośredniego jest trudne, gdyż klasa B jest obudowana przez A i jest dla niej prywatna. Utrudnia to zaobserwowanie zachowania B podczas testu. Jednym ze sposobów obejścia tego problemu jest wykorzystanie wstrzykiwania zależności (zobacz przykłady w [Meszaros 07, rozdział 26]), co nie osadza obiektu zależnego (w naszym przypadku klasy B) jako zmiennej w A, ale przekazuje go do konstruktora A jako parametr tak, aby był

widoczny z zewnątrz. Korzystając z tego podejścia można zastąpić B szpiegiem podczas testów integracyjnych lub innym obiektem zastępczym podczas testowania jednostkowego. Upraszcza to też testowanie jednostkowe klasy A (zobacz rozdział 4).

5.3.2 Integracja podsystemów

Przykłady wykorzystane powyżej opisują, jak integrować i testować dwie klasy. Jeśli sparowane klasy zachowują się zgodnie z oczekiwaniami, mogą być traktowane jako nowy, złożony składnik, który z kolei może być integrowany z innymi klasami. Taki sposób integracji krok po kroku służy do tworzenia klastra klas działających w idealnej harmonii. Klastry takie są często nazywane pakietami lub podsystemami. Typ interfejsu używany przez klaster zależy od używanego języka programowania. Niektóre języki pozwalają na oddzielną specyfikację interfejsu specyficznego dla pakietu, natomiast inne wykorzystują sumę interfejsów API wszystkich klas wewnątrz klastra.

Zawsze lepiej korzystać z interfejsu pakietu, o ile to możliwe. Tak jak rozróżniamy metody publiczne i prywatne na poziomie klasy, tak interfejs specyficzny dla klastra jest wydzielony i utrzymuje minimalną liczbę dostępnych z zewnątrz metod zwiększając tym samym łatwość testowania. Po udanych testach podsystemy utworzone w ten sposób mogą być też traktowane jako osobne składniki.

Jeśli klaster klas został zintegrowany w postaci podsystemu, cały system może być traktowany na wyższym poziomie abstrakcji jako zestaw podsystemów. To podejście pozwala nam na koncepcyjne traktowanie setek klas jako systemu, który składa się z grupy podsystemów, z których każdy może być poddawany osobnym testom jednostkowym. Wymagane przypadki testowe mogą się wywodzić bezpośrednio z przypadków testów jednostkowych i integracyjnych używanych do testowania widocznych na zewnątrz metod interfejsu API podsystemu. Tak jak w przypadku konwencjonalnej integracji klas, podsystemy same mogą być integrowane, aby tworzyły nowe, coraz bardziej złożone podsystemy lub kompletny system, a wymagane przypadki testów integracyjnych mogą się wywodzić i być implementowane w taki sam sposób, jak w przypadku prostych klas.

5.3.3 Integracja systemów

Gdy już wszystkie podsystemy zostaną z powodzeniem zintegrowane i przetestowane, ostatnim krokiem w procesie integracji jest integracja systemów. W przeciwieństwie do poprzednich kroków ten krok obejmuje sprawdzenie, czy gotowy produkt wykorzystuje zgodnie z planem

dane interfejsy do komunikacji z otaczającym środowiskiem. Testy te są przeprowadzane w dużej mierze jako część testowania systemowego (zobacz rozdział 6) wewnątrz środowiska systemu, a wiele projektów nie rozróżnia pomiędzy przypadkami testów systemowych i przypadkami testów integracyjnych systemów.

Integracja systemu informatycznego z komputerami klienta lub jakimś innym urządzeniem niestandardowym może być niezwykle trudna, ponieważ ciężko jest wystarczająco dobrze symulować docelowy sprzęt w środowisku testów systemowych. Chociaż ten krok teoretycznie obejmuje jedynie instalowanie oprogramowania (proces, w którym priorytet mają parametryzacja i akceptacja systemu), często w istocie oznacza przeprowadzenie (dodatkowego) testu systemowego w środowisku funkcjonalnym. Jest to poważna przeszkoda dla zespołu Scrum, jako że planowany, gotowy produkt nie powinien wymagać po zainstalowaniu u klienta niczego oprócz prostej parametryzacji.

Jeśli produkt ujawnia niewykryte usterki po zainstalowaniu, zespół będzie musiał zainwestować w rozszerzenie środowiska testowego i/lub symulatory. Jeśli kroki te nie zostaną podjęte, to zamiast otrzymywać bezusterkowe oprogramowanie, klient będzie regularnie otrzymywał oprogramowanie, które na pewno będzie zawierało jakieś usterki i z punktu widzenia klienta użycie metodyki Scrum okaże się stratą czasu.

Studium przypadku 5-4

Studium przypadku 5-4 sterownika eHome: integracja systemów

a) Udana integracja systemów

System eHome jest instalowany przez technika u klienta. Po zainstalowaniu technik parametryzuje system tak, aby urządzenia domowe były wszystkie poprawnie identyfikowane w systemie. Technik następnie uruchamia program diagnostyczny zawarty w pakiecie i po jego udanym zakończeniu demonstruje klientowi funkcjonalność przełączników i sterowania wszystkimi urządzeniami przy użyciu smartfona.

b) Nieudana integracja systemów

Klient wykorzystuje system magistrali, który jest nowy dla zespołu eHome, a adapter magistrali jest napisany według dostarczonej specyfikacji. Jednakże nowy system magistrali nie był dostępny podczas wewnętrznych testów systemowych producenta, ale system mimo to jest dostarczany do klienta, który ostatecznie anuluje zakup, ponieważ technik nie może zmusić systemu do pracy.

5.4 Tradycyjne strategie integracji

Tradycyjnie zarządzane projekty zwykle przeprowadzają integrację i testowanie integracyjne po testowaniu jednostkowym (zobacz rys. 2-3). Założenie jest takie, żeby wszystkie składniki oprogramowania zostały już zaimplementowane i przeszły testy jednostkowe, zanim zacznie się faza integracji. Zespół integracyjny bierze następnie podzestawy zgrupowanych składników, instaluje je w środowisku integracyjnym i testuje. Jeśli testy kończą się powodzeniem, dany podsystem jest klasyfikowany jako zintegrowany. Jeśli testy kończą się niepowodzeniem, wadliwy składnik musi zostać poprawiony.

Chociaż architektura systemu określa, które składniki należą do którego podsystemu, teoretycznie możliwe jest dowolne zorganizowanie sekwencji integracyjnej. Według [Spillner/Linz 14] może to się odbywać przy użyciu jednej z następujących podstawowych strategii:

- **Integracja z góry na dół:** Integracja zaczyna się od składnika, który wywołuje inne składniki, ale sam jest wywoływany tylko przez system operacyjny.
- **Integracja z dołu do góry:** Testowanie zaczyna się od podstawowych składników, które nie wywołują żadnych innych składników (poza funkcjami systemu operacyjnego).
- **Integracja ad-hoc:** Składniki są integrowane wtedy, gdy są kończone przez programistów.

Czyste techniki integracji z góry na dół i z dołu do góry wymagają zaprojektowania architektury systemowej na podstawie diagramu drzewa. Im bardziej zagmatwana jest podstawowa architektura, tym bardziej trzeba będzie odejść od bazowej strategii integracji. Dodatkowo – co ma miejsce w przypadku projektów zarządzanych przy użyciu modelu V – rzadko zdarza się, aby wszystkie składniki były gotowe i przetestowane jednostkowo, więc integracja często zaczyna się, gdy niektóre składniki są jeszcze w trakcie implementowania. To nieuchronnie prowadzi do wykorzystania strategii integracji ad-hoc.

5.5 Ciągła integracja

Projekty oparte na Scrum stale produkują nowe lub zmienione składniki kodu, więc musimy rozważyć, jak sobie z nimi radzić. Strategia oparta na tradycyjnych metodach wymaga, aby zespół zaczekał, aż wszystkie zadania związane z kodem będą wykonane, a następnie instalował wszystkie zmiany dokonane przez programistów w środowisku testowym w celu przeprowadzenia testowania integracyjnego na koniec sprintu. Jest to jednak zaprzeczenie podstawowych założeń Scrum:

- Informacja zwrotna dla programistów pochodząca z testów integracyjnych jest niepotrzebnie opóźniona. W najgorszym możliwym przypadku programista, który zmienia jakiś kod pierwszego dnia sprintu, musi zaczekać do ostatniego dnia sprintu, aby otrzymać wnioski z testu integracyjnego.
- Ponieważ testy integracyjne mogą ujawnić usterki, zespół musi przewidzieć w swoim planie czas na ich naprawienie. To wymusza dopasowanie sprintu do ścisłej sekwencji kodowania, testowania jednostkowego, testowania integracyjnego i naprawiania usterek.

Wynikiem jest „kaskadowy Scrum" – fazowo zorientowane podejście do sprintów. Żeby tego uniknąć, zespół musi skorzystać z lepszej strategii integracyjnej, mianowicie: ciągłej integracji.

Ciągła integracja jest następnym logicznym krokiem w rozwoju przyrostowej strategii integracyjnej. Integracja przyrostowa oznacza, że każdy element kodu jest instalowany w środowisku integracyjnym i jest integrowany od razu po ukończeniu. Nowe składniki nie są więc grupowane z innymi, które nie zostały jeszcze zintegrowane, ale są integrowane zamiast wcześniej zintegrowanych wersji wewnątrz centralnego środowiska integracyjnego.

5.5.1 Proces ciągłej integracji

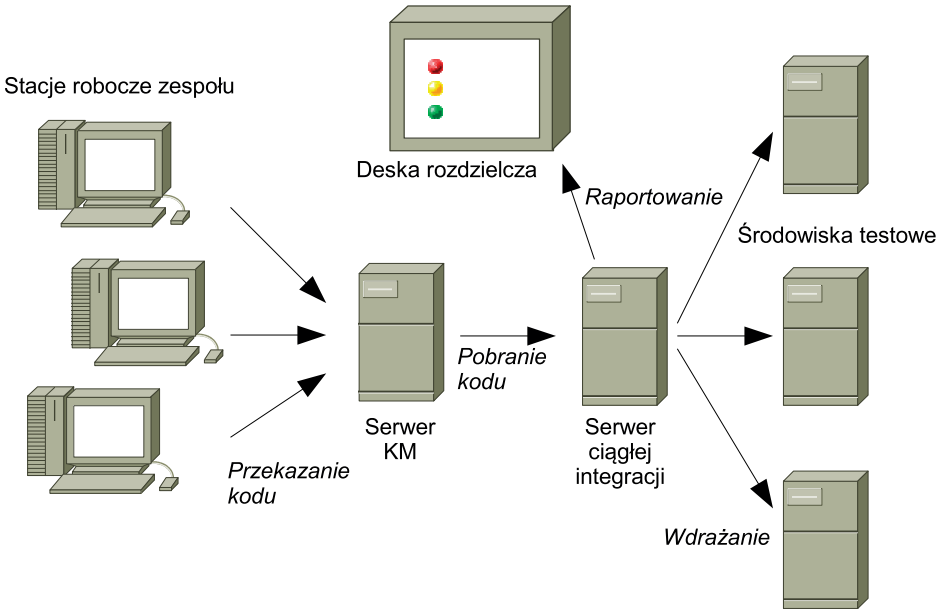
Oprócz centralnego środowiska integracyjnego, innym najważniejszym aspektem ciągłej integracji jest to, że jest ona w pełni zautomatyzowana. Według [Duvall et al. 07], proces ciągłej integracji składa się z następujących kroków i elementów:

- **Centralne repozytorium kodu:** Zespół zarządza kodem programu i zautomatyzowanymi testami we wspólnym, centralnym repozytorium kodu. Zachowywane są poszczególne wersje kodu, a każdy programista musi oddawać swój kod do repozytorium albo codziennie, albo jeszcze lepiej po każdej zmianie (tzn. tak często, jak to możliwe).
- **Zautomatyzowane uruchamianie integracji:** Zwrócenie kodu do repozytorium automatycznie wywołuje proces integracji, który odbywa się na dedykowanym serwerze ciągłej integracji i składa się z następujących kroków:
  - **Kompilacja** Kod jest kompilowany, a ostrzeżenia kompilatora i komunikaty o błędach są rejestrowane na serwerze ciągłej integracji. Kompilacja zwykle zajmuje kilka sekund.
  - **Statyczna analiza kodu** Po udanej kompilacji następuje statyczna analiza kodu, która sprawdza, czy zespół trzymał się swoich własnych wytycznych dotyczących kodowania i miar

jakościowych. Te wyniki również są rejestrowane na serwerze ciągłej integracji. To też zwykle zajmuje kilka sekund. Ważne, aby zespół korzystał ze wskazówek dotyczących kodowania, które mogą być sprawdzane automatycznie. Nie należy nakreślać żadnych wskazówek dla atrybutów kodu, które nie mogą być sprawdzane automatycznie albo takich, które zespół uważa za nieistotne.

- **Wdrażanie w środowisku testowym** Po skompilowaniu i sprawdzeniu statycznym kodu może on zostać wdrożony i zainstalowany w odpowiednim środowisku testowym, które jest najpierw przywracane do wstępnie zdefiniowanego stanu.
- **Inicjacja** Wszystkie kroki inicjujące – takie jak tworzenie i wypełnianie tabel bazodanowych – odbywają się automatycznie. Dane testowe wymagane przez późniejsze testy również są importowane na tym etapie.
- **Testowanie jednostkowe** Następują zautomatyzowane testy jednostkowe dla wszystkich jednostek. Wyniki są rejestrowane na serwerze ciągłej integracji.
- **Testowanie integracyjne** Następne w kolejności są zautomatyzowane testy integracyjne. Wyniki są rejestrowane na serwerze ciągłej integracji.
- **Testowanie systemowe** Na koniec uruchamiane są zautomatyzowane testy systemowe. Te wyniki również są rejestrowane na serwerze ciągłej integracji. Testy systemowe mogą trwać nawet do kilku godzin i dlatego nie są częścią każdego przebiegu ciągłej integracji. Zwykle są uruchamiane raz dziennie, najczęściej w nocy.
- **Informacje zwrotne i tablica rozdzielcza** Serwer ciągłej integracji wyświetla wszystkie wyniki na centralnej „desce rozdzielczej" dostępnej przez przeglądarkę, gdzie można znaleźć natychmiastowe informacje zwrotne w przypadku niepowodzenia testów lub innych problemów. Informacje zwrotne są też aktywnie przekazywane do zainteresowanych programistów poprzez pocztę elektroniczną lub inne środki.

Rysunek 5-4 (za [Duvall et al. 07], z dodatkowymi środowiskami testowymi) pokazuje schemat typowego środowiska ciągłej integracji.



Rys. 5–4
Środowisko ciągłej integracji

Deska rozdzielcza ciągłej integracji

Wyniki każdego kroku są wyświetlane na desce rozdzielczej na bieżąco, a nie na końcu całego przebiegu ciągłej integracji. Szybkość pętli zwrotnej zależy od tego, jak długo trwają poszczególne kroki ciągłej integracji. Kompilacja i analiza statyczna zwykle trwają do kilku minut, natomiast testowanie jednostkowe i integracyjne zwykle może zajmować nieco więcej minut. Z kolei testowanie systemowe zazwyczaj zajmuje kilka godzin. Serwer ciągłej integracji rejestruje wszystkie wyniki testów i wyświetla je w streszczonej formie na desce rozdzielczej, informując cały zespół o stanie każdego przebiegu ciągłej integracji i ogólnej jakości systemu. Rysunek 5-5 pokazuje deskę rozdzielczą wykorzystywaną przez zespół test*Bench* (zobacz studium przypadku 8.2).

Rys. 5–5
Deska rozdzielcza ciągłej integracji pokazująca typowe wyniki

Test Statistics Grid

Job ↓	Success #	%	Failed #	%	Skipped #	%	Total #
iTB Integration Tests	970	100%	0	0%	0	0%	970
iTB Packaging	0	0%	0	0%	0	0%	0
iTB REST Tests	18	100%	0	0%	0	0%	18
iTB Static Analysis	0	0%	0	0%	0	0%	0
iTB_Nightly_2	662	<100%	2	>0%	0	0%	664
iTB_Utilities	1320	100%	0	0%	0	0%	1320
iTEP Export Plugin	5	100%	0	0%	0	0%	5
iTEP4	548	100%	0	0%	0	0%	548
iTORX	5	100%	0	0%	0	0%	5
Word Reporting	163	98%	0	0%	3	2%	166
Total	**3691**	**<100%**	**2**	**>0%**	**3**	**>0%**	**3696**

Test suites overview

Failed (0%) Passed (100%) Skipped (0%)

iTB-unit-tests	0	997	0	997	100%
iTB-integration-tests_ORACLE	0	2936	0	2936	100%
iTB-integration-tests_MSSQL	0	2936	0	2936	100%
iTEP tests	0	548	0	548	100%

5.5.2 Implementowanie ciągłej integracji

Technicznie mówiąc, ciągła integracja nie jest rewolucją. Niektórzy programiści „od zawsze" automatyzowali przebiegi kompilacji przy użyciu skryptów, a większość projektów wykorzystuje systemy zarządzania konfiguracją do zarządzania kodem źródłowym. W wielu projektach również wykonywane są zautomatyzowane procesy conocnych kompilacji, a jeśli menedżer projektu poważnie traktuje jakość, przeprowadza też statyczną analizę tego kodu.

Jeśli zespół już pracuje w ten sposób, do przejścia na w pełni ciągłą integrację konieczne jest tylko zautomatyzowanie wszystkich testów i zebranie ich w procesie ciągłej integracji. Z kolei zespół, który pracuje zgodnie z tylko kilkoma (lub nie pracuje z żadnymi) wskazówkami wymienionymi powyżej, stanie przed dużym wyzwaniem. Z powodów wymienionych na początku podrozdziału 5.5 ignorowanie ciągłej integracji lub opóźnianie jej wprowadzenia nie jest dobrym pomysłem.

Patrząc pod innym kątem, zarządzanie konfiguracją w wielu projektach mogłoby zostać poprawione, a procesy mogłyby być bardziej zautomatyzowane, więc wprowadzenie Scrum stanowi dla zespołu rzeczywistą okazję do poprawienia swojego procesu integracji i jakości pracy.

Przygotowanie środowiska ciągłej integracji przed pierwszym sprintem

Wprowadzenie ciągłej integracji może wiązać się z całkowitą reorganizacją repozytorium oprogramowania zespołu, więc środowisko ciągłej integracji musi być przygotowane przed rozpoczęciem pierwszego sprintu. Ma to zastosowanie wobec wszystkich narzędzi i może też wpływać na strukturę systemu plików dla kodu źródłowego i powiązanych testów. Bez wątpienia trzeba też napisać i przetestować skrypty używane do automatyzowania procesu ciągłej integracji. Jak tylko rozpocznie się pierwszy sprint, a programiści zaczną pracować nad wyznaczonymi zadaniami, system musi być gotowy na pobieranie, zmienianie, odbieranie i testowanie kodu. Zalecamy poniższe kroki podczas budowania środowiska ciągłej integracji:

- **Sprawdzanie bieżącego stanu systemu zarządzania konfiguracją** Które narzędzie do zarządzania konfiguracją jest aktualnie w użyciu? Jak szeroko jest zaimplementowane? Czy są członkowie zespołu lub podprojekty używające innego narzędzia (lub nie używające żadnego)? Jeśli tak, to dlaczego? Jak członkowie zespołu radzą sobie z obsługą narzędzia? Jest proste? Skomplikowane? Jak szybko można skompilować program? Gdzie lub na którym komputerze narzędzie jest uruchamiane? Czy jest zarządzane centralnie, czy zespół steruje narzędziem i repozytorium? Czy są dostępne alternatywne (lub lepsze) narzędzia? Odpowiedzi na wszystkie te pytania można szybko i łatwo znaleźć na warsztatach zespołu,

podobnie jak listę wszystkich elementów systemu zarządzania konfiguracją, które można by poprawić, aby działał szybciej, prościej i bardziej niezawodnie.

- **Poznawanie narzędzi ciągłej integracji** Sprawdzając system zarządzania konfiguracją, zespół może też zacząć przyglądać się dostępnemu oprogramowaniu serwera ciągłej integracji (dobrym punktem wyjścia jest [URL: Testtoolreview]). Serwer ciągłej integracji jest narzędziem informatycznym sterowanym skryptami, które obsługuje poszczególne kroki związane z procesem ciągłej integracji – na przykład uruchamianie i monitorowanie przebiegu kompilatora albo tworzenie pakietu testowego i rejestrowanie wyników. Doświadczeni programiści mogą oczywiście zbudować swoje własne narzędzia, korzystając ze skryptów, choć oprogramowanie dzisiejszych serwerów ciągłej integracji zapewnia gotowe skrypty dla wszystkich ważnych kroków, a deska rozdzielcza oparta na przeglądarce udostępnia wszystkie wyniki. Najlepszym podejściem jest przeprowadzenie warsztatów, na których członkowie zespołu mogą dzielić się wynikami swoich badań i demonstrować próbne instalacje znalezionych narzędzi.

Sprawdzanie bieżącego systemu i wymiana informacji wewnątrz zespołu jest najlepszym sposobem na zdobycie poparcia dla nowego środowiska ciągłej integracji. Mistrz Scrum musi również brać udział w tym procesie, ponieważ nieodpowiednie środowisko ciągłej integracji może być poważnym utrudnieniem. Zbudowanie świetnego środowiska ciągłej integracji jest jednym z pierwszych prawdziwych testów zaangażowania nowego mistrza Scrum. Implementacja systemu ciągłej integracji obejmuje następujące kroki:

- **Wybór narzędzi do zarządzania konfiguracją i ciągłej integracji** Jeśli zespół postanowi wymienić aktualne narzędzie do zarządzania konfiguracją, nowe narzędzie musi dobrze współpracować z wybranym oprogramowaniem serwera ciągłej integracji. Serwer ciągłej integracji musi niezawodnie identyfikować zmiany w kodzie wewnątrz systemu zarządzania konfiguracją tak, aby można go było pobierać w celu przeprowadzenia przebiegu ciągłej integracji.
- **Instalowanie narzędzi do zarządzania konfiguracją i ciągłej integracji** Wybrane narzędzia powinny być zainstalowane na osobnym komputerze. Nie ma sensu instalować narzędzi do zarządzania konfiguracją i ciągłej integracji na komputerze programisty. Należy upewnić się, że wszystkie części systemu są zabezpieczone odpowiednimi systemami kopii zapasowej.

- **Migracja zarządzania konfiguracją** Stare repozytorium kodu musi zostać zaimportowane do nowego systemu. Wszelkie wymagane zmiany w systemie plików mogą wiązać się z dużym nakładem pracy i są podatne na błędy.
- **Tworzenie skryptów ciągłej integracji** Skrypty ciągłej integracji dostarczane z oprogramowaniem muszą zostać zaadaptowane do pracy w środowisku danego projektu. Należy zacząć od skryptów, które monitorują zmiany w repozytorium kodu przed wydaniem kodu kompilatorowi.

W tym momencie zespół ma minimalną funkcjonalność, od której może zacząć. Programiści mogą zdawać kod do centralnego repozytorium, a system automatycznie tworzy nowy przebieg budowania programu. Sprawdzana jest kompatybilność nowego kodu (tzn., czy jest kompletny i składniowo poprawny). Na tym etapie system jest podobny do konwencjonalnego systemu zarządzania konfiguracją, ale nie zawiera jeszcze żadnych testów zautomatyzowanych.

Odpowiednie pakiety zautomatyzowanych testów muszą zostać teraz zaadaptowane i zainstalowane. Zwykle odbywa się to stopniowo w formie określonych zadań testowych planowanych w trakcie następnych sprintów. Poniższe punkty opisują stopień automatyzacji, do którego należy dążyć.

- Każde budowanie natychmiast podlega automatycznym testom jednostkowym. Jest to realistyczny cel, jeśli zespół jest zdyscyplinowany w korzystaniu z programowania sterowanego testami (zobacz rozdział 4). Jeśli zespół już utworzył zautomatyzowane testy jednostkowe, mogą być one zaimportowane do nowego systemu ciągłej integracji. Wersje testowego kodu źródłowego i danych testowych muszą być zarządzane w ramach systemu zarządzania konfiguracją.
- Dodawane są testy integracyjne. Czasami przydatne może być przeniesienie przypadków testów integracyjnych, które są częścią istniejących pakietów testów jednostkowych do oddzielnego pakietu testów integracyjnych. Od tego momentu zespół musi brać pod uwagę typ środowiska wymaganego przez testy integracyjne i czy ma sens przeprowadzanie testów jednostkowych w prostszym, szybszym środowisku. Jeśli zespół postanowi rozdzielić środowiska testowe, będzie musiał nakreślić zadania stworzenia nowych, bardziej skomplikowanych skryptów i skonfigurowania nowego środowiska.
- Testy systemowe mogą też być w znacznej mierze zautomatyzowane i zawarte w systemie ciągłej integracji, zakładając, że zespół rozszerzy zastosowanie zasad sterowania testami (zobacz rozdział 6)

poza testowanie jednostkowe i integracyjne tak, aby obejmowało też testowanie systemowe.

Środowisko ciągłej integracji nie może być zbudowane w biegu.

Skonfigurowanie środowiska ciągłej integracji jest skomplikowanym zadaniem. Ciągłą integrację trzeba dokładnie zaplanować (studium przypadku 8.2 zapewnia szczegółowy przykład), a faza konfiguracji może zająć kilka tygodni, podczas których nie można wykonywać normalnych działań związanych z rozwijaniem produktu.

Nawet jeśli środowisko ciągłej integracji działa, to nie ma gwarancji, że będzie dawać bezusterkowe kompilacje (zobacz [Pichler/Roock 11, podrozdział 4.10]). Jeśli na przykład nie przeprowadzano statycznej analizy kodu przed przejściem na ciągłą integrację, to pierwszych kilka analiz będzie pełnych ostrzeżeń lub komunikatów o błędach, choć składniki i fragmenty kodu zostały już zakwalifikowane przez zespół jako stabilne i gotowe. W takich przypadkach zespół będzie musiał albo złagodzić swoje reguły analizy kodu, albo poprawić cały kod. Ze względu na nagły wzrost pokrycia testami, to samo może się zdarzyć, gdy testy jednostkowe i integracyjne zostaną uruchomione po raz pierwszy. Nawet jeśli te same testy już były uruchamiane z powodzeniem na komputerach poszczególnych programistów, to generowane przez nie wyniki mogą się znacznie różnić, gdy będą wykonywane w centralnie zarządzanym środowisku ciągłej integracji.

Podczas implementowania ciągłej integracji zespół będzie wydawał się na zewnątrz mniej wydajny, niż przed wprowadzeniem Scrum. To doświadczenie może prowadzić do rozczarowań ze strony niektórych członków zespołu. Zadaniem mistrza Scrum jest chronienie zespołu przed nieuzasadnioną krytyką i wspieranie go podczas organizowania nowego środowiska.

5.5.3 Optymalizowanie ciągłej integracji

Przebieg ciągłej integracji musi przetestować system możliwie jak najdokładniej w możliwie jak najkrótszym okresie czasu. Aby wyrównać te na pozór sprzeczne wymagania, zespół może:

- **Tworzyć pakiety testów** Rozróżnienie pomiędzy analizą statyczną a testami jednostkowymi, integracyjnymi i systemowymi dzieli testy na szerokie kategorie, które w razie potrzeby można podzielić na mniejsze pakiety. Mniejsze pakiety testów nie tylko gwarantują, że przebieg testowy zatrzyma się wcześniej w razie niepowodzenia, ale też pozwalają nam włączać je lub wyłączać w zależności od potrzeb aktualnej sytuacji. Na przykład, jeśli dołączymy tylko te pakiety, które dotyczą kodu zmienionego od poprzedniego przebiegu, możemy zaoszczędzić sporo czasu podczas

testowania (zwłaszcza testowania systemowego). Jednakże zapłatą za krótszy czas uruchomienia jest zwiększone ryzyko pominięcia zmienionego kodu i wymaganego zestawu testów.

- **Zatrzymywać przebieg testowy** Trzeba ustalić, czy kontynuować wykonywanie pakietu (lub pakietów) testów, gdy pojedynczy test się nie powiedzie. W wielu przypadkach kolejne testy wymagają określonego zachowania obiektu, jeśli w ogóle mają działać, a nieudany test może to wykluczać. Zatrzymany przebieg jest oczywiście krótszy i oznacza, że pełne możliwości serwera ciągłej integracji będą dostępne wcześniej do przeprowadzenia następnej próby. Z drugiej strony, brak danych wyjściowych z ostatnich testów może utrudniać debugowanie i wykrywanie usterek. Możemy też zakładać, że kolejne testy mogłyby wykryć inne usterki, które można by też od razu poprawić. Skrypty sterujące procesem ciągłej integracji muszą zawierać spójną strategię zatrzymywania/uruchamiania testów uzgodnioną z góry przez zespół.
- **Uruchamiać testy równolegle** Innym sposobem na przyspieszenie procesu testowania jest uruchamianie pakietów testów równolegle, co daje w efekcie przebieg ciągłej integracji trwający tylko tak długo, ile zajmuje wykonanie największego pakietu. Jednakże uruchamianie testów równolegle może być dość skomplikowane z wielu powodów:
  - Równoległe pakiety testów muszą być od siebie niezależne. Innymi słowy żaden test zawarty w pakiecie B nie może polegać na obiektach, na które ma wpływ wykonanie pakietu A. To wymaga ostrożnego skonstruowania pakietów testowych i zawarcia odpowiednio modułowych przypadków testowych w każdym pakiecie.
  - Testowany system musi pozwalać na dostęp równoległy i przetwarzanie równoległe w interfejsie testowym. Jeśli tak nie jest, test będzie wykonywany szeregowo, albo się nie powiedzie. Same testy muszą być napisane w taki sposób, który umożliwi wykonywanie ich w równoległym środowisku testowym. W przeciwnym razie wyniki testów mogą wzajemnie nachodzić na siebie i nie będzie możliwe ustalenie, które wpisy w rejestrze zostały utworzone przez które testy.
- **Utworzyć równoległe środowisko testowe** Zamiast uruchamiać wiele testów na pojedynczym obiekcie, zwykle lepiej jest zainstalować obiekt wiele razy (na przykład na wielu maszynach wirtualnych) i rozdystrybuować pakiety testów pomiędzy różnymi środowiskami. Taki system mógłby na przykład wykonywać testy jednostkowe i proste testy integracyjne w środowisku nr 1,

skomplikowane testy integracyjne w środowisku nr 2, a testy systemowe w środowisku nr 3. W ten sposób każde środowisko może być uruchamiane przy użyciu sprzętu i oprogramowania przeznaczonego dla danego, szczególnego zadania. Może to zwiększyć wymagany budżet na sprzęt, ale sprawia, że testowanie jest dużo bardziej wydajne. Na przykład wszelkie systemy wykorzystujące bazy danych wymagają, aby te bazy danych i wszystkie dane testowe były zainstalowane na potrzeby testowania systemowego. Skonfigurowanie takiej testowej bazy danych wymaga wiele wysiłku i daje w wyniku dość wolny dostęp do danych, więc testy niskiego poziomu są często wykonywane przy użyciu prostej, zastępczej bazy danych znajdującej się w pamięci RAM systemu. Taka konfiguracja jest łatwiejsza w instalacji i działa szybciej.

- **Zaktualizować sprzęt testowy** Wykorzystanie większej ilości sprzętu lub szybszego sprzętu jest oczywistym sposobem na przyspieszenie procesu ciągłej integracji. Niestety ten prosty krok nie może być przyjęty za pewnik, a zamiast wydawać kilka tysięcy złotych na szybszy sprzęt, wiele firm woli zatrudnić członka innego zespołu na tydzień do przeprowadzenia zadań, takich jak optymalizacja środowiska skryptowego.

Powolny proces ciągłej integracji jest poważną przeszkodą. Jeśli system działa powoli, kuszące jest ukrywanie pakietów testów lub nawet unikanie przekazywania nowego kodu dla przyspieszenia procesu. Jeśli praca zespołu cierpi wskutek czegoś tak banalnego, jak korzystanie z powolnego sprzętu, to mistrz Scrum musi zająć się usunięciem tych braków.

Stała optymalizacja środowiska ciągłej integracji

Utrzymywanie i optymalizowanie środowiska ciągłej integracji oraz ograniczanie czasu testowania to stałe wyzwania. Sprawy optymalizacji ciągłej integracji powinny być regularną częścią retrospektyw zespołu i muszą być uwzględniane w planowaniu sprintu. Dla każdego sprintu mistrz Scrum i właściciel produktu muszą ustalić, jak najlepiej podzielić zasoby pomiędzy krótkoterminową wydajność, a średnioterminowe usprawnienia systemu ciągłej integracji, które pomogą zwiększyć ogólną wydajność zespołu.

5.6 Zarządzanie testami integracyjnymi

Najważniejszym aspektem zarządzania testami integracyjnymi jest zapewnienie, żeby wystarczająca liczba odpowiednich przypadków testowych została napisana i uruchomiona. Podobieństwa pomiędzy testami jednostkowymi i integracyjnymi czasem utrudniają programistom ustalenie, jakich testów integracyjnych brakuje i jakie trzeba dodać.

Przypadki testów integracyjnych muszą być nastawione na podstawową architekturę systemu tak, aby testerzy mieli jasność co do struktury planowanej architektury. Z drugiej strony testy integracyjne sprawdzają stopień, w jakim faktyczna architektura reprezentuje planowaną. Testy integracyjne mogą i powinny być projektowane przy zastosowaniu zasady sterowania testami i są przydatnym narzędziem określania architektury systemowej i sprawdzania jej rozwoju podczas kolejnych sprintów. Menedżer testów w zespole może korzystać z programowania sterowanego testami w celu posuwania procesu rozwoju architektury do przodu i może też korzystać z wyników testów integracyjnych, żeby pomagać zespołowi w aktualizowaniu architektury systemu – na przykład przez upraszczanie interfejsów, definiowanie podsystemów, rozpoznawanie i radzenie sobie z problemami wydajnościowymi, itd. Wyniki omawiania takich kwestii będą stanowić zbiorową opinię zespołu na temat architektury docelowego systemu i muszą być ponownie regularnie opracowywane – albo graficznie w postaci diagramów architektonicznych, albo jako zautomatyzowane przypadki testów integracyjnych. Jeśli zespół nie będzie podążał tym szlakiem, wynikiem będzie system zawierający sporo klas o losowej strukturze.

W środowisku Scrum strategia integracji jest w dużej mierze definiowana przez mapę scenariuszy (zobacz rozdział 3), a w przypadku poszczególnych sprintów przez listę zaległości sprintu określającą, które funkcje mają być implementowane podczas którego sprintu, a więc kiedy będą kończone poszczególne składniki. Ponieważ każdy składnik jest natychmiast (lub stale) integrowany z systemem, planowanie sprintu ustala też z góry sekwencję testów integracyjnych. Dlatego ważne jest wzięcie pod uwagę sekwencji integracyjnej, którą dyktuje mapa scenariuszy podczas ustalania, kto powinien zająć się którym zadaniem podczas sprintu. Jeśli wybór jest możliwy, należy zawsze wybierać sekwencję dla nadchodzących zadań, które pociągają za sobą najmniejsze możliwe nakłady integracyjne.

Podczas planowania sprintu trzeba brać pod uwagę nakłady na testowanie integracyjne.

Nakłady związane z projektowaniem, automatyzacją i aktualizowaniem testów integracyjnych muszą być brane pod uwagę przez menedżera testów podczas planowania sprintu. Wymagane nakłady nie skalują się bezpośrednio w relacji do liczby zmienionych lub nowych składników, ale raczej do liczby zależności pomiędzy nimi i mogą być nieproporcjonalnie wysokie. Konfigurowanie środowiska testowego i opracowywanie obiektów zastępczych również wymaga sporo dodatkowego wysiłku. Cały ten wysiłek jest jednak konieczny dla zapewnienia, żeby proces integracyjny dla każdego składnika mógł być automatycznie i stale wykonywany po każdej zmianie w kodzie.

Podczas procesu ciągłej integracji menedżer testów musi również sprawdzać, czy możliwa jest dodatkowa analiza kodu związana

z integracją – na przykład w celu statycznego sprawdzenia spójności interfejsów i ich zgodności z abstrakcyjnymi specyfikacjami[5] albo sprawdzenia, czy żądane pliki i komunikaty są dostępne w odpowiednich formatach[6]. Trzeba też zwrócić uwagę na podział zautomatyzowanych testów integracyjnych na pakiety i stałą optymalizację szybkości procesu ciągłej integracji.

5.7 Pytania i ćwiczenia

5.7.1 Samoocena

Pytania i ćwiczenia pomagające w ocenie, jak zwinny jest naprawdę projekt lub zespół.

1. Czy zespół korzysta ze zautomatyzowanych testów integracyjnych? Z ilu? Ile z nich jest powiązanych z przypadkami testów jednostkowych?
2. Gdzie i jak jest zdefiniowana i udokumentowana architektura systemu? Czy istniejące przypadki testów integracyjnych sprawdzają, czy ta architektura jest implementowana?
3. Czy istnieje proces przeglądu porównujący przypadki testowe z faktyczną architekturą? Czy ma to miejsce regularnie? Jakie wnioski pozwala to wysnuć?
4. Jakie interfejsy są objęte testami integracyjnymi? Jaki jest stopień pokrycia?
5. Jakie są wartości docelowego pokrycia? Jak często wartości te są mierzone?
6. Które wartości uzyskaliśmy teraz/dzisiaj? Gdzie zespół przechowuje te wartości? Gdzie można je znaleźć, jeśli są potrzebne w danym momencie?
7. Jakie środki zostały wprowadzone, żeby poprawić pokrycie testami i/lub jakość testów integracyjnych?
8. Czy korzystamy z monitorów interfejsów? Dla których interfejsów?
9. Czy mamy wdrożony automatyczny proces ciągłej integracji? Jeśli tak, jaka jest jego struktura?
10. Kiedy są wykonywane testy integracyjne? Czy są częścią środowiska ciągłej integracji?

---

5 W oparciu o formalny interfejs i języki definiowania usług, takie jak IDL [URL: OMG] i WDSL [URL: W3C].

6 Korzystając z narzędzi sprawdzających dla stron HTML lub definicji CSS [URL: W3C validator].

11. W którym środowisku testowym jest przeprowadzane testowanie integracyjne? Czy to środowisko jest odpowiednio zdefiniowane i może być niezawodnie odtworzone?
12. Jak można porównać długość trwania testów integracyjnych do testów jednostkowych?
13. Kiedy są projektowane testy integracyjne? W oparciu o opis architektury systemu? Przed rozpoczęciem pisania kodu (sterowanie testami)? Jak tylko składnik jest ukończony? Czy na końcu sprintu?
14. Czy zadania integracyjne są planowane bezpośrednio jako część listy zaległości sprintu?
15. Czy testy integracyjne są częścią definicji gotowości programu?
16. Jak wyglądają teraz/dzisiaj wyniki istniejących testów integracyjnych?
17. Ile różnych rodzajów usterek ujawniają testy integracyjne w porównaniu z testami jednostkowymi?
18. Jak jest przeprowadzane zarządzanie usterkami i poprawianie błędów, jeśli interfejsy podsystemów napisane przez inne zespoły są wadliwe?
19. Jak są dokumentowane uzgodnienia dotyczące interfejsów międzyzespołowych? Czy te problemy są omawiane podczas wspólnych spotkań Scrum?
20. Jakie inne sprawdzenia związane z architekturą i interfejsami są przeprowadzane przez zespół? Przeglądy architektury? Automatyczna analiza? Sprawdzanie poprawności interfejsów i danych?
21. Czy proces ciągłej integracji i jego wyniki są omawiane podczas codziennego spotkania Scrum?
22. Czy sposoby poprawienia procesu ciągłej integracji są omawiane podczas retrospektyw? Jaki jest aktualny stan tej dyskusji? Jakie środki zostały uzgodnione? Nad którymi toczą się prace podczas bieżącego sprintu?
23. Jeśli wiele zespołów pracuje nad tym samym projektem, to czy wykorzystują wspólny proces ciągłej integracji? Kto jest za to odpowiedzialny? Do kogo „należą" narzędzia i sprzęt ciągłej integracji? Czy te problemy są omawiane podczas wspólnych spotkań Scrum?

5.7.2 Metody i techniki

Te pytania pomogą w podsumowaniu treści bieżącego rozdziału.

