testowanie, testy jednostkowe, testy akceptacyjne, testy funkcjonalne, metodyka XP, Rational Robot, SQABasic, xUnit, jakość oprogramowania

Maciej Gabor, Jerzy Nawrocki, Bartosz Walter*

TESTOWANIE EKSTREMALNE I NARZĘDZIA XUNIT¤

Testowanie jest jednym z kluczowych elementów zapewniania jakości. Jego rola w procesie wytwarzania oprogramowania została szczególnie uwypuklona w Programowaniu Ekstremalnym. Metodyka ta kładzie nacisk zarówno na testy jednostkowe, jak i akceptacyjne. W pracy przedstawiono, bazujące na środowisku *xUnit*, podejście do testowania jednostkowego. W zakresie testów akceptacyjnych za punkt odniesienia przyjmujemy *Rational Robot* i *SQABasic* oraz pokazujemy jak zawarte w tych narzędziach pomysły i mechanizmy przenieść do środowiska xUnit.

1. WSTĘP

W kontekście systemów czasu rzeczywistego niezwykle istotna jest jakość oprogramowania i jego niezawodność. Systemy te są bardzo często systemami krytycznymi ze względu na bezpieczeństwo. Nakłada to na oprogramowanie dodatkowe wymagania. Czasem, w celu osiągnięcia wysokiej niezawodności, proponuje się stosowanie metod formalnych. Niektórzy twierdzą jednak, że formalne metody specyfikacji oraz formalne dowodzenie poprawności jest zbyt teoretyczne a istniejące narzędzia niewystarczające by poradzić sobie ze złożonym oprogramowaniem. W praktyce stosuje się głównie przeglądy, inspekcje oraz testowanie oprogramowania.

Poglądy na testowanie ewoluowały wraz z rozwojem informatyki. W latach 70-tych i 80-tych dużą nadzieję wiązano z podejściem matematycznym i formalnym dowodzeniem własności programów. Wielkim zwolennikiem takiego podejścia był E. Dijkstra. Zwrócił on uwagę, że testowanie może jedynie pokazać błędy tkwiące w oprogramowaniu, ale nie może wykazać ich braku. W latach 90-tych z jednej strony nastąpiło pewne zniechęcenie w stosunku do metod formalnych, z drugiej znacznie

* Politechnika Poznańska, ul. Piotrowo 3A, 60-965 Poznań, e-mail: gabi@arni.pdi.net, Jerzy.Nawrocki@put.poznan.pl, Bartek.Walter@man.poznan.pl

¤ Praca wykonana w ramach grantu 91-378/02-BW

dojrzały metody i narzędzia testowania. Pojawiły się takie metodyki, jak Programowanie Ekstremalne [1, 2, 13] (w skrócie *XP* od ang. eXtreme Programming), w których testowanie odgrywa kluczową rolę. Kent Beck, jeden z głównych twórców *XP* twierdzi wręcz, że „*właściwości oprogramowania, których nie można sprawdzić przez automatyczne testy, po prostu nie istnieją*" [1]. W *XP* kładzie się szczególny nacisk na automatyczne testowanie, jako sposób zapewniania jakości. Metodyka ta, należąca do grupy *metodyk lekkich*, zyskuje coraz większą popularność dzięki zorientowaniu na potrzeby klienta, tolerowaniu zmiennych wymagań i sposobowi zapewniania jakości. Programowanie Ekstremalne proponuje szereg interesujących praktyk: grę planistyczną, dowartościowanie roli klienta, krótkie wydania, programowanie parami, opracowywanie przypadków testowych przed kodowaniem, itp.

