Inżynieria oprogramowania

Słynna anegdota opowiada o próbie zorientowania się, czym jest programowanie komputerowe, grupie dyrektorów w firmie zatrudniającej programistów. W ciągu tygodnia nauczono ich programować i dano im do rozwiązania mały problem. Każdemu kierownikowi przydzielono profesjonalnego programistę jako asystenta. Po pomyślnym rozwiązaniu przydzielonych im problemów z „niewielką" pomocą, kadra kierownicza wyszła z tego doświadczenia z poczuciem, że programowanie jest mimo wszystko dość łatwe i nie ma powodu, dla którego ich programiści nie mogliby ukończyć swoich zadań na tak jak oni sami. Oczywiście jest duża różnica między dużymi projektami programistycznymi, obejmującymi miliony lub nawet więcej linijek kodu, a małymi ćwiczeniami programistycznymi, składającymi się z kilkudziesięciu linijek. Tak duża różnica ilościowa powoduje jakościową różnicę w złożoności zadania i wymaga zupełnie innego rodzaju zarządzania. Jedna osoba może z łatwością śledzić wszystkie szczegóły drobnego problemu w swojej głowie. W miarę narastania problemu staje się to trudniejsze i pojawia się potrzeba pisemnych zapisów celów poszczególnych elementów składających się na projekt oraz relacji między nimi. Bez takiej dokumentacji łatwo jest poczynić założenia, w jaki sposób komponent ma być używany podczas jego programowania, ale z naruszeniem tych założeń podczas programowania innych komponentów, które go wykorzystują. Może się to już zdarzyć, gdy program składa się z kilkuset wierszy. Gdy projekty stają się większe, wyrastają poza możliwości jednego programisty. Duże projekty wymagają zespołu programistów współpracujących ze sobą, a nawet kilku zespołów, z których każdy zajmuje się inną częścią całego projektu. Bardzo duże projekty, takie jak nowoczesne systemy operacyjne, składają się z dziesiątek milionów linijek kodu i zatrudniają setki programistów, a nawet więcej. Części całego projektu mogą być opracowywane przez różne firmy. To sprawia, że ukryte założenia i inne rodzaje błędów są nieuniknione. Niestety, nawet zastosowanie najlepszych dostępnych obecnie narzędzi i metod nie gwarantuje, że programy będą wolne od błędów, o czym wie każdy, kto używał komputera od dłuższego czasu. W tej Części omówimy ogólne problemy, które pojawiają się podczas projektowania dużych systemów oprogramowania oraz główne procesy i metodologie stosowane w ich rozwiązywaniu.

Ukryte założenia w lotach kosmicznych

Istnieje niestety wiele przykładów ukrytych założeń w tworzeniu oprogramowania prowadzących do czasami katastrofalnych awarii. Poniższe przykłady to znane wydarzenia w historii lotów kosmicznych. NASA wystrzeliła Mars Climate Orbiter w 1998 roku. Jej misją było okrążenie Marsa i złożenie raportu o jego warunkach pogodowych w ramach przygotowań do lądowania na Marsie w 1999 roku. Orbiter dotarł do Marsa 23 września 1999 roku, ale potem zaginął. Komisja badawcza ustaliła, że trajektoria orbitera była o około 170 kilometrów za niska, ponieważ jedna część programu kontroli naziemnej orbitera wykorzystywała jednostki angielskie, podczas gdy inne części oczekiwały danych w jednostkach metrycznych. Stało się tak pomimo istnienia jasnej specyfikacji, która wskazywała właściwe jednostki do użycia, a także najnowocześniejszych metodologii rozwoju stosowanych przez NASA. Wśród czynników sprzyjających wymienionych w raporcie komisji dochodzeniowej była niewystarczająca komunikacja między zespołami ds. rozwoju i operacji. Mars Polar Lander, który miał wylądować na Marsie około sześć tygodni później, również zaginął. Komisja kontrolna określiła awarię oprogramowania jako najbardziej prawdopodobną przyczynę tej utraty. Silniki lądownika muszą zostać wyłączone, gdy tylko wyląduje, w przeciwnym razie przewróci się. Czujniki na nogach lądownika Polar Lander generują sygnał przy kontakcie z powierzchnią. Czujniki mogą czasami generować fałszywe sygnały, a oprogramowanie jest zaprogramowane tak, aby ignorować takie sygnały, porównując dwa kolejne sygnały i działając na nie tylko wtedy, gdy oba wykazują tę samą wartość. Jednakże, gdy nogi lądownika zostaną wysunięte z pozycji złożonej do pozycji do lądowania, czujniki mogą generować

dłuższe sygnały „kontaktowe”. To samo w sobie nie stanowi problemu, ponieważ nogi są rozmieszczone na wysokości około 1500 metrów, podczas gdy oprogramowanie nie wyłączy silników, dopóki radar nie zgłosi, że lądownik znajduje się mniej niż 40 metrów nad powierzchnią. Niestety fałszywy sygnał wykryty podczas uruchamiania nie jest kasowany i powoduje wyłączenie silnika, gdy tylko lądownik osiągnie wysokość 40 metrów. To wystarczy, aby rozbić się na powierzchni. Raport stwierdza: To zachowanie zostało zrozumiane i oprogramowanie lotu musiało zignorować te zdarzenia; jednak wymaganie nie opisywało tych zdarzeń konkretnie, a co za tym idzie, projektanci oprogramowania nie uwzględnili ich właściwie. Innym spektakularnym przykładem ukrytych przypuszczeń jest eksplozja, która miała miejsce podczas dziewiczego lotu rakiety Ariane 5 Europejskiej Agencji Kosmicznej w dniu 4 czerwca 1996 roku. Około 40 sekund po wystrzeleniu rakieta dokonała gwałtownej zmiany toru lotu, a w rezultacie rozpadł się i eksplodował. Komisja śledcza wyśledziła awarię z powodu błędu oprogramowania określonego typu. Zastosowany język programowania umożliwił programowi naprawienie się po takich błędach, a cztery możliwe przypadki z siedmiu były rzeczywiście odpowiednio chronione. Powód, dla którego pozostałe trzy przypadki nie były chronione, nie został udokumentowany w kodzie, ale później został zidentyfikowany w analizie, która wykazała, że tego rodzaju błąd nie może wystąpić w tych przypadkach. Jak się okazało, ta analiza rzeczywiście była poprawna dla wcześniejszych modeli Ariane, dla których napisano oprogramowanie. Kiedy to oprogramowanie zostało ponownie użyte w Ariane 5, która ma inne charakterystyki toru lotu, nie powiodło się. Nie wchodząc tutaj w więcej szczegółów, należy zauważyć, że chociaż była to główna usterka, w połączeniu z szeregiem innych aspektów systemu spowodowała katastrofalną awarię.

Problem z oprogramowaniem

Te przykłady wyraźnie pokazują potrzebę zdyscyplinowanej metody pisania programów, aby zapewnić ich niezawodność. Badanie takich metod nazywa się inżynierią oprogramowania. Obejmuje aspekty techniczne, takie jak języki programowania i narzędzia do zadań takich jak testowanie, debugowanie i weryfikacja, a także praktyki zarządzania. Inżynieria oprogramowania różni się od innych dyscyplin inżynierskich z powodu odmiennego charakteru przedmiotu: algorytmów i programów. Mają one charakter dyskretny; to znaczy zajmują się pojedynczymi i odrębnymi bytami, a mianowicie bitami. Natomiast inne dyscypliny inżynierskie zajmują się zjawiskami fizycznymi, które zwykle mają charakter ciągły. Ważną miarą złożoności systemu jest liczba jakościowo różnych stanów, jakie może on mieć. Na przykład, podczas gdy samochód może poruszać się z nieskończoną liczbą prędkości od zera do, powiedzmy, 150 kilometrów na godzinę, w celu sterowania samochodem za pomocą manualnej skrzyni biegów istnieje tylko siedem różnych stanów: od trzech do pięciu przełożenia do przodu, położenie neutralne i wsteczny. W każdym z tych stanów prędkość samochodu jest prostą funkcją prędkości obrotowej silnika. Chociaż zmienne układu ciągłego mogą przyjmować nieskończenie wiele wartości, zwykle mają stosunkowo niewielką liczbę jakościowo różnych stanów. Systemy dyskretne, a zwłaszcza komputery, mają ogromną liczbę stanów. Jeśli dodasz jeden bit do pamięci komputera, pomnożysz liczbę stanów przez dwa, ponieważ każdy stary stan dzieli się na dwa nowe: jeden, w którym wartość dodatkowego bitu wynosi zero, a drugi, w którym jest to jeden. Liczba możliwych stanów dla komputera z pamięcią zaledwie 280 bitów przekracza całkowitą szacowaną liczbę atomów we wszechświecie! Współczesne komputery mają pamięć zawierającą miliardy bitów, z niewyobrażalną (ale mimo to skończoną) liczbą możliwych stanów. Złożoność systemów komputerowych jest zatem znacznie wyższa niż systemów ciągłych, co czyni je mniej przewidywalnymi. W wyniku tej różnicy systemy ciągłe są bardziej podatne na analizę matematyczną i umożliwiają inżynierom poleganie na współczynnikach bezpieczeństwa. Na przykład podczas projektowania mostu dla określonego obciążenia, projekt zawsze zakłada większe obciążenie. Przy współczynniku bezpieczeństwa wynoszącym trzy most powinien teoretycznie wytrzymać trzykrotność wymaganego obciążenia. Będzie to wymagało mocniejszej konstrukcji, która może nawet zrekompensować niektóre wady