1. Jakie są najbardziej typowe rodzaje błędów integracyjnych? Wyjaśnić ich przyczyny i symptomy.

2. Jak można systematycznie tworzyć testy integracyjne? Nazwać i opisać potrzebne kroki.
3. Wyjaśnić podobieństwa i różnice pomiędzy przypadkiem testu jednostkowego i przypadkiem testu integracyjnego.
4. Kiedy dwa składniki oprogramowania są od siebie zależne? Jakie są różnice pomiędzy zależnościami bezpośrednimi i pośrednimi?
5. Wyjaśnić termin „łatwość testowania".
6. Jak i dlaczego łatwość testowania interfejsu wpływa na nakłady związane z odpowiadającymi jej testami integracyjnymi?
7. Dlaczego testy integracyjne są szczególnie ważne w systemach zorientowanych obiektowo?
8. W jaki sposób iteratywne podejście Scrum do projektów zwiększa wysiłek związany z testowaniem integracyjnym?
9. Dlaczego refaktoring architektury systemu może ograniczyć wysiłek związany z testowaniem integracyjnym?
10. Nazwać różne typy integracji klas.
11. Co to jest integracja podsystemów?
12. Co to jest integracja systemów?
13. Wyjaśnić, jak planowanie sprintu może wpływać na sekwencję testów integracyjnych.
14. Które elementy i procesy są częścią procesu ciągłej integracji?
15. „Bez ciągłej integracji nie ma Scrum!" Wyjaśnić to stwierdzenie.
16. Które kroki trzeba wykonać przy implementacji procesu ciągłej integracji?
17. Jakie kroki można podjąć, aby stale optymalizować proces ciągłej integracji?

5.7.3 Inne ćwiczenia

Te ćwiczenia pomogą zagłębić się w zagadnienia poruszone w trakcie tego rozdziału.

1. Wyjaśnić, które typy usterek integracyjnych mogą i nie mogą być wykrywane przy użyciu sprawdzania składni podczas kompilacji.
2. Jakie typy błędów integracyjnych występują tylko w asynchronicznie powiązanych składnikach? Dlaczego?
3. Jaki jest efekt przypadkowego podłączenia składnika asynchronicznego w sposób synchroniczny? Wyjaśnić różnice w zachowaniu przez to spowodowanym, posługując się poleceniami sterującymi

żaluzjami w systemie eHome. Jak mógłby wyglądać test, który sprawdza, czy takie polecenie jest wykonywane asynchronicznie?

4. Składniki oprogramowania mogą być zależne bezpośrednio lub pośrednio. Opisać kilka typowych źródeł zależności pośrednich.
5. Wyjaśnić, jak zależność pośrednia może być przekształcona w zależność bezpośrednią. Podać przykład.
6. Załóżmy, że trzy składniki wykorzystują ten sam plik jako zasób. Wyjaśnić, dlaczego negatywne testy dla tych składników wymagają więcej nakładów, niż gdy plik jest obsługiwany przez centralnie zarządzaną usługę. Dlaczego nie jest to przypadek dla testów pozytywnych?

6 Testowanie systemowe i testowanie non-stop

Oprócz testowania jednostkowego i testowania integracyjnego istotną częścią każdego projektu zwinnego jest testowanie systemowe. Ten rozdział wyjaśnia, czym są testy systemowe i czego wymagają od środowiska testowego. Zbadamy różne momenty podczas sprintu, które nadają się do przeprowadzenia testów systemowych, a także omówimy zwinne aspekty testowania systemowego oraz testowanie badawcze i testowanie akceptacyjne.

6.1 Testowanie systemowe

Scrum ma za zadanie wytworzyć potencjalnie gotowy produkt na końcu każdego sprintu. Taki produkt musi być w stanie działać poza środowiskiem ciągłej integracji, musi mieć interfejs użytkownika i zwykle musi mieć możliwość interakcji z innymi systemami klienta. Testy systemowe sprawdzają, czy system działa z punktu widzenia użytkownika korzystającego z interfejsów klienta. Aby testy systemowe były skuteczne, muszą być przeprowadzane w środowisku emulującym środowisko użytkownika końcowego tak dokładnie, jak to tylko możliwe.

Testowanie systemowe sprawdza aspekty produktu, których nie obejmuje testowanie jednostkowe i integracyjne.

Ani testy jednostkowe, ani testy integracyjne nie są w stanie tego zrobić, więc musimy utworzyć i zastosować testy systemowe, które sprawdzają te aspekty systemu, których nie obejmuje testowanie jednostkowe i integracyjne. Wykorzystajmy nasze studium przypadku jako przykład:

Studium przypadku 6-1

Studium przypadku 6-1 sterownika eHome: Przypadki testów systemowych

Jeśli użytkownik otworzy sterownik eHome w przeglądarce i kliknie przycisk „lampa w kuchni", aby ją włączyć, to oczekujemy, że rzeczywista lampa na suficie się włączy. Odpowiedni test systemowy musi więc gwarantować, że:

1. W oparciu o istniejące kryteria akceptacji system spełnia wymagania nakreślone w liście zaległości.

ciąg dalszy na następnej stronie

2. Cały łańcuch funkcjonalny – od interfejsu użytkownika do fizycznego włączenia urządzenia – musi być przetestowany.

Menedżer testów zespołu opracowuje następujące przypadki testów systemowych (TS) w oparciu o listę zaległości sterownika eHome (zobacz podrozdział 3.3):

TS-1: Sterowanie urządzeniami

TS-1.1: Włączanie/wyłączanie lampy w kuchni. Sprawdzić, czy faktyczne przełączanie następuje i jest wizualizowane w przeglądarce.

TS-1.2: Włączyć lampę w salonie, zaczynając od 50% jasności, następnie podwyższyć ją do 70% i ściemnić do 30%. Sprawdzić każdy krok w urządzeniu odbiorczym i w interfejsie przeglądarki. Ściemnić do 0% i sprawdzić, czy stan urządzenia przełącza się na wyłączony.

TS-1.3: Zamknąć żaluzje w salonie i ponownie je otworzyć. Sprawdzić, czy żaluzje reagują prawidłowo i czy ich działanie jest potwierdzane i wizualizowane w przeglądarce.

TS-2: Programowanie przełączania

TS-2.1: Zaprogramować następujące działania: Uchylanie żaluzji w dni powszednie o 7 rano; podnoszenie wszystkich żaluzji od strony południowej o 8.30 rano; zamykanie wszystkich żaluzji o 8.30 wieczorem.

TS-3: Sprawdzanie danych z czujników

TS-3.1: Pobrać wartość „temperatury w salonie", klikając ikonę odpowiedniego czujnika w przeglądarce.

TS-3.2: Odebrać komunikat „burza" od czujnika wiatru i sprawdzić, czy wykonywane jest związane z nim polecenie „opuść wszystkie żaluzje".

Środowisko testowe musi spełniać pewne warunki wstępne (WAR), jeśli wymienione wyżej testy mają się udać – na przykład:

WAR-1: Urządzenie z przeglądarką musi zlokalizować sterownik eHome w sieci i być w stanie się z nim komunikować.

WAR-2: Sterownik musi przekazywać polecenia generowane przez klikanie ikon przeglądarki do odpowiednich urządzeń. W tym celu sterownik musi wiedzieć, która ikona reprezentuje które urządzenie fizyczne. Odpowiednie dane urządzenia muszą być poprawnie wprowadzone i sparametryzowane w bazie danych sterownika eHome (zobacz studium przypadku 3-2 na stronie 27).

WAR-3: Sterownik eHome musi być w stanie kontaktować się ze wszystkimi urządzeniami poprzez magistralę systemową. Adapter magistrali odpowiadający protokołowi magistrali musi być zainstalowany.

WAR-4: Każde urządzenie musi rozumieć otrzymywane polecenia i na nie reagować. W tym celu każdy element wykonawczy musi być prawidłowo sparametryzowany i poprawnie (fizycznie) podłączony.

WAR-5: Wszystkie urządzenia końcowe muszą być albo poprawnie podłączone do odpowiednich elementów wykonawczych, albo właściwie symulowane przez elementy wyświetlające lub instrumenty pomiarowe.

Testowanie przekrojowe

Nasz przykład jasno pokazuje, że testy systemowe sprawdzają cały łańcuch funkcjonalny. Test systemowy wywołuje przepływ danych przechodzący przez cały system – od interfejsu użytkownika, poprzez magistralę do elementu wykonawczego i z powrotem. Ten typ testu jest często nazywany testem przekrojowym[1]. Miejsca, gdzie sprawdzane i obserwowane są dane testowe (punkty kontroli i obserwacji), reprezentują miejsca, gdzie użytkownik końcowy zwraca uwagę na funkcjonalność (lub brak funkcjonalności) produktu.

Sprawdzanie zgodności z wymaganiami w celu odkrycia luk w systemie

Przypadki testów systemowych mogą wywodzić się bezpośrednio z wymagań i kryteriów akceptacyjnych wymienionych na liście zaległości produktowych albo z odpowiednich opisów przypadków użycia. Zespół jednak nie powinien traktować tych źródeł jako pełnych i powinien aktywnie szukać luk w dokumentacji i próbować nakreślać nowe warianty i przypadki specjalne, a także dodatkowe przypadki użycia możliwe do przewidzenia.

Dialog z właścicielem produktu powinien służyć wyjaśnieniu, czy i kiedy wymagania wymienione na liście zaległości powinny być dokładniej zdefiniowane (w oparciu o dane od testerów[2]) i czy trzeba nakreślić dodatkowe wymagania i dodać je do listy zaległości.

Studium przypadku 6-2

Studium przypadku 6-2 sterownika eHome: aktualizacja listy zaległości w oparciu o dane zwrotne z testu systemowego

TS-3.2 jest przypadkiem testu systemowego, który nie ma odpowiednika w postaci wymagania na liście zaległości.

Jednakże zespół i właściciel produktu wiedzą, że system obejmuje wiele dodatkowych czujników (między innymi dla wiatru i światła), które wzbudzają alarmy, gdy osiągnięta zostaje wartość graniczna zdefiniowana przez użytkownika. Właściciel systemu eHome oczywiście oczekuje od systemu reakcji na te sytuacje, więc nowe wymaganie zostaje dodane do listy zaległości (zobacz podrozdział 3.3):

Zagadnienie	**Priorytet**	*Opis / Kryteria przyjęcia*
Reakcja na alarmy	2 3	Czujnik może być przypisany do jednego lub kilku programów, które są aktywowane, gdy adapter magistrali otrzyma i odczyta komunikat alarmowy. ❒ *Sygnał „burza" od czujnika wiatru włącza zaprogramowane polecenie „opuść żaluzje".* ❒ *Program polecenia jest uruchamiany najpóźniej w sekundę po otrzymaniu komunikatu alarmowego.*

1 Nie każdy test systemowy musi być zaprojektowany jako test przekrojowy.

2 Często wystarcza odwołanie do odpowiednich przypadków testowych.

6.2 Środowisko testowania systemowego

Testowanie systemowe zwykle wymaga bardziej złożonego środowiska niż testowanie jednostkowe lub integracyjne – ta sytuacja jest dobrze zilustrowana przez różne środowiska testowe eHome opisane tutaj i w poprzednich dwóch rozdziałach. Środowisko testowania systemowego musi realistycznie emulować prawdziwe środowisko klienta i służy potwierdzaniu, że system działa nie tylko w warunkach laboratoryjnych. Im bardziej realistyczne środowisko testowe, tym bardziej prawdopodobne, że bezbłędny, przetestowany systemowo produkt będzie też idealnie działać po zainstalowaniu we własnym środowisku klienta. Natomiast środowisko testowania systemowego, które jest zbyt proste, zwiększa ryzyko niewykrytych usterek, które będą powodować problemy po zainstalowaniu produktu.

Podobieństwo środowiska testowania systemowego do środowiska produkcyjnego

Środowisko testowe musi odtwarzać środowisko produkcyjne tak dokładnie, jak to tylko możliwe. Oznacza to, że składniki zewnętrzne, z którymi produkt będzie współpracować w środowisku użytkownika końcowego, muszą być reprezentowane w środowisku testowym albo przez rzeczywiste składniki, albo przez realistyczne symulacje. Zamiast elementów testowych i zastępczych, w środowisku testowym trzeba zainstalować możliwie jak najwięcej rzeczywistych składników (takich jak sprzęt komputerowy, oprogramowanie systemowe, sterowniki, sieci i inne systemy). Gotowe środowisko testowania systemowego może być niezwykle skomplikowane (i drogie).

Duża liczba składników oznacza też, że jest wiele różnych sposobów, na które można je zainstalować i skonfigurować. Istnieje wiele typowych konfiguracji używanych przez określone grupy użytkowników, dla określonych scenariuszy użycia lub dla pewnych atrybutów związanych z kompatybilnością testowanego produktu.

Studium przypadku 6-3

Studium przypadku 6-3 sterownika eHome: środowisko testów systemowych

Wymaganie „działa w przeglądarce Firefox 15.0 lub późniejszej" wymienione na liście zaległości produktu (zobacz podrozdział 3.3) nie jest objęte przypadkiem testowym, ale samym środowiskiem testowym, gdzie wersja Firefox 15.0 (i wybrane nowsze wersje włącznie z najnowszą) jest zainstalowana na testowym komputerze klienckim.

Wymaganie „wszystkie urządzenia systemu eHome mogą być sterowane poprzez adapter eHome" jest obejmowane przez fakt, że wszystkie aktualnie dostępne urządzenia magistrali eHome są skonfigurowane jako część środowiska testowego i są połączone przez centralną magistralę.

Kompatybilność z urządzeniami firm trzecich jest testowana przy użyciu podobnych konfiguracji, które zawierają wymagane urządzenia.

Nakłady na testowanie systemowe

Nakłady związane z testowaniem systemowym nie skalują się zgodnie z liczbą przypadków testów systemowych, ale raczej z liczbą różnych środowisk testowych, które trzeba sprawdzić. Jaki wpływ ma to na projekty zwinne?

- Lista materiałów i konfiguracja środowiska testowania systemowego muszą być dokładnie zaplanowane i zdefiniowane, a dopiero wtedy wyniki tych testów będą miały znaczenie. Trzeba jednak zachować odpowiedni kompromis pomiędzy przydatnością wyników a kosztem środowiska użytego do ich uzyskania.
- Początkowe skonfigurowanie środowiska testów systemowych może być skomplikowanym, kosztownym i podatnym na błędy przedsięwzięciem. Zadania związane ze skonfigurowaniem środowiska testów systemowych muszą być zawarte we wczesnych sprintach. Jeśli nie będą, zespół ryzykuje, że otrzyma informacje zwrotne z testowania systemowego na zbyt późnym etapie projektu (zobacz podrozdział 6.8.1).
- Trzeba też zarezerwować czas na utrzymywanie tego środowiska w każdym sprincie. Jeśli nie poświęcimy czasu na utrzymywanie środowiska testowego, szybko się zestarzeje i będzie stanowić poważną zawadę. Konfiguracja systemu nie będzie nadążać za rozwojem produktu, a wszelkie nowe konfiguracje wymagane przez nowe funkcje nie będą zaimplementowane. Uzyskiwane wyniki z testów systemowych będą coraz mniej znaczące albo wręcz nie będzie się dało przeprowadzać dalszych testów systemowych.
- Duża liczba heterogenicznych składników utrudnia automatyzację instalacji i konfiguracji środowiska testów systemowych, ale mimo to jest to warte zachodu. Tutaj też trzeba zapewnić planowanie regularnych zadań automatyzujących.
- Mistrz Scrum powinien korzystać z retrospektyw (zobacz podrozdział 7.2.3), żeby wykorzystywać dostępne opcje optymalizowania środowiska testowania systemowego. Jednakże kroki optymalizacyjne niezbędne dla środowisk testowania jednostkowego, integracyjnego i systemowego znacznie się różnią. Środowiska testów jednostkowych i integracyjnych są zaprojektowane pod kątem maksymalnej szybkości, a każdy test sprawdza tylko jeden niewielki element systemu w celowo uproszczonych warunkach (na przykład przesłanie polecenia przełączenia i zapisanie wyniku w pliku). Składniki zewnętrzne, takie jak magistrala eHome lub przełączające elementy końcowe, są zastępowane przy testowaniu jednostkowym obiektami zastępczymi (zobacz rozdział 4). Testy systemowe też powinny być wykonywane możliwie szybko, ale sensowność wyników ma pierwszeństwo przed szybkością ich wytwarzania. Te różnice w celach różnych typów testów sprawiają,

że konieczne jest budowanie dla każdego z nich oddzielnych środowisk testowych.

- Koszt przygotowania środowiska testowania systemowego oznacza, że często musi być ono wspólnie wykorzystywane przez różne zespoły lub jednostki projektowe. Może to utrudnić planowanie zadań i może oznaczać konieczność koordynowania testów z innymi zespołami lub przeprowadzania testów w określonych przedziałach czasowych. Mistrz Scrum jest odpowiedzialny za koordynowanie przedziałów czasowych podczas wspólnych spotkań Scrum [URL: Scrum Guide].

6.3 Ręczne testowanie systemowe

6.3.1 Testowanie badawcze

Testowanie badawcze jest popularnym podejściem zwinnym do testowania i łączy badanie systemu, który ma być testowany z projektowaniem i przeprowadzaniem testów ręcznych (zobacz [Crispin/Gregory 08]). Najważniejsze cechy tego podejścia to:

- Na początku sesji tester definiuje tylko cele testu – tzn. która funkcja (lub scenariusz użytkownika) ma zostać przetestowana. Sesja skupia się na tym pojedynczym aspekcie produktu. Ogólna struktura testu i poszczególne kroki nie są określone z góry.
- Tester następnie próbuje przeprowadzić dany scenariusz lub zbadać funkcjonalność i obserwuje zachowanie systemu.
- Aby upewnić się, czy obiekt testowy działa zgodnie z założeniami, tester wykorzystuje wszystkie dostępne informacje, które są uważane za przydatne i związane z testowanym obiektem. Do informacji tych może należeć karta zadania, która krótko opisuje funkcje lub innego rodzaju informacje. Braki w dokumentacji są równoważone aktywnym badaniem testowanego systemu.
- Struktura testu i ścieżka wybierana w systemie zależy od obserwacji dokonywanych przez testera podczas sesji. Składniki oprogramowania, które zachowują się normalnie albo które są już znane, są pomijane lub testowane szybko. W razie odkrycia miejsc, gdzie oprogramowanie zachowuje się w sposób nieoczekiwany przez testera, testy są rozszerzane w celu znalezienia przyczyn błędnego działania.
- Od razu notowane są obserwacje, zapytania, ostrzeżenia i wnioski dla programisty(-ów).

Idealne do szybkiego sprawdzania nowych funkcji

Tester nie wymaga szczegółowej specyfikacji obiektu testowego albo testu. Test jest tworzony na bieżąco podczas jego wykonywania i skupia się na podejrzanych lub wadliwych składnikach. To podejście idealnie nadaje się do szybkiego przyjrzenia się nowym lub nieznanym funkcjom.

Jakość testu badawczego zależy w znacznym stopniu od stopnia zdyscyplinowania testera, jego poziomu doświadczenia i wyczucia oprogramowania. Takie testy są trudne lub niemożliwe do automatycznego odtwarzania i muszą być przeprowadzane ręcznie. Testerzy ryzykują też zejściem z zaplanowanego kursu, co może objawiać się niskim poziomem wykrywania usterek.

6.3.2 Testowanie oparte na sesjach

Testowanie oparte na sesjach próbuje przeciwdziałać niektórym z wad testowania badawczego. Najważniejszymi cechami tego podejścia są:

- Tester opisuje cele testu i jego strategię w dwóch lub trzech krótkich zdaniach.
- Czas trwania samej sesji testowej jest ograniczony do maksymalnie 90 minut, które są podzielone na fazy przygotowania, projektu i przeprowadzenia testu, lokalizacji usterek oraz zgłaszania usterek.
- Tworzony jest arkusz sesji, w którym odnotowuje się cele testu, szczegóły testowanego systemu, szczegóły procedury, pokrycie testu, krótkie opisy testowanych funkcji i elementów interfejsu, odkryte usterki, pytania bez odpowiedzi, itd. Zawartość wszystkich dostępnych arkuszy sesji może być elektronicznie podsumowywana i analizowana.

Testowanie oparte na sesjach wykorzystuje wszystkie zalety testowania badawczego i umieszcza tę procedurę w sformalizowanych ramach. Powstałe dzięki temu elektroniczne dzienniki testów są prawdziwym dobrodziejstwem, mogą być używane do odtwarzania testów i pozwalają zespołowi analizować wszystkie podobne testy przy użyciu tych samych kryteriów. Korzystanie z zapisu arkuszy sesji opartego na słowach kluczowych pomaga budować solidną podstawę dla długoterminowej automatyzacji (zobacz też podrozdział 6.4: Zautomatyzowane testowanie systemowe).

Testowanie oparte na sesjach jest też przydatnym narzędziem, dzięki któremu właściciel produktu i klient mogą szybko uzyskać wgląd w aktualny stan produktu. Doświadczenie pokazuje, że jeśli tester bierze udział w takich sesjach, to wnioski mogą być lepiej rejestrowane i później odtwarzane.

6.3.3 Testowanie akceptacyjne

Literatura dotycząca programowania zwinnego często odwołuje się jedynie do testowania jednostkowego i testowania akceptacyjnego, co stwarza wrażenie, że zautomatyzowane testy jednostkowe oraz ręczne, badawcze testy akceptacyjne czynią testy systemowe (znane użytkownikom modelu V) zbędnymi. Niektórzy autorzy ([Crispin/Gregory 08], na przykład[3]) stosują terminy „testowanie akceptacyjne" i „testowanie systemowe" zamiennie. Ale nie każdy przypadek testu akceptacyjnego jest odpowiednikiem przypadku testu systemowego, a przypadki testów jednostkowych i testów integracyjnych mogą być stosowane jako część pakietu testów akceptacyjnych.

Czy zbudowaliśmy właściwy system?

Słowniczek ISTQB [URL: ISTQB Glossary] definiuje test akceptacyjny jako test, który „…umożliwia użytkownikowi, klientowi lub innemu uprawnionemu podmiotowi podjęcie decyzji o akceptacji (lub nie) systemu". Testy akceptacyjne mogą więc być uważane za testy zatwierdzające, które klient lub przedstawiciel klienta (w Scrum – właściciel produktu) wybiera i przeprowadza w celu zatwierdzenia dostarczonego produktu. Celem jest sprawdzenie, czy produkt spełnia zamierzone cele i zapewnia właściwą funkcjonalność („czy zbudowaliśmy właściwy system?"). W Scrum kryteria akceptacyjne dla każdej funkcji są uzgadniane przez zespół i właściciela produktu i są rejestrowane na liście zaległości. Przełożenie tych kryteriów na odpowiednie (ręczne lub zautomatyzowane) przypadki testowe odbywa się podczas sprintu.

Czy prawidłowo zbudowaliśmy system?

Ponieważ właściciel produktu reprezentuje klienta w zespole Scrum, testy akceptacyjne mogą być traktowane jako narzędzie zarządzania jakością, które głównie służy właścicielowi produktu, natomiast testy jednostkowe, integracyjne i systemowe są narzędziem zarządzania jakością służącym zespołowi („Czy prawidłowo zbudowaliśmy system?"). Chociaż klient/właściciel produktu i zespół wykorzystują testy do osiągnięcia różnych celów, zawartość testów akceptacyjnych niekoniecznie różni się znacznie od własnych testów zespołu. Zespół może stosować kroki opisane w teście akceptacyjnym w przypadku testów wykonywanych na innym poziomie, tak jak właściciel produktu może wykorzystywać wiele istniejących testów systemowych do wybrania elementów zestawu testów akceptacyjnych.

Ręczne testy akceptacyjne uzupełniają inne zautomatyzowane testy.

W praktyce testowanie akceptacyjne podczas sprintu będzie ograniczone do tych elementów produktu, które zostały dopiero utworzone lub zostały zmienione podczas sprintu. Zespół przedstawia te przypadki

3 „Testy akceptacyjne sprawdzają, czy wszystkie aspekty systemu, w tym cechy takie jak użyteczność i wydajność spełniają wymagania klienta" [Crispin/Gregory 08, rozdział 6].

testów akceptacyjnych właścicielowi produktu podczas demonstracji sprintu w formie ręcznych testów badawczych. Jeśli wszystkie pozostałe testy są zautomatyzowane, podejście to niesie za sobą pewne ryzyko. Jeśli jednak żadne inne testy systemowe nie odbywają się podczas sprintu, jest bardziej prawdopodobne, że niepożądane efekty powodowane przez zmiany dokonane podczas sprintu pozostaną nieodkryte.

6.4 Zautomatyzowane testowanie systemowe

W projektach Scrum testy jednostkowe i integracyjne są zwykle przeprowadzane automatycznie przy użyciu skryptów testowych xUnit. Jeśli te skrypty testowe są osadzone w środowisku ciągłej integracji, to są one automatycznie uruchamiane za każdym razem, gdy kod się zmienia (zobacz rozdziały 4 i 5). W efekcie testy jednostkowe i integracyjne są wykonywane ciągle i dostarczają stałe informacje zwrotne zespołowi.

Byłoby świetnie, gdybyśmy mogli osiągnąć podobny poziom szybkości i wygody podczas testowania systemowego. Jednakże złożone środowisko (zobacz powyżej) utrudnia osiągnięcie tego celu. Dodatkowo trudniej jest zautomatyzować testy systemowe, niż napisać testy xUnit, ponieważ:

- Najważniejszym interfejsem testu systemowego jest graficzny interfejs użytkownika w danym produkcie, co wymaga użycia narzędzi dedykowanych do testowania interfejsu użytkownika (zobacz [URL: Testtool-review]). Działają one znacznie wolniej od typowych testów xUnit. Testy interfejsu użytkownika muszą też oczekiwać na reakcje i sygnały z innych składników (takich jak bazy danych) lub systemów zewnętrznych. Testy interfejsu użytkownika działają więc znacznie wolniej niż typowe testy jednostkowe.
- Konfiguracja systemu musi być jasno zdefiniowana i musi pozwalać na odtwarzanie stanu bazowego, na którym można by oprzeć skrypty testowe
- Inne interfejsy często odgrywają swoją rolę wymagając odpowiednich dodatkowych testów i narzędzi do automatyzowania testów.
- Wyniki testów często muszą być analizowane ręcznie, ponieważ automatyczne porównywanie oczekiwanego zachowania z faktycznym zachowaniem jest zbyt trudne (lub nawet niemożliwe).
- Z podobnych powodów niektóre przypadki testów systemowych mogą wymagać ręcznej interwencji.
- Na koniec zespół nie ma wielu członków, którzy są doświadczeni w automatyzowaniu testów systemowych.

6.4.1 Testowanie z użyciem rejestrowania/odtwarzania

Interfejsy testów jednostkowych i integracyjnych istnieją na poziomie kodu programu. Przypadki testowe są tworzone przy użyciu xUnit tak, że każdy przypadek testowy dotyczy pojedynczej metody interfejsu API testowanego obiektu, a zawartość każdego przypadku testowego może być bezpośrednio wzięta z planowanej funkcjonalności metody interfejsu API. Słownictwo związane z tymi testami pochodzi od metod interfejsu API i ich parametrów. Inaczej sprawa wygląda z testami systemowymi, które zwykle wykorzystują interfejs użytkownika produktu jako główny interfejs testowy. Testy systemowe są wiec pisane przy użyciu kroków, jakie użytkownik może wykonać poprzez graficzny interfejs użytkownika.

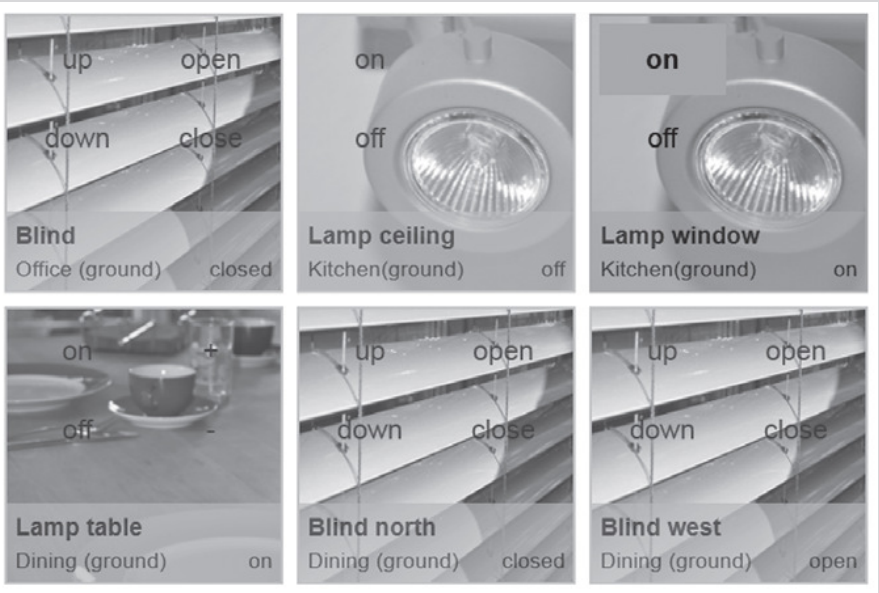
Narzędzia do rejestrowania/odtwarzania nagrywają sekwencje poleceń.

Narzędzia do rejestrowania/odtwarzania służą do nagrywania sekwencji poleceń użytkownika: narzędzie do rejestrowania/odtwarzania nagrywa wszystkie ręcznie wprowadzane polecenia wywoływane klawiaturą i myszą, które tester wykonuje podczas sesji testowej i zapisuje je w formie skryptu. Uruchomienie tego skryptu odtwarza zarejestrowaną sekwencję testową – czynność tę można powtarzać tak często, jak to konieczne (za [Spillner/Linz 14]). Skrypt testowy dla naszego studium przypadku eHome mógłby przybrać następujący kształt:

Studium przypadku 6-4

Studium przypadku 6-4 sterownika eHome: Sekwencja poleceń testu systemowego

Tester uruchamia aplikację przeglądarkową sterownika eHome i wykorzystuje kliknięcia myszy do włączenia i wyłączenia lampy znajdującej się przy oknie kuchennym. Okno przeglądarki wygląda następująco:



Tester przechwytuje polecenia korzystając z narzędzia Selenium do rejestrowania/odtwarzania (zobacz [URL: Testtoolreview]) i otrzymuje skrypt testowy wyglądający następująco:

```
open    http://ehome/eHomeController/index.php
clickAndWait  xpath=(//a[contains(text(),'on')])[2]
clickAndWait  xpath=(//a[contains(text(),'off')])[2]
```

Ten skrypt jedynie klika przycisk włączania/wyłączania lampy i nie sprawdza, czy system faktycznie wykonuje polecenie. Przypadek testowy zakłada angielski interfejs użytkownika (słowa „on" i „off"), w którym lampa przy oknie w kuchni pojawia się jako druga ikona lampy. W ten sposób wiąże przypadek testowy ściśle z graficznym interfejsem użytkownika i środowiskiem testowym – w tym przypadku korzystając z polecenia `clickAndWait`, którego długość może się różnić w zależności od konfiguracji środowiska testowego.

Jeśli korzystamy z narzędzia do rejestrowania/odtwarzania w celu nagrywania przypadków testów systemowych opartych na kliknięciach, łatwo jest wpaść w tego rodzaju pułapki. Podejście to również wiąże nas z wykorzystaniem określonego typu interfejsu (w tym przypadku HTML). Jednakże graficzny interfejs użytkownika jest elementem produktu, który często znacznie się zmienia w trakcie sprintów projektu, więc odpowiednie testy muszą być stale modyfikowane, żeby pasowały do zmieniającego się kształtu struktury poleceń i interfejsu użytkownika. Zwykle lepiej jest poświęcić nieco czasu podczas sprintu na napisanie nowych testów dla nowych funkcji produktu, niż stale utrzymywać i aktualizować istniejące testy. Inną wadą tego podejścia jest to, że nie bierze pod uwagę alternatywnych interfejsów użytkownika lub układów interfejsów. Na przykład sterownik eHome może być obsługiwany poprzez złożony interfejs użytkownika działający w przeglądarce lub prostszy interfejs na smartfonie/tablecie. Tworzenie testów systemowych dla każdego dostępnego interfejsu zajęłoby zbyt dużo czasu i jest po prostu nieopłacalne.

6.4.2 Testowanie sterowane słowami kluczowymi

Zespół może uniknąć tych problemów, jeśli zastosuje bardziej abstrakcyjne słownictwo, zamiast opisywać procedurę testową przy użyciu systemowych poleceń nawigacyjnych. Najlepszym wyborem jest zwykle słownictwo związane z przypadkami użycia systemu lub logiką biznesową.

Jest to znane podejście do tworzenia testów systemowych zwane testowaniem sterowanym słowami kluczowymi. Logika biznesowa sterownika eHome koncentruje się wokół przełączania urządzeń

elektrycznych, a zespół eHome postanawia zdefiniować odpowiedni język specyficzny dla domeny (DSL – Domain Specific Language)[4] do tworzenia swoich testów systemowych:

Studium przypadku 6-5a

Studium przypadku 6-5a sterownika eHome: Opracowywanie języka DSL do testowania systemowego

Właściciel systemu eHome wykorzystuje oprogramowanie eHome do sterowania urządzeniami elektrycznymi (np. lampami) lub wyświetlania stanu różnych czujników (np. temperatury w pokoju). Lampy można włączać, wyłączać lub przyciemniać, natomiast czujniki można odczytywać i nadawać im określone wartości graniczne.

Zdania wykorzystujące prosty język, budowane przy użyciu obiektów z tej domeny mogą być więc tworzone według wzorca <nazwa_obiektu> <polecenie> i mogłyby wyglądać następująco:

```
switch kitchen lamp on;
display living room temperature;
```

Rozwinięcie tego języka pozwala nam na dokładniejsze określanie urządzenia, które chcemy testować:

```
<piętro><pokój><typ_obiektu><nazwa_
obiektu><polecenie><parametr>
```

Ten prosty wzór wystarcza do opisania nawet dość złożonych sekwencji poleceń. Na przykład właściciel systemu eHome mógłby wprowadzić następujące polecenia w interfejsie telefonicznym podczas powrotu do domu:

```
open garage door
living room heating 20 degrees
ground floor lamp on[a]
living room couch lamp dim 60%[b]
television on[c]
```

a Jeśli brakuje nazwy obiektu, wszystkie urządzenia w określonej lokalizacji będą aktywowane.
b Dostępne parametry zależą od używanego polecenia.
c Jeśli nazwa obiektu jest unikalna, to lokalizacja nie musi być podana.

Jak widać, nasz nowy język domenowy dobrze nadaje się do zapisywania przypadków użycia, a jako efekt uboczny definiuje też interfejs użytkownika oparty na poleceniach dla sterownika eHome. Aby korzystać z niego jako specyficznego dla domeny języka do testowania, musimy

4 Ten zapis poleceń/słów kluczowych może być traktowany jako prosty język specyficzny dla domeny (DSL – Domain Specific Language). Języki DSL są ciekawym podejściem do automatyzacji testów na użytek projektów Scrum. Złożone języki DSL mają swoje własne zmienne, struktury sterujące, definicje procedur i inne elementy składniowe. Ciekawe wprowadzenia do budowania języków DSL można znaleźć w [Ghosh 11], [Fowler/Parsons 10] i [Rahien 10]. Podstawy budowania kompilatorów są omówione w [Aho et al. 06].

zapewnić, żeby reakcje systemu na te polecenia były prawidłowo rozpoznawane i porównywane z oczekiwanymi reakcjami. Oznacza to, że musimy dodać funkcję sprawdzającą do działań wymienionych powyżej. Przypadek testu systemowego mógłby więc wyglądać następująco:

Studium przypadku 6-5b

Studium przypadku 6-5b sterownika eHome: Przypadek testu systemowego

Składnia polecenia	Sekwencja wywołań funkcji testowych
`kitchen window lamp on`	`switch('kitchen','lamp','window','on');`
`kitchen window lamp status? on`	`assert('kitchen','lamp','window','on');`

Polecenie sprawdzające `assert` odczytuje stan lampy w kuchni i sprawdza, czy jest ona włączona. W zależności od stopnia inteligencji parsera i sterownika sekwencji testowych (musi on być w stanie wykonywać definiowane przez nas polecenia) możemy stworzyć interfejs użytkownika i język testowy tak wygodny w obsłudze, jak tylko chcemy. Pomocne jest też, jeśli parametry mogą być reprezentowane przez zmienne lub tablice wartości:

Studium przypadku 6-5c

Studium przypadku 6-5c sterownika eHome: Testowanie sterowane danymi

```
switch (FLOOR, ROOM, DEVICE, NAME, 'off');[a]
assert (FLOOR, ROOM, DEVICE, NAME, 'off');
switch (FLOOR, ROOM, DEVICE, NAME, 'on');
assert (FLOOR, ROOM, DEVICE, NAME, 'on');
switch (FLOOR, ROOM, DEVICE, NAME, 'off');
```

FLOOR (PIĘTRO)	ROOM (POKÓJ)	DEVICE (URZĄDZENIE)	NAME (NAZWA)
GF	living room	lamp	couch
GF	kitchen	lamp	window
FF	child	power outlet	television
...			

a Polecenie on/off może być też zawarte w tabeli z danymi testowymi. Jednakże celem tej sekwencji testowej jest upewnienie się, że różne urządzenia mogą być poprawnie włączane i wyłączane, a ten cel będzie wyraźniejszy, jeśli polecenie on/off (włączania/wyłączania) będzie kodowane w teście, a nie wymieniane z innymi danymi testowymi.

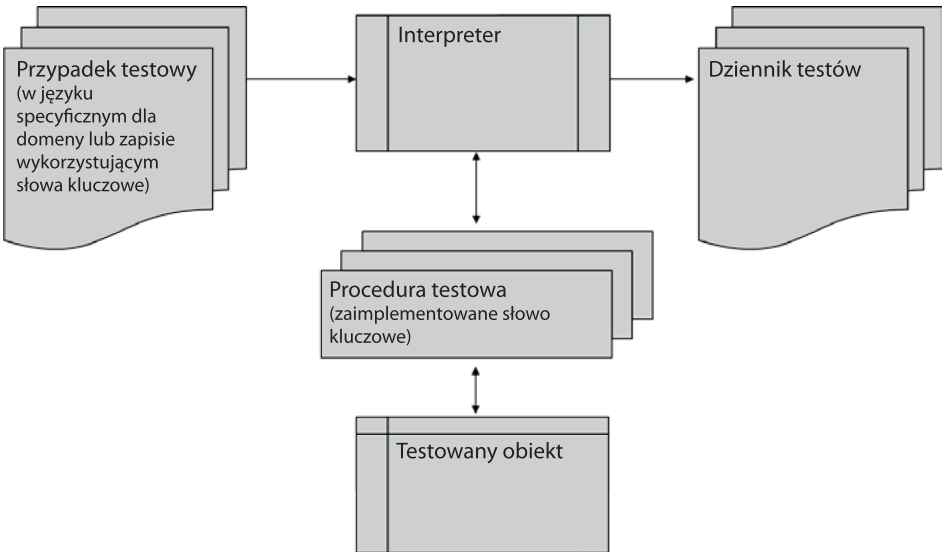
Tabela wartości jest używana w celu wyposażenia sekwencji testowej wymienionej powyżej w odpowiednie dane testowe, tworząc test sterowany danymi, który włącza urządzenia wymienione w tabeli, sprawdza, czy polecenia przełączające zostały wykonane, a następnie przełącza stan urządzenia z powrotem na wyłączone.

Zalety specyficznego dla domeny języka testowego opartego na słowach kluczowych

Do zalet specyficznego dla domeny języka testowego opartego na słowach kluczowych należą:

- Przypadki testowe są pisane wyłącznie przy użyciu języka domeny aplikacji. Reprezentują to, co system powinien być w stanie wykonać, ale nie to, jak działa. Są łatwe do zrozumienia przez testerów, użytkowników, klienta i inne zainteresowane strony oraz zapewniają ważne informacje zwrotne dla wszystkich.
- Nawet jeśli zmieni się implementacja produktu, testy pozostaną ważne i będzie można je przeprowadzać. Testy przykładowe wymienione powyżej mogą równie dobrze służyć do testowania interfejsu przeglądarkowego na komputerze osobistym, jak i dla całkiem innego interfejsu telefonicznego.
- Przypadki testowe pisane w ten sposób mogą też być używane do sprawdzania interfejsów systemowych niższego poziomu bez konieczności ich przepisywania. W naszym przykładzie silnik testowy mógłby tłumaczyć przypadki testowe na polecenia sterownika, które można by dostarczać bezpośrednio sterownikowi eHome. Silnik testowy może służyć jako obiekt zastępczy dla graficznego interfejsu użytkownika umożliwiając uruchamianie testów bez interfejsu użytkownika. To podejście ogranicza czas trwania fazy testowania i umożliwia nam testowanie sterownika na etapie testowania jednostkowego i integracyjnego, nawet jeśli interfejs użytkownika nie jest jeszcze gotowy.