W zakresie testowania automatycznego pojawiło się wiele interesujących narzędzi, zarówno darmowych, jak i komercyjnych. Wśród narzędzi darmowych coraz większą popularność zyskują narzędzia serii *xUnit* (JUnit [11], CPPUnit, itp.). Wśród narzędzi komercyjnych ciekawą propozycją są narzędzia do testowania wchodzące w skład pakietu *Rational Suite*. Umożliwiają one tworzenie skryptów testowych na podstawie zapisu sesji (*Rational Robot* [10], mechanizm *capture&play*), czy też automatyczne generowanie przypadków testowych (*Rational Test Factory*). Do głównych wad pakietu *Rational Suite* można zaliczyć: możliwość automatyzowania testów dopiero, gdy zostaną stworzone interfejsy użytkownika, brak wsparcia dla innych systemów operacyjnych poza Windows oraz problemy z testowaniem niestandardowych, dla tego systemu operacyjnego, obiektów. W tym kontekście wydaje się uzasadnione podjęcie próby przeniesienia przynajmniej części mechanizmów oferowanych przez *Rational Robot* na grunt *xUnit*. Z punktu widzenia systemów czasu rzeczywistego szczególnie istotne są te mechanizmy, które dotyczą testowania wydajności.

W artykule przedstawiono próbę rozszerzenia narzędzi xUnit o wybrane mechanizmy *Rational Robot*, w tym o mechanizmy ułatwiające testowanie wydajności. Krótki opis wybranych praktyk dotyczących Programowania Ekstremalnego przedstawiony został w rozdziale 2. Metodyka *XP* zaleca automatyzację wszystkich testów tworzonych podczas pracy nad rozwojem oprogramowania. Opisany w rozdziale 3. *Rational Robot* oraz język *SQABasic* oferują obszerną funkcjonalność w zakresie automatyzacji testów akceptacyjnych. Podejście do testowania jednostkowego, bazujące na narzędziach *xUnit*, zostało przedstawione w rozdziale 4. W rozdziale 5. zaproponowana została biblioteka VPoints wzbogacająca narzędzia *xUnit* o wybrane mechanizmy *Rational Robot*. Przykłady użycia biblioteki Vpoints zamieszczono w rozdziale 6.

2. ZAPEWNIANIE JAKOŚCI OPROGRAMOWANIA ZGODNIE Z ZAŁOŻENIAMI METODYKI XP

Programowanie Ekstremalne, jest nową, zyskująca popularność metodyką tworzenia oprogramowania. Tom de Marco, twórca analizy strukturalnej, określa *XP* jako

przełom w inżynierii oprogramowania [3]. Poniżej przedstawione zostały wybrane praktyki *XP* [5, 6, 8] bezpośrednio związane z jakością oprogramowania:

- **Klient na miejscu.** Klient ma pełną kontrolę nad procesem rozwoju oprogramowania. Ciągła obecność klienta stwarza programistom szansę szybkiego rozwiązywania napotykanych problemów i wyjaśniania wątpliwości dotyczących cech funkcjonalnych budowanego systemu. Klient aktywnie uczestniczy między innymi w tworzeniu testów akceptacyjnych.
- **Częste wydania.** Kolejne wersje systemu buduje się na zasadzie przyrostów i dostarcza klientowi w możliwie krótkich cyklach (1-2 miesięce). W ten sposób klient ma szansę lepszego poznania swoich potrzeb i wcześniej może skorygować swoje wymagania wobec systemu.
- **Programowanie parami.** Para programistów pracuje wspólnie, przy jednym komputerze, nad wykonaniem określonego zadania. Taki sposób pracy oznacza *de facto* ciągłe przeglądy kodu (w *XP* zrezygnowano z formalnych przeglądów artefaktów).
- **Ciągła integracja.** Kod tworzony przez parę programistów powinien być integrowany z całością systemu tak często, jak jest to tylko możliwe. Przy każdej integracji dodawana jest nowa funkcjonalność do systemu.
- **Refaktoryzacja.** Refaktoryzacja oznacza przebudowę kodu. Tworzony kod oraz struktura systemu ulegają iteracyjnemu poprawianiu i ulepszaniu przy zachowaniu zbudowanej funkcjonalności. Celem takiego działania jest uzyskanie przejrzystego, komunikatywnego kodu, pozbywanie się powielonego kodu oraz zmniejszenie ryzyka pojawienia się kolizji w trakcie procesu integracji.
- **Testowanie**. Proces kodowania zostaje zawsze poprzedzony przygotowaniem przypadków testowych. Taki sposób pracy powoduje, że system jest w pełni zrozumiały dla programistów, czego efektem jest stworzenie prostszej implementacji. *XP* kładzie nacisk zarówno na testy jednostkowe, jak i akceptacyjne. Ważnym wymaganiem jest automatyzacja wszystkich testów. Konieczność taka wynika z częstych integracji (każda integracja systemu kończy się wykonaniem wszystkich istniejących testów) oraz licznych refaktoryzacji.

