

Równowaga między zwinnością a dyscypliną z wykorzystaniem XPrince

Łukasz Olek*

Politechnika Poznańska, Instytut Informatyki
ul. Piotrowo 3A, 60-965 Poznań

Streszczenie. Większość współczesnych projektów informatycznych wymaga równowagi pomiędzy zwinnością a dyscypliną. W tym artykule zaprezentowano metodykę wytwarzania oprogramowania i zarządzania projektami XPrince (eXtreme PRogramming IN Controlled Environments). Jest to połączenie Programowania Ekstremalnego (XP), PRINCE2 i RUP. Opisano również pewne eksperymenty i narzędzia, które stanowią podstawę dla metodyki XPrince.

1 Wprowadzenie

Reakcją na kryzys oprogramowania w późnych latach 60-tych XX wieku było zwiększenie dyscypliny w wytwarzaniu oprogramowania. W ciągu następnych 20-tu lat zaproponowano wiele standardów (IEEE, ISO, itd.), modeli dojrzałości (CMM, ISO 15504, itd.) oraz metodyk zorientowanych na dyscyplinę (np. PSP [13], TSP [14]). Równolegle do tego procesu, w latach 70-tych zastosowano pierwsze podejścia zarządzania projektami do wytwarzania oprogramowania.

Prawdopodobnie pierwszą z nich był PROMPTII, stworzony przez *Sympact Systems Ltd.*, wdrożony w roku 1979 przez *Central Computer and Telecommunications Agency* (CCTA) w Wielkiej Brytanii. W roku 1989 CCTA opracowało własną metodykę nazwaną PRINCE (*Projects IN Controlled Environments*). Kilka lat później zmodyfikowano ją i od tamtego czasu nazywana jest PRINCE2 [18]. PRINCE2 jest szeroko znane jako raczej restrykcyjne, lecz skuteczne podejście do zarządzania projektami.

Jednak zbyt duża restrykcyjność ogranicza inicjatywę i elastyczność, które są bardzo potrzebne do budowy skomplikowanych systemów ze zmiennymi wymaganiami. Aby temu zaradzić, w latach 90-tych powstały tak zwane zwinne metodyki (ang. *agile methodologies*). Kładą one nacisk na efektywną komunikację pomiędzy osobami, są zorientowane na klienta, powstające oprogramowanie oraz szybkie reakcje na zmiany. Prawdopodobnie najbardziej rozpowszechnioną zwinną metodyką jest Programowanie Ekstremalne (XP) [3].

* We współpracy z **Jerzym Nawrockim**, Michałem Jasińskim, Bartoszem Paliświatem, Bartoszem Walterem, Błażem Pietrzakiem i Piotrem Godkiem

Nie ma niestety, jedyne złote rozwiązanie i każde z podejść, zarówno zwinne jak i zorientowane na dyscyplinę, mają swoje zalety i wady. Metodyki zorientowane na dyscyplinę są zazwyczaj krytykowane ze względu na nadmierną pracę papierkową, małą elastyczność, powolne procesy podejmowania decyzji i niezdolność dostosowania się do zmian w trakcie trwania projektu. Słabe strony Programowania Ekstremalnego to wymóg, aby klient pracował razem z zespołem (w wielu projektach klienci są zbyt zajęci i nie mogą spełnić tego wymagania), brak dokumentacji papierowej (komunikacja ustna jest bardzo efektywna, lecz w przypadku bardzo złożonego systemu mogą być trudności z wprowadzaniem zmian po upływie pewnego czasu) oraz czasami zbyt krótka perspektywa planowania.

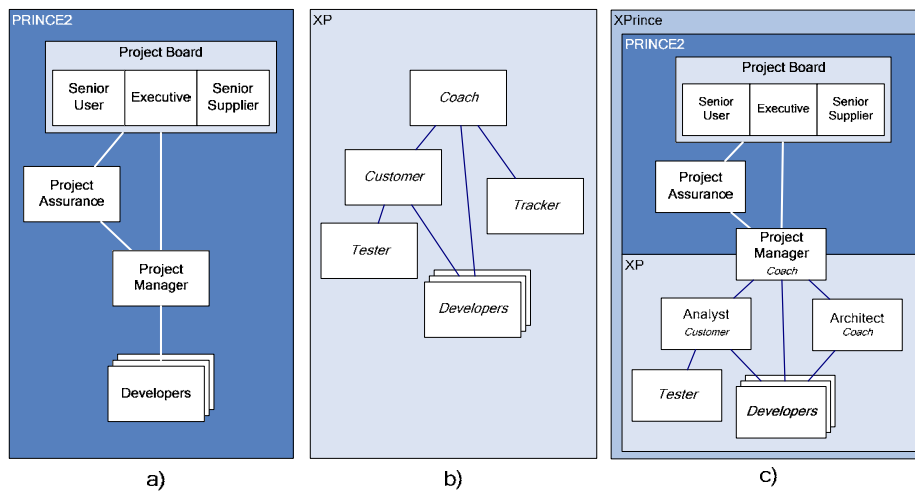
Jak zauważyli Barry Boehm i Richard Turner: *każde pomyślnie przedsięwzięcie w zmieniającym się świecie wymaga zarówno zwinności jak i dyscypliny* ([5], p. 2.). Niniejszy artykuł opisuje zintegrowaną i elastyczną metodologię wytwarzania oprogramowania wraz z towarzyszącymi narzędziami, której celem jest wyważenie między zwinnością i dyscypliną. Nazywa się XPrince (eXtreme PRogramming IN Controlled Environments) i bazuje na trzech innych metodykach: XP [3], PRINCE2 [18] oraz RUP [17]. W następnym rozdziale opisano dwupoziomowe podejście do organizacji zespołu, które powoduje strukturę zespołu zgodną zarówno z XP jak i PRINCE2. W rozdziale 3 przedstawiono cykl życia projektu. Następnie zaprezentowano narzędzia i techniki, których mają zapewnić zwinność i efektywność w zakresie inżynierii wymagań (Roz. 4) oraz konstrukcji oprogramowania (Roz. 5). Naszym celem było rozwiązanie problemów związanych ze słabościami XP oraz zachowanie zwinności. W tym celu zintegrowaliśmy metodykę zarządzania projektem (PRINCE2) z metodyką wytwarzania oprogramowania (XP) oraz stworzyliśmy narzędzia, które pomagają efektywnie zintegrować różne techniki wytwarzania oprogramowania. Zintegrowaliśmy edytor wymagań w postaci przypadków użycia z generatorem makiet funkcjonalnych oraz kalkulatorem pracochłonności (powstałe narzędzie nazywa się *UC Workbench*). Zintegrowaliśmy również ponowne użycie oprogramowania (*reuse*) z testowaniem (przypadki testowe są wykorzystywane jako specyfikacja wyszukiwanych komponentów oprogramowania).

