TRUDNOŚCI W IMPLEMENTACJI WZORCÓW PROJEKTOWYCH W MAŁYCH ZESPOŁACH PROGRAMISTYCZNYCH

Rafał Wojszczyk[1]
Piotr Ratuszniak[2]

Streszczenie

Na rynku IT występuje wiele małych przedsiębiorstw tworzących własne, autorskie oprogramowanie lub wykonujące usługi dla firm trzecich. Małe zespoły często pracują według tzw. zwinnych metodyk wytwórczych, w których ograniczona jest ilość dokumentacji projektowej, a większość procedur znacznie uproszczona. Jednakże małe zespoły nie rezygnują ze stosowania dobrych praktyk, w tym wzorców projektowych.

Celem artykułu jest przybliżenie wybranych trudności, z którymi borykają się małe zespoły programistyczne przy implementacji wzorców projektowych, oraz przedstawienie autorskiej metody wspomagającej wzrost jakości implementacji wzorców projektowych.

Słowa kluczowe: manifest Agile, SCRUM, wytwarzanie oprogramowania, wzorce projektowe.

1. Wprowadzenie

Badania oraz rozwój w dziedzinie inżynierii oprogramowaniu mają na celu ulepszanie wszystkiego co związane z oprogramowaniem. Wzorce projektowe są tylko jednym z wielu rozwiązań, które pozwalają ulepszyć oprogramowanie, mimo to są ważnym, modnym i pożądanym rozwiązaniem przez społeczność branży IT.

Wystąpienie wzorca projektowego w kodzie programu nie jest równoznaczne z poprawną implementacją, a tym bardziej nie oznacza to, że wzorzec został właściwie dobrany. Z takimi trudnościami spotykają się zarówno bardziej, jak też mniej doświadczeni programiści. Klasyczne modele jakości oprogramowania nie znajdują zastosowania do implementacji wzorców projektowych. Sytuacja wygląda podobnie w przypadku popularnych metryk programowania obiektowego. Trudność potęguje

[1, 2] Politechnika Koszalińska, Wydział Elektroniki i Informatyki/Koszalin University of Technology, Department of Electronics and Computer Science.

DOI: 10.19253/reme.2017.02.016

pominięcie wzorców w sztywnych metodykach wytwórczych, natomiast w zwinnych metodykach przeszkodą jest zbyt luźne podejście do wykorzystania wzorców oraz niechęć do porzucania dotychczasowych przyzwyczajeń. Popularne narzędzia oraz środowiska programistyczne zapewniają jedynie podstawowe wsparcie dla wybranych wzorców. Ostatecznie wykorzystanie wzorców projektowych komplikuje brak standaryzacji, spowodowany w dużym stopniu różnorodnością implementacji. Istnieje zatem potrzeba opracowania nowej metody oceny jakości implementacji wzorców projektowych, która rozwiąże wymienione trudności. Metoda powinna być przede wszystkim przeznaczona do wzorców projektowych, bazować na paradygmacie programowania obiektowego, oraz powinna być kompatybilna z istniejącymi modelami jakości lub testowania oprogramowania, aby nie zabierać zbyt wiele czasu w codziennej pracy, w uzyskaniu odpowiedzi na pytanie „czy dana implementacja wzorca spełnia wymaganą jakość?".

Celem publikacji jest przedstawienie metody oceny jakości implementacji wzorców projektowych, która dostarcza wyniki niezbędne do odpowiedzi na w/w pytanie. Na podstawie w/w pytania w rozdziale drugim sformułowano problem badawczy. W rozdziale trzecim i czwartym omówiono kolejno jakość oprogramowania, wzorce projektowe oraz zwinne metodyki wytwórcze. Następnie w rozdziale piątym został szerzej przedstawiony kontekst problemu i proponowane rozwiązanie. Ostatni, szósty rozdział stanowi podsumowanie pracy.