Opisane podejście pozwala utrzymać wysoką jakość wytwarzanego oprogramowania przez cały okres prowadzonych nad nim prac.

3. TESTOWANIE JEDNOSTKOWE Z WYKORZYSTANIEM NARZĘDZI XUNIT

xUnit jest grupą narzędzi automatyzujących proces testowania jednostkowego. Narzędzia te szeroko polecają twórcy metodyki *XP*. Historycznie pierwszym narzędziem był napisany przez Kenta Backa *SUnit* [12] (dla języka Smalltalk). Podejście *xUnit* szybko zyskało popularność. Obecnie istnieją liczne jego implementacje dla większości znanych języków programowania (*JUnit*, *CPPUnit*, *PHPUnit*, *PerlUnit*, itd.). Większość

wersji zawiera prostą aplikację okienkową, która pozwala zarządzać procesem testowania (wybór przypadków testowych, wyświetlanie postępu, prezentacja ostatecznych wyników).

xUnit dostarcza ram do implementowania przypadków testowych w tym samym języku programowania, w którym tworzymy rozwijany system. Jednocześnie w prosty sposób możemy grupować i uruchamiać testy. Poniżej przedstawiamy sposób pracy z narzędziem.

Dla każdego obiektu, który chcemy przetestować budujemy tzw. *TestCase*, czyli oddzielną klasę grupującą wszystkie testy związane z rozważanym obiektem. Stworzenie pojedynczego testu polega na umieszczeniu w tej klasie nowej metody. Jej nazwę rozpoczynamy frazą *test*, na przykład `testCreation`, `testConnection`, itd. Następnie ze stworzonych metod budujemy zestawy testów (*TestSuite*). W zaawansowanych językach programowania (np. *Java*) zestawy takie mogą być tworzone automatycznie poprzez wybranie metod, których nazwy rozpoczynają się frazą `test`. Pracując z prostszymi językami programowania konieczne jest ręczne dodawanie testów do zestawów. Następnie uruchamiamy proces testowania. Wywołanie każdego testu poprzedzane jest automatycznym wykonaniem standardowej metody inicjalizującej (*setUp*), natomiast zakończenie testu powoduje zawsze wykonanie metody czyszczącej (*tearDown*). Metody te można pokrywać w celu odpowiednio tworzenia i niszczenia obiektów niezbędnych do wykonania testów. Podejście takie jest konieczne, by poszczególne testy były wykonywane niezależnie. Poniżej zamieszczamy prosty przykład (w języku Java) ilustrujący test metody `size` klasy `java.util.vector`.

```
public class MyTestCase extends TestCase {
     private Vector v;
     //setUp tworzy potrzebne obiekty
     public void setUp() {
       v=new Vector();
       v.add("String1");
       v.add("String2");
     }
     //tearDown niszczy wszystkie wykorzystywane obiekty
     public void tearDown() {
       v=null;
     }

     //właściwa procedura testująca
     public void testCapacity {
       assertTrue(v.size()==2);
       v.clear();
       assertTrue(v.size()==0);
     }
}
```

Zaletami narzędzi *xUnit* są: implementacje dla wszystkich popularnych języków programowania, niezależność od wykorzystywanego systemu operacyjnego, rozpowszechnianie na zasadzie *OpenSource* oraz ścisłe powiązanie testów z kodem budowanego systemu.