2 Struktura zespołu

Na pierwszy rzut oka, PRINCE2 nie pasuje do XP z wielu powodów. Jednym z nich jest fakt, iż role w PRINCE2 różnią się od ról w XP. Projekt prowadzony wg PRINCE2 jest kierowany przez Zarząd Projektu (ang. *Project Board*), który składa się z trzech ról (zobacz rys. 1a):

- *Dyrektor* (ang. *Executive*) – Reprezentuje inwestora, jest odpowiedzialny za sukces projektu z biznesowego punktu widzenia. Ma prawo anulować projekt, jeżeli jest taka potrzeba.
- *Główny użytkownik* (ang. *Senior User*) – Kieruje użytkownikami końcowymi, skupia się na aspektach użyteczności (ang. *usability*).
- *Główny dostawca* (ang. *Senior Supplier*) – Reprezentuje organizację dostawcy (główny kierownik).

PRINCE2 zakłada, że członkowie Zarządu Projektu są zbyt zajęci, aby troszczyć się o projekt z dnia na dzień. Dlatego też w PRINCE2 istnieje Menedżer Projektu (ang. *Project Manager*), który jest odpowiedzialny za taktyczny poziom zarządzania. Jego zadaniem jest przygotowanie planów, które są następnie akceptowane przez Zarząd Projektu, oraz przygotowanie raportów z postępu projektu. W celu zrównoważenia „naturalnego optymizmu” Menedżera Projektu, istnieje rola Audytora Projektu (ang. *Project Assurance*), której celem jest sprawdzanie, czy raporty dostarczane przez Menedżera Projektu odnoszą się do rzeczywistości. W przypadku małych zespołów programistów są oni koordynowani bezpośrednio przez Kierownika Projektu.



Rys. 1. Minimalna struktura zespołu w PRINCE2 (a). Diagram struktury zespołu w XP (b). Struktura zespołu w XPrince (c)

Programowanie Ekstremalne nie ma konkretnego diagramu struktury zespołu. Jednakże z opisów Kenta Becka [3] można wywieść taki diagram (Rys. 1b), który następnie może być przekształcony w diagram przedstawiony na rysunku 1c. To przekształcenie pozwala ująć zespół XP jako zgodny z PRINCE2. Z punktu widzenia PRINCE2 Menedżer Projektu kontroluje zespół programistów. PRINCE2 nie nakłada żadnych ograniczeń na to, w jaki sposób programiści powinni być zorganizowani, więc z tego punktu widzenia Analityk (ang. *Analyst*) lub Architekt (ang. *Architect*) są również programistami. Z drugiej strony, programiści mogą być nieświadomi z istnienia Zarządu Projektu. Z ich punktu widzenia Analityk (rola pochodząca z RUP) jest klientem, natomiast Architekt i Menedżer Projektu pełnią rolę Trenerów (ang. *Coach*). W XP jest tylko jeden Trener, lecz jego rola jest dwójaka: usuwanie przeszkód organizacyjnych (np. brak papieru do wydruków) oraz interweniowanie w przypadku problemów technologicznych (np. testy jednostkowe wykonują się zbyt długo). Biorąc to pod uwagę, zdecydowaliśmy się rozdzielić te role na dwie osoby. Menedżer Projektu jest odpowiedzialny za prawidłowe środowisko pracy (włączając w to dobry kontakt z Zarządem Projektu), rozwiązuje problemy personalne i raczej motywuje niż kieruje zespołem [4]. Dobry *Menedżer Projektu* powinien zbudować swój zespół zgodnie z zasadami etyki charakteru

zaproponowanymi przez Stephena Coveya [10]. *Architekt* jest dużo bardziej zorientowany na techniczne aspekty projektu. Jest osobą dodatkową, nieobecną w XP ani w PRINCE2, natomiast odgrywa istotną rolę w w RUP. Z punktu widzenia programistów, Architekt jest głównym projektantem (zgodnie z *Chief Surgeon* Brooka), powinien więc posiadać duże doświadczenie i pomagać merytorycznie programistom. Architekt jest odpowiedzialny za początkową architekturę systemu jak również jej późniejszą pielęgnację, natomiast programiści są odpowiedzialni za „wypełnianie” architektury konkretną funkcjonalnością. Bardzo dobry opis roli architekta podali Kroll i Kruchten [17].

3 Cykl życia projektu

Cykl życia projektu jest podstawą planowania. W PRINCE2 (rys. 2a) projekt zaczyna się fazą *Rozpoczęcia* (ang. *Starting-up*): wybieranie zespołu kierowniczego, przygotowywanie *Szkicu projektu* i *Podejścia do projektu*, planowanie fazy inicjacji. Może być ona bardzo krótka — czasem wystarczy kilka godzin. Po niej następuje faza *Inicjacji projektu* (ang. *Initiating a Project*): planowanie projektu, praca nad uzasadnieniem biznesowym i ryzykiem, ustalanie kontroli i plików projektu, stworzenie *Dokumentu inicjacji projektu*. Po tym następuje kilka *etapów* projektu (każdy *etap* posiada listę oczekiwanych produktów i jest kontrolowany wg własnego planu). Projekt kończy się *Zamknięciem*: uzyskaniem akceptacji klienta, identyfikacją dalszych akcji, ewaluacją projektu.