konstrukcyjne. Jednak błąd jednobitowy w programie komputerowym może spowodować, że całkowicie zejdzie z toru, powodując katastrofalną awarię. Ten efekt powiększenia systemów dyskretnych sprawia, że pojęcie współczynników bezpieczeństwa dla oprogramowania jest bezsensowne, a tym samym usuwa jedno z najpotężniejszych narzędzi inżynierskich ze sfery inżynierii oprogramowania. Metody analizy systemów dyskretnych również pozostają w tyle za metodami dla systemów ciągłych. Na przykład, istnieje obszerna wiedza związana z weryfikacją programów komputerowych, jak opisano w Części 5. Badania te doprowadziły nawet do kilku projektów weryfikacyjnych, które zakończyły się sukcesem komercyjnym. Wymagają one jednak dużych nakładów pracy ze strony wysoko wykwalifikowanych badaczy, a formalna weryfikacja większości pisanego oprogramowania jest obecnie niemożliwa. Takie wysiłki skupiają się zatem głównie na rdzeniach systemów krytycznych dla bezpieczeństwa. Co do reszty, musimy zadowolić się mniej skutecznymi, ale bardziej praktycznymi metodami. Oprogramowanie jest znacznie bardziej elastycznym medium niż fizyczne materiały używane w innych dyscyplinach inżynierskich. Po zbudowaniu mostu jego wymiana będzie miała bardzo istotny powód. Podobnie, kupując komputer, spodziewasz się, że będzie on używany przez kilka lat, zanim wymienisz go na nowy. Ponieważ jednak programy komputerowe są najwyraźniej tak łatwe do zmiany - wymaga to jedynie zmiany zawartości dysku twardego komputera - klienci często oczekują (i wiele razy dostaje) częste zamienniki dla ich oprogramowania (zwykle nazywane „aktualizacjami"). Drugą stroną tej monety jest to, że producenci oprogramowania nie martwią się o jakość swojego początkowego produktu w taki sam sposób, jak robią to producenci sprzętu. Zamiast tego polegają na aktualizacjach, aby zapewnić rozwiązania problemów wykrytych po wprowadzeniu produktu na rynek. To niewiele robi, aby stworzyć poczucie zaufania w branży oprogramowania. Co gorsza, założenie, że oprogramowanie jest łatwe do modyfikacji, jest błędne. Ogromna złożoność systemów oprogramowania z jednej strony, a rozmiar problemów, które takie systemy mają rozwiązać z drugiej, oznaczają, że programy są bardzo złożone, znacznie przekraczające nasze możliwości analityczne. Zazwyczaj więc próba naprawienia jednego błędu może spowodować powstanie kilku innych. Metodologie inżynierii oprogramowania, takie jak te omówione w tym rozdziale, mogą być wykorzystane do złagodzenia tego problemu. Jednak wiążą się one z własnym kosztem, który należy uwzględnić w całkowitym koszcie modyfikacji. Ostatnim znanym przykładem jest błąd Y2K. U schyłku dwudziestego wieku działało dużo oprogramowania, które zostało zbudowane przy założeniu, że lata można przedstawić za pomocą dwóch cyfr. Na przykład liczba „80" reprezentuje rok 1980; logicznie rzecz biorąc, liczba „01" oznaczałaby rok 1901. Chociaż w 1901 nie było żadnych komputerów elektronicznych, nadal konieczne byłoby odwołanie się do tej daty; na przykład może to być data urodzenia osoby, która w 1980 r. była uprawniona do świadczeń z zabezpieczenia społecznego. Jednak, gdy zbliżał się rok 2000, stało się oczywiste, że konieczne będzie reprezentowanie lat takich jak 2001, ale liczba „01" została już zajęta! (Pierwsze rzeczywiste przypadki tego problemu miały miejsce w 1998 roku, kiedy niektóre karty kredytowe, których data ważności wynosiła 2000, zostały odrzucone przez komputery myśląc, że był to 1900). W czasie pisania programów ten wybór reprezentacji był rozsądny. Niektóre z tych programów zostały napisane w latach 60. i nikt nie spodziewał się, że będą nadal działać ponad 30 lat później. W tamtych czasach rozmiary pamięci i dysków były znacznie mniejsze i droższe niż obecnie, a przechowywanie nadmiarowych cyfr „19" w każdym polu daty byłyby zbyt kosztowne. (Co zaskakujące, niektóre programy napisane na początku lat 90. nadal używały reprezentacji dwucyfrowej, chociaż tak naprawdę nie było na to usprawiedliwienia tak blisko 2000 roku.) Jedna z rzeczy, które wiemy o tym problemie, to to, że nie można go rozwiązać przez komputer w zasadzie . To nadal nie wyklucza częściowego rozwiązania, które działa w wielu, jeśli nie w większości praktycznych przypadków, ale nawet tak wiele nie jest dziś łatwo dostępnych. takich programów było bardzo dużo, ponieważ daty pojawiają się niemal wszędzie. Aby podać tylko jeden przykład, w programach kontrolnych niektórych wind pojawiają się daty, ponieważ muszą one śledzić harmonogram konserwacji. Niestety, nieubłagany bieg czasu zamienił te programy,

które były poprawne w momencie napisania, w błędne. Spowodowało to przepychankę w celu rozwiązania problemu w połowie i pod koniec lat 90., co okazało się ogromnym przedsięwzięciem. W programach rozpowszechniło się założenie, że lata są przedstawiane za pomocą dwóch cyfr. Konieczne było zbadanie każdego wiersza kodu, aby ustalić, czy w jakikolwiek sposób manipuluje datami. Jeśli tak, to musiało to zostać skorygowane w sposób zgodny z nową strategią dat. Ten projekt, ogólnie bardzo udany, pochłonął dużą inwestycję czasu i pieniędzy oraz opóźnił inne plany rozwoju oprogramowania. Jak widzieliśmy w Części 8, problem z weryfikacją oprogramowania jest nierozstrzygnięty i dlatego nie ma prawdziwej nadziei, że ktoś kiedyś napisze program, który będzie w stanie wykryć wszystkie wystąpienia podobnego błędu. Automatyczne poprawianie błędów jest również nieobliczalne. Chociaż możliwe jest opracowywanie narzędzi, które pomagają ludziom wykrywać i poprawiać błędy (co rzeczywiście zostało zrobione przez tak zwane fabryki Y2K pod koniec lat 90.), procesu nigdy nie da się w pełni zautomatyzować.