Wadą jest tu brak gotowych bibliotek funkcji, z które ułatwiłyby budowanie testów oraz trudności z testowaniem graficznych interfejsów użytkownika.

4. TESTY AKCEPTACYJNE Z WYKORZYSTANIEM RATIONAL ROBOT I JĘZYKA SQABASIC

Rational Robot [10] jest narzędziem automatyzacji procesu testowania aplikacji typu klient/serwer oraz aplikacji internetowych działających pod kontrolą systemu Microsoft Windows (95, 98, NT 4.0, 2000). Robot pozwala tworzyć i odtwarzać skrypty testujące. Tworzenie testu polega na uruchomieniu aplikacji, którą chcemy poddać testowaniu i pracy z nią. *Rational Robot* zapamiętuje wszystkie wykonywane operacje i zapisuje je w postaci skryptu (w języku *SQABasic*). Testowanie odbywa się zatem poprzez interfejsy użytkownika. Robot identyfikuje obiekty głównie na podstawie ich nazw, a nie tylko umiejscowienia na ekranie. W dowolnym momencie możemy w teście osadzać tzw. punkty weryfikacji, które pozwalają skonfrontować stan aplikacji z naszymi oczekiwaniami. Punkt weryfikacji jest wywołaniem funkcji sprawdzającej stan konkretnego obiektu (odpowiednik metody `assert` w xUnit). Poniżej zostały krótko przedstawione wybrane rodzaje punktów weryfikacji:

- *File Comparison* – porównuje zawartość dwóch wskazanych plików dyskowych.
- *File Existence* – sprawdza istnienie pliku dyskowego.
- *Alphanumeric* – pozwala sprawdzić zawartość standardowych elementów sterujących takich jak listy rozwijalne, listy wyboru, pola edycyjne itp.
- *Clipboard* – weryfikuje poprawność danych umieszczonych w schowku systemu.
- *Menu* – pozwala zweryfikować układ menu dla danego okna aplikacji.
- *Object Properties* – sprawdza właściwości standardowych obiektów Windows (np. nazwę okna dialogowego, aktywność obiektów itp.)
- *Object Data* – sprawdza dane umieszczone w standardowych obiektach Windows.
- *Window Existence* – sprawdza istnienie określonego okna aplikacji.

Dla każdego punktu weryfikacji możemy określić dodatkowe właściwości, np. czas oczekiwania tj. maksymalny czas, po którym aplikacja powinna przejść do założonego stanu. Wszystkie testy można również tworzyć bezpośrednio w języku *SQABasic*.

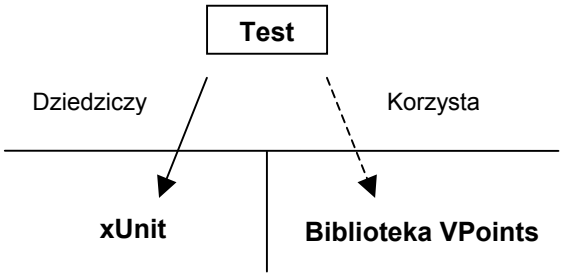
Niewątpliwą zaletą narzędzia *Rational Robot* jest łatwość zapisywania przypadków testowych oraz ogromna funkcjonalność. Nie jest tu konieczna znajomość języków programowania. Robot jest szczególnie przydatny do automatyzacji testów akceptacyjnych, które bardzo często mają postać scenariuszy użycia (sformułowanych przez klienta). Robot doskonale współpracuje z innymi narzędziami dostarczanymi w ramach pakietu *Rational Suite* (np. *Rational TestFactory*, czy *Rational Quantify*).

Problemem jest testowanie obiektów, które nie są standardowymi obiektami Windows. Narzędzie oferuje w takim przypadku kilka sposobów opisu nieznanych obiektów, ale w niektórych przypadkach sposoby te mogą okazać się niewystarczające.