2. Sformułowanie problemu

Dany jest zbiór wzorców projektowych, dokładniej szablonów implementacji opisanych w postaci zgodnej z paradygmatem programowania obiektowego. Dany jest zbiór metryk odwzorowujących elementy wzorców projektowych w wartości liczbowe, zgodnie z wybranymi aspektami implementacji. Jednocześnie dany jest bazodanowy program komputerowy klasy ERP, który stanowi przedmiot umowy pomiędzy odbiorcą a dostawcą, w umowie określony jest minimalny poziom jakości. Dostępny jest kod źródłowy lub pośredni tego programu, w którym zaimplementowane są wyłącznie niewizualne warstwy aplikacji (tj. logika biznesowa, dziedzina, warstwy sterujące). W kodzie programu występują implementacje wzorców projektowych, dla każdego wystąpienia wzorca znane jest miejsce wystąpienia (tzw. rdzeń wzorca), wariant oraz kontekst implementacji. Dany jest również zbiór dopuszczalnych wyników metryk. Niezależnie od punktu widzenia, dostawcy czy też odbiorcy, problem sprowadza się do odpowiedzi na pytanie: Czy dana implementacja wzorców projektowych spełnia wymagany poziom jakości w wybranych aspektach?

Odpowiedź na to pytanie jest istotna dla dostawcy oprogramowania jak również odbiorcy. Odbiorca często nie otrzymuje kodu źródłowego lub informacji o wykorzystanych rozwiązaniach w zamawianym produkcie, w takiej sytuacji nie ma możliwości sprawdzenia czy w produkcie zostały zaimplementowane pożądane rozwiązania. Zatem konieczne jest określenie w kontrakcie pewnego, wymagane-

go poziomu jakości, co pomoże odbiorcy zweryfikować czy otrzymuje zamówiony produkt oraz ułatwi egzekwowanie roszczeń w przypadku różnic. Z perspektywy dostawcy jakość może być wykorzystana przez pracowników na wielu szczeblach przedsiębiorstwa: programiści oceniają czy wykonana przez nich praca spełnia wymagany poziom jakości, kierownicy decydują czy kolejny fragment oprogramowania może zostać oddany do klienta, natomiast menadżerowie oceniają czy projekt można już zakończyć.

3. Jakość oprogramowania i wzorce projektowe

3.1. Klasyczne metryki i modele jakości oprogramowania

Badania nad wyznaczeniem jakości przedmiotów są prowadzone od wielu lat. W przypadku przedmiotów materialnych sytuacja wydaje się być prostsza, ponieważ podstawowe właściwości fizyczne łatwo zmierzyć. Oprogramowanie jest tworem niematerialnym, co przyczynia się do wzrostu trudności pomiarów oprogramowania (Pressman, 2004).

W wielu źródłach jako najpopularniejszą definicję jakości oprogramowania wymieniane jest „brak defektów w produkcie" (Kan, 2006). To stwierdzenie dotyczy wyłącznie wąskiego zakresu własności oprogramowania. Mnogość trudno mierzalnych własności oprogramowania (np. złożoność, funkcjonalność) sprawia, iż wspomniana definicja jest stanowczo niewystarczająca aby współcześnie ją zastosować. Wychodząc od aksjomatu, za (Hołodnik-Janczura, 2007): jakość nie jest wielkością bezpośrednio mierzalną oraz jest właściwością zbiorczą, należałoby wskazać odpowiednie miary, które są niezbędne do wyznaczenia jakości.

Pierwsze miary oprogramowania, wprowadzone w latach 60. XX wieku, dotyczyły łatwo mierzalnych cech np. ilość wierszy kodu LOC (ang. *line of code*). W kolejnych latach pojawiły się nowe, bardziej abstrakcyjne miary niż LOC, np.: liczba punktów funkcyjnych, złożoność cyklomatyczna (Kan, 2006).