Modułowość i interfejsy

Kupując nowy samochód, otrzymujesz instrukcję obsługi, która wyjaśnia przyrządy i sterowanie, gdzie znajdują się bezpieczniki, jak zmienić opony i inne szczegóły techniczne. W części dotyczącej sterowania światłami znajdziesz wyjaśnienia dotyczące obsługi reflektorów, kierunkowskazów, świateł postojowych itd., ale nie instrukcje, które mówią, że należy użyć kierunkowskazu, zanim zechcesz skręcić w prawo, który bezpiecznik sprawdza, czy światła się nie włączają lub które kraje wymagają jazdy z włączonymi reflektorami w ciągu dnia. Są to ważne kwestie, które musisz znać, aby móc obsługiwać swój samochód; jednak jak wiele innych kwestii związanych w taki czy inny sposób z oświetleniem samochodu, pojawiają się one w innym miejscu w instrukcji obsługi lub w zupełnie innych dokumentach. Może fajnie jest mieć wszystkie istotne szczegóły w jednym miejscu, ale jest ich po prostu za dużo. Aby móc szybko i dokładnie uzyskać dostęp do tych informacji, należy je podzielić na osobne tematy, z których każdy pojawia się w innej sekcji jakiegoś dokumentu. A podział musi być na tyle logiczny, abyśmy mogli dowiedzieć się, gdzie znaleźć wszystko, czego potrzebujemy. Ta sama zasada dotyczy programów komputerowych. Często pomijanym aspektem programów jest to, że są one pisane dla ludzi bardziej niż dla komputerów. Na pierwszy rzut oka wydaje się to absurdalne: programy komputerowe są oczywiście napisane do wykonywania na komputerach, a kompilator nie dba o to, jak wyraźnie jest napisany program. Jednak programiści muszą czytać i rozumieć te programy podczas ich pisania i modyfikowania. Ponieważ modyfikacja oprogramowania jest tak powszechna, każdy udany program będzie musiał zostać zmodyfikowany w trakcie jego życia, a programiści odpowiedzialni za modyfikacje niekoniecznie będą tymi, którzy napisali program w pierwszej kolejności. Nawet oryginalny programista zazwyczaj nie pamięta szczegółów programu po kilku miesiącach. Dlatego organizacja programu według rozdziałów i wersetów jest tak samo ważna dla programów komputerowych, jak dla instrukcji obsługi samochodu, i z tych samych powodów: aby ludzie mogli znaleźć potrzebne informacje. Programy komputerowe należy podzielić na małe i spójne części, zwane modułami, z których każdy może być rozumiany oddzielnie. Jest to szczególnie ważne kiedy każdy moduł musi być rozwijany przez osobny zespół programistów. Jeśli jednak istnieje wiele powiązań między modułami, nie można ich zrozumieć w odosobnieniu, a zespoły je rozwijające nie mogą działać osobno, ponieważ każda decyzja jednego zespołu może mieć wpływ na wiele innych zespołów, a cały proces grzęźnie. Dlatego moduły muszą być względnie niezależne; ten cel nazywa się modułowością. Oczywiście moduły nie mogą być całkowicie niezależne, ponieważ niektórzy (klienci) muszą korzystać z usług dostarczanych przez inne moduły (dostawców). Jednak modułowość oznacza, że powinna istnieć minimalna ilość informacji, które muszą być dzielone między klientem a moduły dostawców. Informacje te nazywają się interfejsem modułu dostawcy i powinny zawierać wszystkie szczegóły, które klienci muszą wiedzieć o usługach oferowanych przez dostawcę, ale nic o tym, jak te usługi wykonuje. Na przykład interfejs modułu, który implementuje kolejki składa się z metod

używanych do wykonywania różnych operacji kolejki (dodawanie i usuwanie elementów, pobieranie elementu frontowego itp.), ale nie z implementacji tych metod. Interfejsy powinny być jasno udokumentowane, najlepiej w sposób formalny, który pozwala kompilatorowi sprawdzić, czy programiści nie tworzą niepożądanych zależności. Cel ten nazywany jest ukrywaniem informacji, co oznacza, że klient nie może polegać na implementacji modułu dostawcy, a jedynie na jego interfejsie. To umożliwia podejście do budowania dużych systemów w stylu Lego, w którym moduły można zastąpić innymi implementacjami tego samego interfejsu. Różne typy języków programowania mają różne sposoby dzielenia programu na moduły. Pojęcie modułu może być tak proste, jak zbiór funkcji zapisanych w jednym pliku. Może być bardziej wyrafinowany, a niektóre języki wyraźnie rozróżniają interfejs od implementacji modułu. Jednak język, który nie obsługuje modułowości w taki czy inny sposób, jest po prostu bezużyteczny w przypadku dużych projektów. Na przykład w językach obiektowych naturalną jednostką modułową są klasy. Klasy abstrakcyjne bez implementacji są czystymi interfejsami (i rzeczywiście tak się nazywają w JAVA). Przykładem jest klasa Queue z Części 3. Jak tam wspomniano, interfejs ten może być zaimplementowany przez takie klasy jak LinkedQueue, a programista modułu klienta nie musi być świadomy, która konkretna implementacja jest używana. Kompilator może łatwo sprawdzić, czy rzeczywiście używana jest odpowiednia implementacja. Interfejsy JAVA zawierają nazwy i typy parametrów wszystkich metod, które klienci mogą wywoływać na obiektach kolejki. Niestety to nie wszystkie informacje, które musi znać programista modułu klienta; brakuje znaczenia tych metod, które odróżnia np. stosy od kolejek. Świadczą o tym stwierdzenia dotyczące metodologii projektowania według umowy, opisanej w Części 5. Interfejs z umową zawiera pełne informacje potrzebne klientom. Istnieje wiele metodologii, które próbują poprowadzić proces rozbicia dużego problemu na stosunkowo niezależne moduły. Nie będziemy tutaj wchodzić w szczegóły tych metodologii. Dość powiedzieć, że to zadanie jest nadal bardziej sztuką niż nauką i wymaga dużego doświadczenia.

Modele cyklu życia

Aby opracować metody zarządzania inżynierią oprogramowania, najpierw konieczne jest zrozumienie procesu tworzenia oprogramowania. Proces ten okazuje się dość złożony i trudny do sformalizowania. Istnieją różne jego modele, zwane modelami cyklu życia; każdy model ma inny pogląd na proces iw rezultacie obsługuje różne metodologie. Wszystkie modele cyklu życia oparte są na następujących rodzajach czynności, które zilustrujemy na przykładzie edytora tekstu.

Pozyskiwanie wymagań. (Nazywana również analizą wymagań.) Jest to proces odkrywania tego, czego klient naprawdę potrzebuje. Klienci mogą posiadać wiedzę na temat własnej domeny biznesowej, ale często nie rozumieją implikacji algorytmicznych i systemowych wprowadzenia systemu komputerowego do swojej firmy. Może to prowadzić do nieporozumień, które muszą być dokładnie wyjaśnione, aby doprowadzić do pomyślnego wyniku. Efektem tej czynności jest zwykle dokument wymagań. Czasami nie ma klienta, z którym można by pracować, jak w przypadku, gdy firma produkuje oprogramowanie, które ma być sprzedawane z półki. W tym przypadku producent musi przewidywać potrzeby klienta, co sprawia, że pozyskiwanie wymagań jest trudniejsze, a nie łatwiejsze! (I rzeczywiście, wiele produktów kończy się niską sprzedażą, ponieważ ten proces nie przewidział tego, czego klienci naprawdę potrzebują). W naszym przykładzie dotyczącym przetwarzania tekstu dokument wymagań będzie zawierał listę konkretnych potrzeb klienta, takich jak wykorzystanie złożonej notacji matematycznej, sprawdzanie pisowni i automatyczne generowanie etykiet adresowych.

Projekt. W tym procesie problem jest dzielony na moduły, a odpowiednie algorytmy i struktury danych są wybierane do reprezentowania domeny problemu i wykonywania wymaganych operacji. Czynność ta wymaga gruntownego zrozumienia algorytmiki w celu dokonania wyborów, które zapewnią

poprawność i wydajność powstałego produktu. Projekt przykładowego edytora tekstu może określać moduły do takich zadań, jak graficzny interfejs użytkownika, obsługa tekstu, skład matematyczny i skład liter (w tym generowanie etykiet adresowych). Projekt modułu obsługi tekstu określałby strukturę danych do przechowywania zawartości tekstu; może to być na przykład lista znaków ze specjalnymi symbolami oznaczającymi łamanie linii lub lista linii, z których każdy jest wektorem zawierającym znaki pojawiające się w linii. Moduł ten określi również algorytmy służące do wykonywania takich czynności jak wyrównywanie wierszy i sprawdzanie pisowni. Realizacja. Jest to proces tłumaczenia projektu na określony język programowania. Po pierwsze, każdy moduł jest implementowany osobno. Następnie moduły są składane razem, tworząc kompletny system; proces ten nazywa się integracją i często jest to etap, na którym ujawniają się ukryte założenia. Nasz edytor tekstu jest teraz wrzucony do określonego języka programowania i można z nim poeksperymentować (lub z jego częściami) i zobaczyć, co naprawdę robi.