5. IMPLEMENTACJA WYBRANYCH MECHANIZMÓW JĘZYKA SQABASIC W ŚRODOWSKU XUNIT

Narzędzia serii xUnit zyskują w ostatnim czasie coraz większą popularność. Nie dostarczają one jednak żadnych gotowych bibliotek funkcji, których wykorzystanie znacznie uprościłoby proces implementowania przypadków testowych, skróciłoby czas kodowania oraz uchroniłoby programistów od konieczności powielania kodu testującego. Poniżej proponujemy wzbogacić narzędzia xUnit o bibliotekę VPoints realizującą wybrane rodzaje punktów weryfikacji zaczerpnięte z Rational Robota. W następnym rozdziale przedstawimy przykłady użycia tej biblioteki.

Sposób korzystania z narzędzi xUnit i biblioteki VPoints przedstawiono na Rys.1. xUnit jest wykorzystywany do zarządzania przypadkami testowymi, do ich definiowania i wykonywania, natomiast VPoints dostarcza metody ułatwiające programowanie przypadków testowych.

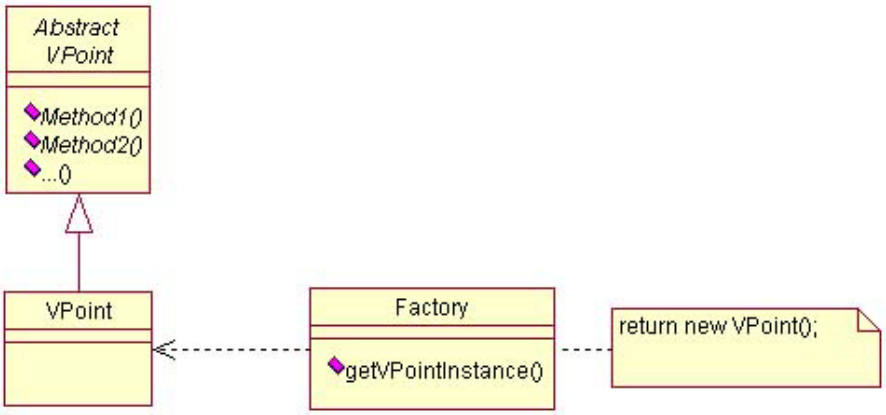


Rys. 1. Sposób korzystania z xUnit i biblioteki VPoints.
Fig 1. Usage of xUnit and the Vpoints library.

Biblioteka VPoints zrealizowana jest w następujący sposób (Rys.2.):

- Każdy rodzaj punktu weryfikacji zrealizowany jest jako oddzielna klasa (lub zestaw klas).
- Programista nie jest odpowiedzialny za prawidłowe inicjalizowanie obiektów koniecznych do wykorzystywania określonego punktu weryfikacji. Proponujemy tu zastosowanie wzorca projektowego *Factory Method* [4]. Programista wywołuje jedną z metod klasy *Factory*, która jest odpowiedzialna za przekazanie mu gotowego do wykorzystywania obiektu.

- Błąd wykonania testu zgłaszany jest przez zastosowany obiekt punktu weryfikacji. Programista nie musi wykorzystywać w tym celu żadnych dodatkowych struktur ani warunków.



Rys. 2. Koncepcja punktu weryfikacji dla xUnit.
Fig 2. Verification Point for xUnit.

W następnym rozdziale przedstawiamy przykładowe realizacje (przygotowane dla środowiska JUnit) wybranych mechanizmów narzędzia Rational Robot. Podajemy również fragmenty kodu obrazujące wykorzystanie tych mechanizmów w testach.