Rosnąca popularność paradygmatu programowania obiektowego spowodowała, że miary, które powstały w drugiej połowie XX wieku dla oprogramowania strukturalnego, stały się nieefektywne. Uzupełnieniem powstającej luki było opracowanie metryk oprogramowania obiektowego (Wojszczyk, 2013), które opierają się na statycznej analizie kodu źródłowego (inaczej metryki statyczne, w odróżnieniu metryki dynamiczne opierają się na uruchomionym programie komputerowym). Metryki oprogramowania obiektowego uwzględniają m.in.: dziedziczenie, hermetyzację, polimorfizm oraz abstrakcję. Szacuje się, że istnieje około 500 metryk oprogramowania obiektowego, jednak za najpopularniejsze uważane są metryki: Shyama R. Chidambera i Chrisa F. Kemerera (w skrócie CK); *Metrics of Object Oriented Design* (w skrócie MOOD); Roberta C. Martina.

Metryki oprogramowania obiektowego są stosowane do wzorców projektowych, np. (Hernandez, 2011), jednakże należy pamiętać, że wspominane metryki są

przeznaczone do badania kodu całych programów. Wzorce projektowe to fragmenty programów komputerowych, zatem wynik metryk zastosowanych do wzorców projektowych może być zaburzony w stosunku do zastosowana zgodnego z przeznaczeniem (Khaer, 2007). Stąd wykorzystanie metryk do wzorców projektowych wiąże się z modyfikacją lub odpowiednią nadinterpretacją wyników (Wojszczyk, 2013).

Ewolucja paradygmatów programowania, zastosowania oprogramowania, dodatkowo badania nad jakością oprogramowania, doprowadziły do opracowania całościowych modeli jakości oprogramowania: CUMPRIMDA stworzone przez IBM oraz FRUPS stworzone przez Hewlett-Packard (Kan, 2006). Oba modele pozwalają na wyrażenie jakości oprogramowania w kilku kategoriach. Kategorii często nie można zmierzyć bezpośrednio, dlatego na wyrażenie każdej z nich składa się kilka charakterystyk, które są wyznaczane na podstawie wartości odpowiednich miar. Charakterystyki zawarte w w/w modelach obejmują szeroki zakres własności oprogramowania, jest to m.in.: funkcjonalność oprogramowania, łatwość instalacji, użytkowania, zarządzania oraz wydajność i niezawodność. Miary skojarzone z wymienionymi charakterystykami dotyczą walorów użytkowych oprogramowania, które nie wynikają bezpośrednio z wykorzystania wzorców projektowych, zatem nie powinny być stosowane do oceny implementacji wzorców projektowych. Obszerność wymienionych modeli jakości wiąże się z dużym nakładem pracy (kosztem), aby je skutecznie wykorzystać, na co nie mogą sobie pozwolić małe zespoły wytwórcze.

3.2. Liczbowe wyrażenie jakości

Charakterystyki w klasycznych modelach jakości i metrykach oprogramowania są abstrakcyjne, wieloznaczne i możliwe do przedstawienia wyłącznie w sposób opisowy. Metryki, które uszczegóławiają charakterystyki powinny być jednoznaczne i obiektywne (Hołodnik-Janczura, 2007) (Kobyliński, 2003), aby to osiągnąć konieczne jest wyrażenie wyników metryk w postaci liczbowej (tzw. jakość techniczna (Bielawa, 2011)). Następnie na podstawie metryk można wyznaczyć jakość charakterystyk.

Jakość wyrażona w postaci liczbowej umożliwia porównywanie produktów (w tym oprogramowania), co pozwala odróżnić od siebie pozornie jednakowe produkty (Durczak, 2011). Poprzez porównanie produktów można utworzyć listę rankingową, która jest często podstawą do podejmowania decyzji (np. o zakupie konkretnego oprogramowania użytkowego). Właściwy wybór przyczynia się do zminimalizowania skutków nietrafionej decyzji.

Badana prowadzone nad liczbowym wyrażeniem wybranych własności wzorców projektowych dotyczą głównie zliczania liczby wystąpień wzorców (Grzanek, 2008), (Rasool, 2010), (Tsantalis, 2006). Liczba wystąpień w danym fragmencie kodu programu jest nie wystarczają informacją, aby uznać to za znamiona jakości implementacji wzorców (świadczy to jedynie o danym fragmencie kodu, a nie o wzorcach w nim zawartych).