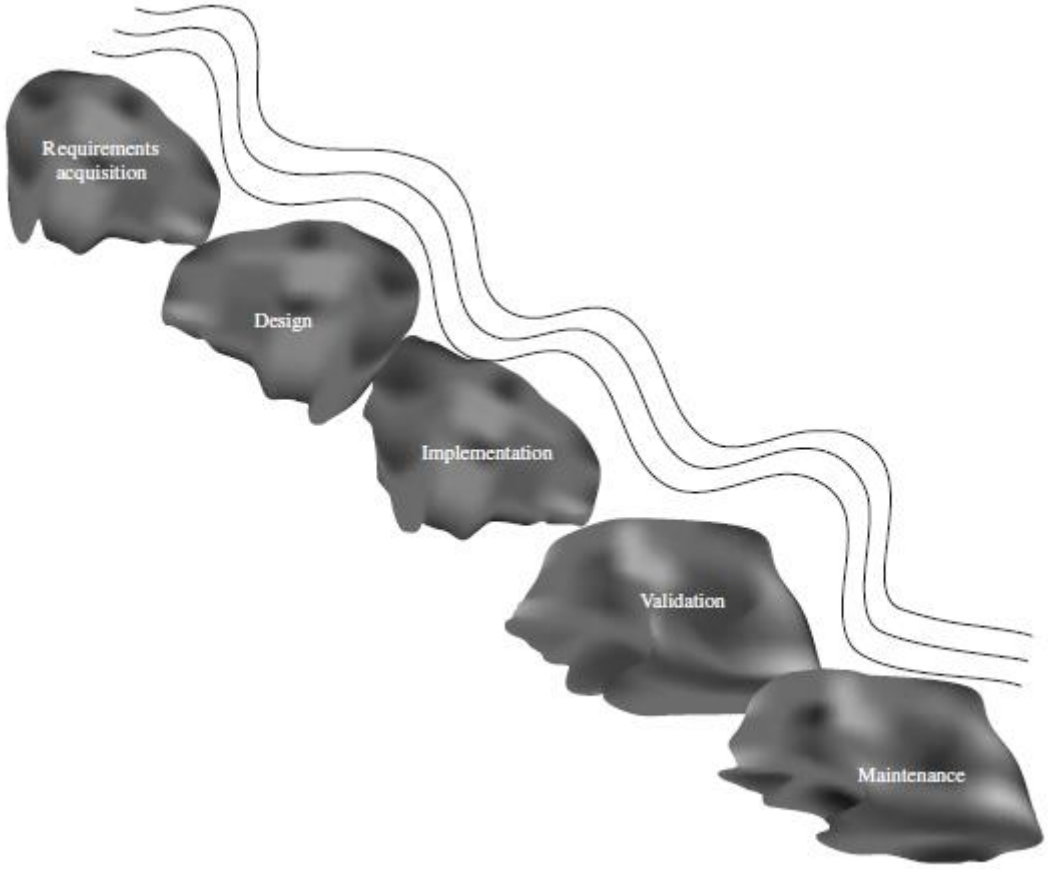
Walidacja. Jest to proces upewniania się, że każdy moduł, a także cały system, spełniają swoje specyfikacje. Walidacja może przybierać różne formy. Zwykle obejmuje obszerne testowanie poszczególnych modułów (tzw. testowanie jednostkowe) oraz całego systemu (tzw. testowanie integracyjne). Niestety, jak wspomniano w rozdziale 5, testowanie nie wystarcza do znalezienia wszystkich błędów. Formalna weryfikacja może zapewnić, że system spełnia swoją specyfikację, ale, jak wspomniano wcześniej, nie jest jeszcze praktyczna do ogólnego użytku. Również fakt, że system spełnia specyfikację, nie musi oznaczać, że robi to, czego chciał klient! Weryfikacja wymaga formalnej specyfikacji w celu sprawdzenia programu, ale przełożenie wymagań na formalną specyfikację jest trudnym problemem i tak samo podatnym na błędy jak programowanie. Nie ma niezawodnego sposobu na sprawdzenie specyfikacji pod kątem wymagań klienta, ponieważ są one w mózgu klienta i są niedostępne dla naszych formalnych narzędzi. W dobrze zaprojektowanym procesie każdy moduł aplikacji do edycji tekstu może być testowany osobno. Wymaga to napisania specjalnych programów testowych, ponieważ moduły nie są zaprojektowane do samodzielnej pracy. Jednak ta „dodatkowa" praca jest warta wysiłku, ponieważ znacznie łatwiej jest znaleźć i naprawić problemy w jednym module niż w całym systemie.

Utrzymanie. To słowo jest tutaj użyte w szczególnym technicznym sensie. W przeciwieństwie do systemów mechanicznych, programy komputerowe nie podlegają procesom środowiskowym, które mogą spowodować ich przegrzanie, zużycie lub w inny sposób utratę swoich właściwości operacyjnych. Nie trzeba „przerabiać" programu po 200 000 mil lub 10 latach (w zależności od tego, co nastąpi wcześniej) - będzie on nadal działał w nieskończoność dokładnie tak samo, jak pierwszego dnia. Na tym właśnie polega problem! Aby programy pozostały użyteczne, muszą się zmieniać. Zmiany mogą być wymagane z powodu zmian w środowisku operacyjnym programu; na przykład, gdy klient zmienia komputer typu mainframe na komputer osobisty lub podczas uaktualniania systemu operacyjnego. Środowisko operacyjne obejmuje znacznie więcej niż sam komputer: na przykład program może wymagać modyfikacji, gdy zmienią się przepisy stanowe regulujące zachowanie domeny aplikacji, co może się zdarzyć w przypadku oprogramowania kasjera, gdy zostanie wprowadzony nowy rodzaj podatku. Zmiany mogą być również wynikiem nowych próśb klientów; gdy klienci zaczynają korzystać z nowego systemu, odkrywają więcej rzeczy, które chcieliby, aby program dla nich zrobił. Wreszcie, niestety, mogą być wymagane modyfikacje, aby naprawić błędy, których dostawca nie wykrył przed udostępnieniem programu klientom. Nowe żądania klientów pojawiające się po wdrożeniu systemu są prawie zawsze nieuniknione, ponieważ wprowadzenie systemu skomputeryzowanego do środowiska, w którym wcześniej go nie było, zmienia to środowisko. To, co początkowo mogło być programem oszczędzającym pracę, wykonującym zadanie, które w innym przypadku można by wykonać ręcznie, teraz staje się istotną częścią biznesu. Na przykład księgowy może zacząć korzystać z komputera dla wygody. Odkrycie, że przyspiesza to jego pracę, może skutkować podjęciem decyzji o pozyskaniu

nowych klientów. Teraz jednak księgowy jest zależny od komputera i nie może się bez niego obejść. Na przykład, jeśli nietypowe zadania, których oprogramowanie nie było w stanie obsłużyć, były wcześniej wykonywane ręcznie, w przypadku większej liczby klientów również muszą one zostać zautomatyzowane. Statystyki pokazują, że największa inwestycja w oprogramowanie jest na etapie utrzymania; może to osiągnąć 80% całości! Dlatego dobrym pomysłem jest włożenie dużego wysiłku w inne czynności wymienione powyżej, aby ułatwić przyszłą konserwację. Większość metodologii inżynierii oprogramowania opiera się na tej przesłance. Jeśli poczta zmieni się z pięciocyfrowych kodów pocztowych na dziewięciocyfrowe, pospieszymy do dostawcy naszej aplikacji do przetwarzania tekstu, aby zmodyfikować generator etykiet adresowych tak, aby obsługiwał dziewięciocyfrowe kody. Jeśli pierwotny projekt traktował pięć jako „magiczną liczbę" bez zastanowienia się nad możliwością, że może się ona zmienić, jest całkiem prawdopodobne, że to założenie jest osadzone w wielu częściach programu. Modyfikacja będzie wtedy bardzo trudna, a dostawca pobierze za to wysoką opłatę. W takim przypadku możemy lepiej wydać nasze pieniądze, przechodząc do innego dostawcy. W rzeczywistości zmiana na dziewięciocyfrowe kody pocztowe to prawdziwy przykład, który spowodował duże koszty dla tych dostawców, którzy nie byli na to przygotowani. Jest to jednak przyćmione przez omówiony już przykład problemu Y2K.