W RUP-ie (rys. 2b) projekt składa się z czterech faz: *Rozpoczęcia* (ang. *Inception*) (dowiadywania się co i w jaki sposób zbudować), *Elaboracji* (ang. *Elaboration*) (pracy nad architekturą, planowania projektu, ograniczania najistotniejszych czynników ryzyka, ustalania środowiska projektu), *Konstrukcji* (ang. *Construction*) (przygotowywania w pełni funkcjonalnej wersji beta systemu), *Tranzycji* (ang. *Transition*) (przygotowywania wdrożenia, szkolenia użytkowników, sprawdzenia czy oczekiwania użytkowników zostały spełnione i czy wdrożenie zostało pomyślnie zakończone).

Najprostszy jest cykl życia projektu wg XP (rys. 2c). Składa się z serii wydań, a każde wydanie z serii przyrostów. Każde wydanie i każdy przyrost zaczyna się sesją planowania (ang. *planning session*), nazywanej również Grą Planistyczną (ang. *Planning Game*). Nie ma tutaj całościowego planu projektu – perspektywa planowania jest ograniczona do pojedynczego wydania. Niekiedy ludzie wykorzystują pierwszy przyrost (tzw. przyrost o zerowej funkcjonalności) jako organizacyjny i przygotowują środowisko projektowe.

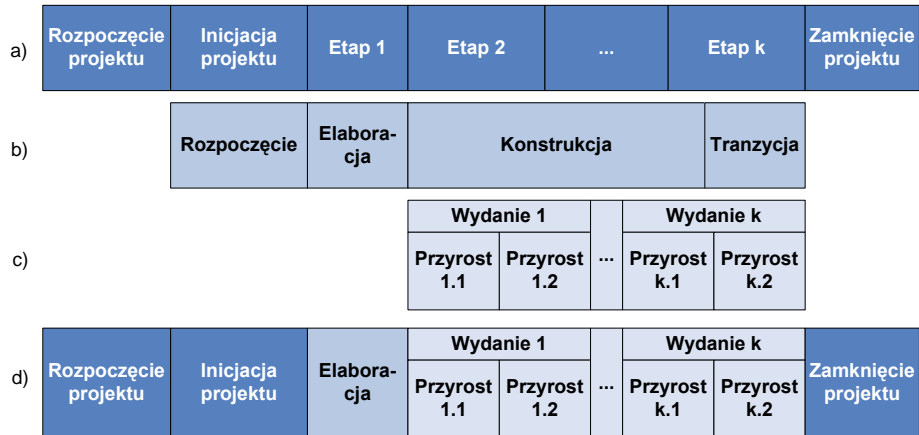


Fig. 2. Cykle życia projektu zaproponowane przez różne metodyki: PRINCE 2 (a), RUP (b), XP (c) i XPrince (d).

Cykl życia projektu w XPrince jest połączeniem wszystkich wspomnianych podejść. Koncepcja „klamer” otwierających i zamykających, obecna w PRINCE2, jest bardzo praktyczna. Są to czynności zupełnie niezwiązane z technologią, więc wydaje się sensowne oddzielenie tego od innych faz (w RUP-ie jest to włączone w fazę Rozpoczęcia i Tranzycji). *Rozpoczęcie projektu* jest w zazwyczaj wykonywane przez Menadżera Projektu, który ma następujące zadania:

- *Ustanowić zespół zarządzania projektem* (zobacz poprzedni rozdział),
- *Stworzyć wizję systemu* (jest to krótsza i bardziej konkretna wersja dokumentów *Szkic projektu* i *Podejście do projektu* z XPRINCE, zawiera on wstępne argumenty biznesowe),
- *Zaplanować fazę Inicjacji projektu*.

Następną fazą jest *Inicjacja projektu*. Jej celem jest dostarczenie planu i stworzenie środowiska organizacyjnego dla projektu. Jest to połączenie Inicjacji projektu z PRINCE2 oraz fazy Rozpoczęcia z RUP. Zadania tej fazy wykonują głównie Kierownik projektu i Analityk z pomocą Architekta:

- *Zrozumieć, co należy zbudować*. Niekiedy konieczne jest stworzenie lekkiej wersji dokumentu ConOps [15], zawierającego model biznesowy bazujący na przypadkach użycia, listę problemów, które należy rozwiązać oraz najważniejszą funkcjonalność wymaganą do rozwiązania tych problemów. Kluczowej funkcjonalności powinna towarzyszyć lista kryteriów jakości i produktów. Osobą odpowiedzialną za to zadanie jest Analityk, który powinien również śledzić ryzyko z tym związane.
- *Zaproponowanie początkowej architektury*. Powinien to być krótki, wysokopoziomowy opis, dostarczający informacji potrzebnej do zaplanowania projektu. Powinien również zawierać listę potrzebnych narzędzi. Osobą odpowiedzialną za to zadanie, wraz z czynnikami ryzyka z nim związanymi jest Architekt, lecz w przypadku gdy architektura rozwiązania wydaje się być oczywista, również Analityk może wykonać to zadanie