Ten stopień wygody ma swoją cenę. Zespół musi uzgodnić zestaw słów kluczowych, który musi pozostać stały i musi napisać interpreter zamieniający polecenia na odpowiednie wywołania funkcji. Musi też utworzyć mechanizm kontrolny sekwencji do wykonywania poleceń i adapter łączący mechanizm kontrolny sekwencji z testowanym obiektem. Aby zapewnić, że adapter będzie prawidłowo identyfikować różne obiekty wymagające testowania (w przypadku naszego interfejsu użytkownika są to przyciski, pola opcji, przyciski wyboru, pola tekstowe, itd.), wszystkie one muszą mieć nadane unikalne identyfikatory, które pozostają niezmienione w czasie trwania sprintu. Jeśli identyfikatory ulegną zmianie, zespół będzie tracił czas na zmienianie kodu testowego pomiędzy sprintami. Rysunek 6-1 pokazuje, jak współdziałają ze sobą różne poziomy takiego zautomatyzowanego systemu testowego.



Rys. 6–1 *Trzypoziomowa architektura testowa*

Trzypoziomowa architektura testowa

Trzypoziomowa architektura testowa nakreśla wyraźne granice pomiędzy różnymi zadaniami i przekazuje zadania, które nie mają nic wspólnego z logiką testową do adaptera lub jednostki sterującej sekwencją testów:

- **Rozdzielanie środowiska testowego** Parametr taki jak limit czasowy nie jest atrybutem przypadku testowego, ale raczej zachowania środowiska testowego i powinien być przekazywany do sterowania sekwencją testów, gdzie może zostać zaimplementowany jako metoda klasy środowiska testowego (`TestEnvironment`). Gdy metoda ta jest wywoływana (na przykład poprzez `TestEnvironment->wait_for MsgAck()`), możemy mieć pewność, że tylko czeka na sygnał potwierdzający ilość czasu określoną przez środowisko testowe, a czas poświęcony na oczekiwanie nie jest już atrybutem logiki testowej. Inne atrybuty środowiska testowego mogą być przekazywane z logiki testowej w podobny sposób.
- **Rozdzielanie interfejsu testowego** Jeśli logika testowa zaczyna się zbytnio splatać z określonym interfejsem testowym, test może zostać przeprowadzony jedynie przy użyciu tego interfejsu. W przeciwieństwie do testów jednostkowych i integracyjnych, niektóre testy systemowe muszą być uruchamiane poprzez wiele interfejsów. Na przykład sterownik eHome może być obsługiwany przez różne interfejsy (przeglądarki na komputerze osobistym, przeglądarki i aplikacje na smartfonie oraz interfejsy wiersza poleceń). Ponieważ sterownik sekwencji testowej ma dostęp do żądanego interfejsu w sposób pośredni poprzez adapter, może komunikować się z innymi interfejsami po prostu przez wymianę adaptera bez konieczności zmieniania logiki testowej. Adapter tłumaczy kroki testu na postać zrozumiałą dla interfejsu.

Aby przeprowadzić testy poprzez interfejs użytkownika obiektu testowego, musimy skorzystać z adaptera, który może sterować odpowiednim narzędziem do rejestrowania/odtwarzania (np. Selenium IDE) oraz tłumaczyć polecenia na postać, którą może przetworzyć.

Automatyzacja testów systemowych opartych na słowach kluczowych ma wysoki koszt początkowy.

Podczas projektowania testów systemowych zespół musi oszacować, ile modularności i elastyczności wymaga dany projekt. Konfigurowanie trzypoziomowej, opartej na słowach kluczowych automatyzacji testów systemowych wymaga znaczących inwestycji w czas i zasoby, jest więc przesadą w projektach krótkoterminowych lub na małą skalę. Są jednak dostępne narzędzia, które upraszczają konfigurację architektury testowej (zobacz [URL: Testtoolreview]).

Zadania konfigurowania automatyzacji testów można też oddzielić od zadań, które są bezpośrednio związane z rozwojem produktu. Rozwój produktu obejmuje implementowanie funkcjonalności i pisanie odpowiednich testów systemowych przy użyciu słów kluczowych. Jeśli mechanizm sterowania sekwencją testową jest już dostępny, testy oparte na słowach kluczowych mogą być natychmiast uruchamiane – jeśli jednak sterowanie sekwencją nie jest jeszcze gotowe, niektóre z testów mogą uruchamiać się automatycznie, natomiast reszta będzie tymczasowo musiała być uruchamiana ręcznie, choć elementy, które tester powinien testować ręcznie, są definiowane przy użyciu słów kluczowych.

Do programowania mechanizmu sterowania sekwencją nie jest wymagana duża wiedza na temat logiki specyficznej dla danego produktu. Wiedza na temat automatyzacji testów jest dużo ważniejsza. Pisanie aplikacji, szkicowanie testów systemowych (przy użyciu słów kluczowych) oraz programowanie sterownika sekwencji testowej są więc zadaniami, które mogą być dystrybuowane wśród członków zespołu, a planowanie sprintu można wykorzystywać do zapewnienia, że każde z nich będzie nadążało za pozostałymi. Opracowanie sterownika sekwencji testów od zera wymaga oczywiście więcej nakładów i dlatego musi mieć nadany wysoki priorytet podczas kilku pierwszych sprintów.

6.4.3 Testowanie sterowane zachowaniami

Innym sposobem nakreślania i automatyzowania testów przy użyciu języka specyficznego dla domeny jest wykorzystanie technik testowania sterowanego zachowaniami (zobacz [URL: BDT]). To podejście wykorzystuje bardziej naturalny język niż podejście oparte na słowach kluczowych i tabelach pokazane wyżej.

Jednak te dwa podejścia mają cechy wspólne i całkiem możliwe jest zapisywanie już zdefiniowanych słów kluczowych w centralnym,

hierarchicznym repozytorium oraz korzystanie z nich do definiowania skryptów testowych.

Dostępnych jest wiele platform testowania sterowanego zachowaniami (np. Fit, FitNesse, Cucumber, JBehave, Specs2 i Behat (zobacz [URL: Testtoolreview])), które obsługują automatyzację przypadków testowych. Automatyzacja odbywa się z wykorzystaniem podobnego podejścia, co testowanie sterowane słowami kluczowymi, ponieważ dana funkcja jest testowana przy użyciu różnych scenariuszy. Scenariusz odpowiada przypadkowi testowemu i jest dzielony na sekcje zgodnie z zasadą `Zakładając-Gdy-To`, co jest w dużym stopniu zgodne z sekcjami `przygotowanie-procedura-sprawdzenie` dla przypadku testu jednostkowego (zobacz podrozdział 4.1.2).

Zespół eHome wybiera zgodną z PHP platformę Behat i wykorzystuje ją do utworzenia następującego skryptu testowego:

Studium przypadku 6-6

Studium przypadku 6-6 sterownika eHome: Przypadek testu systemowego nakreślony jako test sterowany zachowaniami przy użyciu Behat

```
Funkcja: sterowanie urządzeniami
Jako właściciel systemu eHome chcę sterować różnymi
klasami urządzeń (np. lampami, ściemniaczami,
żaluzjami), ale w celu uniknięcia wadliwego działania
sterownik pozwala dla każdej klasy urządzeń na
tylko te polecenia, które mogą być wykonywane przez
to urządzenie.
Scenariusz: Włączanie i wyłączanie lampy
  Given device "lamp" in "kitchen" at "window"
  When switch "on" "lamp" in "kitchen" at "window"
  Then status "lamp" in "kitchen" at "window" is "on"
```

Zespół musi zaimplementować słowa kluczowe i parametry, które występują w tym scenariuszu (np. `switch` i `"lamp"`) w odpowiednich skryptach testowych. Na przykład polecenie włączenia musi być zaprogramowane jako funkcja PHP spełniająca następujący wzorzec:

```
/**
* @When /^switch "([^"]*)" "([^"]*)" in "([^"]*)" at
"([^"]*)"$/
*/
public function switchInAt($arg1, $arg2, $arg3,
$arg4) {

   ...
}
```

Podczas przeprowadzania tego testu Behat wywołuje funkcję PHP korzystając z następujących parametrów:

```
   switchInAt("on", "lamp", "kitchen", "window");;
```

Zastosowanie platformy testowania sterowanego zachowaniami tworzy też wielowarstwową architekturę testową (zobacz rys. 6-1). Podobnie do platform testów jednostkowych platformy testowania sterowanego zachowaniami również wykorzystują interfejs API obiektu testowego jako interfejs automatyzacji. Jeśli interfejs API znajduje się na wyższym poziomie architektury testowej, testowanie sterowane zachowaniami może częściowo zastąpić przypadki testów systemowych, które w przeciwnym razie musiałyby być przeprowadzane poprzez interfejs użytkownika produktu.

Testowanie sterowane zachowaniami a testy API

Przypadki testów jednostkowych (zobacz rozdział 4) mogą być szkicowane również przy użyciu testowania sterowanego zachowaniami, chociaż potrzebne nakłady rzadko znajdują uzasadnienie. W porównaniu z wymaganiami na poziomie systemu interfejs API pojedynczej klasy zwykle implementuje tylko częściową lub podstawową funkcjonalność, którą prościej wyrazić za pomocą metod interfejsu API i ich parametrów niż przy użyciu wymagań systemowych. Jeśli jednak chcemy, aby nasze przypadki testów jednostkowych lub integracyjnych były zrozumiałe dla osób, które nie rozumieją kodu programu lub kodu testów, testowanie sterowane zachowaniami może być przydatnym narzędziem.

6.5 Programowanie sterowane testami przy testowaniu systemowym

Sterowanie testami oznacza pisanie i automatyzowanie jednego lub wielu przypadków testowych, zanim kod, który ma być testowany, zostanie napisany lub zmieniony[5]. Jeśli testy te zakończą się powodzeniem bez ujawniania żadnych usterek, jest to traktowane jako kryterium gotowości dla danego zadania programowego. Podejście to działa bardzo dobrze w kontekście testowania jednostkowego i integracyjnego (zobacz rozdziały 4 i 5), ale czy można je też stosować wobec testowania systemowego?

Ponieważ narzędzia do testowania graficznego interfejsu użytkownika wymagają bezpośredniego dostępu do interfejsu użytkownika aplikacji, jeśli testy są rejestrowane lub kodowane przy użyciu narzędzia testującego graficzny interfejs użytkownika, to obiekt testowy i jego interfejs użytkownika muszą istnieć. Zasady sterowania testami nie mogą być stosowane w tym kontekście.

Oparte na słowach kluczowych przypadki testów systemowych

Jeśli przypadki testów systemowych są szkicowane przy użyciu technik opartych na słowach kluczowych lub wykorzystujących testowanie sterowane zachowaniami (tzn. wykorzystujących słownictwo oparte na

5 „Napisz zautomatyzowany test kończący się niepowodzeniem przed zmianą jakiegokolwiek kodu" [Beck/Andres 04, rozdział 7].

logice biznesowej), wynikiem będą przypadki testowe funkcjonujące niezależnie od technicznej implementacji produktu i jego interfejsu użytkownika. Takie przypadki testowe mogą być więc również przygotowywane, zanim obiekt testowy będzie istniał. Techniki oparte na słowach kluczowych i testowaniu sterowanym zachowaniami nie tylko są więc przydatne, gdy chodzi o modularyzację zadań automatyzacji testów, ale też mogą być używane do implementowania technik sterowania testami na poziomie testowania systemowego.

6.5.1 Repozytorium testów systemowych

Aby zespół mógł udanie pracować przy wykorzystaniu testowania opartego na słowach kluczowych albo testowania sterowanego zachowaniami, musi uzgodnić stosowane słownictwo i zapis. Jeśli nowe słowa kluczowe będą wprowadzane w niekontrolowany sposób, to ryzykujemy ponowne wymyślanie słów kluczowych odnoszących się do funkcji systemowych, które już są objęte istniejącymi słowami kluczowymi. To z kolei prowadzi do tworzenia redundantnych przypadków testowych albo wielu wersji bieżących testów. Ważne jest więc utrzymywanie centralnego słownika, w którym zespół zbiera swoje słowa kluczowe i nimi zarządza.

Test*Bench* (zobacz [URL: Testtoolreview]) jest świetnym narzędziem właśnie do tego. Zespół może wykorzystywać swoje repozytorium do przechowywania słów kluczowych przy zastosowaniu standardowej notacji, a nowe przypadki testowe mogą być budowane z kombinacji słów kluczowych metodą przeciągnij i upuść. Listy odwołań pokazują, które słowa kluczowe i parametry są używane w których przypadkach i mogą służyć do monitorowania zmian w słowach kluczowych lub przypadkach testowych. Narzędzie to znacząco ogranicza ilość wysiłku związanego z zarządzaniem zmianami w obrębie pakietu testów systemowych.

6.5.2 Programowanie w parach

Programowanie w parach jest przydatną techniką dodatkową, która pomoże nam zachowywać zasady sterowania testami podczas testowania systemowego. Różne pary mogą być używane do zajmowania się różnymi typami zadań.

- **Szkicowanie nowych przypadków testów systemowych** Zanim zacznie się kodowanie, programista musi naszkicować żądane przypadki testowe. Zwłaszcza testy systemowe powinny być zawsze przeprowadzane w parze z testerem, który może dopilnować, aby przypadki testów systemowych widziały system z perspektywy

użytkownika i obejmowały wszystkie wymagania. W międzyczasie programista może pomagać testerowi w skupieniu się na pisaniu testów nakierowanych wyłącznie na daną funkcję.

- **Opracowywanie i utrzymywanie słów kluczowych** Wiedza specjalistyczna dotycząca aplikacji i jej domeny jest najważniejszą umiejętnością, jeśli chodzi o nadawanie nazw słowom kluczowym (i ich parametrom) oraz wprowadzanie definicji do repozytorium. To zadanie może być często przeprowadzane przez pojedynczego testera. Jednakże budowanie i utrzymywanie dobrze przemyślanej biblioteki słów kluczowych przez wszystkie sprinty projektu jest dużo trudniejsze, a doświadczenie pokazało, że pary testerów dają lepsze wyniki i pracują wydajniej niż pojedynczy tester pracujący samotnie.
- **Automatyzacja słów kluczowych** To zadanie dotyczy implementowania słów kluczowych przy pomocy skryptów testowych na wszystkich poziomach sterownika sekwencji testowych. Wymaga to zaprogramowania przy użyciu języka skryptowego narzędzia testującego graficzny interfejs użytkownika i może wiązać się z programowaniem w xUnit lub innych językach skryptowych. To zadanie najlepiej przeprowadzać w zespołach składających się z testerów i programistów.

6.6 Testowanie niefunkcjonalne

Testy systemowe, które opisaliśmy na razie, są wszystkie funkcjonalnymi testami systemowymi, które sprawdzają, czy oprogramowanie spełnia wymagania funkcjonalne – tzn. czy wydanie polecenia takiego jak „włącz lampię" faktycznie wykonuje dane działanie.

ISO 25010

Produkty programowe są też poddawane wielu niefunkcjonalnym wymaganiom. Według [Spillner/Linz 14, rozdział 3], „Wymagania niefunkcjonalne opisują atrybuty zachowania systemu – innymi słowy, jak dobrze i na jakim poziomie jakości (pod)system wykonuje swoje zadanie. Implementacja wymagań niefunkcjonalnych ma istotny wpływ na zadowolenie klienta i stopień, w jakim użytkownik końcowy cieszy się, korzystając z produktu." Atrybutami tymi są wydajność, kompatybilność, użyteczność, niezawodność, bezpieczeństwo[6], łatwość utrzymania i przenośność – są one opisane szczegółowo w [ISO 25010][7]. Spełnienie wymagań niefunkcjonalnych musi być

6 Aspekt „bezpieczeństwa" (ang. *safety*) zachowania systemowego w przypadku błędnego użycia lub błędu systemu nie powinien być mylony z „zabezpieczeniami" (ang. *security*).

7 ISO 25010:2011 jest częścią zestawu norm „Inżynieria oprogramowania – wymagania i ocena jakościowa produktu programowego" i w roku 2011 zastąpiła

sprawdzane przy użyciu odpowiednich testów. Według [Spillner/Linz 14][8] należą do nich:

- **Testowanie obciążeń** Mierzenie zachowania systemu przy zwiększającym się obciążeniu – na przykład zwiększającej się liczbie jednoczesnych użytkowników lub transakcji bazodanowych.
- **Testowanie wydajności** Mierzenie szybkości przetwarzania i czasu odpowiedzi dla konkretnych przypadków użycia, zwłaszcza przy zwiększającym się obciążeniu.
- **Testowanie masowe/testowanie wydolnościowe** Obserwacja zachowania systemu dla zmieniających się ilości danych (na przykład przy przetwarzaniu bardzo dużych plików) i podczas przeciążeń.
- **Testowanie zabezpieczeń** przed nieautoryzowanym dostępem do systemu lub danych.
- **Testowanie niezawodności** i stabilności przy ciągłym użyciu (na przykład mierzenie, ile błędów systemowych zdarza się w ciągu godziny dla określonego profilu użytkownika).
- **Testowanie kompatybilności i konwersji danych** Sprawdzanie współpracy z istniejącymi systemami i procesów importu/eksportu danych.
- **Testowanie solidności** w przypadku niewłaściwego użycia, błędów programowych, usterek sprzętowych, itd. oraz testowanie obsługi błędów.
- **Testowanie konfiguracji** dla różnych konfiguracji systemowych – na przykład korzystania z różnych wersji systemu operacyjnego, różnych języków lub różnych platform sprzętowych.
- **Użyteczność** Sprawdzanie, jak łatwo jest nauczyć się i korzystać z systemu. Obejmuje sprawdzanie łatwości zrozumienia danych wyjściowych z systemu w zależności od potrzeb różnych grup użytkowników (zobacz też [ISO 9241]).
- **Sprawdzanie dokumentacji** Sprawdzanie, czy zawartość dokumentacji (podręczniki użytkownika, materiały szkoleniowe, itd.) odpowiada zachowaniu systemu.
- **Łatwość modyfikowania i konserwacji** Sprawdzanie dokładności i aktualności dokumentacji projektowej, struktury sytemu itd.

część 1 poprzedniej normy jakościowej [ISO 9126].

8 [Crispin/Gregory 08] klasyfikuje testowanie oprogramowania korzystając ze schematu czterech kwadrantów. Testy niefunkcjonalne należą do kwadrantu 4, który obejmuje „zautomatyzowane testy zorientowane na technologię". Testy użyteczności są traktowane jako testy ręczne należące do kwadrantu 3.

Sprawdzanie i testowanie tych wymagań stawia zespół zwinny przed wieloma wyzwaniami:

- Testy obciążeniowe, wydajnościowe, masowe i inne podobne testy są z definicji rozbudowane i długoterminowe. Jeśli zostaną wbudowane w proces ciągłej integracji, mogą znacząco go spowalniać i są często zbyt rozciągnięte w czasie, żeby mogły zostać zawarte w conocnym procesie budowania.
- Testy solidności, odporności na błędy sprzętowe i inne podobne testy są trudne do zautomatyzowania i zwykle wymagają ręcznej interwencji podczas konfigurowania środowiska testowego.
- Testowanie przyjazności dla użytkowników, łatwości konserwacji kodu i sprawdzanie dokumentacji wymaga ręcznej interwencji i rozległych przeglądów.

Wymaganiami niefunkcjonalnymi należy zająć się na wczesnym etapie projektu i wymagają one stałego przeglądu i testowania. Najbardziej oczywistym rozwiązaniem jest opakowanie wszystkich tych testów w pojedynczy sprint testowania systemowego. Jednakże ta strategia (zobacz podrozdział 6.8.1) ma poważną wadę, mianowicie informacja zwrotna nie jest zapewniana we właściwym czasie. W powiązaniu z wymaganiami niefunkcjonalnymi stanowi to duże ryzyko, gdyż negatywne atrybuty, takie jak słaba wydajność lub użyteczność są zwykle powodowane przez fundamentalne wady architektury lub projektu systemu, których nie można naprawić przez przepisanie kilku wierszy kodu. Gdy wykryte zostaną tego rodzaju usterki, często wymagany jest kompleksowy refaktoring.

Istnieją różne sposoby radzenia sobie z tym dylematem przez zespół:

- Testowanie długoterminowe: Funkcje, które mają zostać utworzone lub zmodyfikowane podczas bieżącego sprintu, są definiowane podczas fazy planowania sprintu tak samo jak odpowiadające im kryteria akceptacji. Obejmuje to też wszystkie niefunkcjonalne wymagania, na które wpływają lub których wymagają dane funkcje. Projekt i (jeśli to możliwe) automatyzacja odpowiednich testów, które sprawdzają niefunkcjonalne atrybuty w sposób zorientowany na funkcje, stanowią istotną część listy zadań i muszą być przeprowadzane podczas sprintu. W tym celu mistrz Scrum musi przejść przez powyższą listę atrybutów niefunkcjonalnych wraz z zespołem, aby zdecydować, które z nich są istotne dla danej funkcji. Wynikowe testy muszą następnie zostać przeprowadzone co najmniej raz podczas sprintu. To podejście zapewnia, że na końcu sprintu zespół otrzyma przydatne informacje zwrotne dotyczące niefunkcjonalnych aspektów wszelkich nowych funkcji oraz czy istnieją jakieś problemy architekturalne, którymi można zająć się w następnym sprincie.

- Testy solidności, odporności na błędy sprzętowe i inne podobne testy mogą być przeprowadzane jako testy badawcze, choć pełna analiza regresji dla wszystkich testów solidności nie jest zwykle konieczna. Natomiast ograniczona liczba testów solidności jest wybierana z listy i dodawana do listy zadań sprintu.
- Testy przyjazności dla użytkownika, dokumentacji, konserwacji kodu i inne podobne testy są przeprowadzane stale od wczesnego etapu przy zastosowaniu podejścia programowania w parach. Zamiast przeprowadzać rozbudowane, czasochłonne przeglądy na końcu sprintu (albo w oddzielnym sprincie testów systemowych), odpowiednie sprawdzenia dokonywane są codziennie (lub w trybie ciągłym) przez zespoły par. Aby zapewnić niezawodność testów, mistrz Scrum wydaje listę odpowiednich aspektów testowania każdej parze i okresowo sprawdza ustalenia oraz środki zaradcze podjęte przez każdy zespół. Atrybuty niefunkcjonalne rzadko są lokalnymi atrybutami pojedynczej funkcji i najczęściej są atrybutami systemu globalnego (lub architektury systemowej), więc istotne jest, aby mistrz Scrum przeprowadzał regularne przeglądy omawiające wyniki testów z zespołem. Niezależnie od ogólnego obrazu zespół następnie musi zdecydować na przykład, które zadania odnoszące się do poprawienia przyjazności dla użytkowników powinny zostać zawarte na liście zaległości produktowych lub na liście zaległości następnego sprintu.

Sprawdzanie i testowanie wymagań niefunkcjonalnych jest utrudniane przez fakt, że są one często definiowane nieprecyzyjnie. Wymagań takich jak „Systemy muszą być łatwe w użyciu" albo „System powinien reagować szybko" nie da się przetestować i dlatego nie mogą być one używane jako kryteria akceptacji. Jeśli zespół zwinny korzysta ze sterowania testami, to będzie miał dużą pulę testów funkcjonalnych, które może wykorzystać do definiowania i testowania niefunkcjonalnych atrybutów systemu, co znacznie ułatwia ten aspekt zarządzania projektem.

„Scenariusze są wybierane z istniejących testów funkcjonalnych, które reprezentują przekrój ogólnej funkcjonalności systemu, przy czym każdy niefunkcjonalny aspekt, który ma być testowany, musi być możliwy do zaobserwowania w wybranym scenariuszu testowym. Podczas testowania czynnik niefunkcjonalny musi mieścić się w obrębie wstępnie zdefiniowanej wartości progowej, aby test został uznany za udany. Innymi słowy, scenariusz testu funkcjonalnego służy jako instrukcja pomiarowa, która testuje wybrany atrybut niefunkcjonalny" [Spillner/Linz 14, rozdział 3]. W ten sposób wybrane przypadki testowe służą nie tylko jako instrukcje testowania, ale też pomagają definiować wymagania i dokumentują zachowanie systemu.

6.7 Zautomatyzowane testowanie akceptacyjne

Właściciel produktu nie jest ograniczony do korzystania z testów systemowych podczas przygotowywania testów akceptacyjnych. Zautomatyzowane przypadki testów jednostkowych lub integracyjnych również mogą wzbogacać pakiet testów akceptacyjnych, a właściciel produktu może korzystać ze zautomatyzowanych testów systemowych zamiast testowania ręcznego.

Kryteria akceptacyjne, które muszą być spełnione, są krytycznym czynnikiem, jeśli chodzi o zdecydowanie się, z których testów skorzystać i muszą być przeprowadzone przy użyciu odpowiedniego podzbioru dostępnych testów automatycznych. Testy akceptacyjne muszą być przeprowadzane ręcznie tylko dla kryteriów, które nie są jeszcze objęte istniejącymi testami zautomatyzowanymi.

Studium przypadku 6-7

Studium przypadku 6-7 sterownika eHome: Pakiet testów zgodności

eHome Tools udziela licencji na swój produkt partnerowi, który produkuje też urządzenia do automatyzacji domu, które mają być wyposażone w oprogramowanie sterownika eHome. Oczywiście wymagane jest bezproblemowe współdziałanie oprogramowania z urządzeniami, więc rozbudowany, zautomatyzowany zestaw testów zgodności jest ważną częścią procesu certyfikacji, przez który musi przejść każdy produkt.

eHome Tools dołącza pakiet testów zgodności do fazy testowania integracyjnego swojego procesu ciągłej integracji, gwarantując w ten sposób wszystkim przyszłym aktualizacjom zgodność i akceptację ze strony klienta.

6.8 Kiedy powinno odbywać się testowanie systemowe?

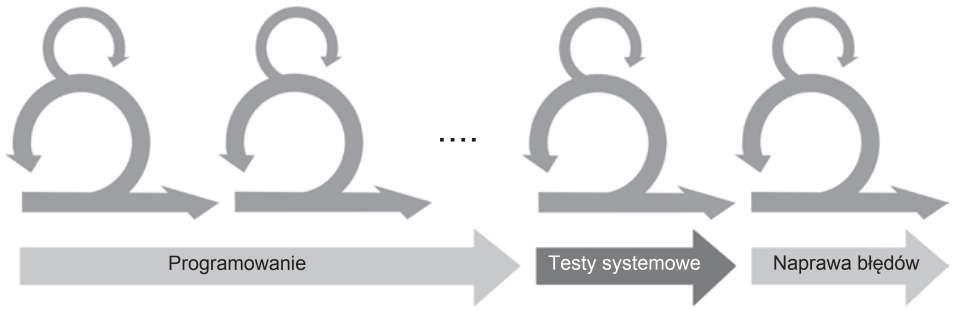
W projektach zwinnych testowanie systemowe jest zwykle dużo mniej zautomatyzowane niż testowanie jednostkowe i integracyjne[9]. Testowanie systemowe zwykle wiąże się ze sporą dawką interwencji ręcznej, która może zająć kilka dni. Ponieważ każdy sprint generuje

9 W tradycyjnie zarządzanych projektach proporcje te są zazwyczaj odwrócone. Niezależnie zorganizowana grupa testowania systemowego buduje rozbudowany, zautomatyzowany system testowania graficznego interfejsu użytkownika, natomiast programiści, którzy mają niewielkie doświadczenie w zarządzaniu jakością i testowaniu, są jedynie na marginesie związani z tworzeniem testów jednostkowych.

nowe funkcje, które wymagają swoich własnych testów systemowych, liczba ręcznych testów systemowych rośnie ze sprintu na sprint.

Już po kilku sprintach staje się jasne, że zespół nie może przeprowadzać wszystkich swoich testów systemowych dla każdego sprintu (a tym bardziej dla każdej zmiany w kodzie) w taki sposób, jak to robi w przypadku swoich testów jednostkowych i integracyjnych. Zespół musi więc ostrożnie planować, kiedy ma przeprowadzać poszczególne testy systemowe. Kolejne podrozdziały opisują wiele sposobów podejścia do ustalania czasu testów systemowych.

6.8.1 Testowanie systemowe w ostatnim sprincie



Rys. 6–2
Testowanie systemowe w ostatnim sprincie

Jest to najprostsza strategia ustalania czasu testów systemowych i polega na zaplanowaniu osobnego „sprintu testów systemowych". To podejście jest łatwe do zaplanowania i może być używane w przypadku ręcznych testów systemowych. Sprint testów systemowych zawiera jedynie zadania testowania systemowego. Wady tego podejścia to:

- Dopiero po zakończeniu sprintu testów systemowych możemy stwierdzić, czy mamy potencjalnie gotowy do dostarczenia produkt. Jeśli sprint testów systemowych nadal trwa, dostarczanie produktu wiąże się z wysokim poziomem ryzyka.
- Liczba poprawek błędów i nakłady na ponowne testowanie mogą być oszacowane dopiero po zakończeniu sprintu testów systemowych i wymagane mogą być dodatkowe sprinty na naprawę błędów lub ponowne testowanie.
- Informacje zwrotne z testowania systemowego są dostępne dopiero po ostatnim sprincie testów systemowych, co często wypada zbyt późno. Nie jest spełniony cel zapewniania programistom informacji zwrotnej w odpowiednim czasie.

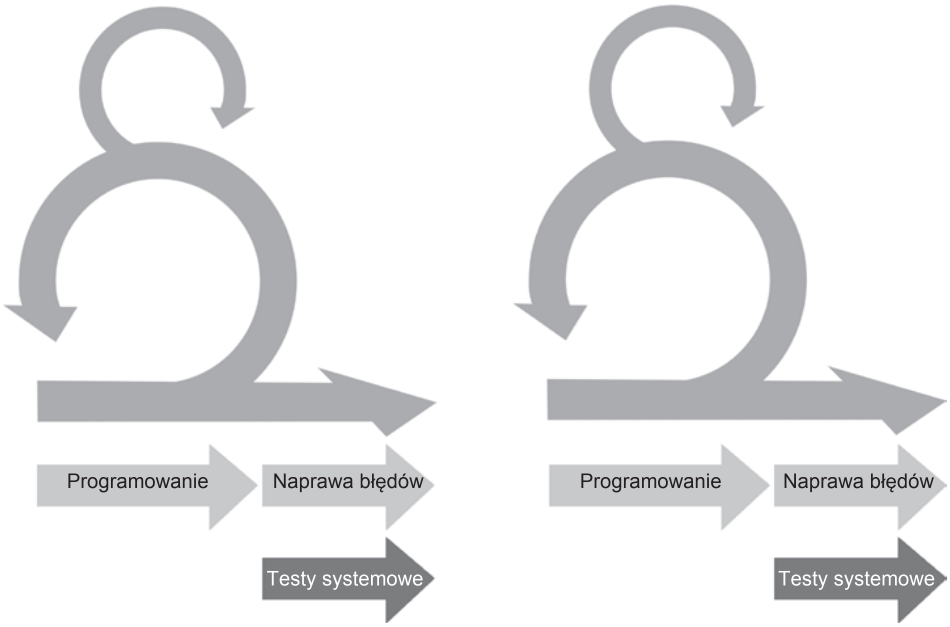
Ogólna procedura jest bardzo podobna do tradycyjnego testowania systemowego w modelu V i utrudnia wykorzystanie zalet praktyk zwinnych – zwłaszcza tworzenia potencjalnie gotowego produktu po każdym sprincie.

Niezależny zespół testowania systemowego

Jednym ze sposobów obejścia tych wad jest utworzenie niezależnego zespołu testowania systemowego (studium przypadku 8.5, „Scrum w środowisku technologii medycznych" opisuje dobry przykład tego podejścia). Nowy zespół może wziąć sprint testów systemowych na swoje barki dając zespołowi produktowemu możliwość pójścia naprzód z następnym sprintem ukierunkowanym na programowanie. W ten sposób programowanie i testowanie systemowe mogą działać równolegle z przesunięciem o jeden sprint. Błędy wykryte w trakcie sprintu testów systemowych mogą być debugowane podczas następnego sprintu programistycznego, a z punktu widzenia programisty pętla informacji zwrotnej ma długość jedynie dwóch sprintów.

6.8.2 Testowanie systemowe na końcu sprintu

Rys. 6–3
Testowanie systemowe na końcu sprintu



Ta strategia polega na przeprowadzaniu testowania systemowego w specjalnie zarezerwowanym momencie na końcu każdego sprintu i nadaje się do wykorzystania w prostszych projektach, które nie wymagają wiele testowania systemowego. Jeśli rozmiar nakładów na testowanie systemowe grozi zbytnim wzrostem, zespół musi wybrać, które testy systemowe przeprowadzić, a które pominąć przy każdym sprincie. W tym przypadku dla menedżera testów ważne jest wyśledzenie sytuacji, które można poprawić przez zwiększenie stopnia automatyzacji testów systemowych. Wady tej strategii to:

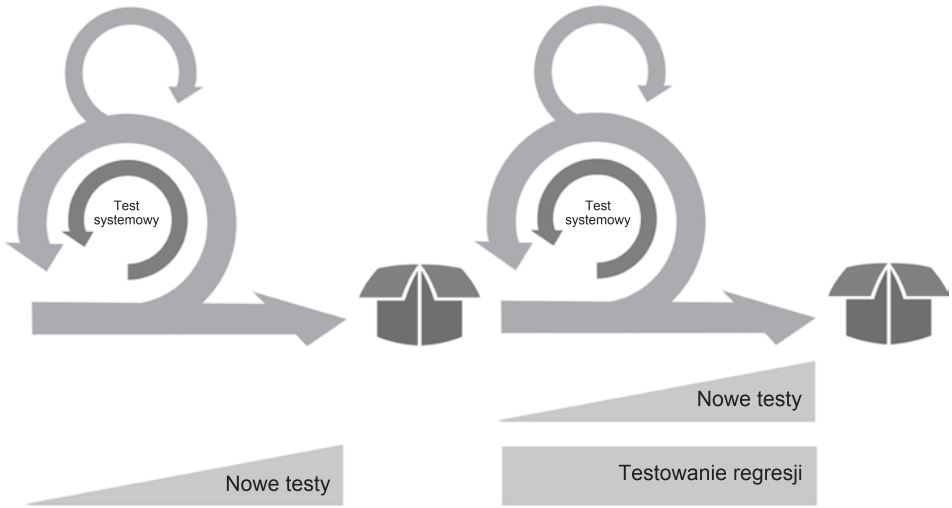
- Jak w przypadku strategii sprintu testów systemowych, to podejście również wytwarza nieznaną liczbę zadań związanych z poprawianiem błędów i ponownym testowaniem na końcu sprintu, co może zagrozić zaplanowanym ramom czasowym.

- Aby zyskać czas na zadania naprawy błędów trzymając się przy tym zaplanowanych ram czasowych, trzeba będzie regularnie usuwać scenariusze użytkowników z listy zaległości sprintu.
- Ponieważ czas jest ograniczony i przeprowadzić można tylko wybrane testy systemowe, wzrasta ryzyko przeoczenia usterek i przeniesienia ich na kolejne sprinty. To ryzyko jest mniej poważne, jeśli przeprowadzimy testowanie regresji dla wszystkich przypadków testów systemowych.
- Programista musi czekać do końca sprintu, aby uzyskać informacje zwrotne. Jeśli odpowiednie przypadki testów systemowych są zawarte w kryteriach gotowości zadania programistycznego (jak powinny), to zadania te pozostają „niezrealizowane" aż do końca sprintu, co uniemożliwia skuteczne szacowanie postępów projektu.

Sprinty stają się uszeregowane jak w podejściu kaskadowym.

Informacje zwrotne dla zespołu następują podczas sprintu, choć zadania są uszeregowane jak w podejściu kaskadowym. Ponieważ testowanie systemowe zawsze ujawnia jakieś usterki, pewna ilość czasu musi być zarezerwowana na naprawianie błędów i ponowne testowanie. Sprint jest więc w nieunikniony sposób dzielony na fazy programowania, testowania systemowego i naprawiania błędów – sytuacja ta często bywa określana jako „kaskadowy Scrum".

6.8.3 Testowanie systemowe non-stop



Rys. 6–4
Testowanie systemowe non-stop
Budowanie co noc

Strategia testowania systemowego non-stop stosuje zasady testowania jednostkowego i integracyjnego wobec testowania systemowego i jest chyba najlepszym rozwiązaniem. „Testowanie systemowe non-stop" oznacza możliwie jak największą automatyzację testów systemowych i integrowanie ich w ramach zautomatyzowanego procesu ciągłej integracji, podobnie jak w przypadku testów jednostkowych

i integracyjnych. Przypadki testów systemowych mogą być pogrupowane na pakiety według wymaganych warunków wstępnych i czasów działania, które następnie mogą być wykonywane lub ukrywane, a środowisko ciągłej integracji zawiera wszystkie testy projektu, od poziomu jednostkowego do systemowego.

Z powodu ilości czasu wymaganego do uruchamiania testów integracyjnych i systemowych nie wszystkie przypadki testowe są wykonywane podczas każdego przebiegu ciągłej integracji. Jednakże zespół ma możliwość wyboru odpowiedniej mieszanki testów jednostkowych, integracyjnych i systemowych, zanim rozpocznie proces ciągłej integracji. Pełne testowanie może się nadal odbywać podczas conocnego budowania dając zespołowi pełne informacje zwrotne (wraz z informacjami zwrotnymi z testów systemowych) co najmniej raz dziennie. Studium przypadku 8.2 podaje przykład testowania systemowego non-stop w praktyce.

Jeśli testy systemowe dla nowej funkcji mają być wykonywane w pierwszym, nocnym budowaniu, to testy systemowe muszą być automatyzowane równolegle z opracowywaniem nowych funkcji. Jest to jedyny sposób zapewniający, że zespół będzie otrzymywać regularne, codzienne informacje zwrotne.

Testowanie systemowe non-stop zakłada, że testowanie systemowe zostało całkowicie zautomatyzowane, a stosowanie zasad sterowania testami względem testowania systemowego jest świetnym sposobem wspierania postępów projektu (zobacz też podrozdział 6.5: Programowanie sterowane testami przy testowaniu systemowym).

6.9 Sprint tworzący wersję produktu oraz wdrażanie

Jeśli dobrze zaimplementowano ciągłą integrację, sprint będzie wytwarzał kilka całkowicie przetestowanych kompilacji produktu. W przeciwieństwie do oczekiwań wielu ludzi, końcowa kompilacja nie zawsze jest gotowym do wydania wynikiem sprintu, a właściciel produktu decyduje, która kompilacja powinna stać się kandydatem na kolejną wersję produktu. Późniejsze kompilacje nie zawsze są lepsze od wcześniejszych i może im brakować na przykład ukończonego graficznego interfejsu użytkownika albo mogą być mniej stabilne od wcześniejszych wersji. Zespół może usunąć funkcję z listy zaległości sprintu, jeśli zajmuje ona zbyt dużo czasu i albo zupełnie ją pominąć, albo ukończyć sprint z kompilacją, która nie będzie jej zawierać.

Właściciel produktu ze wsparciem zespołu podejmuje decyzję dotyczącą tego, którą kompilację wypuścić jako wersję produktu na końcu sprintu, a wybrany kandydat nie jest już rozwijany dalej. Jakikolwiek

pozostały czas może być następnie wykorzystany na uzupełnienie dokumentacji i zakończenie wszelkich zaległych testów ręcznych. Zwykle przybierają one formę testów badawczych lub opartych na sesjach (zobacz też podrozdział 6.3: Ręczne testowanie systemowe). Wszelkie wymagane naprawy błędów odbywają się równolegle. Nacisk w zadaniach realizowanych pod koniec sprintu przesuwa się z programowania i testowania nowych funkcji w kierunku testowania i naprawiania błędów.

Wersje wewnętrzne i zewnętrzne

Kandydaci na wersje niekoniecznie są udostępniani publicznie. Zespoły Scrum, które wytwarzają comiesięcznych kandydatów na wersje produktu, bardzo często akceptują i wydają właściwy produkt co trzy lub sześć miesięcy. Dzieje się tak dlatego, że wersje implementowane na zewnątrz środowiska zespołu generują dodatkową pracę, zwłaszcza dla departamentów wsparcia, marketingu i sprzedaży, które muszą aktualizować swoje systemy i działania po pojawieniu się nowej wersji produktu. Działania te muszą być brane pod uwagę w wewnętrznym przepływie zadań w firmie podczas wprowadzania metodyki Scrum do rozwijania produktu oraz przy decydowaniu, jak często wypuszczać nowe wersje. Zbyt duża liczba wersji nie zawsze jest też dobra dla klienta, ponieważ implementacja każdej nowej wersji zajmuje czas i generuje dodatkowe koszty (nie tylko w departamencie informatyki).