3.3. Wzorce projektowe

Christopher Alexander nazywany jest ojcem wzorców w budownictwie (Alexander, 2008), podobnie tzw. Banda Czterech, czyli E. Gamma, R. Helm, R. Johnson, J. Vlissides, autorzy (Gamma, 1995) nazywani są ojcami wzorców projektowych w programowaniu. Wzorce projektowe przedstawione w (Gamma, 1995) to za (McConnell, 2010): szkielety gotowych mechanizmów, które można wykorzystywać przy rozwiązywaniu typowych problemów projektowania i programowania. Bazując na różnych poziomach abstrakcji można wyróżnić następujący podział wzorców (Wojszczyk, 2017): architektury np. MVC, MVP), projektowe (w tym (Gamma, 1995)), implementacyjne (tzw. idiomy). Niezależnie od podziału wzorce są w pewnym sensie językiem komunikacji (Alexander, 2008) pomiędzy programistami, umożliwiają dzielenie się wiedzą, dokumentowanie projektu oraz kodu źródłowego. Wystąpienie przynajmniej nazwy wzorca w dowolnym artefakcie oprogramowania wskazuje, czego należy oczekiwać od tego artefaktu. Pozwala przewidzieć budowę, możliwości artefaktu oraz pomaga w wyszukiwaniu innych elementów skojarzonych z tym wzorcem. Znajomość i stosowanie wzorców w programowaniu jest niewątpliwą zaletą, która spotyka się z dużym uznaniem w społeczeństwie programistów. Badania (Czyczyn-Egird, 2016) wykazują, że wzorce projektowe są znacznie bardziej popularne w stosunku do wzorców architektury.

Wykorzystanie wzorców projektowych pomaga uniknąć typowych, powtarzających się błędów (Gamma, 1995). Zatem powołując się na stwierdzenie, iż o jakości oprogramowania świadczy ilość błędów, można wywnioskować, że występowanie wzorców projektowych wpływa na wyższą jakość oprogramowania (w uproszczeniu im mniej błędów tym wyższa jakość). Szczegółowego wpływu wzorców projektowych na jakość oprogramowania można doszukiwać się dla każdego wzorca z osobna, ponieważ z każdym wzorcem projektowym skojarzony jest konkretny cel zastosowania (Fowler, 2005). Realizacja każdego celu wnosi do oprogramowania pewne korzyści, związane głównie z jakością wewnętrzną oprogramowania, tj. kodu źródłowego. Cele wzorców projektowych w uogólnionym ujęciu prowadzą m.in. do (Gamma, 1995): powtórnego wykorzystania fragmentów kodu, zmniejszenia pracochłonności potrzebnej na konserwację, rozbudowę oraz modyfikację oprogramowania, dodatkowo poprawiają łatwość zrozumienia i czytelność kodu źródłowego. Wymienione cele pokrywają się z charakterystykami trzeciej części normy ISO 9126 (Kobyliński, 2003).

W procesie implementacji wzorców projektowych przyjęło się, że programista bazując na informacjach przedstawionych w literaturze powinien przy każdej implementacji danego wzorca wzbogacić wytwarzany przez siebie kod o wiele elementów związanych m.in. z nazewnictwem czy również z logiką biznesową, aby lepiej dopasować implementację wzorca do kontekstu programu. Te czynniki zwiększają złożoność i różnorodność implementacji wzorców, czego efektem jest występowanie wielu wariantów implementacji. Nieliczne metody (Nicholson, 2013) (Mehlitz,

2003) mają za cel wspomagać pracę programistów poprzez weryfikację implementacji wzorców (tj. wykazanie zgodności kodu z ogólnymi szablonami wzorców). Niestety, te metody nie są odporne na wspomnianą różnorodność implementacji oraz wymagają kompletnej dokumentacji projektowej, która najczęściej nie jest tworzona w zwinnych zespołach.