6. PRZYKŁADY UŻYCIA BIBLIOTEKI VPOINTS

Przykład pierwszy ilustruje uzupełnienie JUnit o porównywanie binarne plików dyskowych. Tworzymy interfejs, przez który programista będzie odwoływał się do funkcji porównującej pliki. Niech interfejs ten nazywa się `IFileComparator` i definiuje jedną metodę (`compare(File f1, File f2)`) porównującą pliki w sposób binarny. Argumentami tej metody niech będą obiekty reprezentujące wybrane pliki dyskowe. Następnie deklarujemy klasę implementującą ten interfejs. Nazwijmy ją `FileComparatorImpl`. Klasa ta zawierać będzie konkretną implementację metody `compare`. Dobudowaną funkcjonalność moglibyśmy wykorzystać w następujący sposób:

```
public class MyTest extends TestCase {
     public void testCompareFiles() {
             //wykonanie metod testowanej aplikacji, które w
             //efekcie generują plik
       aplication_operations();
             //pobranie obiektu z klasy Factory
       IFileComparator fc = Factory.createFileComparator();
```

```
                //właściwe porównanie plików; argumentami są
                //referencje na nowo utworzony plik oraz plik
                //przygotowany dla testu
          fc.compare(f1, f2);
     }
}
```

Kolejny przykład dotyczy sprawdzenia, czy dana operacja zakończy się w z góry założonym czasie. Interfejs punktu weryfikacji, który zamierzamy wykorzystać nazywa się `IWatchDog`. Udostępnia on metody `start(int time)` oraz `stop()`. Wartość parametru `time` ogranicza od góry czas trwania testowanej operacji. Klasa `WatchDogImpl` implementuje interfejs `IWatchDog`. Działanie metody `start` polega na utworzeniu nowego wątku, który jest „usypiany" na czas określony parametrem. Po upłynięciu tego czasu system operacyjny budzi wątek, który zgłasza błąd wykonania testu. Metoda `stop` kończy działanie wątku odmierzającego czas. Opisany mechanizm możemy zastosować w następujący sposób:

```
public class MyTest extends TestCase {
     public void testOperationDuration() {

               //pobranie obiektu ograniczającego czas trwania
               //operacji z klasy Factory
          IWatchDog wd = Factory.createWatchDog();

          wd.start(500);//czas trwania testowanej operacji nie
                        //powinien przekroczyć 0.5 sekundy
          operations(); //wykonanie testowanych operacji

          wd.stop();  //jeśli czas trwania testowanej operacji
                      //jest krótszy niż 0.5 sekundy metoda stop
                      //zapobiegnie zgłoszeniu błędu testu
     }
}
```

Jeśli wykonanie `operations()` potrwa dłużej niż 0.5 s, to zostanie zgłoszony błąd wykonania testu i będzie wyświetlona nazwa klasy oraz metody, w której nastąpił błąd (w naszym przypadku `testOperationDuration`).

Włączenie testowania maksymalnego czasu wykonania do zautomatyzowanych testów jednostkowych może w istotny sposób wspomóc pielęgnację oprogramowania. Załóżmy, że w programie znajdują się trzy moduły:

- K – Kolejka priorytetowa,
- S – Słownik,
- P – Przydział bloków pamięci.

Moduły K i S oferują możliwość dynamicznego tworzenia elementów, które będą wstawione do słownika lub kolejki. Dlatego też współdzielą one moduł P, który przydziela i zwalnia komórki pamięci zajmowane przez elementy słownika i kolejki. Moduł P przydziela pamięć na zasadzie pierwszego dopasowania (ang. *first fit*) [14]. Po pewnym czasie ktoś wprowadzając poprawkę do systemu decyduje się zmodyfikować moduł S i podzielić elementy słownika na dwie części A, B, aby w ten sposób zmniejszyć zajętość pamięci (część B w wielu elementach powtarza się i bardziej ekonomiczne może wydawać się zastąpienie jej referencją do jednego elementu zawierającego powtarzające się dane). W rezultacie modyfikacji modułu S rośnie czas wykonania modułu K, gdyż spowolnieniu ulega operacja tworzenia elementów realizowana przez moduł P i wykorzystywana również przez K (im więcej elementów tym dłużej działa algorytm pierwszego dopasowania). Jeśli nie zabezpieczymy się poprzez napisanie odpowiedniego testu wydajnościowego, to negatywny efekt takiej modyfikacji może przejść niezauważony, gdyż nie dotyczy bezpośrednio modułu, który został zmodyfikowany. Prosty test bazujący na punkcie weryfikacji typu Watchdog może być tu bardzo przydatny.