Model wodospadu

Najbardziej podstawowy model cyklu życia nazywa się modelem kaskadowym. Zakłada, że powyższe czynności następują po sobie w ścisłej kolejności. Nazwa tego modelu wywodzi się z widoku następujących po sobie czynności, takich jak woda spadająca z klifu w szeregu kroków.



Z tego punktu widzenia dokument wymagań musi być w pełni przygotowany, zanim będzie możliwe rozpoczęcie projektowania. Ponadto po rozpoczęciu projektowania dokument wymagań jest

zamrożony i nie można go modyfikować. Powodem tego jest to, że wszelkie zmiany wymagań poczynione po zakończeniu projektowania lub, co gorsza, po wdrożeniu, wymagają cofnięcia się do początku i odpowiedniej modyfikacji projektu (i wdrożenia). Może to być niezwykle kosztowne. Rzeczywiście, modyfikacje wczesnych etapów procesu wymagają zmian w kolejnych etapach. Późniejsze etapy dodają dużo szczegółów, przez co ich zmiana jest bardziej skomplikowana. W rezultacie im dalej w procesie dokonuje się zmiana, tym jest ona bardziej kosztowna. Oszacowano, że koszt naprawy błędu w fazie wymagań wzrasta dziesięciokrotnie, jeśli jego wykrycie jest opóźnione do fazy implementacji, oraz o czynnik 100, jeśli zostanie wykryty, gdy system już działa. Niestety, na wcześniejszych etapach, w których dokonujemy najważniejszych i najbardziej wpływowych wyborów, problem rozumiemy dość niejasno. W miarę postępów w projektowaniu i wdrażaniu nasze zrozumienie problemu rośnie, aż do momentu, w którym naprawdę go rozumiemy, nie mamy żadnych znaczących wyborów do dokonania! W praktyce na wczesnych etapach nieuchronnie podejmowane będą błędne decyzje, które będą musiały zostać poprawione później. Oznacza to, że przed rozpoczęciem wdrożenia należy włożyć wiele wysiłku w walidację wymagań i projektu. Jest to dość trudne i nie ma zbyt wielu satysfakcjonujących formalnych narzędzi pomocnych w tych zadaniach. Często mamy do czynienia z mniej precyzyjnymi, ale wciąż użytecznymi metodami, takimi jak przeglądy projektów, gdzie wymagania i projekt są przedstawiane najbardziej doświadczonym osobom, które starają się wykryć ewentualne wady. Ponieważ koszt wprowadzenia zmian w wymaganiach i projekcie jest tak wysoki, należy je z wyprzedzeniem przewidzieć. Przy pewnym doświadczeniu można to zrobić całkiem dobrze, ale wyraźnie trzeba będzie wprowadzić zmiany. W związku z tym model kaskadowy jest zwykle pokazywany z „przepływem wstecznym", który dopuszcza tę rzeczywistość, ale metodologie oparte na modelu kaskadowym oferują niewielkie wsparcie, gdy tak się dzieje. Zaletą modelu kaskadowego jest to, że sprawia, że proces jest widoczny, a tym samym łatwiejszy w zarządzaniu. Proces jest zorientowany na dokumenty, a każdy etap modelu ma jasną definicję, jakie produkty (często nazywane produktami dostarczanymi) mają być dostępne po jego zakończeniu. Produkty te, którymi mogą być dokumenty lub programy, są oczywistymi punktami kontroli całego procesu. Kierownictwo (a także klient) może wykorzystać rezultaty do śledzenia postępów projektu. Ta mocna strona modelu jest również słabością, ponieważ wysiłek poświęcony na udokumentowanie procesu jest czasem lepiej poświęcony na zapewnienie, że zaowocuje satysfakcjonującym produktem. Ponadto, gdy zmieniają się wymagania, praktycznie niemożliwe jest rozpoczęcie procesu od zera. W rezultacie programiści „fałszują", próbując zmodyfikować istniejące dokumenty, aby wyglądały tak, jakby nowe wymagania były tam od samego początku. Chociaż jest to przydatne (i zostało nawet usankcjonowane przez niektórych wpływowych metodologów), z konieczności jest podatne na błędy. W miarę wprowadzania kolejnych zmian struktura staje się pełna luk, których łatki nie do końca wypełniają. W końcu każdy dodatek stwarza więcej problemów niż naprawia, w którym to momencie cały proces wymyka się spod kontroli.

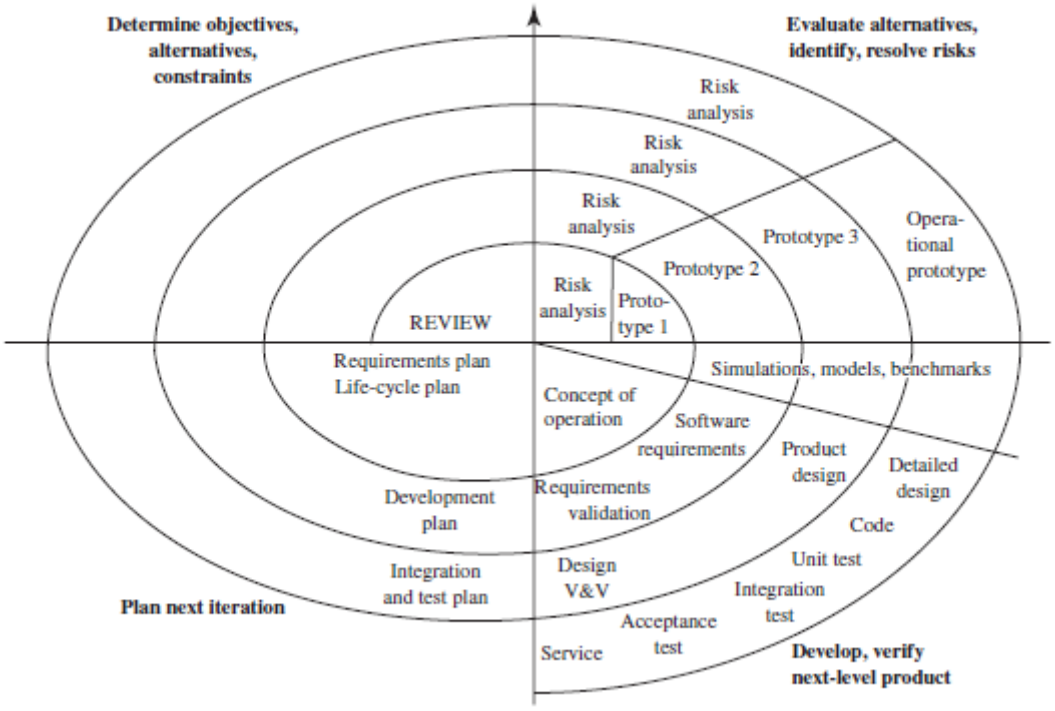
Rozwój ewolucyjny

Model kaskadowy z pewnością ma właściwą kolejność czynności; program napisany bez dobrego projektu będzie wadliwy, a projekt wykonany bez dobrego zrozumienia wymagań nie doprowadzi do rozwiązania problemu klienta. Jednak ocena produktu końcowego jest możliwa dopiero pod koniec tego długiego procesu, a jak już powiedzieliśmy, naprawa wczesnych błędów jest bardzo kosztowna. Ewolucyjny model rozwoju stara się zmniejszyć to ryzyko, zalecając jak najszybsze wyprodukowanie początkowej wersji roboczej. Klient może przekazać przydatne informacje zwrotne na temat przydatności produktu, faktycznie z niego korzystając. Jest to znacznie bardziej prawdopodobne, aby odkryć ukryte założenia i inne problemy niż jakakolwiek ilość czytania dokumentów. Informacje zwrotne z tego są następnie wykorzystywane do tworzenia drugiej wersji, która prowadzi nawet do trzeciej. Każda wersja jest tworzona zgodnie z czynnościami określonymi przez model kaskadowy, ale