Gdy decyzja o dostarczeniu nowej wersji została już podjęta, trzeba zebrać razem gotowy produkt (obejmujący oprogramowanie, podręczniki, samouczki, itd.). Procesy te również mogą być w dużym stopniu zautomatyzowane wewnątrz zespołu Scrum. Wszystkie wymagane pliki są pobierane z systemu zarządzania konfiguracją i opakowywane przy użyciu skryptów. Gotowy pakiet jest następnie przekazywany na serwer, skąd może być pobierany lub wypalany na płycie DVD.

Ciągłe wdrażanie

Jeśli proces wdrażania został zautomatyzowany, to może być zawarty w środowisku ciągłej integracji projektu dając zespołowi możliwość ciągłego wdrażania. Przy założeniu, że wszystkie testy kończą się powodzeniem, oznacza to, że każda zmiana w kodzie generuje nie tylko nową kompilację, ale też nową wersję produktu, która zawiera kod, wymagane pliki konfiguracyjne i odpowiednią dokumentację.

Ponieważ nie wszystkie przypadki testowe są zawsze uruchamiane, a tworzenie dokumentacji użytkownika nie może być w pełni zautomatyzowane, nie każdy przebieg kompilacji da w efekcie wdrożenie. Jednakże codzienne wdrażanie jest realistycznym celem, zwłaszcza w środowiskach rozwijania aplikacji dla sieci Web, gdzie środowisko testowania systemowego i środowisko docelowe niewiele się różnią, a środowisko docelowe jest bezpośrednio dostępne ze środowiska ciągłej integracji. W takich warunkach produkt może być ładowany i automatycznie instalowany na zewnętrznym serwerze WWW jako ostatni krok przebiegu ciągłej integracji.

6.10 Zarządzanie testami systemowymi

Testowanie systemowe stawia zespół Scrum przed wieloma wyzwaniami. Zaczynają się one od decyzji, kiedy pisać i uruchamiać testy systemowe i trwają przez konfigurowanie środowiska testowego oraz architektury automatyzacji. Ostatnim pytaniem jest, które testy przeprowadzać jako testy jednostkowe, integracyjne i systemowe. W tradycyjnym projekcie prowadzonym według modelu V oddzielny pełnoetatowy zespół testowy jest przeznaczony do zajmowania się wszystkimi tymi zadaniami. W zespole Scrum nie powinniśmy po prostu zakładać, że zespół będzie w stanie radzić sobie z testowaniem systemowym razem z wszystkimi innymi zadaniami, którymi już się zajmuje.

Dołączanie zadań testowania systemowego do planowania sprintu

Zadania testowania systemowego muszą być aktywnie uwzględniane podczas planowania sprintu, a część zespołu trzeba jawnie przydzielić do testowania systemowego. W idealnym przypadku zespół będzie śledzić wszystkie aspekty zarządzania testami systemowymi, a jeśli tak nie jest, to zadaniem mistrza Scrum jest rozwiązanie tego problemu – na przykład przez wyznaczenie testera do roli menedżera testów. Zadaniem menedżera testów jest dbanie o to, aby zespół stawał naprzeciw wyzwaniom wymienionym powyżej i ocenianie jakości testów tworzonych przez zespół. Obejmuje to okresowe przeglądanie poszczególnych testów i wyników testów pod kątem tworzenia nowych testów lub przepisywania starych, jeśli to konieczne. Menedżer testów musi też pomagać właścicielowi produktu w interpretowaniu wyników testów i ocenianiu jakości produktu.

Wszyscy członkowie zespołu powinni przyglądać się wynikom testów przeprowadzanych podczas nocnej kompilacji, ale to rolą menedżera testów jest interpretowanie wyników i definiowanie wynikających z nich działań zespołu.

Regularna analiza błędów przed codziennym spotkaniem

Podstawowe reguły Scrum określają, że każda usterka musi być natychmiast usunięta przez programistę, choć najlepiej, aby wszystkie nowe błędy były z grubsza analizowane przed codziennym spotkaniem. Następnie zespół omawia nowe problemy, a jeśli to konieczne, tworzy nowe zadania dla wszelkich problemów, które są zbyt poważne. Błędy, które wymagają swoich własnych, nowych zadań, również muszą być wprowadzane do systemu zarządzania usterkami lub odpowiednio oznaczane wewnątrz zadania.

Jeśli zespół wykorzystuje strategię sprintu testów systemowych lub Scrum kaskadowy, to menedżer testów musi wybrać zakres testów, które mogą być przeprowadzone w czasie przydzielonym na testowanie systemowe. Jeśli obejmuje to testy ręczne, menedżer testów musi też zdecydować, które mają być uruchamiane jako testy regresji codziennie lub co tydzień. Im większy udział testów ręcznych, tym bardziej proces testowania zwinnego będzie się upodabniał do tradycyjnego

zarządzania testami. Jeśli tak jest w przypadku naszego projektu, będziemy musieli szczególne zadbać o wybranie wymaganych testów w oparciu o oszacowanie ryzyka i dokładnie przyglądać się produkowanym przez nie wynikom przy rozdzielaniu testów pomiędzy członków zespołu.

Naiwnością byłoby zakładać, że testy ręcznie nie muszą być monitorowane tylko dlatego, że przeprowadzające je osoby pracują w zespole zwinnym. Praktyki zwinne oznaczają, że każdy członek zespołu ma wiele zadań do zrealizowania, które są dużo ciekawsze od powtarzania testu ręcznego. Programowanie w parach, wstępnie zdefiniowane testy i listy kontrolne dla każdej pary oraz codzienne spotkania Scrum zapewniają, że testowanie jest możliwie wydajne, a menedżer testów może zawsze interweniować w razie konieczności.

Budowanie swojej platformy testowej krok po kroku

Uciążliwość testowania ręcznego jest dobrą zachętą do pracy nad automatyzacją testów w projekcie. Jak widzieliśmy, platformy testowe mogą być bardzo eleganckie, ale im są bardziej skomplikowane, tym większe ryzyko, że zespół poświęci więcej czasu na optymalizowanie platformy niż na szkicowanie i uruchamianie wymaganych przypadków testowych. Tak jak przy opracowywaniu produktu, gdy opracowujemy platformę testową, powinniśmy budować ją krok po kroku i dołączać tylko te składniki, których w danej chwili faktycznie potrzebujemy. Rozwijanie całej platformy z góry w niczym nie pomoże.

6.11 Pytania i ćwiczenia

6.11.1 Samoocena

Pytania i ćwiczenia pomagające w ocenie, jak zwinny jest naprawdę projekt lub zespół.

1. Czy zespół rozróżnia testy jednostkowe, integracyjne i systemowe?
2. Ile przypadków testowych mamy na każdym z tych trzech poziomów? Jak bardzo są one zautomatyzowane? Które (i ile) przypadki testowe są wbudowane w system ciągłej integracji?
3. Czy mamy wiele środowisk testowych?
4. Kiedy zespół przeprowadza testy systemowe? W osobnym sprincie testów systemowych, na końcu sprintu, czy non-stop?
5. Jak automatyzujemy swoje testy systemowe? Korzystając z rejestrowanych testów graficznego interfejsu użytkownika, czy przy użyciu skryptów?

6. Jeśli korzystamy ze skryptów, z którego podejścia zespół korzysta? Słów kluczowych? Języka specyficznego dla domeny? Testowania sterowanego zachowaniami?
7. Kiedy piszemy swoje przypadki testów systemowych? Korzystając ze sterowania testami przed implementacją funkcji? Czy później, gdy funkcja jest częścią bieżącej kompilacji?
8. Które testy niefunkcjonalne wykorzystujemy? Czy obejmują one wszystkie typy testów wymienione w ISO 25010?
9. Czy rozróżniamy testy systemowe i testy akceptacyjne?
10. Kto decyduje, która kompilacja staje się kandydatem na wersję produktu?
11. Kiedy budujemy zewnętrzną wersję produktu? Czy mamy z góry zdefiniowany harmonogram wersji?
12. Jak jest wypełniana rola menedżera testów wewnątrz zespołu? Czy mamy bezpośredniego, czy pośredniego menedżera testów? Czy ta rola została zaniedbana?

6.11.2 Metody i techniki

Te pytania pomogą w podsumowaniu treści bieżącego rozdziału.

1. Wyjaśnić relacje pomiędzy listą zaległości produktu, kryteriami akceptacji i przypadkami testów systemowych.
2. Wyjaśnić pojęcie „testowania przekrojowego".
3. Jakie są podstawowe wymagania odnośnie środowiska testowania systemowego? Jakie są różnice pomiędzy testowaniem systemowym a testowaniem jednostkowym?
4. Wyjaśnić testowanie badawcze i oparte na sesjach.
5. Co to jest test akceptacyjny?
6. Wyjaśnić wady automatyzacji testów oparte na rejestrowaniu/odtwarzaniu.
7. Jak działa test sterowany słowami kluczowymi?
8. Wyjaśnić, jak działa wielopoziomowa architektura testów. Jakie są jej główne zalety i wady?
9. Wyjaśnić podstawowe sposoby planowania czasu testowania systemowego w projekcie Scrum i wymienić zalety i wady każdego z nich.

6.11.3 Inne ćwiczenia

Te ćwiczenia pomogą zagłębić się w zagadnienia poruszone w trakcie tego rozdziału.

1. W odniesieniu do studium przypadku 3-2 na stronie 27, które interfejsy na diagramie architektury eHome są używane w przykładach testów systemowych eHome opisanych w tym rozdziale?
2. Naszkicować przypadki testów systemowych opartych na słowach kluczowych dla wpisu „programowanie przełączników" na liście zaległości w podrozdziale 3.3, korzystając z odpowiednich kryteriów akceptacji wymienionych w ćwiczeniu 3.9.3–2.
3. Naszkicować te same przypadki testowe jako testy sterowane danymi i wpisać odpowiednie dane testowe w tabeli.
4. Które dodatkowe przypadki testowe są nam potrzebne, jeśli funkcja automatycznej odpowiedzi programu może być zdefiniowana przy użyciu pojęcia względnego takiego jak „+2 godziny"? Zastosować sterowanie testami do zdefiniowania żądanego zachowania programu.

7 Zarządzanie jakością i zapewnianie jakości

Tradycyjne zarządzanie jakością jest w dużym stopniu sterowane dokumentami, podczas gdy praktyki zwinne wymagają ograniczenia dokumentacji do minimum. Jaki wpływ ma to na pracę menedżera jakości lub departamentu zapewnienia jakości? W jaki sposób specjaliści od zapewniania jakości mogą korzystać ze swojej wiedzy, żeby przyczyniać się do powodzenia zespołu zwinnego? Jaki jest najlepszy sposób radzenia sobie z wymogami regulacyjnymi i audytami zewnętrznymi? Ten rozdział dostarcza odpowiedzi na wszystkie te pytania.

7.1 Tradycyjne zarządzanie jakością

7.1.1 Norma ISO 9000

System zarządzania jakością (QM – quality management) składa się z reguł procesowych, według których działa przedsiębiorstwo, dokumentacji tych reguł (opisy procesów, instrukcje proceduralne i wykonawcze, najlepsze praktyki i wytyczne) oraz mechanizmów zaprojektowanych do monitorowania i stałego poprawiania samego systemu QM. Działania operacyjne, które stosują te reguły i sprawdzają, że wymagania jakościowe dla produktu są podtrzymywane, są nazywane zapewnianiem jakości (QA – quality assurance) [zobacz podrozdział 7.4].

Wizualizowanie procesów biznesowych

Dokumentacja systemu zarządzania jakością obejmuje wizualizację obowiązków i wszystkich procesów biznesowych, które są związane ze spełnianiem celów jakościowych przedsiębiorstwa. Zwykle obejmuje to programowanie/prace inżynierskie, zamówienia publiczne/zakupy, produkcję i wsparcie/obsługę oraz regulowanie wszystkich procedur zapewniania jakości, które mają miejsce w ramach tych procesów, a także całościowy system zarządzania jakością, ze wszystkimi zasadami jakościowymi, celami, obowiązkami i procesami optymalizacyjnymi.

ISO 9000

Większość dużych organizacji przestrzega wytycznych zarządzania jakością przedłożonych w rodzinie norm [ISO 9000]. [ISO 9001] definiuje minimalne wymagania systemu zarządzania jakością i wymienia te zagadnienia, dla których istnieją reguły i muszą być dokumentowane. Normy te rozwinęły się z pojęć zapewniania jakości stosowanych przez przemysł wytwórczy. Wersja ISO 9001 z 1994 roku definiuje 20 podstawowych elementów zarządzania jakością, które odzwierciedlają procesy związane z wytwarzaniem, od zadań projektowych i inżynieryjnych, poprzez produkcję i kontrolę produktów niespełniających norm, aż po montaż i obsługę klienta. Szeroka interpretacja tych elementów była konieczna, gdy zostały one wykorzystane do reprezentowania procesów stosowanych przez branże usługowe i firmy rozwijające oprogramowanie. Norma ISO 9001:2000 była wersją bardziej zorientowaną na procesy i rozróżnia pomiędzy procesami zarządzającymi (tzn. podejmowaniem decyzji i rolami prawnymi), tworzeniem wartości (projektowanie, produkcja, usługi i wsparcie) oraz (wewnętrznymi) procesami wspierającymi, takimi jak księgowość, zamówienia publiczne/zakupy i zarządzanie łańcuchem dostaw. System zarządzania jakością musi obejmować obowiązki i procesy rządzące wszystkimi tymi działaniami.

Kategoria, do której przypisane jest tworzenie oprogramowania, zależy od jego znaczenia wewnątrz firmy. Firma tworząca oprogramowania najprawdopodobniej przypisze swoją główną aktywność do tworzenia wartości, natomiast firma handlowa najprawdopodobniej potraktuje to jako proces wsparcia zapewniający niezawodne utrzymywanie systemów informatycznych. Scentralizowane zarządzanie konfiguracją, działania związane z zarządzaniem jakością i zarządzanie personelem są zwykle również klasyfikowane jako procesy wspierające.

7.1.2 Zasady PDCA

System zarządzania jakością ISO 9000 jest oparty na cyklu PDCA [ISO 9001, podrozdział 0.2], znanym też jako cykl Deminga [URL: PDCA]:

- **P**lan (planowanie): Proces jest planowany (tzn. definiowany) w odniesieniu do celów, wymaganych kroków, obowiązków, wymagań wstępnych, zasobów i żądanych wyników.
- **D**o (wykonanie): Wstępnie zdefiniowany proces jest przekuwany na praktykę i generuje oczekiwane wyniki lub zaplanowane produkty.
- **C**heck (sprawdzenie): Wyniki procesu są porównywane z wstępnie zdefiniowanymi wymaganiami i tworzone są raporty z tymi sprawdzeniami.
- **A**ct (działanie): Opracowywane są działania naprawcze związane z istotnymi różnicami pomiędzy faktycznymi a zaplanowanymi wynikami i mające na celu stałe poprawianie jakości.

Cykl PDCA na różnych poziomach

Cykle PDCA mogą być wykonywane na różnych poziomach wewnątrz organizacji:

- **Poziom operacyjny** Proces lub działanie jest przeprowadzane (rutynowo) zgodnie z wstępnie zdefiniowanymi wytycznymi. Wyniki są generowane, sprawdzane, poprawiane w razie potrzeby i dostarczane.
- **Poziom zarządzający** Proces jest przerabiany lub całkowicie przedefiniowywany (na przykład podczas reorganizacji firmy) i jest implementowany na przykład jako część projektu pilotażowego. Monitorowana jest skuteczność i dopuszczalność procesu i w razie konieczności jest on dostosowywany i optymalizowany. Po tym następuje certyfikacja użycia poza projektem pilotażowym.
- **Poziom systemu zarządzania jakością** Skuteczność każdego procesu zarządzania jakością jest regularnie sprawdzana przy pomocy danych referencyjnych, raportów o stanie, audytów wewnętrznych i zewnętrznych oraz innych narzędzi, takich jak zbieranie sugestii wewnątrz firmy albo reakcji zwrotnych od klientów. Procesy nieodpowiadające wymaganiom są przerabiane i poprawiane. Cykl PDCA generuje w ten sposób ciągły proces poprawiania jakości.

7.1.3 Mocne i słabe strony

Zarządzanie jakością oparte na ISO 9000 okazało się bardzo skuteczne. Wspiera standaryzację w firmie, motywuje zaangażowane osoby do trzymania się swoich obowiązków i sprawia, że firma jest bardziej przejrzysta dla pracowników i klientów. Zapewnia też wbudowany interfejs do audytów, co ułatwia stronom trzecim uzyskanie szczegółowego przeglądu działania firmy. Może to służyć budowaniu zaufania i wywieraniu presji.

Do wad zarządzania jakością opartego na ISO 9000 należą[1]:

- **Jego zakres** ISO 9000 wymaga opisania (wyspecyfikowania) procesów biznesowych związanych z zarządzaniem jakością jako części dokumentacji zarządzania jakością. Zakres dokumentacji zarządzania jakością zależy od rozmiaru organizacji, typu pracy, w jaką jest zaangażowana, złożoności procesów i kompetencji pracowników [zobacz ISO 9001, dział 4.2.1]. Procesy są jednak często opisywane zbyt szczegółowo, co może pomagać osobom niezaznajomionym z operacyjnymi aspektami procesu w zrozumieniu ich

1 Niektóre z nich były opisane dokładnie w kontekście zarządzania projektem w rozdziale 2.

bez dodatkowych instrukcji. Ponadto audyty procesów muszą być w stanie zapewnić, że proces jest przeprowadzany zgodnie z opisem, co może prowadzić do tworzenia dzienników procesowych i innych zapisów, które nie przyczyniają się bezpośrednio do generowania wartości, a służą jedynie udowadnianiu (wewnętrznym) audytorom, że proces jest przeprowadzany poprawnie. Wszystko to może powodować powstawanie ogromnych zbiorów dokumentów zarządzania jakością obejmujących podręczniki zarządzania jakością, opisy każdego procesu, różne wytyczne dotyczące dostosowywania opisów procesów do ograniczeń projektowych, szablonów, itd.

- **Oporność na zmiany** Zmienianie lub aktualizowanie zawartości poszczególnych kroków procesowych może być dość czasochłonne. Modyfikacja danego kroku może wymagać zmian w innych miejscach w hierarchii dokumentów, które trzeba całkowicie zaktualizować. Dodatkowo systemy zarządzania jakością zwykle odnoszą się do całej firmy, więc dokonywanie zmian wymaga uzgodnień z wieloma osobami i grupami rozproszonymi po całej organizacji, co może zniechęcać do przeprowadzania zmian. Nawet jeśli znajdziemy wystarczającą liczbę zwolenników zmian i uzgodnimy wymagane zmiany zawartości, to wszelkie modyfikacje i tak muszą przejść przez formalne procedury akceptacyjne. Konieczne zmiany są często ignorowane lub przesuwane w czasie, żeby uniknąć całego tego wysiłku.
- **Nieaktualność** Jeśli dokumentacja nie jest wystarczająco często aktualizowana, rzeczywiste procesy wcześniej lub później będą różnić się od swoich odpowiedników na piśmie, co utrudni proces ich uzgadniania z dokumentacją. Ponieważ procesy rzeczywiste będą przebiegać inaczej od sposobu, w jaki są opisane, nie można pokazać ich zgodności z wytycznymi zarządzania jakością i podczas następnego audytu pojawią się rozbieżności.
- **Ryzyko systemu równoległego** Aby uniknąć rozbieżności podczas audytu, kuszące jest tworzenie dzienników wymaganych przez system zarządzania jakością na zasadzie pro forma. Jeśli na to zezwolimy, firma szybko skończy z systemem zarządzania jakością na papierze, który niewiele będzie miał wspólnego z procesami przeprowadzanymi w rzeczywistości. Jeśli system zarządzania jakością jest zbyt teoretyczny albo jeśli jego reguły nie są odpowiednio przyjęte (lub spotykają się z niewielką akceptacją), może to prowadzić do rozbieżności między teorią a rzeczywistością, które uniemożliwią osiągnięcie celów systemu.

7.1.4 Modelowanie procesów a rozwój oprogramowania

Gdy procesy tworzenia oprogramowania są definiowane przy użyciu systemu zarządzania jakością, podkreśla to złożoność procesu i trudności związane z jego modelowaniem. Formalne rozróżnienie pomiędzy poszczególnymi fazami lub krokami jest niezwykle trudne – lub wręcz niemożliwe.

Nawet jeśli proces jest definiowany tylko w formie ogólnie zarysowanych faz, to wystarczająco trudno jest definiować kryteria, które dokładnie opisują początek, koniec lub wyniki tych faz (zobacz podrozdział 2.3, rys. 2-3). Pytanie, czy dany kamień milowy został osiągnięty, jest kwestią interpretacji. Stworzenie dokładniejszej definicji procesu również nie pomaga, a jedynie służy utworzeniu rosnącej liczby przypadków specjalnych, wyjątków i szablonów dokumentów. Innymi słowy, nie jest możliwe dojście do formalnego wniosku dotyczącego tego, czy krok programowania zdefiniowany w modelu został produktywnie i poprawnie zakończony. Nie wynika to tylko z tego, że tworzenie oprogramowania jest procesem niedeterministycznym i empirycznym, ale też ze względu na to, że sekwencja produktów pośrednich (wymagania, specyfikacje zgrubne, specyfikacje dokładne, kod) wytwarzanych przez ten proces nie może być formalnie przypisywana do poszczególnych elementów niezależnie od tego, jak ścisłe metody tworzenia oprogramowania lub narzędzia informatyczne zastosujemy.

Przyjrzyjmy się przykładowi. Zakładając, że chcemy rozróżniać pomiędzy fazami zgrubnych i dokładnych specyfikacji produktu w modelu procesu, departament zarządzania jakością mógłby nakreślić dwa przykładowe szablony korzystając z arbitralnie wybranych struktur i granic pomiędzy obiema fazami. Jednakże to podejście nie gwarantuje, że prawidłowo wypełniona, dokładna specyfikacja opisuje planowany produkt dokładniej lub bardziej realistycznie, niż prawidłowo wypełniona zgrubna specyfikacja. Zespół projektu mógłby po prostu zostawić na boku zgrubną specyfikację, ale nadal utworzyć przydatną specyfikację szczegółową lub nawet zacząć kodowanie bez korzystania z żadnych specyfikacji (choć to ostatnie podejście prawdopodobnie nie da w wyniku dobrego produktu). Z drugiej strony możliwe jest też, że zespół zupełnie minie się z celem, jeśli będzie pracować z szablonami, które są kompleksowe, ale wypełnione słabą zawartością.

Jedynym sposobem na zadowalające rozwiązanie tego problemu jest uzgodnienie z góry żądanych wyników każdej fazy i sprawdzanie każdej z nich. W tym przypadku szczegółowy opis procesu nie stanowi lepszej pomocy, niż ogólnie nakreślony model faz oparty na słowach kluczowych.

7.2 Zwinne zarządzanie jakością

Słabości konwencjonalnych systemów zarządzania jakością wymienione w poprzednim podrozdziale wyjaśniają, dlaczego niektóre zespoły zwinne wręcz odrzucają dokumentację procesów i systemy zarządzania jakością. Takie dogmatyczne podejście często prowadzi do przeciwnej skrajności, tworząc projekty lub zespoły bez zapisanych reguł. Zwykłe procedury firmy są odrzucane na podstawie tego, że nie są zwinne; ale nie tworzy się nowych, lepszych reguł, aby je zastąpić, ponieważ dokumentacja jest traktowana jako bezproduktywna strata. W niektórych przypadkach argument straty jest przydatnym usprawiedliwieniem ignorowania zwykłych reguł i unikania jakiejkolwiek dokumentacji produktu lub zapewniania jakości.

Zwinne procesy tworzenia oprogramowania również wymagają dokumentacji.

Choć nie jest to popularne wśród niektórych członków zespołów, zwinne procesy tworzenia oprogramowania również wymagają dokumentacji, aby były odpowiednio definiowane. Przy prawidłowej interpretacji metodyka zwinna nie odrzuca reguł – zespoły zwinne również potrzebują reguł! Jednakże wszelkie reguły muszą służyć zespołowi, a nie na odwrót[2] i zawsze lepiej trzymać się kilku reguł, niż całkowicie ignorować złożony system reguł, który istnieje tylko na papierze. System zarządzania jakością zaprojektowany do wspierania zasad manifestu Agile [URL: Agile Manifesto] i mający zyskać aprobatę zespołów zwinnych musi być szczupły. Innymi słowy musi być prosty, łatwy w utrzymaniu i zawsze aktualny.

Jeśli firma korzysta już z systemu zarządzania jakością opartego na ISO 9000 (co zwykle będzie prawdą), zwinny proces tworzenia oprogramowania można dołączyć do istniejącej dokumentacji w postaci dodatkowego procesu. Stopień, w jakim zespoły zwinne akceptują to podejście, będzie zależeć od stopnia, w jakim istniejący system zarządzania jakością cierpi z powodu ograniczeń wymienionych w podrozdziale 7.1.3. Zwykle lepiej przeprowadzić rozległą przebudowę istniejącej dokumentacji i kultury zarządzania jakością.

7.2.1 Upraszczanie dokumentacji zarządzania jakością

Zespół musi jasno określić, którą metodykę zwinną stosuje (Scrum, Kanban lub inne), ale nie musi opisywać tego szczegółowo. Wystarczy wymienić wykorzystywane metody, techniki i środki oraz nakreślić, jak są one implementowane na poziomie firmy lub zespołu. Powinno to obejmować formę, położenie i sposób ustalania priorytetów dla listy zaległości, typ i częstotliwość spotkań, podstawowy kształt procesu

2 Manifest Agile mówi: „osoby i interakcje ponad procesami i narzędziami" [URL: Agile Manifesto].

ciągłej integracji, zasady sterowania testami, położenie zapisanych przypadków testowych, itd. Wszystkie te szczegóły mogą odwoływać się do odpowiedniej literatury, podręczników lub wykorzystywanych narzędzi oraz elementów tworzonych przez sam projekt.

Zamiast tworzyć złożony model procesu, wystarczy prosty diagram podobny do pokazanego na rys. 2-1, aby zilustrować podstawową formę projektu. Zamiast szczegółowo przedstawiać złożoną macierz procesów i kroków procesów, ilustracja ta pokazuje wymagane narzędzia, takie jak lista zaległości, codzienne spotkanie Scrum, definicja gotowości, itd. Sposób realizacji projektu jest naturalną konsekwencją zastosowania technik, takich jak uszczegóławianie listy zaległości, poker planistyczny, sterowanie testami, itd. Ponieważ są to warunki wstępne tej metodyki, opis procesu nie musi osobno opisywać tych technik. Opis procesu jest ograniczony do długości kilku stron.

Trzeba uczyć poszczególnych technik.

Nie należy zapominać o szkoleniu personelu w zakresie odpowiednich technik! To że system zarządzania jakością pośrednio wymaga odpowiedniej wiedzy, nie oznacza, że zaangażowani członkowie zespołu są na bieżąco z wymaganą metodyką. Szkolenia bieżące i zaawansowane plany szkoleniowe muszą obejmować literaturę, kursy, doradztwo i zewnętrzne konferencje związane z technikami wykorzystywanymi przez zespół.

Zespoły zwinne często opisują reguły, według których pracują w samodzielnie napisanej karcie zespołu (zobacz podrozdział 3.6). Ten dokument ma znaczenie tylko wewnątrz zespołu. Karta zespołu jest zwykle dość zwarta i dlatego stanowi dobry punkt wyjścia do opisu zwinnego procesu tworzenia oprogramowania w systemie zarządzania jakością. Ponieważ jednak dokumentacja systemu zarządzania jakością jest przeznaczona dla większego kręgu czytelników (inne zespoły, zarząd, zewnętrzni audytorzy, a nawet klienci), zwykle trzeba zmienić hermetyczny język i specjalną wiedzę zawarte w karcie zespołu, zanim zostaną zaadaptowane jako część dokumentacji zarządzania jakością.

Karta zespołu dokumentuje implementację specyficzną dla zespołu.

Jeśli wiele zespołów Scrum pracuje nad projektem, proces tworzenia powinien być opisany w formie ogólnej tak, aby odnosił się do wszystkich zespołów, natomiast precyzyjna definicja pozostaje zadaniem każdego zespołu. Tablica zadań jest typowym przykładem takiego punktu. System zarządzania jakością (albo firmowy proces zwinny) zwykle będzie wymagać, aby każdy zespół miał swoją własną tablicę zadań. Jednakże od danego zespołu zależy to, czy będzie ona prowadzona elektronicznie, czy w postaci tablicy magnetycznej (lub czegoś podobnego). Karta zespołu jest dokumentem, w którym rejestrowana jest każda implementacja specyficzna dla zespołu.

Podczas integrowania metodyki zwinnej z systemem zarządzania jakością musimy decydować, które z istniejących praktyk zarządzania

jakością (wymaganie planowania projektów, wymagane metryki, dokumentacja wyników testów, wykorzystywane narzędzia programistyczne, itd.) powinny być utrzymywane w nowym środowisku zwinnym. Jeśli wszystkie istniejące praktyki zostaną wyrzucone za burtę, ryzykujemy, że cały system zarządzania jakością stanie się przestarzały, ale jeśli zachowane zostaną wszystkie istniejące standardy, system zarządzania jakością będzie utrudniał podejście zwinne lub nawet sprawi, że (formalnie) niemożliwe będzie jego wdrożenie. Odpowiednie reguły (na przykład dotyczące wykorzystywanych narzędzi) powinny być więc sformułowane jako zalecenia. Zasady zwinne pozostawiają zespołowi ustalenie formy procesu tworzenia oprogramowania. Ani dział zarządzania jakością, ani mistrz Scrum, nie mogą dawać zespołowi jednostronnych instrukcji, a mają jedynie doradzać i tworzyć zalecenia.

Ten proces zmiany nie polega tylko na aktualizacji dokumentacji, ale też na przyjęciu i uwzględnieniu praktyk zwinnych przez organizację. W miejscach, gdzie muszą być spełnione zewnętrzne reguły zgodności (zobacz podrozdział 7.3), musimy jasno przekazać zespołom zwinnym, że muszą i tak przestrzegać systemu zarządzania jakością i kilku niezwinnych reguł.

7.2.2 Zmienianie kultury zarządzania jakością

Konwencjonalne zarządzanie jakością jest procesem przebiegającym od góry do dołu. Centralny zespół zarządzania jakością[3] odpowiada za dokumentację, a tam gdzie ma to sens, pojawiają się procesy standaryzacyjne. Zespół tworzy podręczniki procesowe i rozwija ustandaryzowane procesy dla wszystkich działów. Z kolei metodologia zwinna jest procesem przebiegającym z dołu do góry. Zespoły zwinne opracowują swoje własne metody pracy, więc pracownicy zarządzania jakością muszą nauczyć się do nich dostosowywać.

Zespół, który jest przyzwyczajony do unifikowania procesów, musi przekształcić się w grupę pomagającą innym zespołom w dokumentowaniu ich własnych procesów. W ten sposób centralna jednostka wyznaczająca standardy staje się grupą zachęcającą do wymiany informacji pomiędzy zespołami i reklamującą dobrze sprawdzone metody. Zespół zarządzania jakością musi też proponować alternatywy dla metod pracy, które nie funkcjonują zgodnie z planem.

---

3 W niewielkich i średniej wielkości firmach będzie to zwykle dedykowany pracownik i kilku pomocniczych pracowników, natomiast duże organizacje (i jednostki produkcyjne związane z bezpieczeństwem) mają zwykle osobny departament zarządzania jakością. Dla uproszczenia wszystkie te różne formy będą w tym rozdziale nazywane pracownikami zarządzania jakością lub zespołem zarządzania jakością.

Zespół zarządzania jakością staje się grupą usługową.

Zespół zarządzania jakością staje się mentorem i grupą zapewniającą usługi. Zwinne zespoły są postrzegane jako klienci, którzy mogą korzystać z usług tej grupy. Jeśli na przykład zespół chce poprawić swój proces ciągłej integracji przez wprowadzenie nowego narzędzia ciągłej integracji, może poprosić, aby zespół zarządzania jakością poszukał narzędzia spełniającego wymagania zespołu i go zaimplementował. W międzyczasie zespół produkcyjny może skoncentrować się na swojej codziennej pracy. W ten sposób optymalizacja procesów jest inicjowana przez zespół produkcyjny, ale ich implementacja jest delegowana do zespołu zarządzania jakością.

Jeśli zespół zarządzania jakością ma służyć wielu zwinnym zespołom, to musi wyważyć swoje działania, aby wspierać interesy różnych zespołów względem siebie i w kontekście całej organizacji. Nadal jednak to zespół produkcyjny sam decyduje, które rekomendacje zespołu zarządzania jakością przyjmie, nawet jeśli oznacza to, że nie można całkiem przyjąć planowanej standaryzacji. Zespół zarządzania jakością musi stawić czoło temu testowi i w razie odrzucenia którejś ze swoich rekomendacji musi zastanowić się, dlaczego i co można by zrobić, aby poprawić sytuację.

Pracownicy zarządzania jakością jako zespół zwinny

Zespół zarządzania jakością powinien korzystać z okazji, aby zorganizować się jako zespół zwinny – być może wykorzystujący metodykę Kanban. Praca w ten sposób sprawi, że zespół zarządzania jakością będzie szybszy, elastyczniejszy i sprawniejszy, jak również poprawi swoje notowania wśród zwinnych zespołów projektowych. Podobnie do zespołu produkcyjnego zespół zarządzania jakością powinien być w stanie dostarczać gotowy produkt – na przykład w formie opisu procesu, który jest w 100 procentach do przyjęcia przez wszystkie zespoły zwinne albo bezproblemowego wprowadzenia sprawdzonego narzędzia.

Zespół zarządzania jakością a mistrz Scrum

Nie jest niespodzianką, że niektóre z zadań przypisanych do zespołu zarządzania jakością są podobne do zadań mistrza Scrum. Obie strony są odpowiedzialne za zapewnienie stosowania się do wstępnie zdefiniowanych reguł – zespół zarządzania jakością przez implementację ISO 9000 na poziomie przedsiębiorstwa, a mistrz Scrum przez użycie praktyk Scrum w ramach danego zespołu.

Zmiany, które następują, gdy konwencjonalny system zarządzania jakością jest przekształcany w zwinny system zarządzania jakością, mogą ograniczać te różnice albo nawet mogą doprowadzać do ich całkowitego zniknięcia. Zespół zarządzania jakością zapewnia obsługę wszystkim zespołom, natomiast mistrz Scrum musi mieć na uwadze powodzenie nie tylko swojego zespołu, ale też całej organizacji – w obu przypadkach wykorzystywany jest ten sam zwinny system zarządzania jakością. Ten proces przejścia może prowadzić do zintegrowania pracowników zarządzania jakością i mistrzów Scrum w firmie w jeden,

wielofunkcjonalny, zwinny zespół zarządzania jakością, który obejmuje całą wiedzę konieczną do metodycznego i skutecznego wspierania pracy zespołów Scrum. Doświadczenie to obejmuje wszystkie standardy wykorzystywane przez organizację, narzędzia do modelowania procesów oraz wyszkolonych audytorów, specjalistów i szkoleniowców praktyk zwinnych, a także oczywiście mistrzów Scrum.

7.2.3 Retrospektywy i poprawianie procesów

Jednym z centralnych elementów normy ISO 9000 jest wymaganie stałej poprawy, a retrospektywa sprintu (zobacz [URL: Scrum Guide]), która jest wbudowana w metodykę Scrum, jest skutecznym narzędziem, aby to osiągnąć.

Retrospektywa sprintu jest spotkaniem zespołu, które odbywa się po każdym sprincie[4] i służy zbieraniu pomysłów zespołu dotyczących tego, co można by zrobić lepiej jako zespół (a nie, jak poprawić produkt)[5]. Wynikające z tego wnioski mają nadawane priorytety przed dodaniem ich do listy zaległości produktowych zespołu. Wszelkie elementy, które są postrzegane jako poważne przeszkody, mogą być przekazane bezpośrednio mistrzowi Scrum, który może je wprowadzić do listy zaległości blokujących. Podejście do propozycji usprawnień oraz implementacja nowych procesów i technik jest podobna do codziennej pracy zespołu zarządzanej przez listę zaległości.

Poprawianie procesów jest częścią codziennych zadań.

Poprawianie procesów nie jest więc niczym specjalnym dla zespołu zwinnego i jest częścią codziennych zadań. Zintegrowanie poprawiania procesów z procesem planowania sprintów zapewnia więc, że zespół ma wystarczająco dużo czasu, żeby zapanować nad nowymi metodami[6] i zapewnia też, że same poprawki zostaną wprowadzone w życie podczas planowanego sprintu (innymi słowy zajmujemy się tym teraz, a nie wtedy, kiedy będziemy mieć czas). Ponieważ zadania związane z poprawianiem procesów konkurują o zainteresowanie zespołu z zadaniami programistycznymi, poprawianie procesów zostanie szybko ograniczone do wypełniania tylko tych zadań, które można realistycznie zaimplementować. W ten sposób poprawianie procesów

4 [URL: Scrum Guide] podaje „po przeglądzie sprintu i przed następnym spotkaniem planowania sprintu" jako właściwy moment na przeprowadzenie retrospektywy sprintu, co zwykle oznacza co miesiąc. Jednakże zespół Scrum może wybrać inny harmonogram – co trzeci Sprint może wystarczać w przypadku zaawansowanego zespołu. XP [Beck/Andres 04] na przykład proponuje „cykl kwartalny" dla tego typu spotkań.

5 Spotkanie przeglądu sprintu (albo demonstracja sprintu) jest właściwym forum do omówienia, jak poprawić produkt.

6 Oczywiście to działa tylko pod warunkiem, że szacowanie nakładów i planowanie sprintu mają równy priorytet.

staje się ciągłym, krótkoterminowym ćwiczeniem, które nie zajmuje zespołowi dużo czasu.

Podczas kolejnego sprintu zespół może omówić z mistrzem Scrum, czy zaimplementowana poprawka się przyjęła i czy spełnione zostały kryteria akceptacji dla zadania związanego z tą poprawką. Jeśli poprawka wpływa na inne zespoły lub globalny system zarządzania jakością, sensowne jest włączenie zespołu zarządzania jakością do dyskusji, żeby wspierać bezpośrednią wymianę informacji na poziomie zarządzania jakością (zobacz podrozdział 7.2.2).

7.3 Radzenie sobie z wymaganiami dotyczącymi zgodności

Większość firm, które rozwijają oprogramowanie, musi stosować się do narzuconych z zewnątrz wymagań dotyczących zgodności. Większość dużych organizacji wymaga, aby dostawcy udowadniali, że korzystają z systemu zarządzania jakością z certyfikatem ISO 9001. Producenci elementów i systemów elektronicznych, w których oprogramowanie odgrywa rolę związaną z bezpieczeństwem, muszą stosować się do założeń normy IEC 61508-3. Przemysł samochodowy, sprzętu medycznego, kolejowy i lotniczy również musi stosować się do różnych norm przemysłowych[7]. Poniższe podrozdziały zajmują się kwestią tego, czy firmy rozwijające oprogramowanie przy użyciu metodyki zwinnej mogą nadal stosować się do tego typu norm i wymagań dotyczących zgodności.

7.3.1 Wymagania odnośnie procesów tworzenia oprogramowania

Żadna z norm, które wymieniliśmy, nie wymaga użycia tradycyjnego modelu procesu tworzenia oprogramowania takiego jak model V. Jednakże wszystkie one wymagają, aby tworzenie oprogramowania było dobrze zdefiniowane jako proces inżynierski związany z jakością. Oznacza to, że procesy tworzenia oprogramowania muszą być udokumentowane tak, aby projekty programistyczne mogły być traktowane jako działające zgodnie z wstępnie zdefiniowanymi regułami. Wprowadzenie praktyk zwinnych jako nowego lub dodatkowego filaru systemu zarządzania jakością sprawia, że to wymaganie jest łatwe do

7 Na przykład ISO 26262 (Automotive Safety Lifecycle with Automotive Safety Integrity Level (ASIL), IEC 62304 (Medical Device – Software Life Cycle), regulacje U.S.: Food and Drug Administration (FDA), EN 50128 (Railway applications – Software for railway control and protection systems) lub DO 178B (Software Considerations in Airborne Systems and Equipment Certification).

spełnienia, a podrozdział 7.2.1 zapewnia wskazówki, jak to zrobić. Może to mieć wpływ również na inne procesy biznesowe w zależności od tego, które części organizacji stosują praktyki zwinne.