7. PODSUMOWANIE

Metodyka *XP* zaleca automatyzowanie wszystkich testów tworzonych w procesie rozwoju oprogramowania. W pracy przedstawiliśmy dwa narzędzia wspierające tworzenie testów automatycznych, xUnit do testowania jednostkowego i Rational Robot do testów akceptacyjnych, oraz pokazaliśmy jak przenieść część mechanizmów Rational Robot do środowiska xUnit. Szczególnie interesujące wydają się te mechanizmy, które ułatwiają realizację testów wydajnościowych.

Koncepcja tworzenia automatycznych testów ma swoich zwolenników [1] i przeciwników [9]. Wątpliwości budzą głównie dodatkowe nakłady związane z implementacją przypadków testowych. Obecnie, przy współpracy poznańskiej firmy Bestcom, zbieramy dane dotyczące procesu testowania zgodnego z zaleceniami Programowania Ekstremalnego. Na podstawie zgromadzonych danych zamierzamy przeprowadzić analizę opłacalności automatyzacji testów tworzonych zgodnie z *XP*. W przyszłym roku zamierzamy również wprowadzić narzędzie *xUnit* do projektów realizowanych w ramach Studia Rozwoju Oprogramowania [7] na Politechnice Poznańskiej.

BIBLIOGRAFIA

[1] Kent B., Wydajne programowanie. Mikom, Warszawa, 2002.

[2] Jeffries R., Anderson A., Hendrickson C., Extreme Programming Installed. Addison-Wesley, Boston, 2001.

[3] Beck K., Fowler M., Planning Extreme Programming. Addison-Wesley, Boston, 2001.

[4] Gamma E., Helm R., Johnson R., Vlissides J., Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[5] Nawrocki J., Walter B., Wojciechowski A., Toward Maturity Model for eXtreme Programming, 27th Euromicro Conference, Warszawa, 2001, IEEE Computer Press, Los Alamitos, 233-239.

[6] Nawrocki J., Jasiński M., Walter B., Wojciechowski A., Extreme Programming Modified: Embrace Requirements Engineering Practices, 10th Joint IEEE International Conference on Requirements Engineering, Essen, 2002, IEEE Press, Los Alamitos, 303-310.

[7] Nawrocki J., Towards educating leaders of software teams, **w**: P. Klint, J. Nawrocki (eds), Software Engineering Education Symposium SEES'98, Poznań, 1998, Scientific Publishers OWN, Poznan, 149-157.

[8] Nawrocki J., Walter B., Wojciechowski A.: Comparison of CMM Level 2 and eXtreme Programming, 7th European Conference on Software Quality, Helsinki, 2002, Lecture Notes in Computer Science 2349, 288-297.

[9] Bereza-Jarociński B., A complete Guide to Test Automation, **w**: K.Zieliński (red.), II Krajowa Konferencja Inżynierii Oprogramowania, Zakopane 2000, Katedra Informatyki AGH, Kraków, 293-302.

[10] Rational Software Corporation, Using Rational Robot, 2001

[11] JUnit, Testing Resources for Extreme Programming, www.junit.org, 05.2002

[12] Beck K., Simple Smalltalk Testing: With Patterns, www.xprogramming.com/testfram.htm, 05.2002

[13] Extreme Programming: A gentle introduction, www.extremeprogramming.org, 05.2002

[14] Tanenbaum A.S., Operating Systems Design and Implementation, Prentice-Hall, Englewood Cliffs, 1987.

SOFTWARE TESTING IN EXTREME PROGRAMMING

Testing is one of key elements of quality assurance. Its role in software development process is strongly emphasized in the Extreme Programming methodology. XP considers both unit and acceptance tests very important. In this paper we discuss xUnit-based approach to unit testing. We show how to augment xUnit functionality with some features of Rational Robot and SQABasic.