- *Zaplanowanie całego projektu i dopracowanie uzasadnienia biznesowego (ang. business case).* Ten cel jest nadzorowany przez Menadżera Projektu, który jest również odpowiedzialny za śledzenie ryzyka z tym związanego. Plan projektu pokazuje projekt ze strategicznego punktu widzenia. W celu wspierania „zwinności” plan projektu powinien być zbudowany zgodnie z zasadą „rzeczy najważniejsze najpierw” [10]. Powinien precyzować liczbę wydań i przydzielić do nich funkcjonalność (przypadki użycia na wysokim poziomie). Im dłuższy projekt, tym plan projektu powinien być mniej konkretny. Faktyczne planowanie (???) powinno się odbywać na poziomie wydań. W XP nie istnieje plan projektu – są jedynie plany wydań. W XPrince plan projektu został dodany nie tylko ze względu na zgodność z PRINCE2, lecz również aby zapewnić szerszą perspektywę, która okazuje się bardzo potrzebna. Należy zrozumieć, iż plan projektu jest źródłem cennej informacji, nie zaś usprawiedliwieniem odrzucania propozycji zmian. Każda późniejsza propozycja zmiany powinna być zaakceptowana, jeżeli pomaga osiągnąć cele biznesowe.
- *Ustalenie kanałów komunikacyjnych i środowiska zarządzania projektem.* Kanały komunikacyjne obejmują raporty (np. wyniki cotygodniowych testów akceptacyjnych, sugerowanych przez XP). Środowisko zarządzania projektem może być klasyczne, bazujące na plikach i dokumentach, lub może być wspomagane zaawansowanymi narzędziami. Ten cel leży w obowiązkach Menadżera Projektu.

- *Plan fazy elaboracji.*

Faza *Elaboracji* dotyczy głównie architektury. Architekt powinien zaproponować mechanizmy architektoniczne, rozpoznać ryzyko z tym związane (np. za pomocą eksperymentów) oraz stworzyć szkielet, który będzie wykorzystywany przez Programistów. Analityk i Menadżer Projektu w tej fazie udoskonalają wymagania i plan projektu.

Każde *Wydanie* składa się z kilku przyrostów, po których następuje faza tranzycji. Na tym etapie proces wytwarzania oprogramowania bardzo przypomina XP. Architekt i Programiści produkują kod i przypadki testowe. Analityk jest odpowiedzialny za wymagania i testy akceptacyjne, jak również gra rolę klienta będącego na miejscu. Przyrost jest jedynie wewnętrznym punktem kontrolnym. Każde Wydanie jest zakończone fazą tranzycji, w której nowa wersja systemu jest wdrażana i przekazywana użytkownikom końcowym. Podobnie jak w XP, każdy przyrost powinien być tak samo długi – to pomaga Programistom czuć rytm iteracji oraz w rezultacie nauczyć się lepiej planować przyrosty.

Zamknięcie projektu bardzo przypomina odpowiadającą fazę z PRINCE2. Projekt jest zamykany, identyfikowane są dalsze akcje i następuje ocena projektu.

4 Inżynieria wymagań z wykorzystaniem UC Workbench

W tym rozdziale prezentujemy narzędzie wspomagające inżynierię wymagań, bazujące na przypadkach użycia: UC Workbench (UC – skrót od Use Case – przypadek użycia). UC Workbench został stworzony z myślą o Analityku.

4.1 Notacja tekstowa, czy graficzna?

Według popularnego powiedzenia, jeden obraz ma wartość tysiąca słów. Niestety, zasada ta zdaje się nie działać w przypadku inżynierii wymagań. W marcu i kwietniu 2005 przeprowadziliśmy eksperyment na Politechnice Poznańskiej. Celem eksperymentu było znalezienie odpowiedzi na pytanie: jaka reprezentacja wymagań: tekstowa, czy graficzna, jest lepsza z punktu widzenia stopnia zrozumienia przez osoby je czytające.

Jako reprezentanta podejścia tekstowego wybraliśmy przypadki użycia [8,1]. Dla notacji graficznej wybraliśmy zapis BPMN [27], który przypomina UML, lecz jest tworzony z zamiarem opisu modeli biznesowych. Uczestnikami eksperymentu byli studenci 4-tego roku Inżynierii Oprogramowania (IO) oraz Gospodarki Elektronicznej (GE). W sumie było 17 studentów IO oraz 11 GE.

Eksperyment składał się z następujących kroków:

1. *Wykładu* wprowadzającego do konkretnej notacji (90 minut).
2. *Ćwiczeń* w trakcie których studenci otrzymali wysokopoziomowy zapis procesów PRINCE2 wyrażonych w danej notacji, zawierającej celowo posiane błędy. Zadaniem studentów było ich znalezienie. Dokument miał 5 stron, natomiast etap ćwiczeń trwał 90 minut.
3. *Eksperyment*, który przebiegał w podobny sposób jak ćwiczenia. Tym razem jednak zmieniona została dziedzina biznesowa. Zamiast procesu PRINCE2, studenci otrzymali model biznesowy opisujący regulamin studiów (zaliczanie semestrów, zdawanie egzaminów, egzamin dyplomowy, itp.). Studenci w ciągu godziny musieli znaleźć wszystkie możliwe błędy.

Eksperyment został powtórzony dwa razy (za każdym razem dziedzina biznesowa była inna), aby uzyskać bardziej wiarygodne wyniki. Za każdym razem były dwie grupy: jedna pracująca na przypadkach użycia i druga na diagramach BPMN. Mogło się zdarzyć, że dwie grupy różniły się poziomem swoich zdolności. Aby tego uniknąć, za drugim razem zamieniliśmy grupy (grupa pracująca za pierwszym razem na przypadkach użycia, za drugim otrzymała model BPMN i na odwrót). Studenci pracowali indywidualnie. Każdy znaleziony błąd był krótko opisywany w rejestrze błędów. Za stopień zrozumienia wymagań przyjęliśmy liczbę znalezionych błędów w dokumencie. Współczynnik znalezionych błędów (DDR, ang. *Defect Definition Ratio*) dla osoby p był zdefiniowany następująco:

$$\text{DDR}(p) = \frac{\text{Liczba błędów znaleziona przez osobę } p}{\text{Liczba wszystkich błędów}} \cdot 100\%$$

DDR dla przypadków użycia był wyższy niż dla diagramów BPMN i wynik był istotny statystycznie (na poziomie istotności 0,05). Uzasadnia to następujące przypuszczenie: *przypadki użycia są łatwiejsze do zrozumienia niż diagramy BPMN*. Lepiej zatem wyrażać modele biznesowe korzystając z przypadków użycia.