Niektóre normy definiują konkretne wymagania dla pewnych działań i kroków procesu (np. definicja klas zabezpieczeń w IEC [IEC 62304] i różne poziomy funkcjonalnych elementów zabezpieczeń w [ISO 26262]) w połączeniu z wymaganiami na poziomie projektu oprogramowania i metod programistycznych takich jak testowanie oprogramowania (zobacz IEC 61508-3, podrozdział 7.4). W tym przypadku proces rozwoju oprogramowania musi zawierać pewne kroki, które dotyczą wymagań i je niezawodnie implementują. We wspomnianych wyżej przykładach potrzebna jest klasyfikacja ze względu na zalecane poziomy zabezpieczeń, a wynikowy proces musi zapewniać, że elementy oprogramowania są opracowywane i sprawdzane przy użyciu wynikających z tego wytycznych dla określonego poziomu.

Wymagania ustalane na podstawie kryteriów gotowości

Konkretne wymagania dla poszczególnych działań programistycznych mogą po prostu wynikać z odpowiednio sformułowanych definicji gotowości poszczególnych elementów. Jeśli przykładowo ma być zaimplementowana funkcja, która ma wysoką klasyfikację zabezpieczeń, to wymagane pokrycie testami jest używane jako kryterium gotowości dla zadania implementacyjnego. Do tej pory wszystko wydaje się proste. Sprawy się bardziej komplikują, jeśli chcemy skorzystać z procesu do zapewnienia, żeby klasyfikacja zabezpieczeń dla każdej implementacji zadania w każdym sprincie była zawsze analizowana, a analiza wpływu była używana do definiowania odpowiedniego kryterium gotowości. Pomocne mogą być listy kontrolne (ale tylko jeśli będą używane sumiennie) oraz wcześniej drukowane karty zadań, które zawierają ogólne kryteria gotowości (takie jak limity pokrycia). Alternatywnie krytyczne kroki procesowe, których nie można pominąć (takie jak „przeprowadzenie analizy wpływu"), są reprezentowane przez osobną kolumnę na tablicy zadań. Sposób rozwiązania tego z punktu widzenia procesu jest jednym z elementów, które trzeba zawrzeć w opisie procesu systemu zarządzania jakością, a wszystkie zespoły powinny tego ściśle przestrzegać.

7.3.2 Wymagania identyfikowalności

Gdy opracowywane są produkty związane z bezpieczeństwem, odpowiednie normy wymagają, aby decyzje projektowe i zmiany produktu były możliwe do odtworzenia i śledzenia nawet po zakończeniu procesu ich opracowania. Bezpieczeństwo produktu[8] musi być jasno

8 To znaczy zmniejszenie ryzyka dla użytkownika i środowiska.

zdefiniowane w jego atrybutach projektowych i niezawodności implementowanej podczas procesu opracowywania. Odpowiednie testy są następnie używane do sprawdzenia, że produkt i jego funkcje zabezpieczające działają zgodnie z planem.

W praktyce oznacza to, że każdy element oprogramowania musi być całkowicie identyfikowalny – tzn. które wymagania produktu spełnia, w jaki sposób jego specyfikacja zapewnia spełnienie tych wymagań i które testy (oraz wyniki testów) dowodzą, że jego funkcjonalność autentycznie spełnia swój cel. Późniejsze zmiany projektu lub implementacji produktu nie mogą ograniczać albo w inny sposób wpływać na planowany poziom bezpieczeństwa. Oznacza to, że trzeba przeprowadzić analizę wpływu przed dokonaniem zmian i potrzebne są sugestie, jak przeciwdziałać potencjalnemu ograniczeniu bezpieczeństwa. Każdy etap każdej zmiany musi być też identyfikowalny od momentu jego zaproponowania poprzez wyniki analizy wpływu aż do wyników zastosowania zmiany wobec produktu.

Identyfikowalność

Ponieważ projekty zwinne wiążą się ze stałą zmianą, nie jest łatwo zagwarantować identyfikowalność wewnątrz procesu zwinnego. Jeśli mechanizm identyfikowalności jest skomplikowany lub zawodny, to szybko spowoduje wytwarzanie dużej ilości dokumentacji lub powstawanie luk. Zespół musi też wziąć pod uwagę poziom szczegółowości, na którym gwarantowana ma być identyfikowalność. Minimalnym wymaganiem jest możliwość śledzenia połączeń pomiędzy wymaganiami produktowymi a przypadkami testów systemowych, co umożliwi korzystanie z zakończonych powodzeniem testów do udowadniania, że wymagania zostały spełnione[9]. Aby proces ten się powiódł, musimy być w stanie uzasadnić, dlaczego jeden lub kilka przypadków testów systemowych może sprawdzić spełnienie wymagania. Oznacza to, że trzeba sprawdzać adekwatność przypadków testów systemowych. Jeśli usterki będą ujawniane, konieczna będzie niezawodna analiza i proces poprawiania (zwykle kierowany przez system zarządzania usterkami). Aby prześledzić usterkę z powrotem do jej źródła, komunikat o usterce musi być połączony z testem, który ją spowodował. W praktyce niezawodna identyfikowalność może być gwarantowana tylko wtedy, jeśli korzystamy z odpowiednio połączonego wymagania i narzędzi do zarządzania testami.

Na pierwszy rzut oka może się wydawać korzystne, że ten poziom formalizmu jest wymagany jedynie w przypadku zewnętrznej wersji produktu. Ponieważ jednak zespół zwinny często decyduje dopiero na końcu sprintu, czy wynikowy produkt ma być używany na zewnątrz, nie jest to zbyt pomocne. Dokumentowanie danych o identyfikowalności

9 Ten proces jest odpowiednikiem kroku sprawdzania poprawności w modelu V.

po fakcie nie jest zalecane i ważne jest zapewnianie, żeby kompilacje generowane bez użycia działającego mechanizmu identyfikowalności nie były wersjami końcowymi. Zawsze lepiej, aby mechanizm identyfikowalności był standardową częścią każdego sprintu.

To, czy identyfikowalność na poziomie testowania wymagań jest wystarczająca i czy należałoby włączyć bardziej szczegółowe elementy, zależy od poziomu, na którym podejmowane są decyzje projektowe[10] i poziomu architektury oprogramowania, na którym obserwowana jest dana funkcjonalność. Oznacza to, że w niektórych przypadkach proces opracowywania oprogramowania musi być identyfikowalny aż do poziomu testowania jednostkowego.

7.3.3 Wymagania dotyczące atrybutów produktu

Sprawy dotyczące zgodności oprogramowania nie wpływają tylko na działania związane z tworzeniem oprogramowania. Użycie oprogramowania jako części infrastruktury informatycznej organizacji też może być uzależnione od wymagań dotyczących zgodności, takich jak ochrona i archiwizowanie danych (w tym przechowywanie i odzyskiwanie danych). W przypadku branż, które tworzą produkty związane z bezpieczeństwem i oparte na oprogramowaniu, często wymagane jest, aby narzędzia używane przez nie w procesach inżynieryjnych były certyfikowane przed pierwszym użyciem i po każdej aktualizacji ich wersji.

To oznacza, że producenci oprogramowania muszą być zaznajomieni z wymaganiami dotyczącymi zgodności, które obowiązują w branżach klientów i obsługiwać je przy użyciu dedykowanych funkcji produktów lub poprzez zapewnianie odpowiednich usług. Funkcje, których zadaniem jest spełnianie standardów, nie muszą być traktowane inaczej niż zwykłe funkcje, ale i tak pewnie będą otrzymywać wyższy priorytet. Bliska współpraca z klientem i regularne tworzenie działających wersji ułatwia zespołom zwinnym identyfikowanie i dostrajanie dodatkowych usług wsparcia oraz zapewnianie ich klientom w odpowiednim czasie.

Regulacje, którym podlegają niektóre branże (na przykład FDA OTS i FDA Validation w przypadku technologii medycznych), wskazują na konieczność formalnego sprawdzania nie tylko oprogramowania systemowego, ale też określonych narzędzi. Nakłady klienta związane z takim sprawdzaniem mogą znacząco spowolnić pracę zespołu zwinnego, gdyż każda wersja (a czasami każda poprawka) oprogramowania musi być sprawdzana. Dłuższe cykle wydawnicze

10 Niektóre wymagania dotyczące zgodności wymuszają ten poziom z góry.

ograniczają wymagane nakłady, jeśli więc zespół zwinny dostarcza gotową wersję zbyt często, klient prawdopodobnie będzie ignorował lub pomijał niektóre wersje. W obu przypadkach zespół nie otrzymuje użytecznych informacji zwrotnych od klienta, który może czuć się poddany niechcianej presji dotyczącej aktualizacji oprogramowania. W takich przypadkach programowanie oparte na tradycyjnym modelu V jest chyba bardziej wskazane.

7.4 Tradycyjne zapewnianie jakości

Podczas gdy zarządzanie jakością (QM – quality management) zajmuje się zawartością, jakością i optymalizacją procesów biznesowych, zapewnianie jakości (QA – quality assurance) koncentruje się na jakości produktów wytwarzanych przez organizację.

7.4.1 Narzędzia do zapewniania jakości

Tradycyjne metody tworzenia oprogramowania wyróżniają konstruktywne i analityczne środki zapewniania jakości. Szablon, który zapewnia, że dokument będzie tworzony w określony sposób, jest przykładem konstruktywnego zapewniania jakości, tak samo jak wzorzec używany do projektowania oprogramowania zgodnie z ustalonymi najlepszymi praktykami. Wszystkie typy testowania, sprawdzania, przeglądania i analizy są określane jako analityczne środki zapewniania jakości.

Zapewnianie jakości a planowanie testów

Tradycyjnie zarządzane projekty opisują, planują i wdrażają środki zapewniania jakości, które są często oparte na elementach normy [IEEE 730]. Taki plan jest zwykle przygotowywany przez menedżera zapewniania jakości projektu, menedżera testów lub menedżera projektu. Planowi zapewniania jakości towarzyszy plan testowania (często opracowywany według [IEEE 829]). Oszacowanie nakładów związanych z zastosowaniem tak zaplanowanych środków zapewniania jakości jest następnie dodawane do ogólnego planu projektu.

7.4.2 Organizacja

Menedżer testów albo menedżer zapewniania jakości otrzymuje zadanie zastosowania zaplanowanych środków. W większych projektach osobie tej przydzielana jest grupa kolegów/testerów, która tworzy podprojekt.

Rozróżnianie pomiędzy pracownikami programującymi i testującymi

To podejście tworzy wyraźne rozróżnienie pomiędzy pracownikami programującymi i testującymi, a w tradycyjnie zarządzanych projektach, gdzie nie dokonuje się tego rozróżnienia, powoduje niewystarczające zapewnianie jakości. Dzieje się tak, ponieważ menedżer projektu

i członkowie zespołu koncentrują się na wspieraniu ukończenia produktu, a zadania testowania i przeglądu są traktowane jako opóźniające ten proces. Jednakże oddzielna grupa do spraw zapewniania jakości będzie się skupiać wyłącznie na zadaniach związanych z zapewnianiem jakości i będzie rozliczana tylko z tego, jak dobrze je przeprowadzi i jak wiele usterek pomoże odkryć przed dostarczeniem produktu. Jest to zaleta tradycyjnej organizacji i zarządzania projektami, którą warto docenić.

W przebiegu niektórych projektów delegowanie zadań związanych z zapewnianiem jakości dedykowanemu zespołowi powoduje wyraźnie obniżenie zainteresowania i ogólnego poczucia odpowiedzialności za jakość. Widać to w pomijaniu testów jednostkowych lub niechlujnych praktykach związanych z testowaniem usprawiedliwianych pewnością, że zespół zapewniania jakości się tym zajmie. Jednakże takie zachowanie może być też wynikiem nieodpowiedniego zdefiniowania, która część zespołu jest faktycznie odpowiedzialna za testowanie jednostkowe. Ostatecznie każdy zakłada, że wszyscy inni zajmą się tym problemem.

Aby zapobiec tego rodzaju nieporozumieniom, menedżer projektu musi zapewnić, że wszystkie części zespołu będą się regularnie ze sobą komunikowały. Może to przybierać postać codziennego spotkania Scrum z przedstawicielami wszystkich zaangażowanych zespołów.

7.5 Zwinne zapewnianie jakości

Zwinne zespoły są wielofunkcjonalne (zobacz podrozdział 2.1). Cały zespół jest odpowiedzialny za produkt oraz jego jakość i nie ma formalnego podziału ról albo dedykowanej grupy do zapewniania jakości, która byłaby wyłącznie odpowiedzialna za zapewnianie jakości i zadania testowe. Każdy członek zespołu oferuje swoje własne umiejętności specjalne (architekt oprogramowania, programista, tester, itd.), ale nie jest wyłącznie przywiązany do tej określonej roli. Każdy może wykonywać dowolne z wielu typów zadań zawartych na tablicy zadań, w tym zadania związane z zapewnianiem jakości i testowaniem.

Organizacyjny punkt zaczepienia dla zadań zapewniania jakości jest tracony podczas przechodzenia na praktyki zwinne i – jak pokazano w kolejnym podrozdziale – musi zostać zastąpiony w metodyczny sposób, jeśli zespół ma nadal utrzymywać wysoką jakość tworzonych produktów.

7.5.1 Zasady i narzędzia

Zapewnianie jakości w zwinnym zespole oparte jest na zasadzie „inspekcji i adaptacji”[11] zdefiniowanej przez przewodnik Scrum [URL: Scrum Guide] następująco:

- **Inspekcja** Użytkownicy Scrum muszą często badać elementy Scrum i postęp w kierunku celu sprintu, aby wykrywać niepożądane odchylenia. Inspekcja nie powinna być na tyle częsta, żeby przeszkadzała w pracy. Inspekcje są najbardziej przydatne, kiedy są pilnie wykonywane przez wykwalifikowanych inspektorów.
- **Adaptacja** Jeśli inspektor ustali, że jeden lub więcej aspektów procesu wychodzi poza dopuszczalne granice i że wynikowy produkt będzie nie do przyjęcia, trzeba poprawić proces lub przetwarzany materiał. Poprawka musi być dokonana jak najszybciej, aby zminimalizować dalsze odchylenia.

To oznacza, że każdy pojedynczy element musi być badany! Jednakże w przeciwieństwie do tradycyjnie zarządzanego projektu kwestia, kiedy i jak zostanie przeprowadzone to sprawdzenie (i przez kogo), nie jest zaplanowana z góry, a jest ustalana od nowa dla każdego nowego sprintu w trakcie rutynowych spotkań:

- **Spotkanie planujące sprint (zobacz podrozdział 3.5)** Zadania wymagane do ukończenia każdego elementu na liście zaległości sprintu są definiowane w trakcie spotkania planującego sprint. Odpowiednie kryteria akceptacji są już nakreślone przez właściciela produktu i zawarte na liście zaległości produktu (zobacz podrozdział 3.3). Zajmują one miejsce celów i atrybutów jakościowych, które są oddzielnie definiowane w konwencjonalnych planach zapewniania jakości. Następnie musimy zdecydować, jakiego typu sprawdzenia będą odpowiednie i konieczne, aby sprawdzić, czy każde kryterium akceptacji zostało właściwie spełnione. Dostępne narzędzia, z których zespół może wybierać, obejmują wszystkie typy analitycznych środków zapewniania jakości i typy testów (zobacz rozdziały 4, 5 i 6), a także testy badawcze opisane szczegółowo w podrozdziale 6.3. Jeśli postanowimy przeprowadzić wyczerpujące sprawdzenie, muszą być one przygotowane w osobnym zadaniu (tzn. trzeba określić i zautomatyzować przypadek testowy), które jest zawarte w planowaniu sprintu i odpowiada właściwemu zadaniu programistycznemu. Jeśli za wystarczające uznamy istniejący przegląd z programowania w parach lub istniejący, zautomatyzowany test regresji, to sprawdzenie może być

11 Ten pomysł został dodany do manifestu Agile [URL: Agile Manifesto] jako dwunasta zasada przewodnia.

odnotowane pośrednio w samym zadaniu programistycznym. Zadania zapewniania jakości są więc obsługiwane w dokładnie taki sam sposób jak wszystkie inne zadania i stają się częścią codziennego przepływu zadań bez konieczności tworzenia osobnego planu zapewniania jakości.

- **Codzienny Scrum** To codzienne spotkanie jest wykorzystywane[12] przez zespół do odzwierciedlania stanu sprintu i aktualizowania tablicy zadań. Trzeba być w stanie udowodnić, że uzgodnione kryteria gotowości zostały spełnione dla każdego zadania, które jest oznaczone jako wykonane. Jeśli zdefiniowano bezpośrednie zadanie testowania, musi być ono przeprowadzone, a wyniki (np. dzienniki testowe) muszą być dostępne do sprawdzenia. Jeśli nie jest to możliwe, cel jakościowy dla danego elementu nie został osiągnięty i zadanie nie może być zakwalifikowane jako wykonane. Jeśli na przykład okaże się, że dana funkcja jest trudniejsza do zaimplementowania, niż to pierwotnie zakładano, wszystkie wady ujawnią się nie tylko wobec zaangażowanego programisty, ale wobec całego zespołu. Codzienne spotkanie Scrum służy więc nie tylko przekazywaniu zespołowi obrazu ogólnego postępu, ale też powiadamianiu o wszelkich problemach, przed którymi stoją aktualnie pojedynczy członkowie zespołu. Członkowie zespołu mogą więc reagować szybko i pomagać sobie nawzajem. Wszelkie bieżące przeszkody są również omawiane podczas codziennego spotkania Scrum, co daje mistrzowi Scrum okazję do reagowania i implementowania odpowiednich rozwiązań. Codzienny Scrum podejmuje więc kwestie jakości produktu i związane z procesami.
- **Przegląd sprintu** Przegląd sprintu odbywa się na końcu każdego sprintu i jest używany przez zespół do przedstawiania i wyjaśniania najnowszego stanu produktu właścicielowi produktu i klientowi. W tym momencie zespół otrzymuje natychmiastową informację zwrotną od klienta na temat nowej wersji produktu. Pomaga to zespołowi i klientowi upewnić się, że produkt rozwiązuje problem klienta. Przegląd sprintu przybiera więc postać nieformalnego sprawdzenia poprawności i zapewnia informację zwrotną dotyczącą jakości poszerzającą informacje zwrotne zwykle dostarczane przez testy. Informacje te mogą być następnie wykorzystane do implementacji wszelkich koniecznych poprawek produktu i do dodawania wszelkich nowo odkrytych poprawek procesu do listy zaległości produktowych.
- **Retrospektywa sprintu** Zespół wykorzystuje retrospektywę sprintu do omówienia problemów związanych z procesem

12 „...15-minutowe zdarzenie, podczas którego zespół synchronizuje działania i tworzy plan na następne 24 godziny." [URL: Scrum Guide].

i sposobów poprawienia procesu tworzenia oprogramowania (zobacz podrozdział 7.2.3). W tym miejscu omawiane i hierarchizowane są środki poprawy projektu określone w przeglądzie sprintu lub innych krokach służących poprawieniu ogólnej skuteczności (na przykład poprawienie pokrycia testami przez wykorzystanie większej liczby zautomatyzowanych testów). Implementowanie takich środków służy stabilizacji i poprawie jakości produktu we wszystkich kolejnych sprintach.

Zapewnianie jakości przez rutynę i przeźroczystość

Choć brak jest jawnie określonej osoby lub grupy, która jest odpowiedzialna za zadania zapewniania jakości, istnieją dwa główne powody, dla których zespół nie traci zapewniania jakości z pola widzenia w trakcie codziennego przepływu zadań. Codzienne spotkania wymienione powyżej zapewniają, że sprawy zapewniania jakości są omawiane codziennie, a podstawowe zasady Scrum zapewniają, że wszelkie braki jakościowe są identyfikowane na wczesnym etapie. Mogą to być nieudane testy, które pojawiają się na desce rozdzielczej, karty zadań, które nie mogą zostać przeniesione, ponieważ ich kryteria realizacji nie zostały spełnione, albo komentarze przedstawione przez klienta podczas przeglądu sprintu. Ponieważ cały zespół jest stale na bieżąco z całym procesem tworzenia oprogramowania, wielu błędów można uniknąć od samego początku, a tendencja poszczególnych członków zespołu do pracy w izolacji jest w znacznym stopniu eliminowana. W zespole wielofunkcyjnym dyscyplina rutynowych spotkań i przeźroczystość bezpośrednio zastępują indywidualną odpowiedzialność przypisywaną członkom konwencjonalnego zespołu opracowującego oprogramowanie.

7.5.2 Mocne i słabe strony

Zespół, który wdrożył w praktyce zwinne środki zapewniania jakości opisane powyżej, zwykle osiągnie świetne wyniki. Mocnymi stronami tego podejścia są:

- **Każdy opracowywany element jest testowany** Obejmuje to testowanie wykonywalnego kodu programu, przeglądanie i inspekcję diagramów architektonicznych, diagramów klas i innych dokumentów projektowych, sprawdzanie poprawności planu projektu poprzez zawartość tablicy zadań lub listy zaległości sprintu.
- **Środki zaradcze są wprowadzane tak wcześnie, jak to możliwe** Gdy zautomatyzowany przypadek testowy się nie powiedzie, przyczyna jest natychmiast analizowana. Jeśli to możliwe, odpowiedzialny programista natychmiast poprawia usterkę lub oferuje rozwiązanie w ramach bieżącego sprintu. Jeśli problem nie

ustępuje, środki zaradcze (takie jak zmiana w projekcie produktu) są dodawane do zaległości produktowych, a odpowiednie rozwiązanie jest wprowadzane w życie zgodnie ze zwykłym procesem nadawania priorytetów.

- **Konstruktywne narzędzia zapewniania jakości wysuwają się naprzód** Techniki takie, jak czysty kod, automatyzacja testów, sterowanie testami, itd. w oczywisty sposób podnoszą poziom technicznego rzemiosła i wiedzy wewnątrz procesu tworzenia oprogramowania. Zespół rozwija swoje standardy wysokiej jakości wobec produktu i technik używanych do jego zbudowania.
- **Poprawki procesu są dokonywane z dołu do góry** Kroki są wprowadzane szybko zgodnie z faktycznymi i natychmiastowymi potrzebami zespołu.

Ryzyko związane ze zwinną kulturą zapewniania jakości

Jeśli zespół wykorzystuje te mocne strony, może rozwinąć swoją własną, zdyscyplinowaną i wysoce efektywną kulturę zapewniania jakości. Jednakże sukces nie jest automatyczny i podejście Scrum rodzi pewne ryzyko, jeśli chodzi o stałe zapewnianie jakości produktu. Do nas należy decyzja, czy następujące punkty będą stanowić ryzyko, któremu trzeba przeciwdziałać, czy będą ogólnymi słabościami Scrum lub podejścia zwinnego do tworzenia oprogramowania. W każdym razie powinniśmy wiedzieć, że istnieją:

- **Świat idealny kontra codzienna rzeczywistość** Scrum, XP i inne procesy zwinne są wszystkie oparte na pojęciu zespołów wielofunkcjonalnych[13]. Zespół Scrum składa się z członków, którzy między sobą mają wszystkie umiejętności wymagane do osiągnięcia celu sprintu [Schwaber/Beedle 02, strona 37]. Jednakże codzienna rzeczywistość pracy w organizacji jest inna. Wymagana wiedza nie zawsze jest obecna w zespole, a niektórzy członkowie zespołu są bardziej wydajni niż inni spełniając rolę „głównego gracza" lub „guru". Myśląc o wprowadzeniu Scrum lub pracując już w zespole Scrum trzeba uznać, że zespół nie jest idealny i założyć, że może to utrudniać przebieg potencjalnie gładkich procesów, które metodyka Scrum ma generować.
- **Uczenie się a umiejętności** Praca w parach i wspieranie się pomaga wszystkim członkom zespołu w poprawieniu swoich umiejętności. W ten sposób zespół może uzupełniać brakujące umiejętności i dawać każdemu członkowi okazję do podzielenia się wiedzą i nauczenia się samemu nowych rzeczy. Jednakże w rzeczywistości nie każdy członek zespołu będzie w stanie płynnie

13 W XP [Beck/Andres 04] pojęcie to jest nazywane zasadą jednego zespołu, natomiast Scrum [Schwaber/Beedle 02] odwołuje się do zespołu wielofunkcjonalnego.

i efektywnie pracować z wszystkimi pozostałymi i nie każdy lubi dzielić się swoją wiedzą. Ponieważ metodyka Scrum zmusza do „ram czasowych" przeciwdziałających ograniczeniom czasowym, które pośrednio jej dotykają, czas poświęcony na naukę nowych umiejętności jest rzadkim zasobem i zespół zwykle w momencie, gdy trzeba ukończyć jakieś zadanie, będzie polegał na kimś, kto ma odpowiednie umiejętności, niż na kimś, kto nadal się uczy. Każdy członek zespołu ma swoje własne doświadczenie akademickie i rozpoczyna naukę od innego punktu. Dlatego mało prawdopodobne (a z punktu widzenia firmy nieekonomiczne) jest, aby członkowie zespołu uczyli się całkowicie nowych umiejętności podczas pracy. Tam, gdzie takie podejście ma sens, niektórzy będą sobie z nim radzić, niektórzy nie, a inni w ogóle nie będą próbować. Ogólnie mówiąc, wykorzystanie istniejących mocnych stron jest zawsze lepsze niż próba eliminowania słabszych[14].

- **Wypychanie kontra dociąganie** Scrum, XP i inne modele zwinne są oparte na założeniu, że zespół sam się organizuje. Jednakże tak samo jak w świecie tradycyjnie zarządzanych projektów można znaleźć lepszych i gorszych menedżerów projektów oraz liderów zespołów, tak samo niektóre zespoły zwinne są lepsze w organizowaniu się niż inne. Różnorodne są czynniki, które przeszkadzają w skutecznej samoorganizacji. Tak jak w innych środowiskach również projekty Scrum składają się z różnorodnych zadań od ciekawych i ekscytujących po dość monotonne, co na pewno będzie wpływać na planowanie sprintu i ustalanie listy zaległości. Oczywiście zespół dopilnuje, żeby niepopularne zadania nie zostały całkowicie zignorowane, ale nie może automatycznie zapobiec, żeby bardziej interesujące zadania nie uzyskiwały wyższych priorytetów niż to konieczne. Tak samo niektórzy członkowie będą (świadomie lub nieświadomie) wybierać bardziej interesujące zadania spośród aktualnie oferowanych. Niezależnie od zasady „dociągania", na której opiera się Scrum, „wypchnięcie" ze strony właściciela produktu lub jakiegoś członka zespołu może być czasami na miejscu! Podobnie do innych zespołów tak samo w zespole Scrum będzie kilku ambitnych i zdolnych członków. Takie tendencje mogą prowadzić do tworzenia nieoficjalnych hierarchii w zespole, które są dość podobne do tych w zespołach dla tradycyjnych projektów. Zespół może być w stanie to tolerować, ale może w pewnym momencie odkryć, że nie jest już wcale zespołem Scrum.

14 Fredmund Malik uważa wykorzystanie znanych mocnych stron za najważniejszą ze swoich sześciu zadań skutecznego przywództwa (zobacz [Malik 09, rozdział 4]).

- **Dyscyplina a szybkość** Praca w rytmie zorganizowanych sprintów może wydawać się prostą, przyjemną i dobrze uregulowaną. Jednakże ścisłe przestrzeganie zasad przeźroczystości i ciągłe ograniczenia czasowe związane z osiąganiem celu na końcu sprintu mogą generować dość duże napięcia. Teoretycznie zespół wykorzystuje planowanie sprintu do regulowania ilości pracy tak, aby ustanowić stałą prędkość działania zespołu. W praktyce jednak często będzie można odkryć, że dyscyplina jest poświęcana na rzecz szybkości. Zasady sterowania testami nie są ściśle stosowane ze względu na brak czasu; nikt nie pisze tradycyjnych specyfikacji, gdyż dokumentacja z gruntu nie jest procesem zwinnym; przypadki testowe, które można by zautomatyzować, są stosowane ręcznie, ponieważ nie ma czasu na ich zautomatyzowanie; konieczny (lub pożądany) refaktoring jest opóźniany, ponieważ głównym priorytetem klienta są nowe funkcje, itd. To wszystko spycha zespół na ścieżkę prowadzącą w końcu do nieudanego projektu.

Każdy mistrz Scrum i zespół zwinny musi stawiać czoło tym i podobnym ryzykom, które wpływają na skuteczność procesu zapewniania jakości. Jednakże mocne strony zwinnych procesów zapewniania jakości wymienione na początku tego podrozdziału mogą przeważać, jeśli to ryzyko będzie utrzymywane pod kontrolą.

7.6 Testowanie zwinne

Nawet jeśli konstruktywne środki mają wysoki priorytet w zespole zwinnym, testowanie wciąż jest najważniejszym narzędziem zapewniania jakości. Rozdziały 4, 5 i 6 opisują szczegółowo, jakie są możliwości testowania i jak podchodzi się do testowania w środowisku programowania zwinnego. Poniższe podrozdziały podsumowują najważniejsze czynniki sprawiające, że testowanie jest zwinne.

7.6.1 Krytyczne czynniki udanego testowania zwinnego

„Testowanie zwinne jest testowaniem oprogramowania w ramach zwinnego projektu programistycznego [...] Testowanie zwinne stosuje zasady przedstawione w manifeście Agile i stosuje zasady zwinnej metodyki testowania oprogramowania" (za [URL: Agile Testing]).

Testowanie non-stop z codzienną informacją zwrotną

Głównym wymaganiem wobec zwinnego testera jest dostarczanie szybkiej informacji zwrotnej, a wszystkie wysiłki związane z testowaniem skupiają się na tym celu. Zamiast sekwencyjnych faz testowania (i w związku z tym powolnej informacji zwrotnej), podczas każdego sprintu mają miejsce ciągłe, równoległe testy zapewniające testowanie

non-stop z codzienną informacją zwrotną. Następujące czynniki są krytyczne dla powodzenia tego podejścia:

- **Automatyzacja testów** Szybka informacja zwrotna może być stale zapewniana tylko wtedy, jeśli wszystkie testy (tzn. testy jednostkowe, integracyjne i systemowe) będą wystarczająco zautomatyzowane. Taka sieć zautomatyzowanych testów umożliwia ciągły refaktoring kodu programu i jest koniecznym warunkiem wstępnym dla niezawodnego wprowadzenia podejścia czystego kodu (zobacz [Martin 08]).
- **Testowanie badawcze** Ponieważ nie jest możliwe natychmiastowe zautomatyzowanie każdego przypadku testowego, wymagane jest dodatkowe (szybkie) testowanie ręczne. Można to osiągnąć przy użyciu testowania badawczego – techniki, która radzi sobie bez wstępnej specyfikacji testu i daje testerowi wolność intuicyjnej pracy umożliwiając niezwykle krótkoterminowe testowanie nowych funkcji, których spodziewane zachowanie jest tylko ogólnie zarysowane na odpowiedniej karcie zadania. Oznacza to też, że tester musi być w stanie aktywnie podchodzić do wszystkich zainteresowanych stron i innych potencjalnych źródeł danych testowych. Ta technika wymaga talentu, ale można się jej nauczyć.
- **Doświadczenie w testowaniu wewnątrz zespołu** Zespół zwinny odpowiada za testowanie. Działania związane z testowaniem są planowane i kontrolowane w ten sam sposób jak wszystkie inne działania w ramach sprintu. Każdy członek zespołu może (i powinien) przeprowadzać zadania związane z testowaniem zgodnie ze swoimi własnymi umiejętnościami. Uczynienie z testowania zadania zespołowego wymaga od zespołu odpowiednich umiejętności od samego początku, a tradycyjny zespół programistyczny przechodzący na metodykę Scrum zwykle nie jest odpowiednio przygotowany. W większości przypadków zewnętrzni testerzy lub członkowie istniejącego zespołu testowania systemowego będą oddelegowani do zespołu Scrum, żeby wzmocnić jego umiejętności związane z testowaniem. Osoby te muszą nauczyć się testowania w ramach zespołu zwinnego, a nie w niezależnym środowisku testowym, do którego są przyzwyczajone. Odchodzi się od niezależnego testowania z oddzielnie zdefiniowanymi rolami i wydzielonymi strukturami organizacyjnymi. To przejście nie jest pozbawione ryzyka i może się udać tylko wtedy, gdy zapewnimy, aby zespół również miał swoich własnych ekspertów od testowania, a mistrz Scrum lub specjalista od testowania przyjął rolę menedżera testów. Jednak nawet jeśli zespół zawiera ekspertów od testowania, bliska współpraca (będąc nieodłącznym elementem zespołu zwinnego) może osłabić proces testowania, ponieważ (jak

pokazano w studium przypadku 8.1) nawet pełnoetatowi testerzy mają tendencję do patrzenia na projekt z punktu widzenia programisty, co prowadzi do mniej krytycznej oceny wyników testów, niż mógłby to zrobić niezależny tester.

- **Wiele zespołów** Jeśli wiele zespołów Scrum pracuje nad pojedynczym projektem, musimy spojrzeć z szerszej perspektywy, aby zapewnić, że wszystkie opracowywane funkcje będą ze sobą poprawnie współpracować. Nawet jeśli poszczególne zespoły przeprowadzają testy jednostkowe, integracyjne i systemowe dla swoich funkcji, to ryzykujemy zaniedbanie kompleksowych testów ogólnosystemowych. W celu przeciwdziałania tej tendencji menedżer testów i pełnoetatowi testerzy muszą spotykać się regularnie i wymieniać uwagi, tak jak mistrzowie Scrum podczas wspólnego spotkania Scrum. Ta wymiana zdań może przyjmować formę procesu Scrum testerów, jak opisano w studium przypadku 8.3. Sensowne może być też utworzenie nadrzędnego zespołu do testowania systemowego (zobacz podrozdział 6.8.1), który zapewni nie tylko powstanie scenariuszy testów międzyzespołowych, ale też będzie obsługiwać platformy testowe i dostarczać je innym zespołom jako usługę.

7.6.2 Planowanie testów w Scrum

Jak już wspomniano, działania testowe w projekcie Scrum są planowane i kontrolowane w ten sam sposób, jak wszystkie inne działania w ramach sprintu – przy użyciu zadań przenoszonych z listy zaległości produktowych do listy zaległości sprintu, a następnie na tablicę zadań w trakcie planowania sprintu. Podczas planowania testów opartych na Scrum trzeba mieć na uwadze następujące punkty:

- **Definicja gotowości** Definicja gotowości zespołu jest listą kontrolną wykorzystywaną przez właściciela produktu do zapisywania i sprawdzania jakości scenariuszy użytkowników i do której trzeba się odwoływać, gdy scenariusze są pobierane z listy zaległości produktowych do listy zaległości sprintu. Zespół może określić, czy scenariusz użytkownika jest gotowy, przyglądając się mu z punktu widzenia testera. Jeśli nie można nakreślić odpowiednich przypadków testowych lub jeśli nie jest jasne, kiedy zaklasyfikować wynik testu jako poprawny lub niepoprawny, to scenariusz oczywiście nie jest wystarczająco dobrze zdefiniowany i powinien być odrzucony jako niegotowy. Alternatywnie zespół może uzupełnić luki w scenariuszu stosując zasady sterowania testami i dodając przypadki testowe sprawdzające brakujące wyniki. Praca właściciela produktu w parze z testerem jest świetnym sposobem zrealizowania tego zadania.

- **Definicja wykonania** Definicja wykonania jest dodatkową listą kontrolną opisującą cele, które zespół musi osiągnąć, zanim scenariusz będzie można zaklasyfikować jako gotowy do włączenia w przegląd sprintu. Definicja wykonania obejmuje czynniki, takie jak wymagane typy testów i pokrycie testami oraz kryteria definiujące test jako wykonany (zwykle oznacza to wyeliminowanie wszystkich usterek). Definicja wykonania jest więc bezpośrednio związana z zapewnianiem jakości produktu i zadowolenia klienta.

Zadanie testowania może być jawnie zdefiniowane jako osobne zadanie lub może istnieć pośrednio w formie kryterium wykonania dla zadania programistycznego. Jeśli wszystkie potrzebne testy zostały już napisane i zautomatyzowane, przyjęcie podejścia pośredniego nie stanowi problemu – jeśli jednak scenariusz lub funkcja są testowane po raz pierwszy (albo testy są w trakcie procesu automatyzowania), to zaleca się obsługę takich zadań oddzielnie i jawnie.

7.7 Umiejętności, szkolenia, wartości

W ciągu ostatnich 10 lat testowanie oprogramowania stało się pełnoprawnym zawodem. Wiele projektów powierza działania związane z testowaniem i zapewnianiem jakości certyfikowanym testerom, którzy odgrywają znaczącą rolę w powodzeniu projektu.

Istotne są wysokie poziomy zapewniania jakości i umiejętności testowania oprogramowania.

Wysokie poziomy testowania oprogramowania i umiejętności zapewniania jakości są istotne w coraz bardziej złożonych systemach, które są budowane przez dzisiejsze zwinne zespoły tworzące oprogramowanie. Zwinne testowanie nie oznacza testowania mniej dokładnie niż wcześniej, a wręcz odgrywa coraz bardziej znaczącą rolę. Przypadki testowe definiują system zamiast specyfikacji (sterowanie testami), testy są automatyzowane i wykonywane ciągle (testowanie non-stop). Projektowanie przypadków testowych wysokiej jakości jest wymagającym zadaniem, które wymaga specjalistycznego szkolenia. Automatyzacja testów wymaga znajomości narzędzi programistycznych i testowych oraz analizy kodu, ciągłej integracji, technik przeglądu, itd.

Ponieważ Scrum nie definiuje jawnie roli testera i nawet nie wspomina o roli menedżera testów, zespołom Scrum często brakuje umiejętności potrzebnych do udanego przeprowadzania testów oprogramowania. Jest to ryzykowna sytuacja, ponieważ Scrum polega w dużej mierze na pętlach zwrotnych, a testy są jednymi z najważniejszych źródeł informacji zwrotnych w środowisku tworzenia oprogramowania.

Ma to wpływ na wszystkich członków zespołu wielofunkcjonalnego, więc istotne jest, aby wszyscy – w tym programiści – mieli przynajmniej podstawowe umiejętności związane z testowaniem. Z kolei

testerzy muszą być biegli w podstawach programowania, aby mogli przeprowadzać zadania automatyzacji testów i analizy kodu. Wszystkie te techniki wymagają praktyki i dział kadr musi uzgodnić z mistrzem Scrum i pracownikami zapewniania jakości (zobacz podrozdział 7.2.2) harmonogram szkoleń dla zespołu.

Certyfikowani testerzy ISTQB

Uznawaną międzynarodowo kwalifikacją testerów oprogramowania jest tytuł ISTQB Certified Tester. Kurs na poziomie podstawowym obejmuje wszystkie podstawowe techniki od partycjonowania równoważnościowego przez analizę wartości granicznych po testowanie oparte na stanach, a więc dotyczy wszystkich etapów procesu, z którymi spotka się zespół Scrum, od testów jednostkowych po testy akceptacyjne (zobacz rozdziały 4, 5 i 6). Kursy ISTQB Advanced oraz Expert są bardziej dogłębne i nadają się dla członków zespołu, którzy zajmują się testami i zapewnianiem jakości. Kursy te są publikowane przez organizację ISTQB i jej oddziały narodowe, które odpowiadają też za testowanie i monitorowanie oferowanych kursów szkoleniowych [URL: ISTQB].

Szkolenie zwinnych technik testowania

Jeśli ktoś jest już certyfikowanym testerem, powinien wziąć pod uwagę udział w kursie podstaw Scrum, być może w połączeniu ze zwinnymi technikami testowania (na przykład „Testing in SCRUM" [URL: iAkad] lub kurs Certified Agile Tester [URL: iCAT]). Oficjalnym programem dla takich kursów jest nowy program ISTQB: „Foundation Level Extension Syllabus Agile Tester" [URL: ISTQB]. Kursy, które koncentrują się na indywidualnych technikach testowania, takich jak testowanie badawcze, są przydatnym dodatkiem (ale nie alternatywą) dla szkolenia ISTQB na poziomie podstawowym. Jak już pokazaliśmy, testowanie w projekcie zwinnym wymaga wiedzy, która obejmuje całe spektrum technik testowania – techniki badawcze są jedynie częścią układanki.