szybciej i przy znacznie mniejszej dokumentacji. W konsekwencji powstałe systemy mogą być mniej dobrze skonstruowane, a zatem bardziej kosztowne w utrzymaniu. Model najlepiej zastosować, gdy ryzyko wczesnych błędów jest szczególnie wysokie, na przykład przy tworzeniu nowego typu aplikacji, z którą programista (a być może nawet klient) nie ma doświadczenia. Aby jak najszybciej dostać się do działającej wersji, jest on zwykle produkowany jako system barebone, koncentrując się na najważniejszych funkcjach i pomijając resztę. Taka wersja nazywana jest prototypem. Kolejne wersje rozwijają pierwotny prototyp, dodając więcej funkcji, aż do uzyskania w pełni funkcjonalnego systemu końcowego. Proces ten nazywa się programowaniem eksploracyjnym. Bardziej radykalnym podejściem jest wyrzucenie pierwotnego prototypu i rozpoczęcie od zera po raz drugi. Takie podejście, zwane szybkim prototypowaniem, ma tę zaletę, że początkowy prototyp można stworzyć bardzo szybko, ponieważ jego wewnętrzna jakość nie jest istotna. Wyciągając wnioski z początkowego doświadczenia, programiści mogą teraz zbudować drugi prototyp wyższej jakości, nie martwiąc się o doposażenie starego projektu w nowe spostrzeżenia. Co więcej, szybki prototyp można napisać w innym języku programowania niż drugi. Niektóre języki, takie jak LISP, doskonale nadają się do szybkiego prototypowania, ponieważ uwalniają programistę od niektórych bardziej nudnych aspektów innych języków, takich jak konieczność deklarowania typów wszystkich zmiennych. Wynikająca z tego szybkość rozwoju rekompensuje ryzyko związane z ich ignorowaniem. Drugi prototyp można wtedy napisać ostrożniej w bardziej restrykcyjnym języku. Jeśli programiści dobrze rozumieją dziedzinę problemu, rozwój ewolucyjny może nie być konieczny, ponieważ prawdopodobnie mogliby stworzyć szczegółową specyfikację bez eksperymentowania. Miałoby to miejsce na przykład, gdyby doświadczony zespół programistów pracował nad nowym produktem, podobnym do tych, które opracował wcześniej. Szybkie prototypowanie jest szczególnie przydatne, gdy problem do rozwiązania wymaga szczególnie innowacyjnych pomysłów lub po prostu nie jest dostatecznie dobrze zrozumiany, jak np. sztuczna inteligencja.

Model spiralny

Spiralny model cyklu życia jest uogólnieniem poprzednich.



Opiera się na procesie iteracyjnym, dzięki czemu umożliwia rozwój ewolucyjny, a także szybkie prototypowanie, gdzie każdy cykl może być oparty na modelu kaskadowym. Głównym wkładem modelu spiralnego jest skupienie się na zarządzaniu ryzykiem. Każda iteracja na spirali rozpoczyna się od fazy, w której określane są cele iteracji wraz z alternatywnymi sposobami ich osiągnięcia (u góry po lewej na rysunku). Po tym następuje faza oceny ryzyka, w której badane są zagrożenia związane z każdą alternatywą (prawy górny róg rysunku). Na tym etapie można na przykład ocenić wybór platform sprzętowych, języka programowania i narzędzi programowych. W przypadku nowej technologii, gdzie ryzyko jest szczególnie wysokie, może być konieczne przeprowadzenie oceny technologii na małą skalę w celu prawidłowej oceny i zmniejszenia związanego z nią ryzyka. Po wybraniu alternatywy jest ona realizowana i oceniana w trzeciej fazie (prawy dolny róg rysunku). Ta ocena jest następnie wykorzystywana do planowania następnej iteracji (na dole po lewej). Sam model spiralny nie określa szczegółów każdej iteracji, która mogłaby podążać za modelem kaskadowym lub w inny sposób przydzielać czynności. Podobnie jak model kaskadowy, model spiralny jest przede wszystkim ukierunkowany na sterowanie procesem i dlatego jest również oparty na dokumentach. W rzeczywistości ma dodatkowe rezultaty, w wyniku czego z dodatkowych trzech faz dodanych w celu zarządzania ryzykiem. Jest to uważane za niewielką cenę do zapłacenia w projektach wysokiego ryzyka.

Rozwój obiektowy

Jak widzieliśmy w Części 3, języki programowania obiektowego wyrosły z potrzeby modelowania zdarzeń w świecie rzeczywistym. Jest zatem naturalne, że koncepcje klas i obiektów są szczególnie odpowiednie do pozyskiwania wymagań i projektowania. Etapy obiektowej inżynierii oprogramowania są podobne do etapów innych metodologii przedstawionych powyżej. Jednak metodologie zorientowane obiektowo mają ważną zaletę polegającą na stosowaniu wspólnego słownictwa w ramach działań, opartego na pojęciach klas i obiektów. Np. program do zarządzania sklepem w sieci supermarketów jest w naturalny sposób określony pod kątem obiektów oznaczających klientów, kasjerów, terminale kasowe, artykuły spożywcze, magazyny, dostawców itp. Faza projektowania może dodawać inne rodzaje obiektów dla wewnętrznych celów algorytmicznych, ale jest to raczej rozwinięcie specyfikacji niż zupełnie inny rodzaj dokumentu. We wcześniejszych metodologiach przejście między czynnościami wiąże się z przechodzeniem między różnymi formalizmami. Ta potrzeba zmiany reprezentacji jest jednym ze słabych punktów metodologii niezorientowanych obiektowo, ponieważ prawdopodobnie zerwie połączenie między powiązanymi działaniami. Na przykład, często zdarza się, że programiści ignorują opracowane dla nich projekty, ponieważ są ukształtowani w formalizmie, który nie jest bezpośrednio związany z używanym przez nich językiem programowania. Powszechne słownictwo w metodologiach zorientowanych obiektowo sprawia, że jest to znacznie mniej prawdopodobne. Pozwala również na pewne mieszanie działań; na przykład projekt może być przeplatany programowaniem. W ten sposób doświadczenie zdobyte przy wdrażaniu jednej części systemu może wyjaśnić kwestie przydatne przy projektowaniu innych części. Dopóki proces jest dobrze zarządzany, może to przyspieszyć rozwój, zapewniając bardziej terminowe informacje zwrotne bez negatywnego wpływu na jakość. Jednym z ważnych celów inżynierii oprogramowania zawsze było ponowne wykorzystanie: możliwość korzystania z artefaktów (programów, projektów, a nawet wymagań) generowanych dla jednego projektu w kolejnym projekcie. Zaleta ponownego wykorzystania jest oczywista: pozwala deweloperowi szybko tworzyć nowe projekty po początkowej inwestycji w pierwszy tego typu projekt. Technologia obiektowa oferuje techniczne środki umożliwiające ponowne wykorzystanie. Ponieważ klasy hermetyzują typy danych wraz ze skojarzonymi z nimi operacjami, są one naturalną jednostką do ponownego użycia. Jest to jedna z głównych korzyści rozwoju obiektowego.