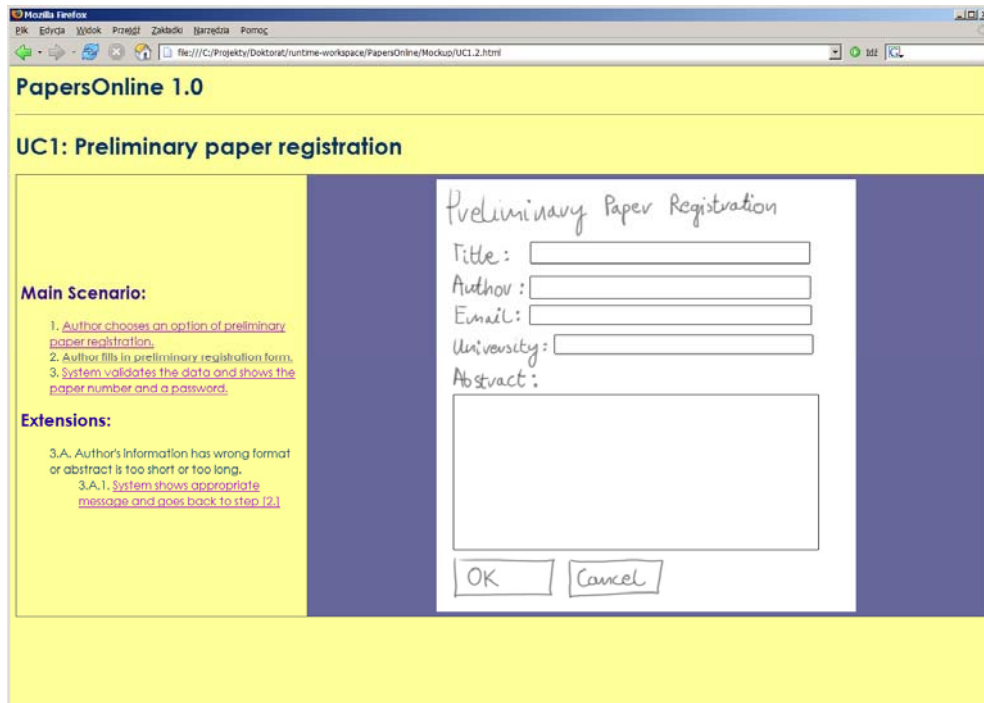
Przeprowadziliśmy również kolejny eksperyment. Tym razem jedna grupa studentów otrzymała model biznesowy wyrażony jedynie za pomocą przypadków użycia, druga natomiast oprócz przypadków użycia otrzymała diagramy BPMN opisujące te same procesy biznesowe. Okazało się, iż ta druga grupa znalazła więcej błędów, a wynik był również istotny statystycznie. Diagramy BPMN są więc pozytywnym dodatkiem do przypadków użycia. Biorąc pod uwagę wyniki

eksperymentów inżynieria wymagań w XPrince bazuje na przypadkach użycia, natomiast diagramy służą jedynie jako przydatne dodatki.

4.2 UC Workbench

UC Workbench to narzędzie wspomagające zarządzanie wymaganiami i modelowanie dziedziny biznesowej bazujące na przypadkach użycia, rozwijane na Politechnice Poznańskiej [21]. Byliśmy zaskoczeni faktem, że nie istnieje żadne dobre narzędzie służące do inżynierii przypadków użycia. UC Workbench posiada następujące funkcje:

- *Edytor przypadków użycia* z automatyczną renumeracją kroków w głównym scenariuszu, jak również w rozszerzeniach.
- *Automatyczne przeglądy* z wykrywaniem potencjalnych błędów (np. niezdefiniowany lub nieużywany aktor, za długie bądź za krótkie scenariusze, rozszerzenie bez kroków itd.)
- *Generowanie makiet funkcjonalnych* z zebranych przypadków użycia. Automatycznie generowana makiet jest otwierana w przeglądarce internetowej i składa się z dwóch okien (zobacz rys. 3): *okna scenariusza* (prezentuje aktualnie animowany przypadek użycia) oraz *okna ekranów* (pokazuje projekt wyglądu formularzy aplikacji). W przypadku procesów biznesowych okno ekranów może zawierać diagramy BPMN dla przeglądanych procesów.
- *Komponowanie dokumentu specyfikacji wymagań* bazujące na standardzie IEEE 830-1998. UC Workbench generuje specyfikacje wymagań z przypadków użycia.
- *Kalkulator pracochłonności bazujący na metodzie UC Points* [16] wspiera Grę Planistyczną z Programowania Ekstremalnego. W XPrince planowanie przebiega na trzech poziomach: metoda UC Points (najniższy poziom) podaje domyślne szacunki, które mogą być następnie dostosowywane przez ekspertów; metoda delficka jest używana do szacowania pracochłonności przez ekspertów oraz Gra Planistyczna (najwyższy poziom) służy do uzgodnienia zakresu następnego wydania pomiędzy klientem (i Analitykiem) oraz Programistami kierowanymi przez Architekta.



Rys. 3. Zrzut ekranowy makiety wygenerowanej automatycznie przez UC Workbench.

Wierzmy, że prawidłowe wykorzystanie narzędzi może być bardzo pomocne w równoważeniu zwinności i dyscypliny. Mogą one dostarczyć dużo szybciej i taniej informacji dostępnej w metodykach zorientowanych na dyscyplinę. Ze względu na to zmiana wymagań nie stanowi już tak dużego problemu, jakim było do tej pory.