Zmiana wartości

Przejście na metodykę Scrum wymaga zmiany sposobu pracy. Z punktu widzenia procesów zarządzania jakością i zapewniania jakości istnieją dwie zmiany, które są szczególnie oczywiste. Zarządzanie jakością przekształca się z procesu odgórnego w proces oddolny, a zespół zarządzania jakością świadczy usługi zespołowi zwinnemu. Operacyjne zapewnianie jakości nie jest już zadaniem wykonywanym przez zespół zewnętrzny, a staje się integralną częścią przepływu zadań zespołu Scrum. Zmiany te koniecznie wymagają przewartościowania priorytetów testerów i specjalistów zapewniania jakości, którzy pracują w zespole zwinnym:

- Konstruktywne relacje pomiędzy członkami zespołu są ważniejsze niż procesy i narzędzia testowe.
- Testowane oprogramowanie jest ważniejsze niż kompleksowa dokumentacja testów.

- Ciągła współpraca z klientem jest ważniejsza niż formalne testy akceptacyjne przeprowadzane na końcu projektu.
- Reagowanie na zmiany jest ważniejsze niż postępowanie według ścisłego planu testów.

Do potrzebnych umiejętności miękkich, które dobrze uzupełniają te postawy, należą współpraca, umiejętność pracy w parach, komunikacja zamiast dokumentacji, samoorganizacja, inicjatywa oraz aktywne poszukiwanie i wyciąganie informacji.

To sprawia, że metodyczne umiejętności zaawansowanych specjalistów od testowania i zapewniania jakości stają się jeszcze ważniejsze. Sposób pracy z pewnością się zmieni, a w firmie imbus mówimy:

Zmień swoje nastawienie! Zachowaj swoje metody!

7.8 Pytania i ćwiczenia

7.8.1 Samoocena

Pytania i ćwiczenia pomagające w ocenie, jak zwinny jest naprawdę projekt lub zespół.

1. Jak dobrze działa firmowy system zarządzania jakością? Czy rzeczywista praktyka pokrywa się z regułami zarządzania jakością, czy też stanowią dwa równoległe światy?
2. Czy reguły są aktualne? Kiedy system był ostatnio aktualizowany?
3. Czy twoje własne zadania i obowiązki są odpowiednio zdefiniowane? Czy stosujesz się do reguł odnośnie swoich osobistych obowiązków?
4. Czy znasz wszystkie dokumenty zarządzania jakością, które są związane z twoimi obowiązkami? Jak długo zajmuje ci znalezienie odpowiedniego dokumentu?
5. Jak są implementowane nowe lub zmieniane procesy? Czy sesje szkoleniowe lub informacyjne są wystarczające?
6. Jak jest zdefiniowany proces tworzenia oprogramowania? W których fazach procesu aktualnie uczestniczysz? Które kryteria są ważne do zakończenia bieżącej fazy?
7. Jeśli brałeś udział w audycie, jakie były wyniki i czy wprowadzono od tego czasu jakieś środki naprawcze?
8. Jak skuteczna jest współpraca pomiędzy twoim działem/zespołem a pracownikami zarządzania jakością? W czym pomogłyby praktyki zwinne?

9. Jeśli zespół korzysta już z praktyk zwinnych, jak dobra jest współpraca pomiędzy mistrzem Scrum a pracownikami zarządzania jakością? Czy zagadnienia omawiane podczas retrospektyw sprintów są skutecznie implementowane? Jak można by poprawić implementację?
10. Które analityczne środki zapewniania jakości (poza testowaniem) i które konstruktywne środki stosujesz w swoim projekcie? Które środki lub techniki nie są stosowane, pomimo że miałyby sens? Dlaczego?
11. Jeśli zespół korzysta już z praktyk zwinnych, to jak jest zorganizowane testowanie? Jak dobrze zespół spełnia krytyczne czynniki dla udanego testowania zwinnego?
12. Jak obecnie wygląda plan testowania projektu? Czy można z niego korzystać do identyfikowania, które zadania testowania są do wykonania dziś/w tym tygodniu? Czy te zadania są faktycznie realizowane? Czy pomagają wykrywać usterki na czas?

7.8.2 Metody i techniki

Te pytania pomogą w podsumowaniu treści bieżącego rozdziału.

1. Wyjaśnić termin „cykl PDCA".
2. W odniesieniu do planu projektu pokazanego na rys. 2-3, które cykle PDCA można zidentyfikować?
3. W odniesieniu do procesu Scrum pokazanego na rys. 2-1, które cykle PDCA można zidentyfikować?
4. Wyjaśnić różnicę pomiędzy kartą zespołu a opisem procesu programowania zwinnego opartym na zarządzaniu jakością.
5. Wyjaśnić narzędzia przeglądu sprintu i retrospektywy sprintu.
6. Co oznacza identyfikowalność? Dlaczego odpowiednie standardy tworzenia oprogramowania dla produktów związanych z bezpieczeństwem wymagają wysokiego stopnia identyfikowalności?
7. Wymienić zalety i wady rozdzielania programistów i testerów oraz wykonywanych przez nich zadań.
8. Wyjaśnić zwinną zasadę „inspekcji i adaptacji".
9. Wymienić zalety i wady zespołów wielofunkcjonalnych. Jakie ryzyko jest z tym związane?
10. Nazwać i wyjaśnić czynniki, które są krytyczne dla udanego testowania zwinnego.

7.8.3 Inne ćwiczenia

Te ćwiczenia pomogą zagłębić się w zagadnienia poruszone w trakcie tego rozdziału.

1. Opisać wykorzystywany proces tworzenia oprogramowania maksymalnie na jednej stronie niezależnie od tego, czy się pracuje tradycyjnie, czy korzystając z metodyki zwinnej.
2. Wybrać drugi znany projekt lub zespół programistyczny i opisać różnice pomiędzy wykorzystywanymi tam metodami a metodami opisanymi w poprzednim ćwiczeniu. Uogólnić opis procesu tak, aby obejmował oba zestawy charakterystyk.
3. W przypadku pracy nad projektem zwinnym opisać w formie karty zespołu (zobacz podrozdział 3.6), jak działa w zespole planowanie sprintu i szacowanie nakładów. Czy ten opis odnosi się też do innych zespołów?

8 Studia przypadków

Ten rozdział przedstawia studia przypadków dotyczące firm przemysłowych, zajmujących się handlem elektronicznym oraz wytwarzaniem oprogramowania. Każde studium pochodzi z wywiadów przeprowadzanych przy wprowadzaniu i implementowaniu praktyk zwinnych w poszczególnych organizacjach.

8.1 Wykorzystanie Scrum do tworzenia oprogramowania do produkcji wideo i audio

dr Stephan Albrecht,
Menedżer w firmie AVID w Monachium, Niemcy

Firma AVID została założona w roku 1987 w Burlington niedaleko Bostonu, w USA. Jest to firma notowana na giełdzie NASDAQ, która rozwija i sprzedaje rozwiązania do cyfrowej produkcji audio i wideo. Zakres produktów firmy rozciąga się od cyfrowych systemów edycji wideo dla użytkowników końcowych (Pinnacle Studio, Avid Studio) po profesjonalne systemy do produkcji filmowej i telewizyjnej, takie jak Avid Interplay i Avid Media Composer. Wiele spośród najbardziej udanych reklam, teledysków, programów telewizyjnych i filmów na świecie zostało wyprodukowanych przy użyciu produktów AVID.

Pracując w tak dynamicznym środowisku firma musi tworzyć złożone, wyspecjalizowane oprogramowanie w krótkich cyklach, a wszelkie usterki w produktach są dosłownie słyszalne i widoczne dla klienta.

Ze względu na naturę swoich produktów firma AVID zawsze korzystała z mocno iteracyjnych metod wytwarzania oprogramowania, a w 2009 roku postanowiła wprowadzić metodykę Scrum. To studium przypadku opisuje doświadczenia firmy związane z wprowadzaniem Scrum do działu Zarządzania Mediami i Aktywami Produkcyjnymi, który zatrudnia około 100 programistów pracujących w Burlington, Kijowie, Szanghaju, Kaiserslautern i Monachium.

Powody przechodzenia na Scrum

Przy tradycyjnym podejściu iteracyjnym tworzenie oprogramowania odbywało się w różnych, niezależnych zespołach pracujących w różnych lokalizacjach. Na przykład biuro w Monachium składało się z zespołu projektantów, kliku zespołów programistów i swojego własnego zespołu do spraw testowania systemowego i zarządzania jakością. Menedżer produktu odpowiadał za planowanie produktu i decydował, które funkcje mają być zawarte w danej wersji.

Na ogół podejście iteracyjne działało dobrze, ale miało się poczucie, że różne zespoły nie komunikują się wystarczająco ze sobą, co prowadziło do nieporozumień i nieuniknionych usterek. Było też oczywiste, że niezależnie od tego, jak często i jak bardzo dokładnie plany projektowe będą opracowywane i przeglądane, to nigdy nie nadążą za rzeczywistością codziennego przepływu zadań produkcyjnych. Wprowadzenie Scrum było oczywistym sposobem na poradzenie sobie z obydwoma problemami i potencjalnie miało przyspieszyć i poprawić ogólną jakość pracy.

Zmiana

Pierwotne zespoły projektowe, programistyczne i zarządzania jakością zostały rozwiązane i utworzono zespoły wielofunkcjonalne z nowym zakresem obowiązków. Teraz każdy zespół ma wszystkie potrzebne umiejętności i odpowiada za pełny rozwój danego produktu. Innymi słowy każdy zespół ma swoich projektantów, programistów, testerów i specjalistów od zarządzania jakością. Liderzy zespołów zostali przeszkoleni poza firmą jako mistrzowie Scrum i każdy z nich przejął kierowanie zespołem jako mistrz Scrum.

Przeszkody

Mistrzowie Scrum początkowo wracali do swoich starych przyzwyczajeń jako liderów zespołów i byli zbyt dominujący w swoich nowych rolach. Stawało się to szczególnie wyraźne, gdy planowane funkcje nie były gotowe na końcu sprintu. Członkowie zespołu nie byli zadowoleni ze sposobu, w jaki mistrz Scrum próbował wymuszać ukończenie rzeczonych funkcji, choć od początku było jasne, że nie jest to możliwe. W takich przypadkach mistrzowie Scrum po prostu decydowali, że pewne funkcje mają być ukończone w ramach sprintu przez określoną osobę w taki sam sposób, jak to robili jako przywódcy zespołów.

Wypełnianie roli właściciela produktu również generowało pewne trudności. Wysoce złożona natura produktów i olbrzymia liczba zawartych w nich funkcji oznaczała, że tylko niewielka liczba osób mogła się nadawać do tego zadania – znowu pierwszym wyborem był

zwykle dawniejszy lider zespołu. To oznaczało, że niektórzy z nowych mistrzów Scrum podejmowali (lub otrzymywali) też rolę właściciela produktu lub pełnomocnika właściciela produktu (tzn. lokalnej osoby kontaktowej właściciela produktu).

Oba te problemy zostały rozwiązane. Zarząd wybrał pojedynczego właściciela produktu, który odpowiadał za uszczegółowianie listy zaległości produktowych dla całego działu Zarządzania Mediami i Aktywami Produkcyjnymi. Ta sama osoba przejęła też odpowiedzialność za ustalanie czasu i zawartości wypuszczanych na zewnątrz wersji, a więc również planowanie sprintów. Mistrzowie Scrum stopniowo dostosowali swoje umiejętności przywódcze do nowej sytuacji, a ci, którzy sobie nie radzili, byli zastępowani innymi członkami zespołów. Różne metody pracy i zasoby służące poprawie komunikacji pomiędzy zespołami (na przykład forma list zaległości) zostały ustandaryzowane pomiędzy wszystkimi zespołami.

Główne zmiany

Stara organizacja w stylu matrycowym (departamenty × projekty) została zastąpiona zespołami wielofunkcyjnymi, aby umiejętności (takie jak testowanie systemowe), które dawniej były zapewniane centralnie, weszły w zakres odpowiedzialności poszczególnych zespołów. Każdy zespół synchronizuje swoją pracę poprzez codzienne spotkanie na stojąco, a synchronizacja pomiędzy zespołami odbywa się przy użyciu wspólnego spotkania Scrum prowadzonego poprzez wideokonferencję.

Jak można się było spodziewać, początkowo zespołom brakowało niektórych umiejętności (takich jak zarządzanie jakością), które dawniej były zapewniane przez wyspecjalizowane departamenty. To sprawiło, że praca w nowych zespołach Scrum stała się bardziej zróżnicowana, ale też bardziej wymagająca dla członków zespołu. Luki w doświadczeniu członków zespołu mogą być wypełniane przez innych członków zespołu, którzy są proszeni o wsparcie przez dzień lub dwa, ale każdy członek zespołu musi wcześniej czy później nabyć odpowiednie umiejętności.

Za wyjątkiem kilku specjalnych odgałęzień kod nie należy już do pojedynczych programistów, ale do całego zespołu. To oznacza, że każdy może modyfikować i rozwijać cały kod programu. Niektórzy członkowie zespołu postrzegają to jako utratę odpowiedzialności, natomiast inni jako okazję do zajęcia się nowymi aspektami produktu. W każdym razie, ponieważ złożoność produktów czyni je podatnymi na usterki, firma AVID nie zezwala na przekazywanie własności kodu między zespołami, więc koledzy spoza zespołu nie mogą edytować kodu. Pociągałoby to za sobą koordynację pomiędzy zespołami, która i tak byłaby zbyt skomplikowana. W niektórych przypadkach kod

i związane z nim umiejętności są tak bardzo wyspecjalizowane, że własność kodu w przypadku niektórych części produktu pozostaje przy wyznaczonych specjalistach.

Zmiany dotyczyły też opracowywania wymagań. Wprowadzono status gotowości dla scenariusza użytkownika, co oznacza, że scenariusz użytkownika (lub zależne od niego zadania) może być dodawany do listy zaległości sprintu tylko wtedy, jeśli spełniony jest stan gotowości scenariusza użytkownika. To zapewnia, że częściowo zrealizowane scenariusze użytkownika nie będą utrudniać prowadzenia sprintu.

Zmiany procedur testowania

Testerzy byli dawniej członkami zespołu testowania systemowego wewnątrz centralnego departamentu zapewniania jakości. Ten departament został rozwiązany, a specjaliści od automatyzowania testów systemowych, specjaliści od automatyzowania testów jednostkowych, testerzy wykonujący ręczne testy systemowe i testerzy wydajności pracują teraz samodzielnie wewnątrz poszczególnych zespołów Scrum. Na krótką metę oznacza to, że kadra testująca nadal wymaga dostępu do specjalistycznych umiejętności dawniejszej kadry zapewniania jakości. Ta sytuacja jednak daje też członkom zespołów okazję do nauczenia się nowych rzeczy i dzielenia się swoją wiedzą odnośnie testowania. Niektórzy członkowie zespołów, którzy pracowali dawniej jako testerzy (zwykle testerzy jednostkowi), teraz piszą też kod, natomiast inni dawniejsi testerzy systemowi rozszerzyli swoje umiejętności o zapewnianie wsparcia dla użytkowników i pisanie dokumentacji. Raczej trudne okazało się zaadaptowanie dawniejszych programistów do wykonywania zadań testowych. Wielu programistów AVID nadal skupia się na pojedynczej funkcji i szczegółach jej implementacji.

Uzyskane doświadczenie

- Wprowadzenie Scrum skróciło wewnętrzne cykle wydawnicze (tzn. sprinty) z kilku miesięcy do jednego miesiąca i mniej. Jednakże nie każdy sprint daje w wyniku wersję do publikacji na zewnątrz.
- Cała praca skupia się na zadaniach, do których zespół się zobowiązał.
- To z kolei oznacza, że zadania są czasami wybierane arbitralnie z listy zaległości, co prowadzi do przedkładania krótkoterminowego postępu w określonych funkcjach klienckich ponad długoterminową funkcjonalność podstawową i działania stabilizacyjne.
- Rozwiązanie starej struktury zespołów jest prawdziwym dobrodziejstwem. Programiści i testerzy pracują teraz ramię w ramię

i stale zapewniają sobie nawzajem informacje zwrotne. To powoduje szybsze reakcje na wewnętrzne i zewnętrzne raporty o usterkach.

- Z kolei śledzenie błędów jest teraz możliwe tylko w sprawach, które dotyczą wielu zespołów lub sprintów. Wszystkie inne błędy nie są dokumentowane, ale są natychmiast obsługiwane szybko i nieformalnie w ramach zespołów, co utrudnia identyfikowanie problemowych punktów zapalnych.
- Bliska współpraca z programistami i coraz większe dryfowanie pracy testerów w kierunku programowania czasami wywołuje programistyczny sposób myślenia wśród testerów. Niektórzy testerzy zaczynają patrzeć na sprawy z punktu widzenia programisty i wydają oświadczenia zaczynające się od słów „Nie możemy tego zrobić, ponieważ…". Zauważyliśmy też, że niektórzy testerzy nie chcą zgłaszać się podczas codziennego spotkania Scrum pod koniec sprintu i informować zespół, że funkcja nie jest gotowa.
- Zwiększona przeźroczystość zapewniana przez listę zaległości i karty zadań na tablicy pomaga każdemu. Otwarta kultura firmy AVID pomogła bardziej wycofanym członkom zespołów brać aktywniejszy udział w porannych spotkaniach.
- Rozwijanie podstawowej funkcjonalności musi się też odbywać przyrostowo i powinno dostarczać nową, ukierunkowaną na klienta funkcjonalność na końcu każdego sprintu.
- Stopień automatyzacji testów ogromnie się zwiększył. Jednakże wiele testów trudno jest zautomatyzować w sposób opłacalny, więc teraz przeprowadzamy dodatkowy sprint testowy przed każdym zewnętrznym wydaniem wersji produktu. Składa się on ze złożonych, zwłaszcza ręcznych testów, które nie mogą być przeprowadzane dla każdej kompilacji.
- Błędy i nieporozumienia nie są całkowicie wyjaśniane, ale są rozpoznawane i szybciej usuwane na wszystkich poziomach.
- Jeśli coś nie działa, reagujemy natychmiast i poprawiamy dany proces, choćby to oznaczało odejście od książkowego podejścia do Scrum. Na przykład w czasie przeprowadzania tego wywiadu zajmowaliśmy się sprawdzaniem niezależności swoich testów i sposobami poprawienia jednorodności testów. Warunki te wymagają przeprowadzenia sprintów w zespołach, które są tymczasowo łączone w tym celu. Ponieważ też scentralizowane standardy nadal działają lepiej w pewnych kontekstach, rozważamy, czy nie wprowadzić ponownie scentralizowanego testowania systemowego i obowiązków związanych z zapewnianiem jakości. Wciąż dyskutujemy, jak najlepiej podejść do tej zmiany.

Wnioski

Ogólnie rzecz biorąc Stephan Albrecht wyciąga pozytywne wnioski i mówi nam „Bliska współpraca pomiędzy programistami i testerami oraz umożliwiane przez to krótkie iteracje są głównymi zaletami użycia Scrum. Jednakże to podejście wymaga też, aby testerzy byli niezwykle obiektywni względem testowanych obiektów".

8.2 Testowanie systemowe non-stop – Wykorzystanie Scrum do opracowywania narzędzia Test*Bench*

Joachim Hofer, Menedżer Rozwoju TestBench
oraz Dierk Engelhardt, Menedżer Produktu TestBench w firmie imbus AG

imbus AG jest niemiecką firmą specjalizującą się w zapewnianiu jakości i testowaniu oprogramowania. Podczas pisania tej książki firma imbus miała ponad 200 pracowników pracujących w Niemczech, Sousse (Tunezja) i Szanghaju (Chiny). Firma oferuje usługi konsultingowe, testowania oprogramowania i outsourcingu testów, a także swoje własne narzędzia do testowania oraz szkolenia. Jej klientami są producenci oprogramowania i departamenty wytwarzające oprogramowanie w agencjach rządowych i firmach z różnych branż przemysłu.

Test*Bench* jest potężnym narzędziem do zarządzania testami opracowanym przez imbus. Obejmuje wszystkie aspekty procesu testowania, od planowania, projektowania i automatyzacji do wykonywania testów i raportowania wszelkich zadań związanych z testowaniem oprogramowania. Program Test*Bench* jest używany w branżach technologii medycznych, kolejowej, samochodowej oraz bankowości i ubezpieczeń.

Narzędzie to zostało opracowane przy użyciu języka Java przez zespół składający się z 12 do 16 członków. Produkt jest implementowany i integrowany u klientów przez specjalistycznych konsultantów Test*Bench*. Do roku 2010 produkt ten był opracowywany przy pomocy iteracyjnego procesu zorientowanego na fazy z programistami i testerami pracującymi w osobnych grupach i tworzącymi jedną lub dwie wersje programu rocznie.

Cele poprawy

Ten proces tworzenia oprogramowania powodował typowe problemy, z których znane są modele zorientowane fazowo. Testowanie systemowe zaczynało się dopiero, gdy implementacja była gotowa, a programiści świętowali już ukończoną iterację. Testerzy systemowi

regularnie ostudzali entuzjazm programistów przekazując raporty o usterkach. Z punktu widzenia programistów raporty te przychodziły zbyt późno i często dotyczyły kwestii, które wymagały interwencji, a na które nie było już dostępnego czasu w bieżącej iteracji. Każda iteracja była więc dzielona na fazę implementacji i następującą po niej fazę poprawiania błędów, przy czym ta druga odbywała się pod ogromną presją ze względu na ograniczony czas i konieczność doprowadzenia do sukcesu. W wyniku otrzymywano stabilny produkt wysokiej jakości, ale proces jego tworzenia zawsze wiązał się z dużym wysiłkiem.

Aby złagodzić tę sytuację, wyznaczono jako cel równoległe wykonywanie zadań programistycznych i testowych. Chodziło o zapewnienie programistom szybszych informacji zwrotnych z testowania i dalsze zautomatyzowanie procesu testowania, aby zapewnić większe bezpieczeństwo refaktoringu kodu dla przyszłych wersji produktu.

Wprowadzenie zwinnych technik programowania

Począwszy od roku 2010 menedżer Joachim Hofer zaczął wprowadzać cały szereg praktyk zwinnych w celu osiągnięcia tych celów:

- **Zarządzanie wymaganiami** Wcześniej wymaganie przybierało formę nagłówka w narzędziu do zarządzania wymaganiami Caliber połączonego ze szczegółowym dokumentem programu Word. Implementacja mogła zostać rozpoczęta dopiero, gdy cały dokument został zatwierdzony. Podjęto decyzję o odejściu od dużych dokumentów dotyczących wymagań i przejściu na mniejsze scenariusze użytkowników budowane i rejestrowane przy użyciu narzędzia do zarządzania problemami Jira.
- **Nocna kompilacja** Zacieśniono też reguły tworzenia conocnych kompilacji. Istniało też centralne środowisko kompilacji i integracji, a programiści rejestrowali swój kod, gdy był gotowy (średnio co dwa, trzy dni). To podejście tworzyło pewną ilość niedokończonego kodu, który nie był jeszcze zgłoszony, więc podjęto decyzję, aby wszyscy programiści zgłaszali cały swój kod co wieczór. Istniejące zautomatyzowane testy jednostkowe były następnie przeprowadzane na całym kodzie, co początkowo prowadziło do różnego rodzaju problemów. Jednakże było to cenne doświadczenie, które nauczyło zespół codziennej pracy w kierunku ukończonego kodu wykonywalnego. Pakiety kodu, z którymi programiści zmagali się każdego ranka, stawały się mniejsze, a planowane zmiany były w większości przypadków gotowe na wieczór.
- **Nocne zautomatyzowane testy systemowe** Oprócz dalszego automatyzowania testowania jednostkowego zespół doprowadził też do automatyzacji testowania systemowego. Środowisko

testowe zostało rozszerzone tak, aby po testach jednostkowych i integracyjnych następowały conocne, zautomatyzowane testy systemowe uruchamiane bezpośrednio ze środowiska kompilacyjnego. Wszystkie nowe testy systemowe były od razu projektowane i implementowane przy użyciu narzędzia QF-Test tak, aby można je było dołączać do środowiska conocnych testów.

- **Ciągła integracja** Pierwotnie wytworzenie każdej kompilacji zajmowało cztery godziny, co pozwalało na przeprowadzanie kompilacji co noc, ale było zbyt powolne, aby zapewnić, że nocne testy będą gotowe, zanim praca rozpocznie się następnego dnia. Czas kompilacji został ograniczony do 15 minut przez podzielenie każdej kompilacji na ciąg podprojektów i przebudowanie środowiska budowania jako środowiska Jenkinsa/Hudsona. Wszystkie zautomatyzowane testy jednostkowe, integracyjne i systemowe zostały przepakowane i osadzone w poprawionym środowisku kompilacji. W zależności od wykonywanego pakietu testów czasy reakcji zostały ograniczone do 15 minut – tzn. nie dłużej niż przerwa na kawę!
- **Statyczna analiza kodu i mierzenie pokrycia** Dynamiczne testy jednostkowe w środowisku ciągłej integracji zostały rozszerzone tak, aby obejmowały dodatkową statyczną analizę kodu działającą równolegle do zautomatyzowanego testowania integracyjnego. Na przykład narzędzie FindBugs jest wykorzystywane do wynajdywania typowych problemów z kodowaniem w języku Java, takich jak nieprawidłowe wywołania API.
- **Zorientowanie na zadania** Aby osiągnąć wystarczająco dokładną kontrolę zadań w każdej iteracji, wprowadzono zorientowane na zadania metody pracy oparte na scenariuszach użytkowników. W celu nadawania priorytetów zadaniom wprowadzono system punktacji, który ocenia priorytet zadania biorąc pod uwagę punkt widzenia klienta, liczbę związanych z nim wymagań oraz wewnętrzne głosowanie w ramach zespołu.
- **Codzienne spotkanie** Zespół spotyka się teraz każdego ranka na 15 minut. Każdy członek zespołu raportuje, nad czym aktualnie pracuje, jak postępują prace i jakie napotyka problemy. Każdy sam za siebie decyduje, które zadania omawiać. Na razie nie ma prawdziwego planowania sprintu, do którego można odnieść raportowane zadania, ale codzienna natura spotkań sprawia, że jest to dobre przygotowanie do nadchodzących codziennych spotkań Scrum.

Wszystkie te praktyki zostały wprowadzone przy zachowaniu pierwotnego, iteracyjnego modelu tworzenia oprogramowania. Jednakże wykorzystanie scenariuszy użytkowników oraz ciągłej integracji

spowodowało rozwiązanie ścisłych faz wcześniejszego procesu tworzenia oprogramowania i pomogło je poprzeplatać i zrównoleglić. Następny krok obejmował wprowadzenie zwinnych praktyk zarządzania produktami i projektami oraz przekształcenie zespołu w jednostkę samoorganizującą się.

Wprowadzenie Scrum

Nowe techniki tworzenia oprogramowania wymienione wyżej były wprowadzane głównie przez menedżera Joachima Hofera. Nadszedł czas, aby zaangażować cały zespół w proces Scrum. Joachim Hofer i menedżer produktu Dierk Engelhardt postanowili dokonać tego przejścia w kilku odrębnych krokach:

- **Badanie i informowanie** Rozważano Kanban, Scrum, XP i inne metodyki zwinne, ale dyskusja i dalsze planowanie skupiły się szybko na metodyce Scrum. Zorganizowano wewnętrzne forum, na którym zespół mógł dodawać swoje przemyślenia i zgłaszać pomysły dotyczące sposobów wdrożenia nowego procesu tworzenia oprogramowania opartego na Scrum. Wszyscy członkowie zespołu kierowani własną ciekawością czytali literaturę dotyczącą Scrum i odwiedzali witryny internetowe takie jak scrum.org, aby dowiedzieć się, na czym polega metodyka Scrum. Procesowi temu towarzyszyły wewnętrzne warsztaty i regularne dyskusje w zespole.
- **Restrukturyzacja zespołu** Przejście na Scrum wymagało oczywiście zmiany ról, obowiązków i zadań wewnątrz zespołu. Joachim Hofer został mistrzem Scrum, menedżer testów został jego zastępcą, a menedżer produktu przejął rolę właściciela produktu. Podział między testerami i programistami został obalony i wprowadzono programowanie w parach, gdzie pary tester/programista zwykle odpowiadają za zadania testowania i integrowania, pary składające się z dwóch programistów za zadania programistyczne, a pary testerów za zadania testowania systemowego.
- **Sprinty** Nie ma idealnego albo łatwego momentu na porzucenie starych przyzwyczajeń i rozpoczęcie pierwszego sprintu – trzeba się po prostu w którymś momencie na to zdecydować. W firmie imbus pierwsze spotkanie planowania sprintu odbyło się w poniedziałek na początku 2011 roku. Właściciel produktu przeniósł już najważniejsze jego zdaniem wymagania ze starego planu projektu do listy zaległości projektu, która została przekształcona przez zespół w zbiór kart zadań dla czterotygodniowego sprintu podczas pierwszej jednodniowej sesji planowania sprintu. Później zespół przeszedł na trzytygodniowy cykl sprintów.

Główne zmiany

- Zniesienie różnic pomiędzy tradycyjnymi rolami testerów i programistów oraz przyjęcie uniwersalnej roli członków zespołu zostało powszechnie uznane za najpoważniejszą zmianę. Wprowadzenie programowania w parach odgrywało ważną rolę w udanym przejściu na nową metodykę.
- Programowanie w parach uczyniło przeglądy kodu regularną częścią przepływu zadań. Każdy programista jako rzecz oczywistą przekazuje nowy kod swojemu partnerowi do przeglądu. Ta praktyka zachęca też do dzielenia się wiedzą wewnątrz zespołu.
- Podejście wykorzystujące kamienie milowe w planowaniu zadań zostało zastąpione przez podejście zorientowane na zadania. Ograniczyło to znaczenie narzędzia do zarządzania wymaganiami Caliber i zwiększyło znaczenie narzędzia do zarządzania problemami Jira oraz wtyczki Greenhopper (do zarządzania zaległościami, rankingiem zadań, tablicą zadań oraz metrykami/wykresami).
- Wprowadzono główne narzędzia Scrum: listę zaległości, planowanie sprintu/poker planowania, retrospektywy i zaczęto je odtąd stosować w trwały i zdyscyplinowany sposób.
- Scenariusze użytkowników pozwoliły na wprowadzenie ściślejszych reguł tworzenia oprogramowania. Od samego początku zespół musiał pisać przypadki testowe dla każdej zmiany kodu lub nowego scenariusza użytkownika w narzędziu Jira. Wcześniej członkowie zespołu mogli pisać testy po zakończeniu zmiany w kodzie. Teraz para programista/tester odpowiada za pisanie testów jednostkowych, integracyjnych i systemowych z góry dla każdego scenariusza użytkownika. Programiści wykorzystują narzędzie TestNG do implementowania oraz przeprowadzania testów jednostkowych i integracyjnych, natomiast testerzy automatyzują testy systemowe przy użyciu narzędzia QF-Test.
- Wprowadzenie ciągłej integracji umożliwia równoległą realizację zadań programistycznych i testowych. Za każdym razem, gdy zmiana w kodzie jest zatwierdzana w systemie, uruchamiane jest budowanie produktu. Składa się ono co najmniej z kompilacji i testów jednostkowych, a przekazanie informacji zwrotnej do programisty zajmuje około trzech minut. Później następują trwające 15-30 minut testy integracyjne.
- Oprócz wywoływanych przekazaniem kodu przez programistów uruchomień ciągłej integracji, które mają miejsce kilka razy dziennie, zespół uruchamia też budowanie co noc. Każdy przebieg budowania produktu obejmuje uruchomienie maszyny wirtualnej ze świeżo zainstalowanym systemem operacyjnym i zainstalowanie

ostatniej udanej wersji produktu pochodzącej z procesu ciągłej integracji. Następuje zautomatyzowane testowanie systemowe sterowane przez narzędzie TestBench i składające się obecnie z 15000 kroków sterowanych danymi, których ukończenie zajmuje około 10 godzin. Kod jest obsługiwany automatycznie i obecne testy osiągają około 40% pokrycia wierszy kodu. Wykonywane są zrzuty wideo dla nocnych testów systemowych, które umożliwiają pracownikom wizualne prześledzenie nieudanych testów następnego ranka. Całkowite pokrycie kodu w ramach środowiska ciągłej integracji wynosi około 60% i osiąga niemal 100% dla implementowanych na nowo funkcji lub scenariuszy użytkownika. Starszy kod, dla którego nie istnieją zautomatyzowane testy, obniża niestety ogólne pokrycie kodu.

- Specyfikacje testów wykorzystywane wcześniej zostały w dużej mierze zastąpione kodem testowym z komentarzami, a w przypadku testów systemowych specyfikacjami testów sterowanych słowami kluczowymi. Narzędzie Test*Bench* jest ściśle powiązane z Jira oraz środowiskiem Jenkins i jest wykorzystywane do zarządzania wszystkimi testami systemowymi.
- Testy systemowe nie są całkowicie zautomatyzowane i w przyszłości testy ręczne nadal będą konieczne do sprawdzania wykresów, raportów, użyteczności, itd., więc dodatkowy półdniowy, oparty na sesjach, badawczy test systemowy odbywa się na końcu każdego sprintu. Przejście z iteratywnej do zwinnej metodyki tworzenia oprogramowania ograniczyło jednak drastycznie nakłady na testowanie ręczne z kilku osobo-tygodni do jednego osobo-dnia raz na trzy tygodnie.
- Sprinty trwają trzy tygodnie i dają w wyniku przetestowaną wewnętrzną wersję produktu. Tak jak wcześniej, wersje zewnętrzne są wypuszczane dwa razy do roku. Zmiany w wymaganiach klienta mogą zostać wprowadzone na początku sprintu (czyli na trzy tygodnie przed dostarczeniem).

Uzyskane doświadczenie

- Wprowadzenie zwinnych technik tworzenia oprogramowania (takich jak ciągła integracja) przed przejściem na wykorzystanie Scrum dało nam stabilną podstawę do rozbudowywania sprintów od samego początku.
- Konfigurowanie infrastruktury narzędziowej (głównie dla ciągłej integracji) wymagało znacznego wysiłku. Prace z tym związane ograniczają potencjalną wydajność pierwszych kilku sprintów.
- Programowanie w parach jest kluczowe dla powodzenia zespołu, choć niektórym członkom zespołu łatwiej jest pracować

z pewnymi kolegami, a nie ze wszystkimi. Trzeba pozwolić na oddolne tworzenie się par, a nie przymuszać ludzi do współpracy.
- Programowanie sterowane testami poprawia ogólną architekturę kodu, a liczba usterek, którymi trzeba zarządzać, została znacznie ograniczona.
- Regularne retrospektywy sprintów dają ciągły strumień pomysłów na poprawienie różnych elementów (np. poprawione szacowanie nakładów lub omawianie pytań typu „Czym dokładnie jest punkt scenariusza?").
- Scrum nie ogranicza ogólnego nakładu pracy i nie tworzy dodatkowych zasobów! Jednakże końcowym efektem pracy jest faktycznie ukończony produkt. Nie musimy już sobie radzić z dużą liczbą usterek i żmudnych rund poprawiania błędów.
- Mapy drogowe produktu zostały zastąpione planowaniem strategicznym.
- Przejście na Scrum wymaga zaangażowania ze strony menedżerów oraz zespołu i zajmuje czas. Wymaga też badań, szkoleń i konfigurowania nowej infrastruktury, czego nie da się osiągnąć z dnia na dzień.

Wnioski

Po rocznym doświadczeniu z metodyką Scrum Dierk Engelhardt i Joachim Hofer potwierdzili, że byli w stanie osiągnąć cele, które sobie założyli. Zespół wykorzystuje teraz ważne techniki zwinne, takie jak programowanie sterowane testami i ciągłą integrację w zdyscyplinowany i trwały sposób. Programowanie sterowane testami i strategia „zero usterek" są szczególnie skuteczne, a integracja narzędzi Test*Bench* i Jenkins pozwala w praktyce zrealizować ideę testowania non-stop (czyli ciągłego, zautomatyzowanego testowania jednostkowego, integracyjnego i systemowego). Rozbudowany refaktoring również stał się mniej ryzykowny, a wysiłki związane z tworzeniem zewnętrznego wydania produktu zostały znacznie ograniczone. Zespół jest naprawdę zadowolony z nowych ustaleń.

Jednakże zespół nie zamierza spocząć na laurach i już planuje następne usprawnienia procesu tworzenia oprogramowania. Obejmuje to wykorzystanie narzędzia Atlassian Confluence do opisywania wymagań systemu i dalszego przyspieszania procesu ciągłej integracji przez zrównoleglanie testów i aktualizacje sprzętowe serwera służącego do testów i budowania.

8.3 Wykorzystanie Scrum przy tworzeniu sklepu internetowego

Sabine Herrmann,
Zwinny tester w firmie zooplus AG w Monachium

zooplus AG jest firmą zajmującą się handlem elektronicznym w zakresie sprzedaży detalicznej wszelkiego rodzaju produktów dla zwierząt domowych i ich właścicieli. Firma oferuje około 8000 produktów dla psów, kotów, ptaków, gadów, koni, małych gryzoni i ryb, w tym pasze znanych firm, produkty pod własną marką, zabawki, produkty do pielęgnacji i rozmaite inne akcesoria. Witryna internetowa firmy zawiera też wiele darmowych informacji na tematy weterynaryjne, a także interaktywne aplikacje, fora i blogi.

Firma została założona w roku 1999 i początkowo skupiała się na rynku niemieckim i austriackim, ale stale rozszerzała działalność na inne rynki europejskie począwszy od roku 2005. Obroty firmy rosły w tempie 33% rocznie przez ostatnie cztery lata, a liczba pracowników też odpowiednio wzrosła.

zooplus AG obecnie zatrudnia 200 osób, spośród których 51 pracuje w dziale informatycznym, a 102 w marketingu.

Jak było wcześniej

Szybkiemu wzrostowi firmy w ostatnich 10 latach towarzyszyło też szybkie rozszerzanie infrastruktury informatycznej.

Przed przejściem na metodykę Scrum w roku 2008 istniały osobne zespoły obsługujące tworzenie oprogramowania sklepowego i wspierającego dla dziewięciu sklepów internetowych firmy. Był też osobny zespół odpowiedzialny za testowanie nowej funkcjonalności sklepów (testowanie integracyjne) i podstawowej funkcjonalności (testowanie regresji).

Tworzenie oprogramowania w tym czasie odbywało się zgodnie z tradycyjnym modelem trójfazowym:

Programowanie → Testowanie → Produkcja.

Problemy istniejące przed przejściem na Scrum:

- Nie było jasno zdefiniowanego procesu zarządzania zmianami.
- Nie było prawdziwej koordynacji pomiędzy zespołami programistycznymi i testowymi (na przykład przy nadzorowaniu problemów z zarządzaniem wersjami).
- Było zbyt wielu wysoce wyspecjalizowanych ekspertów, a zbyt mało wysiłku zespołowego.

- Wraz z firmą szybko rosła funkcjonalność oprogramowania coraz bardziej utrudniając testy regresji.

Przejście na Scrum

Przejście na metodykę Scrum odbyło się w dwóch fazach:

Faza 1

Zespoły tworzące oprogramowanie sklepów i oprogramowanie zaplecza zaczęły wykorzystywać elementy Scrum, wprowadzono codzienne spotkania Scrum, mianowano dwóch mistrzów Scrum i wprowadzono narzędzie do śledzenia błędów Jira. Wymagania były dokumentowane w dedykowanym serwisie Wiki.

Faza 2

Utworzono nowe wielofunkcjonalne zespoły przy pomocy trenerów Scrum, a pod koniec roku 2009 nastąpił „wielki wybuch" i zespół przeszedł całkowicie na metodykę Scrum.

Pierwszych kilka sprintów było przeprowadzanych przez trzy zespoły Scrum przy założeniu, że każdy zespół może robić wszystko i odchodzimy od odizolowanych specjalności. Procesowi temu towarzyszyło wiele dyskusji na temat nowych procesów dokumentowania wymagań i zarządzania zmianami.

Następujące problemy pojawiły się po zmianie metodyki:

- Nie wszystkie działy uczestniczyły we wprowadzeniu nowych procesów, co sprawiło, że zmiany procesów tworzenia oprogramowania były mniej przeźroczyste, niż powinny.
- Zewnętrzny zespół testujący nie był odpowiednio zaangażowany i w efekcie stanął przed nowym wyzwaniem przeprowadzania testów dla wszystkich trzech zespołów.
- Jeden właściciel produktu nie wystarczał, aby zająć się pracą z trzema zespołami, więc zespoły przeprowadzały wiele własnych analiz podczas sprintów.