4. Zwinne metodyki wytwórcze

4.1. Manifest Agile

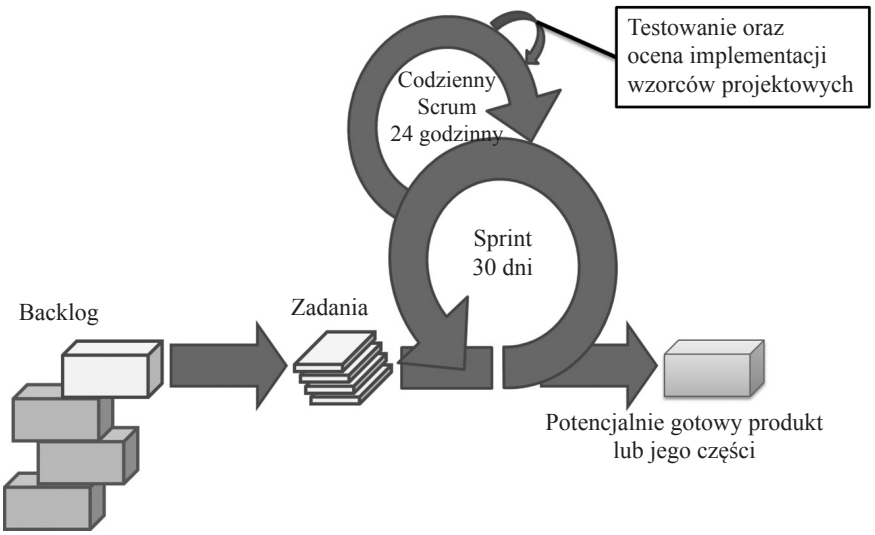
W procesie wytwarzania oprogramowania powszechnie przyjęło się występowanie kilku faz (Roman, 2015): zbieranie wymagań, projektowanie systemu, implementacja, testowanie oraz konserwacja. Wymienione fazy zostały uporządkowane oraz zdekomponowane w różnych metodykach wytwórczych. Jedną z najbardziej rozbudowanych jest metodyka RUP (od ang. *Rational Unified Process*), w której zostały szczegółowo opisane: role udziałowców, wytwarzane artefakty (np. dokumentację, testy, kod źródłowy i inne), względną pracochłonność każdej fazy oraz wytyczne (np. implementację programu można rozpocząć dopiero, gdy zostanie zaprojektowane 90% przypadków użycia). Według badań (Forrester, 2009) 21% pracowników branży IT (ang. *Information Technology*) pracuje według tzw. sztywnych metodyk, do których należy RUP.

Wzorce projektowe bardzo często wiązane są wyłącznie z etapem projektowania oprogramowania, a nie bezpośrednio z wytwarzaniem kodu źródłowego (Kerievsky, 2005). Jest to wąski punkt widzenia, ponieważ wzorce bardzo często wykorzystywane są w procesie refaktoryzacji kodu źródłowego (inaczej ulepszaniu) do wzorców projektowych (Kerievsky, 2005) (Fowler, 2011), oraz podczas naprawy i rozbudowy oprogramowania. Wymienione działania bez wątpienia są związane z wytwarzaniem bądź modyfikowaniem kodu źródłowego, podczas czego dokumentacja może być niedostępna lub przynajmniej nieaktualna po wykonanych zmianach.

Ograniczona ilość dokumentacji jest cechą charakterystyczną zwinnych metodyk wytwórczych (Martin, 2008), ponieważ jeden z postulatów manifestu Agile stanowi, że ważniejszy jest działający program od szczegółowej dokumentacji. Manifest Agile to dokument opracowany w 2001 roku przez grupę ekspertów z branży IT (m.in. Martin Fowler, Robert C. Martin, Kent Beck). Dokument ma celu popularyzację wytwarzania oprogramowania, jest to alternatywa dla sztywnych metodyk. Popularyzowane wartości zostały spisane w postaci następujących postulatów: ludzie i interakcje ponad procesy i narzędzia, współpraca z klientem ponad negocjację umów, reagowanie na zmiany ponad realizację założonego planu oraz wspomniane działające oprogramowanie ponad szczegółową dokumentację. Z badań (Forrester, 2009) wynika, że 35% pracowników branży IT pracuje według metodyk zwinnych, jest to najliczniejsza grupa w przytoczonym badaniu.