W celu ewaluacji narzędzia UC Workbench przeprowadziliśmy prosty eksperyment mający na celu porównanie UC Workbench z edytorem tekstowym ogólnego użytku (MS Word). W eksperymencie wzięło udział 12 uczestników (studentów 4-tego roku, specjalizacji Inżynieria Oprogramowania). Byli podzieleni na dwie równoliczne grupy. Jedna z nich korzystała z MS Word, natomiast druga z UC Workbench. Studenci otrzymali zarys czterech przypadków użycia (każdy zawierał 6-9 kroków). Eksperyment składał się z dwóch etapów. W pierwszym uczestnicy mieli przepisać dostarczone przypadki użycia za pomocą dostarczonego edytora. W drugim etapie byli poproszeni o wprowadzenie pewnych zmian do specyfikacji. Okazało się (zobacz rys. 4) że korzystanie z UC Workbench zaoszczędza około 25% pracy podczas pierwszego opracowywania przypadków użycia oraz około 40% podczas wprowadzania zmian.

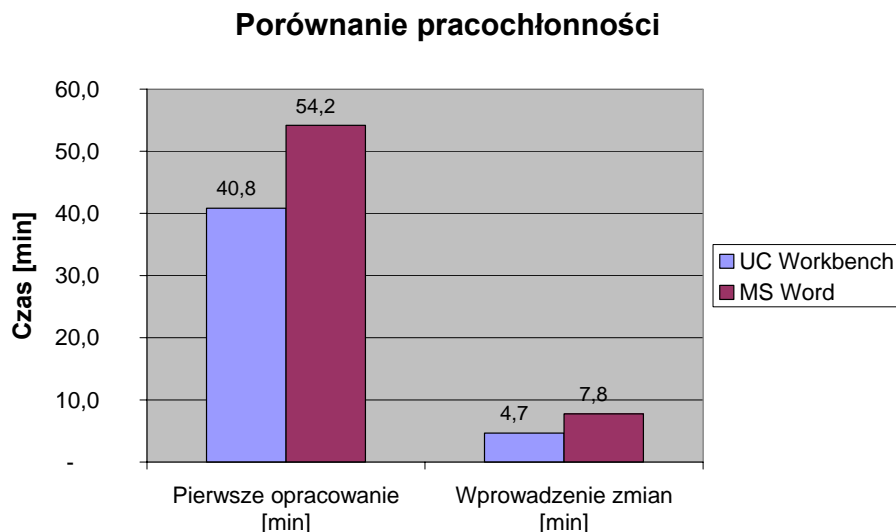


Fig. 4. Pracochłonność potrzebna do wprowadzenia przypadków użycia za pierwszym razem (po lewej) oraz wprowadzenia zmian do specyfikacji (po prawej).

5 Rozwój oprogramowania

5.1 Programować parami, czy nie programować parami?

Programowanie parami jest kluczową praktyką w XP. Para programistów wyposażona w jeden komputer jest przydzielana do zadania programistycznego. Jeden z programistów pisze kod, natomiast drugi śledzi jego pracę, zadaje pytania i proponuje przypadki testowe, tak więc zapewnia to tzw. ciągły przegląd. Inne podejście do programowania zespołowego to programowanie Side-by-Side (SbS), które zostało zaproponowane przez Cockburn'a [9]. W tym podejściu pojedyncze zadanie jest rozwiązywane przez dwóch programistów, lecz każdy z nich posiada osobny komputer.

Wyniki badań eksperymentalnych dotyczących wydajności programowania parami oscylują pomiędzy optymistycznymi (przyspieszenie rzędu 40% czasu i narzut 20% pracochłonności przy porównaniu z programowaniem indywidualnym [23, 28, 29]), a całkiem pesymistycznymi (przyspieszenie rzędu 20%, narzut 60% [20]). Niestety, nie ma opublikowanych żadnych aktualnych wyników eksperymentów dotyczących szybkości programowania SbS.

Ostatnie eksperymenty przeprowadzone na Politechnice Poznańskiej [22] pokazują, że klasyczne programowanie parami jest mniej wydajne niż programowanie SbS. Niecałe 30 studentów pracowało przez 6 dni w kontrolowanym środowisku.

Budowali aplikację internetową zarządzającą przyjmowaniem artykułów konferencyjnych i przeglądem recenzji. Studenci byli podzieleni na trzy grupy: pary SbS, pary XP oraz indywidualistów. Okazało się, iż pary SbS były o 13% szybsze od par XP i 39% szybsze od indywidualistów. W konsekwencji pracowalność związana z programowaniem SbS była o 26% niższa niż w przypadku par XP oraz o 22% wyższa od indywidualistów. Ten eksperyment pokazuje, że programowanie SbS jest interesującą alternatywą do programowania parami znanego z XP oraz indywidualnego programowania.

Inny eksperyment przeprowadzony na Politechnice Poznańskiej w 2005 roku potwierdza tę obserwację. W eksperymencie wzięło udział 44 ochotników, studentów drugiego roku informatyki. Zaliczyli oni już różne przedmioty dotyczące programowania (m.in. C++ i Java) – w sumie ponad 400 godzin zajęć. Również w tym eksperymencie zdecydowaliśmy się na kontrolowane środowisko (praca jedynie na uczelni pod okiem prowadzącego). Część studentów pracowało w parach XP, inni w parach SbS. Otrzymali oni dwa zadania oszacowane na 9 godzin pracy i pracowali zgodnie ze zdefiniowanym wcześniej procesem. W rezultacie programowanie SbS okazało się znowu szybsze od programowania parami z XP o około 16%-18%.

Interesujący jest fakt, iż uczestnikom eksperymentu bardziej odpowiadało programowanie zespołowe (55%) niż indywidualne (40%). Co więcej, 70% wszystkich osób wolało podejście SbS od XP.