Pozytywnymi efektami przejścia były:

- Zmniejszenie rozproszenia specjalistycznej wiedzy.
- Zastosowanie sprintów i listy zaległości produktu zapewniło zwiększoną przeźroczystość procesu tworzenia oprogramowania dla zarządu i wszystkich zainteresowanych stron.
- Funkcje oprogramowania stały się łatwiejsze do planowania.
- Wczesne testowanie poprawiło jakość.
- Istniejące przyzwyczajenia związane z pracą zostały zarzucone.
- Wiedza domenowa jest dostępna wewnątrz zespołów ekspertów.

- Nowy nacisk na pracę zespołową i techniki Scrum dobrze się przyjął.
- Synchronizacja pomiędzy zespołami została znacząco poprawiona.

Jak to wygląda dzisiaj

Stały wzrost wymagań doprowadził do zwiększenia działu informatycznego i obecnie tworzenie oprogramowania odbywa się w pięciu zespołach Scrum.

Podczas przechodzenia na Scrum było dużo dyskusji i oporu, ale ostatecznie nikt nie opuścił firmy. Dzisiaj zmiany są na ogół bardziej akceptowane. Zespoły pracują inaczej, a retrospektywy, które odbywają się na końcu każdego projektu, pomogły w ustanowieniu procesu ciągłych zmian i usprawnień.

Zewnętrzny zespół testujący nie był już dłużej w stanie zajmować się wszystkimi zadaniami testowania dla pięciu oddzielnych zespołów Scrum, więc wprowadzono rolę zwinnego testera w każdym zespole.

Zwinne testowanie doprowadziło do całkowitej likwidacji oddzielnych faz programowania i testowania oraz umożliwiło kończenie danego scenariusza w obrębie sprintu. Testowanie kryteriów akceptacji jest teraz integralną częścią definicji wykonania produktu.

Obecnie testujemy znacznie wcześniej, a wprowadzenie testera do każdego zespołu oznacza, że jest stała, bezpośrednia wymiana informacji pomiędzy programistami a testerem. Tester jest mocno zaangażowany w planowanie każdego scenariusza i dokumentowanie kryteriów akceptacji, jest także w ciągłym kontakcie z właścicielem produktu i innymi zainteresowanymi stronami.

Automatyzacja testów regresji podczas sprintów umożliwia dostarczanie nowych funkcji szybciej niż dawniej. Cały zespół rozwinął lepsze rozumienie procesu testowania, co pozwoliło osiągnąć większą synergię pomiędzy programistami a testerami.

Zapewnianie jakości jest teraz wspólnym obowiązkiem całego zespołu, a nie tylko zespołu testującego i w efekcie zespół stale poszukuje sposobów poprawienia swoich procesów.

Zwinni testerzy utworzyli wirtualny zespół testowy. Scrum testerów spotyka się regularnie w celu omawiania automatyzacji testów i ogólnych zagadnień związanych z testowaniem.

Wnioski

Włączenie zarządu w procesy zmian było krytycznym czynnikiem powodzenia przejścia do metodyki Scrum. Samo przejście nigdy nie zostało naprawdę zakończone i jest raczej procesem stałego doskonalenia.

Wszystkie procesy są pod ciągłą obserwacją i mogą być zawsze optymalizowane, co prowadzi do tworzenia uniwersalnego procesu łączącego się idealnie z istniejącą kulturą zespołów i całej firmy.

8.4 Wprowadzenie Scrum w firmie ImmobilienScout24

Eric Hentschel,
Inżynier testowy w ImmobilienScout24 w Berlinie

ImmobilienScout24 jest największym internetowym pośrednikiem w handlu nieruchomościami w krajach niemieckojęzycznych. Według comScore witryna internetowa odnotowuje ponad 7,5 miliona unikalnych odwiedzających miesięcznie. Mająca siedzibę w Berlinie firma działa w sieci od ponad 13 lat i ma ponad 500 pracowników.

Około 160 z nich jest zatrudnionych w 22 zespołach informatycznych opartych na metodyce Scrum. Zwinne opracowywanie produktów jest częścią filozofii firmy od 2009 roku i obejmuje wykorzystanie hybrydowego procesu „ScrumBan" – mieszaniny Scrum i Kanban. Każdy zespół Scrum zwykle składa się z czterech, pięciu lub sześciu programistów, właściciela produktu, inżyniera do spraw testów, projektanta, architekta i menedżera aplikacji.

Powody wprowadzenia Scrum

Firma ImmobilienScout24 przez lata dynamicznie rosła. Do roku 2009 tworzenie oprogramowania opierało się na tradycyjnym modelu V, a pracownicy w osobnym departamencie zapewniania jakości odpowiadali za analizę testów, automatyzację testów, testowanie i zarządzanie testami. Analiza testów była przeprowadzana przez inżyniera piszącego przypadki testowe w oparciu o dokumentację wymagań napisaną przez menedżera produktu oraz tworzącego sekwencje przypadków testowych jako podstawę do automatyzacji testów. Grupa automatyzująca testy wykorzystywała następnie te sekwencje do opracowywania zautomatyzowanych testów. Komunikacja pomiędzy menedżerem produktu, zespołem programistów i zespołem zapewniania jakości/testowania była często ograniczona do kontekstu samego testu, co oznaczało, że niekompletne lub nieprawidłowo interpretowane wymagania były odkrywane dopiero na etapie testowania systemu. Podobnie do wielu firm z systemami programowymi, firma ImmobilienScout24 musiała zmagać się z długimi cyklami tworzenia oprogramowania, zbyt dużymi projektami, zapewnianiem jakości na późnym etapie, skomplikowaną i nieaktualną dokumentacją oraz długimi cyklami wydawniczymi,

które często wynikały z użycia modelu V do opracowania produktu. Te czynniki utrudniały szybką reakcję na potrzeby klientów i usuwanie usterek w portalu internetowym.

Niemniej jednak firmie wciąż udawało się utrzymywać solidne środowisko testowania. Przypadki testowe były zarządzane przy użyciu narzędzia HP Quality Center oraz automatyzowane przy użyciu HP Quick Test Professional. Do czasu wprowadzenia Scrum systemy te umożliwiały firmie przeprowadzanie niezawodnych, zautomatyzowanych testów regresji.

Przebieg wszystkich testów regresji (ręcznych i zautomatyzowanych) zwykle zajmował około 12 godzin, ale nie mógł być swobodnie powtarzany bez ręcznego resetowania testowych baz danych. Całkowity czas związany z budowaniem nowej wersji produktu, wdrażaniem jej w środowisku testowym, oceną wyników błędnych testów i dostarczaniem pełnej informacji zwrotnej zespołowi programistycznemu zwykle zajmował około 3-5 dni. Czasy reakcji były do przyjęcia, ale średni czterotygodniowy cykl wydawniczy dla nowej wersji portalu był nadal zbyt długi. Każdy czterotygodniowy cykl zawierał czterech lub pięciu kandydatów na wersje finalne, zawierających poprawki błędów, a czasami też nowe funkcje. Cały proces testowania był przeprowadzany dla każdego kandydata na wersję. Czterotygodniowy cykl wydawniczy jest niespełnionym marzeniem dla wielu producentów oprogramowania, ale portale internetowe często działają na zupełnie innej skali czasowej. Istotne jest przechodzenie na szybszy, bardziej zwinny system tworzenia oprogramowania, żeby nadążyć za szybko uciekającym światem handlu internetowego.

Wprowadzenie zmian

Przejście na metodykę Scrum zaczęło się od projektu pilotażowego obejmującego pojedynczy zespół. Na początek zarządzanie produktem, programowanie i zapewnianie jakości pozostało w różnych fizycznych lokalizacjach, ale komunikacja pomiędzy nimi znacząco się poprawiła dzięki regularnym spotkaniom w sprawach szacowania nakładów i planowania sprintów. Ponieważ były one nadal osadzone w istniejących nieelastycznych procesach analizy i automatyzacji testów, wciąż problem stanowiły niewygodne procesy automatyzacji i zapewniania jakości.

W drugiej fazie dodatkowe zespoły i inne departamenty dołączyły do metodyki Scrum i fizyczny podział pomiędzy różnymi zainteresowanym stronami został rozwiązany. W tym kontekście oddzielny departament zapewniania jakości był dłużej nie do utrzymania, a każdy zespół Scrum otrzymał swojego własnego inżyniera do spraw testów, który odpowiadał za wszystkie zadania związane z testami (analizę,

automatyzację, zarządzanie i samo testowanie). Wbudowanie zadań testowania w strukturę zespołu znacznie przyspieszyło proces opracowywania produktów, ale stosowanie wciąż narzędzia HP Quality Center stanowiło wąskie gardło w procesie testowania. Własny język skryptowy tego narzędzia uniemożliwiał adaptację testów wewnątrz zespołu programistycznego, a zmiany wymagały interwencji dedykowanego eksperta od automatyzacji testów, co obniżało ogólny poziom akceptacji tego narzędzia. Problem ten został pokonany przez zastąpienie narzędzia HP opartym na języku Java interfejsie Selenium 2.0 WebDriver API. Ta platforma testowa jest stale rozwijana przez aktywną społeczność i cieszy się dużym stopniem poparcia wśród testerów i programistów zespołu. Gdy inżynier do spraw testów był w stanie przygotować wstępnie napisane przypadki testowe, cały zespół programistyczny mógł tworzyć i zarządzać zautomatyzowanymi testami, przyspieszając w ten sposób proces automatyzacji testów.

Zespoły programistyczne w ImmobilienScout24 mogą swobodnie wybierać swoje metody pracy, ale większość wykorzystuje dwutygodniowe cykle Scrum. Niektóre zespoły łączą techniki Scrum i Kanban tworząc zwinny system bez wstępnie zdefiniowanych cyklów (sprintów). Te zespoły przeprowadzają ustalanie priorytetów na liście zaległości i szacowanie nakładów w taki sam sposób, jak typowe zespoły Scrum, ale nie definiują z góry liczby scenariuszy, które mają być zawarte w następnym sprincie. Zamiast tego po prostu pracują nad zaległościami sprintu od góry do dołu, scenariusz po scenariuszu.

Przeszkody

W przypadku pracowników zapewniania jakości początkowa faza przejścia na metodykę Scrum była ogromnym wyzwaniem, a nowo mianowani w zespołach inżynierowie do spraw testów otrzymali ogromne poszerzenie swoich uprawnień. Analitycy testów osadzeni w każdym zespole również musieli zdobyć odpowiednią wiedzę dotyczącą programowania umożliwiającą im opracowywanie zautomatyzowanych testów nie tracąc przy tym z oczu analizy testów. Czynnik ludzki podczas dokonywania takiej transformacji nie powinien być lekceważony – w firmie ImmobilienScout24 około roku zajęło zespołom przystosowanie się do nowego wielofunkcjonalnego środowiska zwinnego.

Zastąpienie narzędzia HP Quality Center przez WebDriver i ogólne zalety zwinnej metodyki znacznie poprawiły szybkość i jakość tworzenia oprogramowania w zespołach.

Zależności istniejące pomiędzy poszczególnymi zespołami są funkcją złożoności opracowywanych aplikacji i nadal pozostają problemem. Natura architektury systemu utrudnia wyraźne oddzielanie zespołów i opracowywanych przez nie podsystemów. Oddzielne testowanie

systemowe obejmujące prace wszystkich zespołów programistycznych nadal jest istotne. Ten typ testów będzie też konieczny w przyszłości, ale zamierzamy ograniczyć jego złożoność, na ile to możliwe.

Rozwiązanie technicznych zależności pomiędzy różnymi elementami portalu jest ogromną częścią wyzwania związanego z przejściem na metodykę zwinną. Historycznie pomiędzy jedną trzecią a jedną piątą dostępnych zasobów poświęcano na projekty związane z systemami informatycznymi. Pomimo trwającej modularyzacji niektóre większe projekty nie zostały nigdy zakończone, a wynikłe z tego tarapaty doprowadziły do utworzenia specjalnych zespołów i grup zadaniowych do rozwiązywania problemów. Wcześniejsza monolityczna struktura platformy została obecnie znacznie zmodularyzowana, choć proces ten nadal nie został całkowicie zakończony. Opracowywanie baz danych jest nadal szczególnie podatne na zakorkowanie, co oznacza, że przestoje wdrożeniowe nadal się czasem zdarzają.

Dalsza automatyzacja procesu testowania również okazała się trudna zwłaszcza w odniesieniu do odtwarzalności zautomatyzowanych testów dla podstawowej funkcjonalności – zadania te zostały dodatkowo skomplikowane przez niespójną dostępność systemów testowania i niekompletne dane testowe. Złożone struktury bazodanowe, czasochłonne importowanie i wycofywanie danych również ograniczyły możliwość odtwarzania testów. Nie było też dostępnej możliwości równoległego testowania.

Z drugiej strony pokusa automatyzowania po prostu wszystkiego przeważała na początku zmiany metodyki na Scrum zwłaszcza w odniesieniu do programowania. To prowadziło do tworzenia dużej liczby zadań bez ustalonych priorytetów, co nieuchronnie powodowało niestabilne warunki testowania. Podczas analizowania wyników dużej liczby testów trudno jest rozróżnić pomiędzy usterkami, za które odpowiada aplikacja, a tymi, które są powodowane przez wewnętrzną niestabilność systemu. Ponadto bardzo duża liczba testów jest niezwykle trudna do zarządzania i generuje coraz więcej wyników bez żadnego znaczenia. Próby ustabilizowania niestabilnych testów zwykle się nie udają ze względu na liczbę i złożoność potencjalnych przyczyn.

Główne zmiany

Metodyka Scrum poprawiła komunikację pomiędzy wszystkimi biorącymi udział w procesie wytwarzania oprogramowania i ogólnie sprawiła, że wytwarzanie oprogramowania stało się szybsze i wydajniejsze. Mniejsze scenariusze użytkowników, które są omawiane na poziomie zespołu, powodują, że wymagania są jaśniejsze i ograniczają ilość dokumentacji. Wielofunkcjonalna natura Scrum sprawia, że wszyscy członkowie zespołu są bardziej świadomi pracy wykonywanej przez

innych – ten czynnik przyczynił się do ogromnego wzrostu jakości produktu.

Minusem tego podejścia jest to, że poszczególni członkowie są zorientowani bardziej na swoje własne zespoły i trudniej im pomagać innym zespołom. Problemy spoza zespołu nie są istotne dla inżynierów do spraw testów, co prowadzi nas bezpośrednio do pytania, jak bardzo niezależne powinny być zespoły. Fakt, że każdy zespół ma względnie swobodny wybór narzędzi i języków programowania utrudnia zmianę tego nastawienia ukierunkowanego na zespół.

Wymagania stawiane zarządowi również się zmieniły. Rosnąca niezależność zespołów początkowo prowadziła do ograniczenia wysiłków koordynacyjnych, ale zagrożenie izolacją niektórych zespołów doprowadziło do ponownego wzrostu wysiłków koordynacyjnych. Zarząd musi pozwolić, aby zespoły programistyczne nadal pracowały niezależnie, zapewniając przy tym, że będą respektować priorytety właściciela produktu i współpracować ze sobą nad osiągnięciem celów firmy. Koordynację można osiągnąć korzystając z różnych środków, w tym cotygodniowej zwinnej narady w sprawie usuwania przeszkód wykorzystywanej przez zarząd IT i mistrzów Scrum do naprawiania przeszkód organizacyjnych. Mistrzowie Scrum organizują też codzienne spotkania w celu omówienia bieżących problemów i zapobiegania potencjalnym wąskim gardłom. Menedżerowie produktów organizują cotygodniowe narady epickie, podczas których właściciele produktów omawiają aktualny stan produktów i wszelkie planowane rozwiązania. Tworzenie oprogramowania jest wspierane przez architektów systemowych, którzy dbają, aby praca poszczególnych zespołów pasowała do ogólnej koncepcji przedsiębiorstwa.

Wprowadzenie Scrum skróciło cykl wydawniczy o 25% – do trzech tygodni. W ostatnich trzech latach został on dodatkowo skrócony do jednego tygodnia. Testowanie wersji produktu odbywa się teraz w dwóch fazach. Początkowe testy regresji są przeprowadzane jako część rozwijania funkcji podczas sprintu, zanim zintegrowana wersja zostanie poddana końcowemu testowi regresji na dedykowanym systemie próbnym. Przybiera to formę testów badawczych (ręcznych) przeprowadzanych przez istniejący zespół składający się z jednego menedżera testów i pięciu testerów, po których następuje ponowne uruchomienie zautomatyzowanych testów.

Zmiany procedur testowych

Testy są teraz integralną częścią definicji wykonania determinującej, które kroki są konieczne do ukończenia scenariusza. Integracja zapewniania jakości w każdym kroku procesu ukazuje usterki możliwie wcześnie przy zachowaniu aktualności testów i łatwości ich

modyfikacji. To podejście zapewnia informację zwrotną, jeśli testy są niewystarczające lub zbyt kompleksowe. Ponieważ jakość jest atrybutem, na którym koncentrują się też programiści, łatwiej jest spójnie implementować zwinną piramidę testowania.

Pomysł, aby pojedynczy inżynier do spraw testowania miał odpowiadać za wszystkie testy funkcjonalne i niefunkcjonalne, okazał się niemożliwy do zaimplementowania. Ogromna różnorodność testów zabezpieczeń, obciążeń, wydajności oraz zautomatyzowanych i badawczych testów systemowych nie jest możliwa do opanowania przez jedną osobę. Rozwiązanie leży w przekazaniu innym dedykowanym specjalistom przeprowadzania różnych dodatkowych typów testów takich jak testy zabezpieczeń.

Uzyskane doświadczenie

Nauka płynąca z restrukturyzacji naszej głównej aplikacji i wprowadzenia metodyki zwinnej w naszych zespołach doprowadziła do skonstruowania całkiem nowego portalu. Nasz debiut rynkowy w Austrii w 2012 roku dał nam okazję do zbudowania nowego produktu całkowicie zgodnie z praktykami zwinnymi. Na potrzeby projektu austriackiego skonstruowaliśmy środowisko umożliwiające ciągłą integrację od samego początku, a wszyscy przyjęli zorientowaną na jakość kulturę tworzenia oprogramowania. Programowanie jest sterowane testami i prowadzone zgodnie ze zwinną piramidą testowania. Wiele agenckich modułów umożliwia równoległe testowanie, dając nam szybką informację zwrotną i nowe kompilacje przy minimalnych zmianach. Im mniej zmian dana kompilacja zawiera, tym łatwiej jest zidentyfikować źródło wszelkich usterek, które mogą się pojawić. Modularyzacja umożliwia zmienianie pojedynczych składników bez wpływania na inne części systemu. Każdy test zapewnia swoje własne dane testowe oparte na wstępnie zdefiniowanych warunkach i może być wykonywany niezależnie od innych stanów bazy danych. Dane są importowane przy użyciu dedykowanych wewnętrznych interfejsów API. Ponieważ nie chcemy zatrzymywać regularnych testów badawczych, nie każda kompilacja daje w wyniku gotowe wdrożenie, ale średnio nowa wersja nadająca się do wdrożenia będzie gotowa przynajmniej co 24 godziny.

Aby zapewnić powodzenie przejścia do metodyki Scrum, musimy zwracać tyle samo uwagi na czynniki społeczne, co na techniczne. Najważniejszymi wnioskami dla nas były:

- Nigdy nie zapominać, że przyzwyczajenie jest drugą naturą człowieka.
- Zależności w architekturze systemu generują zależności pomiędzy poszczególnymi zespołami Scrum, które wymagają aktywnej koordynacji.

- Programowanie sterowane testami musi być implementowane bardzo ściśle.
- Odpowiednie środowisko budowania i równoległe testy są istotnymi czynnikami.
- Zautomatyzowane testy nie są uniwersalnym lekarstwem. Nie możemy zautomatyzować intuicji zaawansowanego testera i dodatkowe testy badawcze są niezastąpione.

Wnioski

Przejście na zwinne tworzenie oprogramowania wiązało się ze znaczącymi zmianami w metodach pracy wielu naszych pracowników. Sam zakres zadań, z którymi muszą sobie radzić nasi inżynierowie do spraw testów, jest nadal problemem, który będziemy rozwiązywać przez dalszą specjalizację. Niemniej jednak podejście zwinne jest na ogół odczuwane jako niezwykle wydajne. Poprawiona kooperacja pomiędzy członkami zespołów znacząco zwiększyła efektywność testowania. Innymi znaczącymi ulepszeniami są szybszy czas dotarcia na rynek i widoczne ograniczenie liczby błędów.

8.5 Scrum w środowisku technologii medycznych

Andrea Heck,
Siemens AG Healthcare, Erlangen

Andrea Heck ma dyplom uniwersytecki z informatyki i pracuje w branży tworzenia oprogramowania od ponad 20 lat. Jest menedżerem do spraw przechodzenia na metodykę zwinną w firmie Siemens Healthcare.
http://www.linkedin.com/in/andreaheck
Blog: *http://andreasagileblog.blogspot.com*

Siemens Healthcare (www.siemens.com/healthcare) jest światowym liderem w obrazowaniu medycznym i innych obszarach technologii medycznych. Oprogramowanie *syngo* jest kamieniem węgielnym wielu systemów tej firmy i stanowi podstawę aplikacji skanujących oraz narzędzi do przepływu zadań, które za rozsądną cenę umożliwiają szpitalom i placówkom medycznym tworzenie wysokiej jakości diagnoz i terapii. *syngo.via* łączy wszystkie aplikacje oparte na syngo w złożone rozwiązanie stanowiska roboczego do oceny obrazów medycznych. Aplikacje oparte na WWW również umożliwiają personelowi medycznemu przeglądanie obrazów zdalnie lub przy łóżku pacjenta. Jest to bardzo dynamiczny rynek, ale klienci oczekują wyników niezwykle

wysokiej jakości. Musimy się też stosować do ścisłych krajowych i międzynarodowych praw i wytycznych.

Kolejne strony opisują przejście na zwinne zarządzanie projektami w organizacji obejmującej siedem lokalizacji w pięciu krajach.

Powody wprowadzenia zwinnego tworzenia oprogramowania i technik zarządzania projektami

Cele, które chcieliśmy osiągnąć przechodząc na nową metodykę:

- Szybsze dostarczanie nowych produktów i funkcji klientom
- Zapewnianie bardziej wartościowych produktów o wyższej jakości, które są jak najlepiej dopasowane do potrzeb klienta
- Zmotywowane i wysoce efektywne zespoły programistyczne
- Ograniczone koszty

Przejście

Przejście odbyło się w trzech fazach, które duże organizacje często wykorzystują do implementacji dużych projektów:

1. **Nauka i pilotaż**
   Pierwszy zespół zwinny został utworzony w roku 2008 i przyjął oddolne podejście do tworzenia oprogramowania, ucząc się od innych firm i organizacji, studiując literaturę oraz odwiedzając konferencje i konsultantów. Uzyskano wsparcie zarządu dla tej zmiany i utworzono pilotażowe zespoły Scrum. Przeanalizowano informacje zwrotne i ogłoszono wyniki wszystkim pracownikom.
2. **Duże przekształcenie**
   Później nastąpiła długa, przygotowawcza faza budżetowania, zewnętrznych konsultacji i planowania. Rola naszych dostawców zmieniła się na bardziej partnerską. Przeszkolono i zaimplementowano zespoły Scrum, a właściciel produktu rozpoczął swoją pracę na wczesnym etapie procesu. Wiele procesów, struktur organizacyjnych i projektów zostało przekształconych jednocześnie w środku roku 2010, powstały też zespoły funkcyjne. Rozpoczęto duży projekt z 20 zespołami Scrum, a zespoły rozpoczęły proces stałego poprawiania przez samoocenę i szkolenia.
3. **Ciągłe poprawianie**
   Zaczęliśmy inwestować więcej w techniczną doskonałość, a wyniki retrospektyw zaczęły przynosić owoce w postaci poprawionych procesów i narzędzi. Sprawdzaliśmy swoje wyniki na tle innych podobnych organizacji i wprowadziliśmy elementy smukłej produkcji oprogramowania.

Ważne zasady:

- Każdy członek zespołu przyczynia się do dostarczania wartości ważnych dla klienta.
- Unikanie strat: Należy unikać przestojów i nadprodukcji oraz minimalizować prace w toku.
- Dostawcy zostają partnerami: Zarządzanie dostawcami jest objęte procesem przejścia. Formułowane są wspólne cele, a każda organizacja planuje swoją własną implementację. Następuje wzajemna pomoc, jeśli to konieczne.
- Zespoły są zachęcane do samodzielnego organizowania się, aktywnej współpracy oraz stałego kwestionowania status quo: Czy pracujemy tak wydajnie, jak to możliwe? Co możemy poprawić?
- Zespoły i właściciel produktu przechodzą szkolenie w zakresie Scrum. Scrum staje się platformą zwinną, w której implementowane są inne zwinne praktyki, takie jak programowanie oparte na zasadach XP.
- Fazy testów są przesuwane naprzód tak, aby miały miejsce podczas iteracji programowania i są uruchamiane wiele razy w formie testów regresji. Automatyzacja testów dla istniejącego kodu jest stale ulepszana.
- Ciągłe uczenie się: Grupy ćwiczeniowe pomagają członkom zespołu uczyć się poza granicami swoich własnych zespołów i poprawiają ich wydajność oraz wiedzę techniczną.

Istotne wyzwanie: zmiana z zespołów komponentowych na zespoły funkcyjne

We wcześniejszej, tradycyjnej konfiguracji projektu tworzenie oprogramowania było rozproszone po wielu lokalizacjach, a specyfikacje wymagań, programowanie i testowanie odbywało się wszędzie. Dodatkowo zespoły programistyczne były podzielone według warstw architektury aplikacji, natomiast testerzy byli podzieleni według poziomu testowania. Strategia zlecania zadań na zewnątrz przed zmianą metodyki wykorzystywała zasadę rozszerzonego pulpitu roboczego: Poszczególne składniki były dostarczane przez zewnętrznych dostawców, a każdy dostawca odpowiadał tylko za swoje składniki.

Teoretycznie specyfikacje reprezentowały interfejs. Ponieważ jednak zespoły tworzące składniki musiały dzielić poszczególne funkcje na wiele małych pakietów, stary system wymagał analityków przygotowujących każdy krok, zanim zespoły integracyjne składały je razem oraz wielu pośrednich menedżerów rozwiązujących wszelkie problemy z integracją. Nawet przy korzystaniu z praktyk zwinnych proces ten powoduje mini efekt kaskadowy [Larman/Vodde 09].

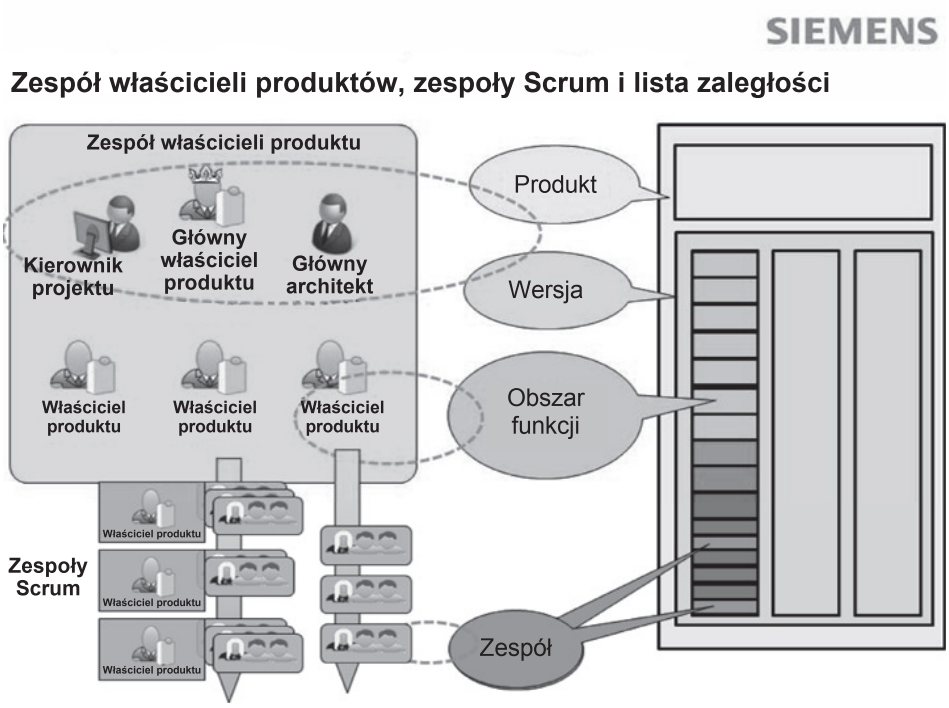
Przebudowaliśmy ten system, aby stworzyć organizację koncentrującą się wokół łańcucha wartości. Zaczyna się to od wymagań klienta a kończy (z punktu widzenia klienta) na instalacji u klienta ukończonego produktu zawierającego wymagane funkcje. Trudno jest optymalizować łańcuch wartości w organizacjach hierarchicznych, które mają podziały funkcjonalne, więc zorganizowaliśmy zespoły tak, aby każdy mógł tworzyć kompletny łańcuch wartości. To doprowadziło do utworzenia zespołów wielofunkcjonalnych. Każdy zespół ma swoich własnych specjalistów od definiowania wymagań, programistów, testerów i architektów. Najlepiej, aby programiści pochodzili z różnych zespołów składnikowych i zapewniali wiedzę obejmującą wiele poziomów architektury oprogramowania.

Niestety zwykle to nie wystarcza. Istotna wiedza jest rozproszona wewnątrz zespołów i dzielona pomiędzy nimi. Kilka miesięcy może zająć przekształcenie dawniejszych specjalistów w specjalistów z bardziej ogólną wiedzą.

Główną zaletą tej konfiguracji jest to, że organizacja jako całość staje się bardziej elastyczna, a pojedynczy programista nie będzie już stanowił wąskiego gardła dla wielu funkcji. Oznacza to też, że większość kodu programu może być teraz zmieniana i edytowana przez każdego, zamiast przynależeć do określonego zespołu. Aby pomóc ludziom odnaleźć się w nowej strukturze, wyznaczyliśmy Opiekuna Składników dla każdego zespołu. Jest to osoba zajmująca się kwestiami typu „Jaką funkcjonalność zapewnia składnik X i jak ma być implementowany?" albo „Które części składnika X są już pokryte zautomatyzowanymi testami i gdzie musimy poprawić pokrycie kodu?", albo „Czy proponowana zmiana projektu ma sens i czy łączy się dobrze z istniejącą funkcjonalnością?"

Hierarchiczny zespół właścicieli produktów

Aby przeprowadzić projekt zwinny na taką skalę, zespół właścicieli produktów musi być odpowiednio przeskalowany. Obecnie jest jeden główny właściciel produktu, wielu właścicieli produktów dla poszczególnych funkcji, a w razie konieczności podwłaściciele produktów, którzy opiekują się zespołami Scrum. Jest tylko jedna lista zaległości produktowych dla całego produktu, choć oferuje wiele widoków – na przykład tylko funkcje z najwyższej warstwy hierarchii albo poszczególne funkcje z danego obszaru. Mogą być one następnie dzielone dalej na funkcje, którymi ma się zająć określony zespół lub nawet na funkcje i zadania należące do określonego sprintu.



Rys. 8–1
Ułożona według priorytetów lista zaległości produktowych oferuje różne widoki poszczególnych funkcji. Na tej ilustracji wyższe priorytety są niżej na liście.

Spełnianie wymagań prawnych

Na początku projektu nasze środowisko procesowe spełniało na przykład wymogi norm FDA (Food and Drug Administration). Jednakże taka certyfikacja wiązała się ze znacznym dodatkowym wysiłkiem i zastanawialiśmy się, czy nie powinniśmy porzucić pewnych nieistotnych części procesu dla zaoszczędzenia nakładów, osiągając przy tym takie same wysokojakościowe wyniki. Musimy też stale wypatrywać regulacji, które możemy po prostu zignorować, ponieważ są już integralną częścią procesów Scrum i ciągłej integracji.

Zgodność z wymaganiami regulacyjnymi jest teraz w dużej mierze częścią kryteriów wykonania produktu, które są wykorzystywane przez właściciela produktu do akceptacji funkcji. Na przykład nasza lista kontrolna kryteriów wykonania stwierdza, że w przypadku ryzyka produktowego (na przykład zagrożenia dla zdrowia lub potencjalnej utraty danych) dana funkcja musi też mieć wbudowane odpowiednie środki zaradcze.

Czołowi konsultanci FDA obecnie przyznają, że praktyki zwinne mogą spełniać wymagania FDA lepiej niż kaskadowe techniki tworzenia oprogramowania [Olivier/Dere 11].

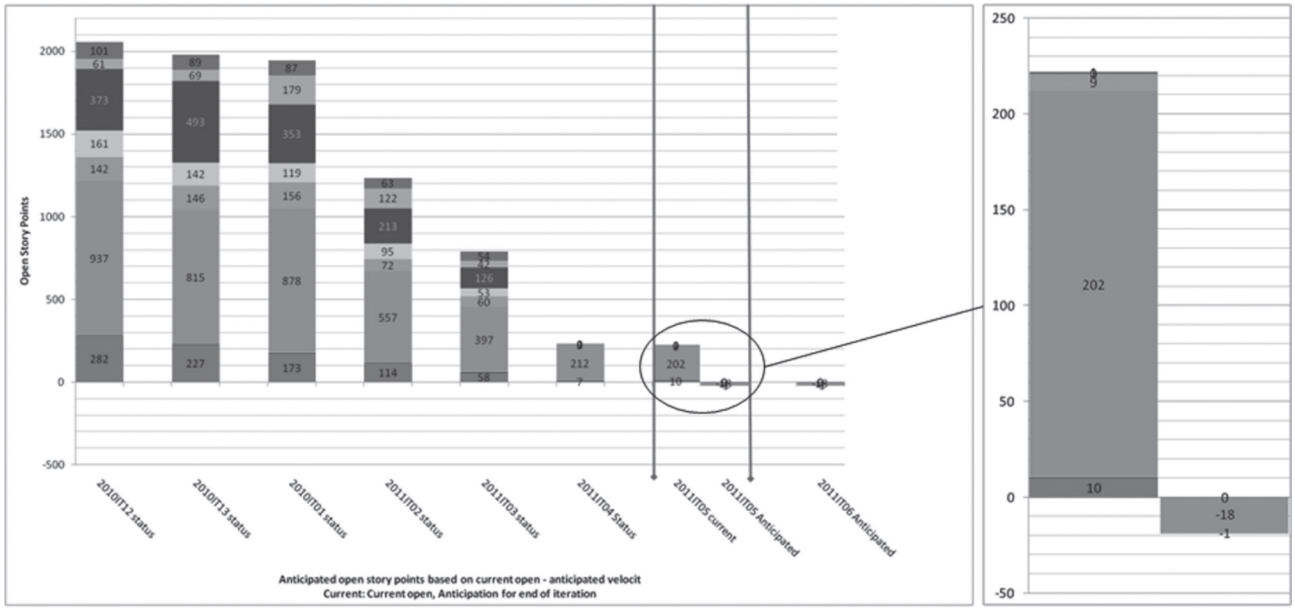
Gdzie jesteśmy dzisiaj?

Obecnie, rok później ukończyliśmy swój pierwszy projekt Scrum wykorzystujący 20 zespołów. Postępy w ramach tego projektu były przeźroczyste od wczesnego etapu i byliśmy w stanie implementować poprawki, które umożliwiły nam osiąganie planowanych kamieni

milowych. Większość zespołów ma mnóstwo doświadczenia w planowaniu i szacowaniu nakładów.

Nasze zespoły Scrum nie są jeszcze tak wydajne, jakbyśmy chcieli, ale tworzą oprogramowanie zgodnie z naszymi oczekiwaniami zwłaszcza w obliczu stromej krzywej uczenia się i olbrzymiej ilości transferu wiedzy. Nadal pracujemy z kilkoma starymi procesami i narzędziami, które wbudowaliśmy w nasze nowe iteracje. Wprowadzenie praktyk zwinnych sprawiło, że wykrywanie zmarnowanego potencjału jest bardziej systematyczne i stale monitorujemy procesy, które wydają się zbyt złożone.

Wyczerpywanie się listy zaległości w punktach scenariuszowych



Rys. 8–2
Iteracja przed zakończeniem implementacji. W tym momencie poziomy ryzyka są niskie.

Sukces zespołu właścicieli produktów

Zespół właścicieli produktów regularnie zaprasza klientów z całego świata do zgłaszania uwag odnośnie planowanych funkcji. Proces ten często ujawnia priorytety dla pewnych funkcji, które różnią się od naszych planów, co pozwala nam na bieżąco dostosowywać priorytety. W niektórych przypadkach funkcja i jej implementacja spotykają się z dużym entuzjazmem klientów, natomiast implementacja innych pomysłów nie przystaje dobrze do działań klienta. Takie informacje zwrotne są ważną częścią procesu planowania i pomagają nam szybko zmieniać elementy produktu, aby zachować konkurencyjność.

Zespół właścicieli produktów bardzo dobrze zna swoje systemy i funkcje i przeprowadza rozbudowaną demonstrację sprintu dla każdej zakończonej iteracji. Właściciele produktów pomagają też zespołom testować przepływy zadań i generować informacje zwrotne, zapewniając

nieformalne testowanie przez klientów podczas opracowywania danej funkcji (i po jej opracowaniu).

Jesteśmy bliżej celów zapewniania najpierw najważniejszych funkcji i orientowania naszej pracy na konkretne wymagania klientów, ale nadal musimy ograniczać długość całego cyklu od pomysłu do akceptacji.

Postęp w tworzeniu oprogramowania

Nasi programiści mają teraz dużo jaśniejszy pogląd na wymagania klienta. Omawiają scenariusze użytkowników z właścicielem produktu w celu wyłapania nowych funkcji, a większość też przechodziła dodatkowe szkolenia specjalistyczne.

Większość zespołów poprawiła swoje umiejętności ciągłej integracji i korzysta z możliwości gotowego zestawu zautomatyzowanych przypadków testowych, które pomagają im w rozpoznawaniu efektów ubocznych wprowadzanych zmian. Ponieważ zespoły nie pracują już nad konkretnymi składnikami, ale nad całą bazą kodu, programiści często dodają testy innych zespołów do swoich własnych pakietów testowych. Czują się też odpowiedzialni za dostosowywanie przypadków testowych, które już nie działają, ze względu na zmiany w kodzie produktu.

Niektóre testy są nadal automatyzowane na zbyt wysokim poziomie. Innymi słowy nadal testujemy na poziomie interfejsu użytkownika, choć jest to najbardziej czasochłonne podejście i wymaga wielu nakładów. Przeprowadzaliśmy już regularne przeglądy kodu i statyczną analizę kodu, żeby mieć pewność, że przestrzegamy określonych wytycznych dotyczących kodowania. Kod, który nie przejdzie tych dwóch testów, nie może być zatwierdzony. Oprócz testowania i przeglądów kodu dodaliśmy też programowanie w parach do swojego arsenału zapewniania jakości. Nasi programiści decydują w ramach zespołu, które metody będą najlepsze dla danego zadania. Zdyscyplinowane programowanie w parach może w znacznym stopniu zastąpić przeglądy kodu, a tylko kod związany z bezpieczeństwem musi być sprawdzany przy użyciu dodatkowych przeglądów.

Testowanie w zespołach Scrum

Wcześniej tylko testy jednostkowe były pisane i wykonywane przez programistów, natomiast testy integracyjne, systemowe i badawcze były wszystkie wykonywane przez specjalistyczne zespoły testujące.

Teraz zespoły Scrum przeprowadzają wszystkie testy jednostkowe i integracyjne dla pojedynczych składników i podsystemów. Na początku projektu każdy zespół Scrum miał co najmniej jednego testera, który był specjalnie wyszkolony w planowaniu i przeprowadzaniu

testów wymaganych przez organy regulacyjne. Odpowiednia wiedza też się rozprzestrzenia. Współpraca pomiędzy testerami a programistami jest na ogół bardzo dobra i wielu testerów cieszy się z okazji do programowania. Jednakże pomimo kreatywnej natury procesu testowania programiści nie są zbyt chętni do zdobywania nowych umiejętności związanych z testowaniem. Planowanie testów pokazało, że nasze aktualne narzędzia są zbyt silnie skupione na centralnej roli menedżera testów, co ogranicza ich przyjęcie przez zespoły. Obecnie pracujemy nad tym, jak je zastąpić bardziej przyjaznymi narzędziami.

Niektórzy programiści teraz piszą nowe przypadki testów integracyjnych i je automatyzują, a nawet zajmują się automatyzowaniem istniejących przypadków testowych. Średnio terminową zaletą nowego procesu jest to, że nowy kod jest lepiej przygotowany do automatyzacji testów.