4.2. Metodyka SCRUM

Metodyka SCRUM jest najpopularniejsza pośród metodyk zwinnych. Wytwarzanie według metodyki Scrum składa się z tzw. sprintów, czyli przebiegów trwających od dwóch do czterech tygodni, realizowanych przez zespoły liczące od pięciu do dziewięciu osób. W ramach każdego przebiegu realizowany jest *backlog* sprintu, jest to lista prac nad produktem. Każdy dzień pracy członkowie zespołu Scrum rozpoczynają od krótkiego spotkania, podczas którego rozdzielane są zadania z *backlogu* sprintu oraz każdy z członków odpowiada na trzy pytania: „co robiłeś do tej pory? Co będziesz robił? Jakie miałeś problemy?". Zadania dopierane są w taki sposób, aby mogły zostać wykonane w trakcie jednego roboczodnia. Dobrą praktyką jest, gdy wykonane zadanie zakończy się wydaniem potencjalnie gotowego fragmentu oprogramowania. Następnie przygotowany fragment wymaga przeprowadzenia podstawowych testów, z którymi równoważna może być ocena implementacji wzorców projektowych, zostało to zaznaczone na rys. 1.



Rys 1. Ogólny model metodyki Scrum
Źródło: opracowanie własne

4.3. Trudności w implementacji wzorców projektowych

Badania mające na celu zidentyfikowanie trudności z jakimi borykają się małe zespoły wytwórcze w implementacji wzorców projektowych zostały przeprowadzone na podstawie obserwacji z perspektywy Scrum Mastera (osoba odpowiedzialna za zgodność z zasadami metodyki Scrum), trwającej łącznie 3 lata w dwóch różnych zespołach oraz na podstawie wywiadów przeprowadzonych z niezależnymi członka-

mi różnych zespołów. Wnioski wyciągnięte z badań uwidoczniły następujące trudności:

— decyzje o rozwiązaniu napotkanego problemu programistycznego poprzez implementację wzorca projektowego są podejmowane na szybko podczas porannych spotkań, następnie praca nad implementacją jest oddelegowana do jednego z członków zespołu (Wojszczyk, 2016);
— jedna osoba odpowiada za wiele różnych zagadnień, zatem jej umiejętności mogą być niewystarczające w zależności od powierzonego zadania (Mazur, 2010), a umiejętności w zakresie implementacji wzorców mogą być również ograniczone;
— programista chcąc poznać lub odświeżyć informacje o danym wzorcu projektowym zagląda do przykładowego kodu przedstawionego w jednym z katalogów, np. (Gamma, 1995), a następnie przystępuje do implementacji tego wzorca, bez skrupulatnego przemyślenia lub konsultacji z innymi członkami zespołu (Kerievsky, 2005);
— członkowie zespołu wytwórczego wybierają rozwiązania według swoich przyzwyczajeń oraz posiadanego stanu wiedzy, zamiast poznać nowe rozwiązania. Wyrażają szczególną niechęć do przyswajania wiedzy o innych paradygmatach lub językach programowania niż wykorzystywany przez nich (Wojszczyk, 2016);
— pozostawanie ciągle przy znanych rozwiązaniach oraz ograniczona ilość dokumentacji prowadzi do uwsteczniania się członków zespołu w zakresie implementacji rzadziej stosowanych wzorców projektowych oraz trudności w rozumieniu kompletnej dokumentacji projektowej, w tym zapominania elementów notacji UML.

Skutkiem wymienionych sytuacji są różne usterki w oprogramowaniu, a w tym w implementacji wzorców projektowych oraz tzw. pozorna implementacja wzorców. Oprogramowanie zawierające pozornie zaimplementowane wzorce może działać prawidłowo oraz przechodzić wymagane testy, jednakże implementacja danego wzorca będzie niezgodna z zaleceniami i nie będzie spełniała postawionego celu. Problemy wynikające z tego faktu będą najbardziej zauważalne dopiero po pewnym czasie, np. przy próbie rozbudowy lub modyfikacji danego fragmentu oprogramowania.