W XPrince Programiści mogą wybrać pomiędzy programowaniem indywidualnym, SbS lub parami z XP. Jeżeli wybiorą jednak programowanie indywidualne, przydzielany jest recenzent w celu sprawdzenia jakości pracy (jest nim inny Programista).

5.2 Refaktoryzacja kodu

Refaktoryzacja jest jedną z kluczowych technik wspomagających pielęgnację oprogramowania [3]. Polega ona na wprowadzaniu zmian do kodu źródłowego, polepszających czytelność i dostosowujących go do zmiennych wymagań, zapewniając przy tym takie same obserwowalne zachowanie (to pozwala na użycie w tym celu zwykłych testów regresyjnych) [12].

Projekty „zwinne” przez większość czasu znajdują się w fazie pielęgnacji, gdyż podlegają ciągłej ewolucji. Refaktoryzacja (lub inna forma pielęgnacji kodu) odgrywa w tej fazie dużą rolę.

Niestety, refaktoryzacja jest zarazem techniką kosztowną i podatną na błędy. Zmiany w kodzie często wprowadzają błędy i nieprzewidziane efekty uboczne, które zmieniają zachowanie programu. Przeciwdziałanie temu wymaga dodatkowego nakładu pracy, sięgającego nawet 80% całego kosztu realizacji projektu [6]. Opłaca się jednak wykonać tą pracę w przypadku kolejnych zmian, gdyż według niektórych autorów [25], koszt związany z refaktoryzacją zwraca się po wprowadzaniu szóstej modyfikacji do systemu. Eksperyment przeprowadzony na małą skalę na Politechnice Poznańskiej ze studentami czwartego roku Inżynierii Oprogramowania pokazał, iż narzut związany z refaktoryzacją w kolejnych przyrostach zmalał z 75% do 7% już w trzecim cyklu rozwojowym, w stosunku do podobnego procesu przyrostowego prowadzonego bez refaktoryzacji kodu [26]. Tak więc utrzymywanie dyscypliny w

zakresie refaktoryzacji kodu poprawia jakość kodu i opłaca się w przypadku projektów z kilkoma przyrostami funkcjonalnymi lub cyklami pielęgnacyjnymi.

W projektach XPrince projekty posiadają wiele przyrostów, tak więc refaktoryzacja i środowiska programistyczne ją wspierające (np. Eclipse IDE) odgrywają w nich istotną rolę..

5.3 Integracja ponownego użycia kodu i kodowanie poprzedzone testowaniem

Fakt, iż ponowne użycie kodu zmniejsza koszty wytwarzania oprogramowania i zwiększa niezawodność, jest powszechnie znany. Przykładowo, Toshiba odnotowała dzięki temu zmniejszenie liczby defektów o 20-30%, a Hewlett-Packard nawet o 76% [11]. Główny problem dotyczący ponownego użycia kodu polega na znalezieniu odpowiedniego fragmentu kodu, który może zostać użyty w danej sytuacji. Jest to zadanie nietrywialne, szczególnie gdy rozmiar repozytorium i liczba zaangażowanych osób są bardzo duże (co jest warunkiem, aby ponowne użycie się opłacało). Tak trudnego zadania nie da się rozwiązać bez wsparcia zarówno dobrze zorganizowanych procesów, jak również specjalnych narzędzi.

Jednym z najbardziej interesujących podejść do wyszukiwania komponentów jest wyszukiwanie na podstawie zachowania (podejście behawioralne) [2, 24]. Zachowanie klasy lub metody jest definiowane za pomocą małego programu pokazującego wejście i oczekiwane wyjście. Ten pomysł został zaproponowany przez Podgurskiego [24]. Niestety, nie jest on szeroko stosowany w praktyce, gdyż panuje powszechne przekonanie o trudności zdefiniowania wejścia i wyjścia, oraz że będzie szybciej samemu napisać dany fragment kodu, zamiast szukać go w repozytorium (podejście behawioralne jest zorientowane na relatywnie małe fragmenty kodu, takie jak funkcje lub klasy).

W XP (oraz XPrince również) kodowanie jest poprzedzone przez przygotowanie przypadków testowych (nazywa się to kodowaniem poprzedzonym testowaniem, *ang. test-first coding*). W celu wspierania ponownego użycia i kodowania przed testowaniem stworzyliśmy narzędzie, które pobiera przypadki testowe stworzone z wykorzystaniem JUnit'a i wykorzystuje podejście behawioralne do wyszukania komponentu z repozytorium, który potencjalnie pasuje do dostarczonych przypadków testowych. W przypadku odnalezienia kilku komponentów, Programista może sprawdzić dokładniej, czy dany kod spełnia jego wymagania. Jeżeli nic nie zostało odnalezione, może on przystąpić do programowania, tak samo, jakby robił to w typowym kodowaniu poprzedzonym testowaniem.

Zaproponowana technika nie ma na celu zastąpienie innych metod przeszukiwania repozytoriów. Jest raczej rozwiązaniem komplementarnym, zaprojektowanym do wyszukiwania małych jednostek kodu, dla których powszechnie używane techniki bazujące na tekście lub klasyfikacji aspektowej mogą okazać się niewystarczające.

W celu oceny proponowanego narzędzia, przeprowadziliśmy prosty test. Dziewięciu programistów (studenci 5-tego roku) otrzymali opis 10-ciu stosunkowo prostych jednostek programowych (opisy były przedstawione w języku naturalnym). Ich zadaniem było zaproponowanie przypadków testowych, które by wykorzystali w celu znalezienia jednostki kodu w repozytorium. W 9 na 10 przypadków programiści przygotowali przypadki testowe, które dały poprawne rezultaty. Jedna klasa jednak

okazała się problematyczna. Jej zadaniem było przedstawienie łańcuchów znaków, które mogą pasować do wyrażeń regularnych (4 z 9-ciu programistów poczyniło błędne założenia).