Ekstremalne programowanie

Podstawową zasadą modelu wodospadu jest to, że koszt zmian dramatycznie wzrasta z czasem, tak że naprawa błędów na początku procesu tworzenia staje się coraz bardziej kosztowna w miarę przechodzenia z etapu na etap. W rezultacie konieczne jest jak największe przewidywanie przyszłych zmian i uwzględnienie ich w projekcie. Może to sprawić, że projekt będzie bardziej złożony i nieporęczny niż to konieczne. Niedawno opracowano technikę zwaną refaktoryzacją, która ma pomóc w rozwiązaniu tego problemu. Zamiast zamrażać projekt na początku wdrożenia, ten pogląd przyjmuje odwrotne podejście: projekt powinien być elastyczny i zmieniać się wraz z wdrożeniem. Nie jest to łatwe. Zwykle zdarza się, że implementacja jest modyfikowana w odpowiedzi na presję klientów, zmieniające się środowiska i wykryte błędy, ale odbywa się to za pomocą „łat", które próbują zmodyfikować jakiś aspekt programu bez ogólnego widoku. W rezultacie program odbiega od pierwotnego projektu i coraz trudniej go modyfikować. Metodologia refaktoryzacji stara się zachować jakość projektu pomimo zmian w kodzie. Programowanie zmienia się między modyfikacjami, które dodają nowe funkcje lub naprawiają błędy, a refaktoryzacją, co poprawia strukturę programu bez wpływu na jego funkcjonalność. Jest to podobne do tego, co dzieje się, gdy musisz dokonać modyfikacji w swoim domu. Na przykład, kiedy dodajesz nowe gniazdko elektryczne, najpierw otwierasz ścianę, aby uzyskać dostęp do okablowania. Po zainstalowaniu nowych przewodów i gniazdka, zakrywasz otwór i odmalowujesz ścianę. Te ostatnie działania nie mają nic wspólnego ze sposobem funkcjonowania nowego gniazdka; dostarczy prąd, nawet jeśli nie zakryjesz dziury. Wykonuje się je w celu przywrócenia muru do pierwotnego stanu. Jeden z ekspertów w dziedzinie technologii obiektowej porównał sprzątanie projektu do spłacania długów, podczas gdy modyfikowanie programu podczas psucia projektu jest jak zaciągnięcie pożyczki. Niespłacona pożyczka nalicza odsetki, a jeśli nie jest spłacana przez długi czas, może przekroczyć zdolność kredytobiorcy do spłaty. Podobnie, jeśli program zbytnio odbiega od projektu, staje się niemożliwy do zarządzania. W końcu dalsze modyfikacje staną się niemożliwe, ponieważ każda taka próba przyniesie więcej problemów niż rozwiąże. Czasami warto pożyczyć pieniądze, ale dług należy spłacić jak najszybciej. Podobnie, czasami przydatne jest wprowadzenie szybkich zmian w programie, aby wykorzystać jakąś okazję. Jednak program powinien zostać zrefaktoryzowany tak szybko, jak to możliwe, aby ponownie dopasować go do dobrego projektu. Refaktoryzacja oferuje zestaw szczegółowych metod ulepszania projektu programu bez wpływu na jego funkcjonalność. W ten sposób nowe spostrzeżenia dotyczące struktury programu są wykorzystywane do ulepszania jego projektu. Takie spostrzeżenia mogą być . Istnieje również ryzyko ponownego użycia, jak pokazuje historia eksplozji Ariane 5 wspomniana na początku . Niestety są one znacznie mniej oczywiste. zdobyte podczas procesu dodawania nowej funkcjonalności, znajdowania i poprawiania błędów, czy podczas przeglądów kodu. Dzięki refaktoryzacji możliwe jest korygowanie błędów projektowych (lub w inny sposób modyfikowanie początkowego projektu) bez ponoszenia dużej kary nawet na znacznie późniejszym etapie procesu. Może to nie być skuteczne w bardzo dużych projektach, ale działa całkiem dobrze w małych i średnich projektach. Ponieważ błędy projektowe nie są tak zaporowe, początkowy projekt może być prosty, co upraszcza również późniejsze etapy. Wszystko to prowadzi do zupełnie innego stylu programowania, zwanego programowaniem ekstremalnym. Ekstremalna metodologia programowania wysoko ceni prostotę. Nazywa się to „ekstremalnym", ponieważ wykorzystuje każdą dobrą praktykę do maksimum. Na przykład zaleca częste i automatyczne testowanie (co jest również podstawową przesłanką do refaktoryzacji). Rozwój ewolucyjny jest również doprowadzony do logicznego zakończenia na wielu poziomach. Po pierwsze, integracja systemów odbywa się często (przynajmniej raz dziennie). Oznacza to, że cały czas dostępny jest w pełni działający system. Chociaż taka przejściowa wersja systemu może nie spełniać wszystkich wymagań, może być wykorzystana do oceny aktualnego stanu rozwoju w najbardziej bezpośredni sposób – poprzez faktyczne jego wykorzystanie. Po drugie, rozwój opiera się na krótkich cyklach (około

trzech tygodni), z których każdy skompresuje pozyskiwanie wymagań, projektowanie, implementację i walidację w jedną ciągłą czynność. Finalnie wydania klientów są dokonywane co kilka miesięcy, co oznacza, że klient musi być zaangażowany w projekt przez cały czas, a nie tylko na początkowym etapie. Rzeczywiście, przedstawiciel klienta na miejscu jest podstawowym wymogiem dla ekstremalnego programowania. W programowaniu ekstremalnym przeglądy kodu wykonywane są w sposób ciągły poprzez praktykę programowania w parach, w której każdą czynność programistyczną wykonują dwie osoby. W dowolnym momencie jedna z par będzie używać komputera do pisania lub modyfikowania programu, podczas gdy druga „spogląda przez ramię", aby spróbować zidentyfikować problemy, od błędów w pisowni po błędy logiczne. Okresowo „kierowca" i „nawigator" zamieniają się rolami. Mogłoby się wydawać, że dwie osoby wykonujące pracę jednej będą o połowę mniej produktywne. Jednak badania wykazały, że programowanie w parach jest prawie tak samo wydajne, jak dwie osoby pracujące oddzielnie pod względem ilości wykonywanej pracy, ale skutkuje programami o znacznie wyższej jakości. Wciąż nie ma wystarczającego doświadczenia z tą metodologią do pełnej oceny, ale kiedy działa, wydaje się być bardzo skuteczna. Spodziewamy się, że zdobędzie znaczące miejsce obok innych, cięższych metodologii.

Psychologia inżynierii oprogramowania

Projektanci metodologii często zapominają o najważniejszym czynniku w inżynierii oprogramowania: analitykach, projektantach, programistach i menedżerach. To wszystko są ludzie, z ludzkimi mocnymi i słabymi stronami. Jedną ze słabości jest tendencja aby łamać zasady, co utrudnia egzekwowanie wymagań metodologii. Możliwe jest zmuszenie programistów do tworzenia obszernej dokumentacji; prawie niemożliwe jest zmuszenie ich do stworzenia użytecznej dokumentacji. Skłonność ludzi do łamania zasad jest również mocną stroną, jeśli jest właściwie stosowana. Powszechnie wiadomo, choć często zapomina się, że poza metodologicznymi istnieje wiele czynników, które mają duży wpływ na wydajność i produktywność ludzi. Słynna anegdota opowiada o dużej świetlicy dla studentów w uniwersyteckim centrum obliczeniowym, w którym na jednym końcu znajdowało się kilka automatów. Kiedy kilku studentów narzekało na hałas dobiegający z tego rogu pokoju, administracja przeniosła maszyny w inne miejsce. Zaraz potem pojawił się poważniejszy problem: dwaj konsultanci, których centrum obliczeniowe pomagało studentom w rozwiązywaniu ich problemów, zostało zalanych tworząc długie kolejki przed ich pokojem. Okazało się, że hałas wokół automatów był często powodowany przez studentów rozmawiających ze sobą o swoich problemach komputerowych, a ponieważ problemy często były podobne, większość z nich rozwiązywano na miejscu. Do konsultantów zgłaszano tylko nietypowe problemy. Wraz z usunięciem tej nieformalnej, ale bardzo skutecznej usługi, cała sytuacja zmieniła się na gorsze. W ostatnich latach pojawiło się kilka metodologii, które próbują rozwiązać problem człowieka. Zamiast nazywać siebie „lekkimi metodologiami", co jest negatywnym terminem, który odróżnia ich od innych „ciężkich" metodologii, zaczęli nazywać siebie zwinnymi. Termin ten leży u podstaw zmiany punktu ciężkości z zarządzania dokumentami na zarządzanie oparte na ludziach. Metodologie zwinne, których jednym z przykładów jest programowanie ekstremalne, próbują dać ludziom motywację i wsparcie, których potrzebują do wykonywania swojej pracy, i ufają im, że wykonają ją dobrze. W rezultacie kładą duży nacisk na komunikację między klientami, projektantami i programistami i opowiadają się za tym, aby wszyscy pracowali blisko siebie; jeśli to możliwe, w tym samym pokoju. Gdy członkowie zespołu skutecznie się komunikują, potrzeba znacznie mniej dokumentacji papierowej. Zamiast śledzić postępy za pomocą dodatkowej dokumentacji, metodyki zwinne wykorzystują działające oprogramowanie. O wiele łatwiej jest ocenić działający produkt niż dokument projektowy. Oczywiście oznacza to, że proces musi często wspierać produkcję stabilnych wersji. W wyniku tej filozofii zespoły stosujące metodyki zwinne mogą szybciej reagować na zmiany niż te, które korzystają ze stałych planów. Tradycyjne metodologie zgodne ze strategią „linii produkcyjnej" i postrzegają programistów (takich jak analitycy, projektanci, koderzy i testerzy) jako