Istniejące metody zliczania wystąpień i weryfikacji implementacji wzorców projektowych nie sprawdzą się w rozwiązaniu wymienionych trudności. Jest to spowodowane w dużej mierze brakiem kompletnej dokumentacji w małych zespołach wytwórczych, jak również niską dokładnością tychże metod oraz brakiem wariantów (różnorodności implementacji, która wykraczałyby poza aktualny stan wiedzy programisty). Dodatkowo uzasadnia to pracochłonność potrzeba na wykorzystanie dodatkowych metod oraz konieczność przyswojenia odmiennego paradygmatu.

5. Przykład implementacji wzorca Singleton

5.1. Kontekst problemu

Oprogramowanie zmienia się w czasie (Fowler, 2011), co powoduje konieczność refaktoryzacji. Na proces refaktorazacji składają się różne przekształcenia kodu źródłowego, m.in. usuwanie powtórzeń, upraszczanie złożonych elementów logiki, porządkowanie mało przejrzystego kodu oraz refaktoryzacja do wzorców projektowych. Prezentowany w dalszej części przykład refaktoryzacji został oparty na kodzie źródłowym programu komputerowego przeznaczonego do obsługi sprzętowych interfejsów kontrolno-pomiarowych, napisany jest w języku C# zgodnie z wzorcem architektury MVP, składa się z 158 klas. Program został wykonany przez jeden z obserwowanych zespołów wytwórczych, zatem można uznać ten program za odpowiedni materiał badawczy.

Listing 1 (w tabeli 1) przedstawia fragment kodu źródłowego klasy odpowiedzialnej za komunikację z urządzeniem podłączonym do interfejsu szeregowego RS-232 (tzw. port COM). Klienty tworzą egzemplarz tej klasy a następnie uzyskują dane poprzez metodę *GetData*. Przedstawiony kod działa prawidłowo, zdaje poprawnie testy jednostkowe (ang. *unit tests*) a wyniki wymienionych wcześniej metryk (zastosowanych zgodnie z przeznaczeniem) mieszczą się w zakresie dopuszczalnych wartości (Wojszczyk, 2013). Niestety, taka implementacja stwarza pewne zagrożenia:

1. Spadek wydajności w sytuacji częstych odczytów danych. Jest to spowodowane koniecznością utworzenia egzemplarza klasy i inicjalizacji połączenia z urządzeniem (urządzenie potrzebuje kilku sekund na start i wykonanie wewnętrznych testów).
2. Zablokowanie urządzenia w sytuacji niesynchronicznego utworzenia obiektu (np. z różnych wątków aplikacji), co może spowodować również przekłamania w zwracanych danych.

Istnieje przynajmniej kilka rozwiązań, które mogą wyeliminować powyższe zagrożenia oraz ryzyko zwiększonych kosztów po stronie odbiorcy i dostawcy systemu. Jednym z zalecanych rozwiązań jest refaktoryzacja kodu z listingu 1 do wzorca projektowego Singleton (Fowler, 2011), listing 2 przedstawia kod po refaktoryzacji. Wzorzec Singleton zapewnia, że klasa będzie miała tylko jeden egzemplarz, który będzie dostępny globalnie. To ograniczenie jest wystarczające aby rozwiązać wymienione zagrożenia.

Przedstawiony kod z listingu 2 działa prawidłowo i pozytywnie zdaje testy jednostkowe. Niestety również w tym przypadku występuje drobna pomyłka, która wynika z przeoczenia programisty. Pomyłka to pomięcie modyfikatora dostępu przy właściwości zwracającej instancję obiektu Singleton (wiersz *static RS232Reader Instance* na listingu 2). W języku C# brak modyfikatora dostępu równoważny jest modyfikatorowi *internal*, który zawęża zakres widoczności danego elementu. Taka, z pozoru drobna pomyłka, może spowodować znaczące i kosztowne utrudnienia

w przyszłości, tj. po zakończeniu prac nad danym fragmentem programu, który zostanie zamknięty w postaci osobnej biblioteki.dll. Obserwacje pokazują, że wraz z upływem czasu kod źródłowy nie zawsze jest dostępny, nawet w obrębie tej samej organizacji. Występująca pomyłka uniemożliwi ponowne wykorzystanie tej biblioteki z jednoczesną rozbudową w nowym programie, tzn. instancja Singletonu nie jest dostępna dla innych bibliotek (w tym programów exe). Naprawa tej pomyłki na etapie tworzenia kodu pozwoli uniknąć dodatkowych kosztów.