Ponieważ wiele testów i poprawek jest przeprowadzanych podczas tworzenia oprogramowania, a programiści i testerzy rozumieją wymagania klientów lepiej niż wcześniej, nowe funkcje mają znacznie mniej usterek, które później wychodzą na jaw.

Wewnętrzne i zewnętrzne szkolenia w zespołach poprawiły jakość naszych przypadków testowych i skróciły czas testowania.

Istniejące przypadki testowe trzeba było na nowo rozprowadzić pomiędzy nowe zespoły funkcyjne. Nie każdy przypadek testowy jest przypisywany do zespołu, który nim zarządza (na przykład konwersja ze starego formatu ClearCase do nowego środowiska Team Foundation Server). Jeśli zespół dokona zmiany, która zepsuje przypadek testowy, test będzie musiał zostać naprawiony przez ten zespół, zanim nowy kod zostanie opracowany.

Ponieważ ręczne testy są teraz przeprowadzane przez zespoły, uruchamianie testów początkowo zajmowało więcej czasu. Czas testowania obecnie jest powoli skracany, w miarę jak coraz więcej testów jest automatyzowanych.

Nasze testy jednostkowe (dla języka C# w NUnit) są obecnie całkowicie zautomatyzowane. Nasze testy integracyjne nadal zawierają dużo starego kodu z niskimi współczynnikami pokrycia i wskaźnikami zautomatyzowania na poziomie 50-100% w zależności od konkretnych funkcji.

Testowanie systemowe

Mamy dedykowany zespół testowania systemowego, który testuje kod z punktu widzenia klienta – często w kontekście medycznym i na systemach testowych, które są wyposażone we wszystkie potrzebne interfejsy RIS i PACS. Testy obciążeniowe są również przeprowadzane przez ten zespół.

Zespół testowania systemowego opracowuje przypadki testowe dla nowych funkcji równolegle z ich opracowywaniem albo z góry przed następną iteracją. Przypadki testowe są automatyzowane, kiedy tylko ma to sens. Testy niefunkcjonalne mają wyższy wskaźnik automatyzacji niż testy funkcjonalne.

Zespół testowania systemowego dowiaduje się o planowanych funkcjach z listy zaległości, szczegółów specyfikacji i od właściciela produktu. Zespół testowania systemowego ogląda też demonstracje funkcji przedstawiane przez zespoły Scrum i wykorzystuje swoje odkrycia do tworzenia nowych testów koncentrujących się na kliencie.

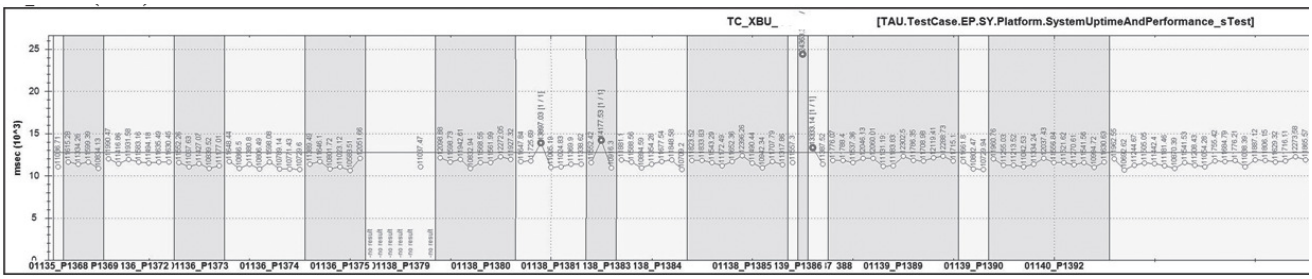
Zespół testowania systemowego przeprowadza też wszystkie oficjalne (tzn. udokumentowane i zatwierdzone) testy systemowe dla określonych kamieni milowych. Testy te odbywają się po wykonaniu testów zautomatyzowanych, badawczych i ręcznych oraz gdy system jako całość osiąga łączne pokrycie testami na poziomie 100%. Raporty elektroniczne są wystarczające dla wszystkich innych typów testów.

Testowanie integracyjne

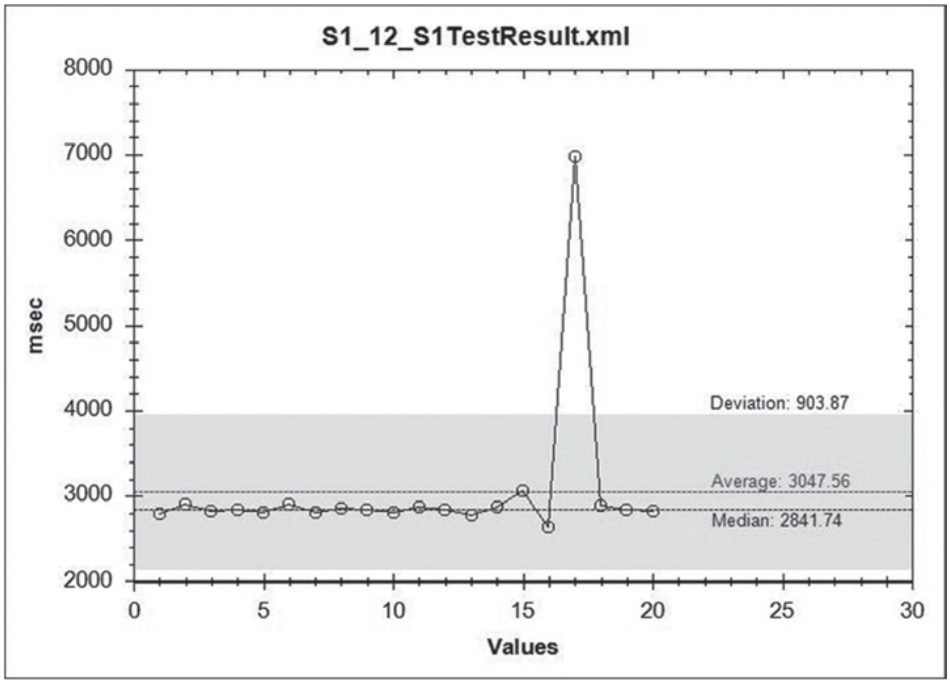
Aby zagwarantować wysoką jakość integracji, wybrane testy integracyjne są zawsze przeprowadzane, zanim zmiana w kodzie zostanie wprowadzona. Testy te są uruchamiane na wielu równoległych komputerach, a ich czas trwania jest ograniczony do dwóch godzin.

Zestaw testów wydajnościowych jest uruchamiany cztery razy dziennie dla wszystkich aktualnych zmian w kodzie. Testy wydajnościowe wykorzystują rzeczywiste przepływy zadań użytkowników i obecnie wykazują znacznie ograniczoną liczbę problemów wydajnościowych, niż miało to miejsce dawniej w czasie późniejszych etapów testowania.

Wszystkie zautomatyzowane testy pisane przez wszystkie zespoły są zbierane w jeden duży pakiet, który jest uruchamiany na wstępnie zainstalowanych systemach testowych (bez interwencji ręcznej). To podejście pomaga wykrywać usterki, które mogły zostać niezauważone przez zespoły programistyczne. Nieudane testy generują informacje zwrotne dla zespołu Scrum, który wprowadził daną zależność. Aby poprawić usterkę, dany zespół albo doda nieudany test do swojego własnego pakietu, albo napisze nowe testy niskiego poziomu.



Rys. 8–3
Trend tworzony przez testy wydajnościowe uruchamiane na przestrzeni wielu iteracji.



Rys. 8–4
Testy wydajnościowe są wykonywane kilka razy podczas każdego przebiegu testowego i czasami kończą się niepowodzeniem.

Co dalej?

Ciągłe poprawianie nadal stanowi główną część przyszłego planowania:

- Zespoły Scrum poprawiają swoje własne metody i środowisko pracy przy pomocy retrospektyw.
- Odbywa się to poprzez szkolenia, wykłady i transfer wiedzy.
- Programowanie sterowane testami i programowanie w parach są stosowane w niektórych zespołach, ale nadal nie w takim stopniu, jakbyśmy chcieli. Wiedza na temat programowania zwinnego jest stale poprawiana poprzez szkolenia techniczne, wykłady, konkursy i demonstracje wśród społeczności programistów.
- Wymiana informacji z innymi firmami odbywa się na konferencjach oraz poprzez współpracę i wzajemną analizę porównawczą.
- Inicjowane są projekty strategiczne w celu poprawienia procesów i narzędzi, które mają zbyt wiele interfejsów lub zbyt duże obciążenie. Na przykład ostatnio zastąpiliśmy narzędzie ClearCase własnym systemem budowania opartym na oprogramowaniu Team Foundation Server firmy Microsoft.
- Uczenie się szczupłych zasad i metod. Wprowadzone ma zostać stałe poprawianie przepływu zadań, jak w firmie Toyota (Kaizen, A3) i zamierzamy uczynić z systematycznego rozwiązywania problemów integralną część codziennego przepływu zadań w zespołach.
- Przekształcenie w uczącą się organizację.

8.6 Testowanie w procesie Scrum w firmie GE Oil & Gas

Terry Zuo, menedżer oprogramowania
w Software Development Center (SSDC),
Measurement & Control, GE Oil & Gas w Szanghaju, Chiny.

GE Oil & Gas jest światowym liderem w zaawansowanych technologiach i usługach, zatrudniającym 37000 pracowników w ponad 100 krajach i obsługującym klientów w wielu branżach – od wydobycia ropy i gazu po transport. GE Measurement & Control (M&C), jeden z kluczowych członków GE Oil & Gas, zajmuje się poprawianiem efektywności elektrowni, rafinerii i innych krytycznych systemów przemysłowych poprzez zaawansowane technologie testowania, sterowania i monitorowania oraz jest wiodącym innowatorem w zaawansowanych rozwiązaniach obejmujących pomiary oparte na czujnikach, testowanie niedestrukcyjne, monitorowanie oraz sterowanie przepływem i procesami (np. platformy biznesowe i narzędzia wspierające). GE M&C obsługuje 2 miliony użytkowników końcowych na całym świecie każdego miesiąca.

Powody przechodzenia na technologie zwinne

W GE wykorzystujemy model linii produkcyjnych, regionów i funkcji, co oznacza, że wszystkie organizacje wykorzystują raporty matrycowe. Model ten jest zaprojektowany dla biznesu operacyjnego, a w przypadku projektów inżynieryjnych wykazuje następujące wady:

- Złożona struktura podejmowania decyzji może sprawić, że rola właściciela produktu będzie dużym wyzwaniem
- Wpływy funkcji regionalnych, produktowych i globalnych

Aby pokonać te niedogodności i przyspieszyć dostarczanie gotowych produktów, GE Oil & Gas zaczął wprowadzać Scrum i praktyki zwinne w roku 2008. To przejście było stale rozwijane przez ostatnie pięć lat i obecnie (w 2013 roku) obejmuje ponad 150 zespołów w Stanach Zjednoczonych, Europie oraz rejonie Azji i Pacyfiku.

Przejście na Scrum w GE

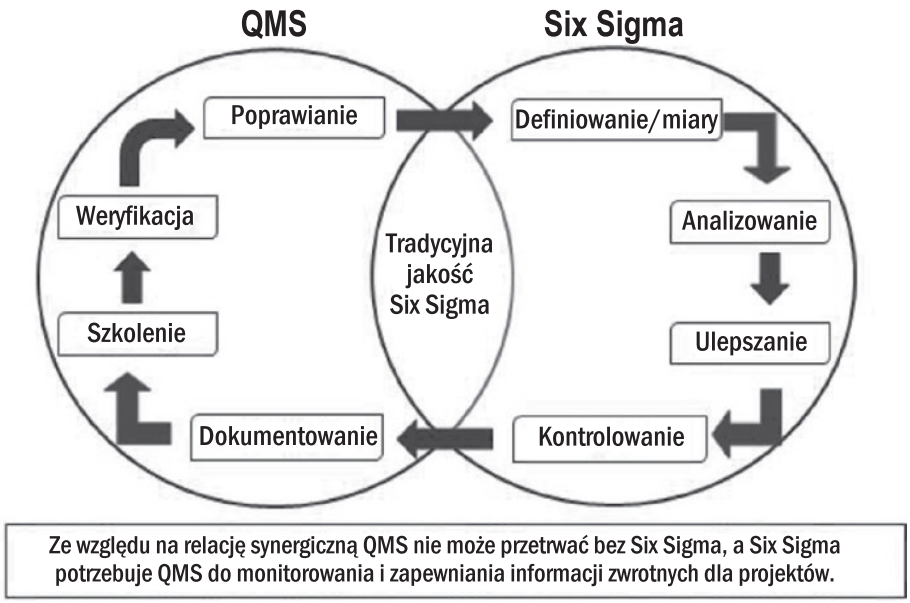
Jak więc osiągnęliśmy te cele szybkiego podejmowania decyzji i realizowania projektów, a jednocześnie przestawiliśmy nasz obecny system QMS tak, aby odpowiadał wysokiemu zapotrzebowaniu na zasady zwinne? Zespół inżynierski opracował i zaimplementował następujące zadania:

- Ożywienie kultury zwinnej w oparciu o Lean Six Sigma™ firmyGE
  - Rozruszanie zespołu „czarnych pasów" skupiającego się na kluczowych problemach.
  - Współpraca z różnymi zespołami poprzez projekty i inicjatywy tam, gdzie może to mieć największe znaczenie.
- Zwiększenie wiedzy na temat Lean Six Sigma i innych praktyk zwinnych
  - Oferowanie sieciowych kursów dotyczących Lean Six Sigma oraz praktyk zwinnych/Scrum w sieciowej platformie szkoleniowej firmy GE.
  - Prowadzenie szkoleń dla osób kierujących kluczowymi projektami.
  - Wyznaczanie osób do uczestnictwa w działaniach zespołowych i kluczowych projektach.
- Połączenie technik zwinnych, QMS i Lean Six Sigma
  - Wcześniej korzystaliśmy głównie z systemu QMS i Lean przy tworzeniu oprogramowania. Istnieją pewne różnice, ale praktyki zwinne mają wiele wspólnego z praktykami Lean. Zaczerpnęliśmy wiele dobrych elementów z dawnego podejścia i przenieśliśmy je do praktyk zwinnych.

Podejście szczupłe (lean)	Praktyka zwinna (agile)
Kaizen Ciągłe poprawianie	Sesje planowania iteracji Retrospektywy procesu i projektu
Kanban Wizualne systemy zarządzania	Listy zaległości produktowych Listy zaległości iteracji Wykresy wyczerpywania się list Suwaki projektu i jakości Deski rozdzielcze zautomatyzowanych testów
Ograniczenie instalacji Zdolność adaptacji do szybkich zmian	Zautomatyzowane kompilacje Ciągła integracja Programowanie sterowane testami Zautomatyzowane testowanie
Czas taktowania Dostawa oparta na popycie klienckim	Iteracyjne cykle tworzenia oprogramowania Inkrementacyjne tworzenie oprogramowania
Komórki robocze Rozmieszczane zasoby dla danego zadania	Zespoły wielofunkcjonalne Środowiska współpracujących zespołów Uogólnianie ról specjalistycznych Programowanie w parach

Rys. 8–5
Metody zwinne jako podejście Lean (szczupłe) (©2014 GE, wykorzystano za zgodą GE)

Rys. 8–5 (cz. 2) *Metody zwinne jako podejście Lean (szczupłe) (©2014 GE, wykorzystano za zgodą GE)*



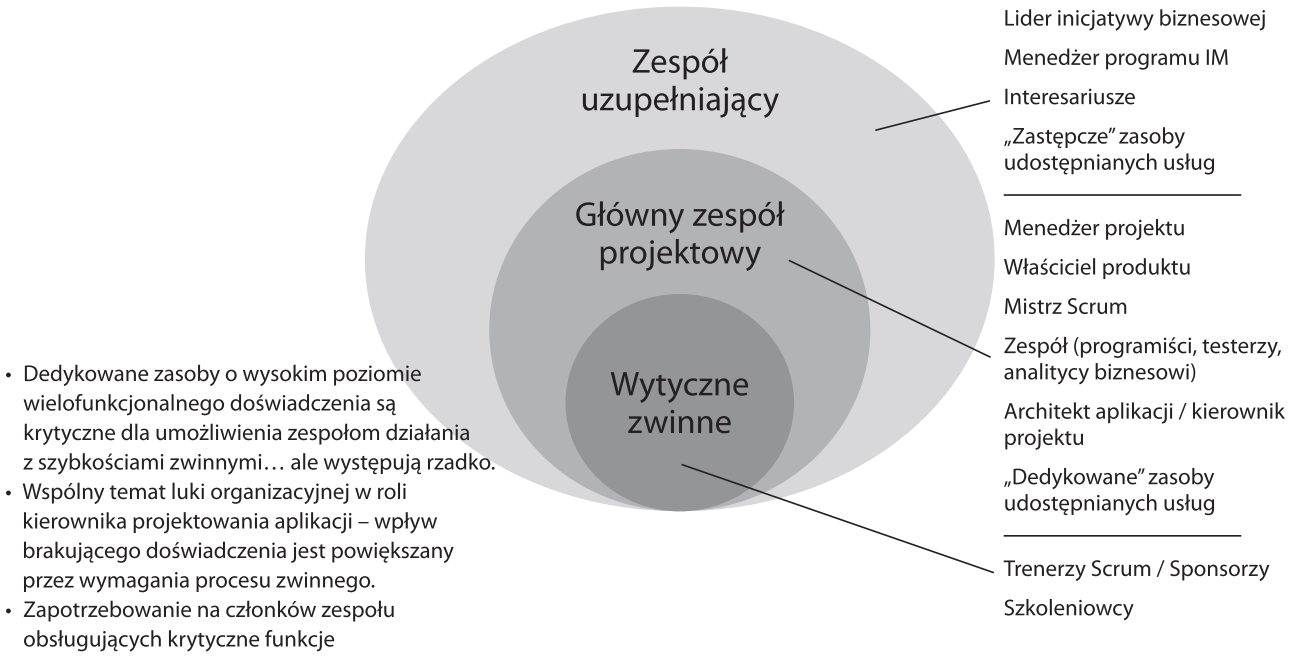
Przejście na metodykę Scrum w zespole Shanghai SSDC

Zespół Shanghai SSDC rozpoczął przejście na metodykę Scrum pod koniec roku 2008 podczas dwóch projektów pilotażowych. Jednym z nich była implementacja naszej platformy HMI. Podczas tego projektu pilotażowego uzyskaliśmy 20-procentowy wzrost wydajności zwłaszcza dzięki unikaniu implementacji funkcji o ograniczonej wartości, które zostałyby zaimplementowane, gdybyśmy korzystali z naszej klasycznej metody kaskadowej. Byliśmy też w stanie dostosowywać swój projekt do zmieniających się wymagań biznesowych i ewoluującego zrozumienia, które możliwości mają największą wartość. Przy tym wszystkim poprawiliśmy zadowolenie użytkowników i zadowolenie pracowników zaangażowanych w projekt.

Utwierdziło nas to w przekonaniu o dużym potencjale metodyki zwinnej i przeszliśmy do przekształcania do niej wszystkich projektów, gdzie to miało sens. Dział kadr i główny zarząd pomogły w przeglądzie dostępnych zasobów i przeprowadzeniu reorganizacji w strukturę zespołów projektowych typowych dla Scrum:

Podczas przejścia z metodyki kaskadowej do zwinnej (Scrum) przekształciliśmy rolę menedżera produktu w rolę właściciela produktu. Żeby szybciej iść dalej, rola menedżera projektu została zachowana i zaadaptowana do praktyk Scrum. Zespół zmienił swoje podejście do codziennej pracy i działań. Oto kluczowe pomysły:

- Obowiązki właściciela produktu są dzielone pomiędzy biznesowego właściciela produktu i menedżera projektu.
- Mistrz Scrum stanowi dominującą rolę w projektach pilotażowych.



Rys. 8–6
Struktura zespołów projektowych Scrum (© 2014 GE, wykorzystane za zgodą GE)

- Ważność wewnętrznych zasobów zapewniania jakości, zaawansowanych zasobów projektowych oraz zasobów analizy biznesowej.
- Elastyczny model wspólnych zasobów usługowych.

Wraz ze wzrostem firmy/zespołu zastąpiliśmy pojedynczy, duży zespół kilkoma minizespołami. Minizespół jest w pełni odpowiedzialny za projekt i implementację produktu i zwykle składa się z właściciela produktu, projektanta, jednego do trzech programistów, jednego lub dwóch testerów oraz autora dokumentacji. Minizespół podejmuje wszystkie decyzje techniczne, rozważa pomysły i zapewnia rozwiązania. Jest autonomiczny i skoncentrowany na produkcie, co pozwala nam działać wydajniej i reagować szybciej. Obecnie mamy sześć minizespołów w SDDC Shanghai. Przeglądając zmiany procesów w ostatnich pięciu latach poczyniono następujące obserwacje:

- **Punkty – brak oszacowań:** Ta zmiana stała się możliwa dzięki przyjęciu Kanban. W iteracyjnym tworzeniu programowania należy szacować nakłady pracy. Teraz mamy tylko poziome oszacowania, takie jak „dni", „tygodnie", „miesiąc", „miesiące". Czasami właściciel produktu myli się w szacunkach, ale przy nowych planach szacunki te stały się zaskakująco dokładne.
- **Rejestrowanie czasu – brak rejestrowania czasu:** Od pierwszego dnia rejestrowaliśmy poświęcany czas. Rejestrowanie czasu było głęboko w nas zakodowane i nawet pensje były oparte na liczbie przepracowanych godzin. Złamanie tej zasady było dość trudną decyzją, ale wszystko poszło zaskakująco gładko. Właściwie nie

	2009	2011	2012	2013
Zarządzanie źródłami i odgałęzieniami	VSS	TFS		TFS
TDD/BDD	TDD (programowanie sterowane testami) jest wielce zalecaną praktyką. Testy jednostkowe są konieczne dla każdego scenariusza użytkownika.	Nowy kod jest pokrywany przez testy jednostkowe. Sytuacja jest gorsza w przypadku C#, dopiero zaczęliśmy stosować TDD dla interfejsu użytkownika. Wczesne próby zastosowania BDD (programowania sterowanego zachowaniami).		Wyraźne skupienie się na BDD.
Zautomatyzowane testy funkcjonalne	Testy **Selenium/ TestComplete**	Zautomatyzowane testy funkcjonalne są oparte na TC/Selenium. Utworzyliśmy w C# platformę upraszczającą tworzenie testów.	**Testy funkcjonalne** działają równolegle. Pokrycie testami jest dużo lepsze.	
Ciągła integracja	Bardzo podstawowa.	Bardzo podstawowa.	Korzystamy z oprogramowania Cruise Control i mamy sporo różnych konfiguracji. Budowanie **wersji wstępnych**, **wersji docelowych**, itd. Testy funkcjonalne działają równolegle, a wykonanie ich wszystkich zajmuje 1 godzinę.	Celem jest kompletny proces ciągłej integracji.
Pokrycie testami	Brak pomiarów		Mamy 10% pokrycia testami jednostkowymi i 30% pokrycia testami funkcjonalnymi.	Mamy 20% pokrycia testami jednostkowymi i 50% pokrycia testami funkcjonalnymi.

Rys. 8–7
Przejście do metodyki zwinnej w SSDC

ma prawie żadnych powodów do rejestrowania czasu, jeśli nie szacujemy nakładów pracy. Niektórzy początkowo narzekali, ale po dwóch lub trzech miesiącach wszyscy byli zadowoleni z tej decyzji.

- **Planowanie wersji – Brak – Mapy drogowe:** Te zmiany mogą się wydawać nieco dziwne. Początkowo prowadziliśmy ścisłe planowanie wydań wersji produktu, ale po przyjęciu Kanban zrezygnowaliśmy z tej praktyki. Wydawało nam się, że nie był to bardzo dobry pomysł. Problem polegał na tym, że zgubiliśmy cel. Dobrze jest, gdy wydanie ma jasno wyznaczony cel; pomaga to zdefiniować, co należy zrobić, a czego nie. Bez wyznaczonego celu robimy wiele małych i dużych rzeczy, których nie łączy wspólny cel. To rozmywa sens naszych działań i wydajemy po prostu zbiór funkcji. Postanowiliśmy używać zamiast tego „map drogowych". Jednocześnie opracowywane mogą być trzy lub cztery główne scenariusze użytkownika. Każdy scenariusz ma jasno określony cel, a jego ukończenie zabiera od 3 do 12 miesięcy. Mapa drogowa

pokazuje po prostu wszystkie główne scenariusze w toku realizacji oraz niektóre przyszłe.

- **Scenariusze użytkownika się dzielą:** Co ciekawe, wciąż ciężko nad tym zapanować. W naszym przypadku scenariusze użytkownika są zawsze funkcjami wysokiego poziomu połączonymi ze standardami domenowymi i nie jest łatwo je podzielić. Jednakże ściśle stosujemy się do modelu INVEST, co bardzo pomaga nam w ulepszaniu scenariuszy użytkownika.
- **Codzienne spotkanie:** Ta praktyka przetrwała pięć lat bez modyfikacji. Staraliśmy się, aby spotkanie zawsze było krótkie i konkretne. Łatwo możemy przeprowadzić spotkanie z 15 osobami w 15 minut.
- **Spotkania:** Istnieje tendencja do organizowania mniejszej liczby ogólnych spotkań, a większej konkretnych. Jedynym okresowym spotkaniem jest obecnie codzienne rutynowe spotkanie, pozostałe organizowane są na życzenie. W Scrum mamy wiele formalnych spotkań, co może być dobre, jeśli właśnie zaczęliśmy przechodzić na praktyki zwinne. Z czasem ich liczba się zmniejsza. Spotkania na żądanie są świetnym pomysłem: mniej ludzi, lepsza koncentracja i lepsze wyniki.

Główne zmiany

W ostatnich latach wszystkie zespoły inżynierów przeszły z praktyk kaskadowych na Scrum, a następujące działania zostały podjęte w naszej jednostce, co naprawdę poprawiło codzienną wydajność pracy i ograniczyło koszty. Nie tylko zmienił się rytm działań, ale też sposób pracy wpłynął na ogólną wydajność projektu:

- Dopasowane zostały role różnych zawodów i wymagania odnośnie rekrutacji
- Przeprojektowana została ogólna struktura organizacyjna i obszar roboczy
- Nowoczesne praktyki i działania zespołowe poprawiają wydajność pracy:
  - Codzienne spotkania / Sprinty / Wypuszczane wersje produktu
  - 100 procent ciągłej integracji
- Entuzjastyczna zmiana kulturowa w 2012 roku

Wyniki i korzyści

- 96 spośród 267 wersji systemów było opracowywanych zgodnie z metodyką zwinną.

- 91 procent spośród zwinnych wersji zostało opracowanych na czas.
- Po wydaniu każdej wersji mierzymy ogólne zadowolenie klienta odnośnie wyników i działania produktu. Zadowolenie jest mierzone w skali od 1 do 10. Średni wynik dla tych wersji wynosił 8,8 – co oznacza wzrost o 0,4 w stosunku do projektów tworzonych kaskadowo.
- Inną ważną korzyścią, z której zdaliśmy sobie sprawę, był znacznie skrócony czas dostarczania produktów: średni czas od rozpoczęcia projektu do wydania pierwszej wersji albo czas pomiędzy wersjami produktu wynosił 4 miesiące. W przypadku naszego klasycznego procesu kaskadowego był to jeden rok!

W niektórych przypadkach przyspieszony czas cyklów wynikał jednak z typu projektu, ale samo przejście na metodykę zwinną przyczyniło się co najmniej do dwukrotnego przyspieszenia go. Nic lepiej nie zwiększa zaufania i zaangażowania klienta niż znacznie szybsze otrzymanie wartościowego rozwiązania. Nasi pracownicy też zyskują dodatkową motywację!

Uzyskane doświadczenie

- Przeszkody:
  - Brak chęci do zmian.
  - Jasność ról i obowiązków w modelu zwinnym/Scrum.
  - Skonfigurowanie ogólnej platformy do automatyzowania wszystkich testów jest długotrwałym procesem; wymaga ciągłych nakładów.
  - Ciągła integracja wymaga wysiłku.
- Ogólne czynniki kluczowe sukcesu:
  - Zdobywanie i przekazywanie wiedzy o metodykach zwinnych zwiększa odpowiedzialność.
  - Zapewnienie dostarczania produktów i usług zgodnie z wymaganiami klientów zwiększa reputację, wzrost, innowację i lojalność klientów.
  - Zapewnianie kluczowych miar przyszłych usprawnień procesu w celu eliminowania strat, wykonywania wartościowych działań, stałego rozwiązywania problemów (z jakością, wydajnością, itd.) i opracowywania rozwiązań poprawiających produkty i usługi.

- Wnioski związane z personelem:
  - Wyznaczyć dedykowanego, pełnoetatowego lidera transformacji!
  - Brak szkoleniowców/ekspertów – brak metodyki zwinnej. Trzeba zainwestować w szkolenia i zatrudnić wcześniej pełnoetatowych szkoleniowców.
  - Wymagać doświadczenia w metodykach zwinnych w procesie rekrutacji/wymaganiach dla stanowisk pracy.
  - Zatrudnić doświadczonego mistrza Scrum.
  - Zmodyfikować formalną metodologię zarządzania projektami.
  - Stosować umowy oparte na punktach. Wydajność personelu oparta na systemie punktowym i wpływająca na umowy o pracę i plan awansów.
  - Łączenie specjalistycznej wiedzy w centrach doskonałości

Podsumowanie i co dalej

Rozmiar zespołu zwinnego w SSDC wzrósł ponad dwukrotnie i nadal rośnie. Choć mamy ponad 150 zespołów zwinnych w M&C, to nadal przed nami długa droga. Niektóre zespoły są bardzo zwinne; inne stosują procesy mini-kaskadowe i nazywają je zwinnymi. Zmiany są trudne, a zmienienie firmy tak dużej jak GE czasami wygląda jak sterowanie wielkim statkiem po małym bajorku. Zrozumieliśmy, że cierpliwość jest ważna tak samo, jak pamiętanie, że nawet najmniejsze, stopniowe usprawnienia mogą mieć ogromne znaczenie, jeśli dokonujemy ich na dużą skalę. W miarę nabierania doświadczenia w korzystaniu z praktyk zwinnych będziemy coraz bardziej zwiększać prędkość wytwarzania oprogramowania:

- Większa liczba automatycznych testów jest krytyczna dla przyspieszenia cyklów budowania i testowania przy jednoczesnym poprawianiu jakości produktu.
- Programowanie sterowane testami:
  - Wymusza solidne zrozumienie celu poszczególnych składników
  - Zapewnia, że wszystkie moduły mają zautomatyzowane testy
  - Dopiero zaczynamy to wdrażać w działach informatycznych
- Ciągła integracja:
  - Zapewnia, że zawsze mamy zintegrowaną kompilację systemu gotową do testowania i wypuszczenia nowej wersji

- Wymusza przyrostowe rozwiązywanie problemów i konfliktów między niezależnymi modułami
- Wymaga automatycznych kompilacji ograniczających ogólny czas cyklu budowania/testowania

Zespoły i strategie zwinne stale ewoluują. Jedyną pewną stałą u nas jest zmiana.

Dodatki

A Słowniczek

Termin	Definicja	Źródło
analityczne zapewnianie jakości	Środki oparte na diagnozie – na przykład testowanie w celu mierzenia lub szacowania jakości produktu	[Spillner/Linz 14]
falsyfikat	Obiekt, który nie jest ani bezpośrednio sterowany, ani obserwowany przez test i który zastępuje funkcjonalność prawdziwego składnika zależnego alternatywną (uproszczoną) implementacją	[Meszaros 07]
imitacja	„Inteligentny" obiekt zastępczy, który zwraca różne wyniki do testowanego systemu w zależności od parametrów, z jakimi jest wywoływany. Działa jako punkt obserwacyjny dla pośrednich danych wyjściowych systemu testowanego.	[Meszaros 07]
integracja	Proces łączenia składników w większe podzespoły	[Spillner/Linz 14]
język specyficzny dla domeny	Komputerowy język programowania o ograniczonej składni, skoncentrowany na określonej domenie	[Fowler/Parsons 10]
kompilacja, budowanie	Skompilowana wersja programu albo proces jej tworzenia	*http://en.wikipedia.org/wiki/Build*
kryteria akceptacji	Kryteria wyjściowe, które dany składnik lub system musi spełniać, aby zostać przyjętym przez użytkownika, klienta lub inną uprawnioną jednostkę	[URL: ISTQB Glossary]
obiekt zastępczy	Obiekt, który zastępuje składnik zależny i ma identyczny interfejs tak, że test może sterować pośrednimi danymi wejściowymi dla testowanego systemu.	[Meszaros 07]
piramida testowania	Metafora dla zestawu planowanych lub istniejących przypadków testowych i ich rozdziału pomiędzy poziomy testowania jednostkowego, integracyjnego i systemowego	
programowanie zwinne	Grupa metod tworzenia oprogramowania oparta na programowaniu iteracyjnym i przyrostowym, gdzie wymagania i rozwiązania ewoluują poprzez współpracę pomiędzy samoorganizującymi się, wielofunkcjonalnymi zespołami. Promuje planowanie adaptacyjne, programowanie ewolucyjne oraz podejście iteracyjne oraz zachęca do szybkiego i elastycznego reagowania na zmiany. Jest to platforma koncepcyjna, która promuje ścisłe interakcje w trakcie całego cyklu tworzenia oprogramowania.	*http://pl.wikipedia.org/wiki/Programowanie_zwinne*

Termin	Definicja	Źródło
pseudo-obiekt	Obiekt zastępczy, który jest przekazywany do testowanego systemu jako argument (lub atrybut argumentu), ponieważ jest składniowo wymagany, ale nigdy nie jest faktycznie używany	[Meszaros 07]
retrospektywa sprintu	Retrospektywa sprintu jest dla zespołu Scrum okazją do przeprowadzenia wewnętrznego badania i utworzenia planu usprawnień do wprowadzenia podczas następnego sprintu. Retrospektywa sprintu występuje po przeglądzie sprintu i przed planowaniem następnego sprintu.	[URL: Scrum Guide]
Scrum	Platforma, w ramach której programiści mogą rozwiązywać złożone problemy, jednocześnie dostarczając w sposób wydajny i kreatywny produkty najwyższej jakości. Platforma Scrum składa się z zespołów Scrum i związanych z nimi ról, zdarzeń, artefaktów i reguł.	[URL: Scrum Guide]
składnik	1. Element oprogramowania o minimalnym stopniu złożoności, który można testować w izolacji 2. Element oprogramowania, który spełnia zalecenia implementacyjne określonego modelu składników oprogramowania (EJB, CORBA, .NET itd.)	[Spillner/Linz 2014]
składnik zależny	Pojedyncza klasa lub większy składnik, od którego zależy testowany system	[Meszaros 07]
sprawdzanie poprawności	Potwierdzenie przez badanie i obiektywne dowody, że wymagania dla określonego planowanego użycia aplikacji są spełnione	[ISO 9000]
sprint	Jednomiesięczny lub krótszy przedział czasowy, podczas którego tworzony jest gotowy, użyteczny i potencjalnie nadający się do wydania produkt. Sprinty składają się z planowania sprintu, codziennych spotkań Scrum, właściwej pracy programistycznej, przeglądu sprintu i retrospektywy sprintu.	[URL: Scrum Guide]
symulator	Urządzenie, program komputerowy lub system używany podczas testowania, który zachowuje się lub działa jak dany system, gdy się mu zapewni zestaw kontrolowanych danych wejściowych.	[URL: ISTQB Glossary]
szpieg	Obiekt zastępczy z dodatkową możliwością cichego rejestrowania wszystkich wywołań swoich metod. Zarejestrowane dane mogą być używane do sprawdzania pośrednich danych wyjściowych testowanego systemu.	[Meszaros 07]
test funkcjonalny	1. Weryfikacja wymagań funkcjonalnych 2. Testowanie oparte na analizie specyfikacji funkcjonalnej składnika lub systemu	[Spillner/Linz 14]
test integracyjny	Testowanie przeprowadzane w celu wykrycia usterek w interfejsie i interakcjach pomiędzy zintegrowanymi składnikami lub systemami	[Spillner/Linz 14]

Termin	Definicja	Źródło
test jednostkowy (test składnikowy)	Test pojedynczego (wyizolowanego) składnika oprogramowania	
test niefunkcjonalny	Testowanie atrybutów składnika lub systemu, które nie są związane z funkcjonalnością – np. wydajności, użyteczności, łatwości utrzymania i przenośności.	[Spillner/Linz 14]
test sterowany danymi	Technika skryptowa, która przechowuje dane wejściowe do testów i oczekiwane wyniki w tabeli lub arkuszu kalkulacyjnym tak, aby pojedynczy skrypt sterujący mógł przeprowadzić wszystkie testy z tabeli.	[URL: ISTQB Glossary]
test systemowy	Test zintegrowanego systemu sprawdzający, czy spełnia on określone wymagania	[Spillner/Linz 14]
test wydajności (test obciążeniowy)	Testy, które określają zmiany wydajności produktu programowego wraz z rosnącym obciążeniem	[Spillner/Linz 14]
testowanie akceptacyjne	Formalne testowanie w odniesieniu do potrzeb użytkownika, wymagań i procesów biznesowych przeprowadzane w celu ustalenia, czy system spełnia kryteria akceptacyjne i pomagające użytkownikowi, klientowi lub innej uprawnionej jednostce w ustaleniu, czy ma przyjąć system, czy nie.	[URL: ISTQB Glossary]
testowanie sterowane słowami kluczowymi	Technika skryptowa, która wykorzystuje pliki danych zawierające nie tylko dane testowe i oczekiwane wyniki, ale też słowa kluczowe związane z testowaną aplikacją. Słowa kluczowe są interpretowane przez specjalne skrypty wspierające, które są wywoływane przez skrypt sterujący testem.	[URL: ISTQB Glossary]
testowanie zwinne	Praktyka testowania dla projektu stosującego metodyki programowania zwinnego, obejmująca techniki i metody takie jak programowanie ekstremalne (XP), traktowanie programowania jako klienta testowania oraz podkreślająca paradygmat programowania sterowanego testami	[URL: ISTQB Glossary]
wdrażanie	Wszystkie działania, które sprawiają, że system programowy staje się dostępny do użytku, zwłaszcza jego instalacja i aktywacja w środowisku wykonawczym	*http://pl.wikipedia.org/wiki/Wdro%C5%BCenie_systemu*
weryfikacja	Potwierdzenie przez badanie i obiektywne dowody, że określone wymagania są spełnione	[ISO 9000]
zapewnianie jakości	Wszystkie środki zarządzania jakością, które skupiają się na generowaniu przekonania, że wymagania jakościowe produktu zostaną spełnione	[Spillner/Linz 14]

Termin	Definicja	Źródło
zapobiegawcze zapewnianie jakości	Użycie metod, narzędzi i procedur, które przyczyniają się do poprawy jakości projektowej produktu. W wyniku ich zastosowania można uniknąć błędów przy programowaniu lub ograniczyć prawdopodobieństwo ich wystąpienia, co daje nam produkt z wyłącznie pożądanymi cechami i niewieloma usterkami.	[Spillner/Linz 14]
zarządzanie jakością	Skoordynowane środki zarządzania organizacją w odniesieniu do jakości	[ISO 9000]
zgodność	Stan lub fakt bycia w zgodzie lub spełniania reguł albo standardów	*http://www.oxforddictionaries.com*

B Źródła

B.1 Literatura

[Aho et al. 06]
Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman
Compilers: Principles, Techniques, and Tools
Addison Wesley, 2nd edition, 2006

[Anderson 10]
David J. Anderson
Kanban: Successful Evolutionary Change for Your Technology Business
Blue Hole Press, 2010

[Bashir/Goel 99]
Imran Bashir, Amrit Goel
Testing Object-Oriented Software: Life Cycle Solutions
Springer, New York, 1999 (reprint 2012)

[Beck/Andres 04]
Kent Beck, Cynthia Andres
Extreme Programming Explained: Embrace Change (2nd edition)
Addison-Wesley Longman, Amsterdam, 2004

[Beedle et al. 99]
Mike Beedle, Martine Devas, Yonat Sharon, Ken Schwaber, Jeff Sutherland
Scrum: An Extension Pattern Language for Hyperproductive Software Development
Addison-Wesley Longman, Amsterdam, 1999

[Bergmann/Priebsch 11]
Sebastian Bergmann, Stefan Priebsch
Real-World Solutions for Developing High-Quality PHP Frameworks and Applications
Wrox, 2011