wymiennych w ramach ich poszczególnych kategorii. Metodologie zwinne skupiają się zamiast tego na indywidualnym rzemiośle, bez sztywnych granic i gdzie programiści zdobywają wiedzę i doświadczenie współpracując z innymi. Zaczynają jako praktykanci, wykonując prostsze i bardziej żmudne części pracy, ale nie są trzymani z dala od zadań, które wymagają większej wiedzy. W miarę postępów podejmują się niektórych bardziej skomplikowanych zadań, aż osiągną pozycję, w której sami stają się rzemieślnikami. Ważne jest, aby zrozumieć, że nie odpowiada to zwykłemu przejściu od programisty do menedżera. Rzemieślnik przyjmuje na siebie więcej obowiązków niż praktykant, w tym obowiązki związane z zarządzaniem, ale to nie uniemożliwia mu robienia tego, co robi najlepiej -rzeczywistego tworzenia oprogramowania. . Nie trzeba dodawać, że takie podejście kładzie nacisk na jednostkę, a nie na proces, i jest bardzo preferowane przez samych programistów. Oczywiste jest, że większe zespoły lub te opracowujące aplikacje krytyczne dla życia muszą używać bardziej formalnych procesów z większą ilością dokumentacji niż mniejsze zespoły opracowujące mniej krytyczne oprogramowanie. Rzeczywiście, istnieje wiele odmian zwinnych metodologii, ukierunkowanych na różne typy projektów. W miarę jak podejście to stanie się bardziej znane i rozpowszechnione, prawdopodobnie zostanie ono przetestowane w większych organizacjach, w tym w tych, które opracowują krytyczne aplikacje. Miłośnicy zwinnych metodologii tworzenia oprogramowania twierdzą, że nie ma niczego, czego nie można zrobić przy użyciu tych metodologii. Czas pokaże, czy mają rację.

Etyka zawodowa

Ze względu na coraz większe wykorzystanie oprogramowania w wielu krytycznych systemach z jednej strony, a naszą niewystarczającą zdolność weryfikacji jego poprawności z drugiej, duży nacisk należy położyć na rzetelność i profesjonalizm osób, które specyfikują, projektują i rozwijają systemy komputerowe. Muszą dołożyć wszelkich starań, aby tworzone przez nich oprogramowanie było najwyższej jakości, zgodnie z najlepszymi dostępnymi praktykami w tej dziedzinie. Dwie główne profesjonalne organizacje komputerowe, ACM i IEEE Computer Society, przygotowały kodeksy etyczne dla specjalistów komputerowych. Wyszczególniają one szczegółowe obowiązki programistów systemów w odniesieniu do ogółu społeczeństwa, a w szczególności organizacji, dla których pracują. Na przykład, poucza się ich, aby w swojej pracy przyczyniali się do dobrobytu społeczeństwa i ludzi, unikali krzywdzenia innych, byli uczciwi i uczciwi oraz szanowali prywatność, poufność i własność intelektualną innych. Powinni dążyć do najwyższej jakości i efektywności swojej pracy zawodowej, edukować społeczeństwo w zakresie implikacji systemów skomputeryzowanych oraz doskonalić własną edukację techniczną. Menedżerowie powinni tworzyć środowiska pracy, które ułatwiają pracownikom przestrzeganie kodeksu i muszą mieć pewność, że potrzeby wszystkich osób, na które mają wpływ ich produkty, są brane pod uwagę. Powiązany problem dotyczy certyfikacji. Czy profesjonaliści komputerowi powinni być certyfikowani przez jedno ze stowarzyszeń zawodowych i czy taka certyfikacja powinna być wymogiem przy pracy nad krytycznymi projektami? Stowarzyszenie Komputerowe IEEE ma dla swoich członków dobrowolny program certyfikacji, który oprócz wymagań technicznych dotyczących certyfikacji, wymaga również przestrzegania kodeksu etyki. Oczekuje się, że takie programy, wraz z włączeniem kursów etycznych do programu nauczania informatyki, doprowadzą do wyższych standardów etycznych w zawodzie, a tym samym do wyższej jakości produktów.

Badania nad inżynierią oprogramowania

Pod koniec lat siedemdziesiątych kandydat do pracy przybył do jednego z ośrodków badawczych IBM na wykład i rozmowę kwalifikacyjną. Temat, na który wykładał, miał coś wspólnego z porównywaniem języków programowania. Po około 15 minutach jego wystąpienia stało się jasne, że planuje zaprezentować metodę porównywania składni języków programowania. Jedna z osób na widowni, ekspertka od języków formalnych, podniosła palec i powiedziała potulnie: „Ale ten problem jest

nierozstrzygnięty. Pokażę ci prawdziwy praktyczny algorytm.” Oczywiście nigdy nie dostał pracy . . . W dyscyplinach inżynierskich istnieje tendencja do ciągłego napięcia między teorią a praktyką i potrzeba czasu, aby idee teoretyczne dotarły do praktyków. Niektóre pomysły szybko się przyjmą, inne zabierają więcej czasu, a niektóre pozostają czystą teorią (co często nie umniejsza ich znaczenia). W algorytmice względy wydajnościowe (rozdział 6) bardziej bezpośrednio przemawiały do praktyków niż poprawność. Wynika to prawdopodobnie z dwóch powodów. Po pierwsze, rażąco niewydajny program jest bezużyteczny, podczas gdy program z kilkoma rzadko ujawnianymi błędami nadal może być używany (choć z pewną frustracją). Po drugie, odkrycie nowego, wydajniejszego algorytmu, coś najczęściej wykonane przez teoretyków, jest zwykle łatwe do przetłumaczenia na kod, a gdy już to zrobi, kod może być używany w wielu programach. Z drugiej strony udowodnienie poprawności programu jest niezwykle trudne i musi być przeprowadzane osobno dla każdego nowego programu.

Badania nad poprawnością algorytmów wpłynęły na inżynierię oprogramowania na wiele sposobów. Najintensywniejsze wykorzystanie metod formalnych w przemyśle to weryfikacja, czyli formalne udowodnienie, że produkt spełnia swoją specyfikację.. Istnieją również dość wyrafinowane narzędzia, które mogą pomóc zweryfikować dowód przedstawiony przez eksperta. Narzędzia te mają dwie zalety. Po pierwsze, konstruują całkowicie formalne dowody, na poziomie szczegółowości nieosiągalnym dla człowieka (nie dlatego, że jest to dla człowieka zbyt trudne intelektualnie, ale z przeciwnego powodu: jest zbyt długi i żmudny). Po drugie, pozwalają użytkownikom skoncentrować się na koncepcjach i strategiach wysokiego poziomu, pozwalając narzędziu wypełnić szczegóły. Mimo to weryfikacja wymaga dużo czasu i wysiłku i jest stosowana głównie przez producentów sprzętu, ponieważ błędy sprzętowe są znacznie bardziej kosztowne w naprawie niż błędy oprogramowania. Na przykład wszystkie podstawowe operacje zmiennoprzecinkowe na mikroprocesorze AMD Athlon zostały mechanicznie zweryfikowane pod kątem zgodności ze standardową specyfikacją. Prace zostały wykonane w AMD przed wyprodukowaniem Athlona i odkryto kilka błędów. Motorola i Intel również odniosły imponujące sukcesy w korzystaniu z narzędzia weryfikacji. Ważnym wkładem formalnych metod badawczych do praktyki programowania jest projektowanie na podstawie umowy, o którym wspomniano w Części 5. Twierdzenia, które można napisać w językach takich jak EIFFEL, nie są wystarczająco silne, aby wyrazić wszystkie matematyczne właściwości programu potrzebne do weryfikacji. Są one jednak wykonywalne w tym sensie, że komputer może sprawdzić, czy są one prawdziwe podczas wykonywania programu. Zatem projektowanie na podstawie umowy jest dobrym przykładem użytecznego formalnego rozumowania dotyczącego programów. Zwiększa jakość programów, a także szybkość rozwoju (biorąc pod uwagę, że pozwala to na wczesne wykrywanie błędów), a to rekompensuje dodatkowy wysiłek wymagany przy pisaniu asercji. Ponadto zachęca programistów do zastanowienia się nad formalnymi właściwościami ich kodu, a nawet do udowodnienia sobie (na razie bez automatycznej pomocy), że ten konkretny problem nazywa się problemem równoważności dla języków bezkontekstowych i został udowodniony być nierozstrzygalnym. są poprawne. W miarę jak narzędzia do dowodzenia twierdzeń stają się coraz bardziej wyrafinowane, będą w stanie zaoferować większą pomoc w tym procesie. Badania doprowadziły również do postępu w zakresie języków wymagań i specyfikacji, głównie tych, które są zarówno wizualne, jak i matematyczne, i które są używane do określania złożonych zachowań.