WNIOSEK!!!

6 Zakończenie

Poprzez połączenie różnych metodyk i wsparcia odpowiednich narzędzi można uzyskać równowagę pomiędzy zwinnością i dyscypliną. Rozwiązania zaproponowane w artykule wynika z siedmiu lat doświadczeń w prowadzeniu Studia Rozwoju Oprogramowania na Politechnice Poznańskiej. Opisana metodyka (XPrince) oraz pierwsza wersja narzędzia UC Workbench zostały sprawdzone w praktyce – zostały wykorzystane w komercyjnym projekcie realizowanym na zlecenie organizacji rządowej.

Podziękowania

Jesteśmy wdzięczni firmie PB Polsoft z Poznania, za uwagi pochodzące z realnych projektów przemysłowych prowadzonych zgodnie z XPrince. Szczególnie dziękujemy Grzegorzowi Leopoldowi, który stworzył taką sposobność.

Badania przedstawione w tej pracy zostały sfinansowane z grantu KBN nr 4 T11F 00123 (lata 2002-2005).

Bibliografia

- [1] Adolph, S., Bramble, P., Cockburn, A., Pols, A., *Patterns for Effective Use Cases*, Addison-Wesley, 2002.
- [2] Atkinson S., *Examining behavioural retrieval*, WISR8, Ohio State University, 1997
- [3] Beck, K., *Extreme Programming Explained. Embrace Change*, Addison-Wesley, Boston, 2000.
- [4] Blanchard, K., Zigarmi D., Zigarmi P., *Leadership and the One Minute Manager*, 1985.
- [5] Boehm, B., Turner, R., *Balancing Agility and Discipline. A Guide for Perplexed*, Addison-Wesley, Boston, 2004.
- [6] Bossi P., *Repo Margining System*. <http://www.communications.xplabs.com/lab2001-1.html>, visited in 2004..
- [7] Brooks, F., *A Mythical Man-Month*, Addison-Wesley, Boston 1995.
- [8] Cockburn, A., *Writing Effective Use Cases*, Addison-Wesley, Boston, 2000.
- [9] Cockburn, A., *Crystal Clear. A Human-Powered Methodology for Small Teams*. Addison-Wesley, Boston, 2005.
- [10] Covey, S., *The Seven Habits of Highly Effective People*, Simon and Schuster, London, 1992.
- [11] Ezran M., Morisio M., Tully C., *Practical Software Reuse*, Springer, 2002.
- [12] Fowler M., *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, Boston, 1997.

14 **Jerzy Nawrocki**, Łukasz Olek, Michał Jasiński, Bartosz Paliświat, Bartosz Walter, Błażej Pietrzak, Piotr Godek

- [13] Humphrey, W., *A Discipline for Software Engineering*, Addison-Wesley, Reading MA, 1995.
- [14] Humphrey, W., *Introduction to the Team Software Process*, Addison-Wesley, Reading MA, 2000.
- [15] *IEEE Guide for Information Technology – System Definition – Concept of Operations (ConOps) Document*, IEEE Std. 1362-1998.
- [16] Karner, G., *Use Case Points – Resource Estimation for Objectory Projects*, Objective Systems SF AB, 1993.
- [17] Kroll, P., Kruchten, Ph., *The Rational Unified Process Made Easy*, Addison-Wesley, Boston, 2003.
- [18] *Managing Successful Projects with PRINCE2*, TSO, London, 2004.
- [19] Nawrocki, J., Jasiński, M., Walter, B., Wojciechowski, A., *Extreme Programming Modified: Embrace Requirements Engineering Practices*, [10th IEEE Joint International Requirements Engineering Conference](#), RE'02, Essen (Germany), IEEE Press, Los Alamitos (2002) 303-310.
- [20] Nawrocki, J., Wojciechowski, A.: *Experimental Evaluation of Pair Programming*. In: Maxwell, K., Oligny, S., Kusters, R., van Veenendaal, E. (eds.): *Project Control. Satisfying the Customer. Proceedings of the 12th European Software Control and Metrics Conference ESCOM 2001*. Shaker Publishing, London (2001) 269-276.
- [21] Nawrocki, J., Olek, L., *UC Workbench – A Tool for Writing Use Cases and Generating Mockups*. In: Baumeister, H., Marchesi, M., Holcombe, M., (Eds.) *Extreme Programming and Agile Processes in Software Engineering, Lecture Notes in Computer Science 3556*, 230-234.
- [22] Nawrocki, J., Jasinski, M., Olek, L., Lange, B.: *Pair Programming vs. Side-by-Side Programming*. Proceedings of the European Software Process Improvement and Innovation Conference, *Lecture Notes in Computer Science* (in print) (2005).
- [23] Nosek, J. T., *The Case for Collaborative Programming*. *Communications of the ACM*, Volume 41, No. 3 (1998) 105–108.
- [24] Podgurski, A., Pierce, L., *Retrieving reusable software by sampling behavior*, ACM TOSEM, Volume 2 , No. 3 (1993) 286 – 303.
- [25] Stroulia E., Leitch R., K., *Understanding the Economics of Refactoring*. In: Proc. of the Fifth ICSE Workshop on Economics-Driven Software Engineering Research. Portland, 2003.
- [26] Walter B., *Analysis of Software Refactorings*. PhD dissertation, Poznań University of Technology, Poznań (Poland), 2004 (in Polish).
- [27] White, S., *Introduction to BPMN*, <http://www.bpmn.org/Documents/Introduction%20to%20BPMN.pdf>, visited in 2005.
- [28] Williams, L.: *The Collaborative Software Process*. PhD Dissertation at the Department of Computer Science, University of Utah, Salt Lake City (2000).
- [29] Williams, L. at al.: *Strengthening the Case for Pair Programming*. *IEEE Software*, Volume 17, No. 4 (2000) 19–25.