Tabela 1. Fragmenty kodu źródłowego programu

Listing 1

```
public class RS232Reader
{
 public RS232Reader)
 {
 this.Initialize();
 }
 private void Initialize()
 { ... }
 public byte[] GetData()
 { ... }
}
```

Listing 3 – fragment

```
...
public static RS232DeviceReader
Instance
...
```

Listing 2

```
public class RS232Reader
{
 private RS232Reader instance;
 private RS232Reader()
 {
 this.Initialize();
 }
 static RS232Reader Instance
 {
 get
 {
 if (instance == null)
 instance = new RS232Reader();
 return instance;
 }
 }
 private void Initialize()
 { }
 public byte[] GetData()
 { }
}
```

Źródło: opracowanie własne

Przedstawiona pomyłka jest trudna do wykrycia w sposób automatyczny. Różnice pomiędzy wynikiem metryk dla listingu 2 i 3 (poprawiona pomyłka, widoczny jest wyłącznie jeden wiersz kodu z zaznaczoną zmianą) są pomijalne i w obu przypadkach mieszczą się w zakresie zalecanych wartości. Dedykowane metody zliczające ilość wystąpień wzorców projektowych w obu przypadkach wykryją to samo, dokładnie jedno, wystąpienie wzorca Singleton, natomiast metody weryfikacji implementacji wzorców projektowych wykażą zgodność implementacji z dopuszczalnymi rozwiązaniami.

5.2. Proponowane rozwiązanie problemu

Listingi 2 i 3 prezentują dwa, pozornie podobne fragmenty kodu. Jak zostało już wspomniane, aby porównać dwa podobne elementy należy wyznaczyć jakość tych elementów (Durczak, 2011). Następnie dysponując jakością wyrażoną w postaci liczbowej można ocenić, który element jest „lepszy" w zadanym kontekście i stanowi właściwe rozwiązanie, oraz który z elementów spełnia wymagany poziom jakości.

W celu wyrażenia jakości w sposób liczbowy konieczne jest wskazanie odpowiednich metryk. Aby uwidocznić opisaną wcześniej pomyłkę należy wykorzystać metrykę $m_1 \in M$ wyrażoną wzorem (1). Metryka ta kwantyfikuje modyfikator dostępu instancji obiektu Singleton. Tabela 2 przedstawia interpretacje możliwych wyników w przyjętym kontekście (udostępnianie globalnego zasobu), zalecany wynik to 2 (modyfikator publiczny).

$$f(x) = \begin{cases} 0, \text{gdy private} \\ 1, \text{gdy internal lub domyślny} \\ 2, \text{gdy public} \end{cases} \quad (1)$$

gdzie x to element składowy udostępniający instancję obiektu Singleton, tzn. pole, metoda lub właściwość.

Tabela 2. Wyniki oraz interpretacja metryki m_1.

Modyfikator dostępu	Wynik metryki	Interpretacja
public	2	Zalecane, udostępnia instancję Singletonu wszystkim klasom, nie ogranicza wykorzystania.
Internal lub domyślny	1	Dopuszczalne, jednakże ogranicza zasięg widoczność wyłącznie do biblioteki. Może powodować problemy przy próbie integracji z oprogramowanie zawierającym wystąpienie wzorca.
private	0	Niedopuszczalne, uniemożliwi dostęp do instancji, wzorzec nie będzie spełniał swojego przeznaczenia.

Źródło: opracowanie własne

Wynik metryki m_1 dla listingu 2 (dokładniej dla właściwości udostępniającej obiekt) to $y = 1$, natomiast dla listingu 3 $z = 2$. Z tabeli 1 wynika, że zalecany jest najwyższy wynik metryki, zatem $MAX(y, z) = z$. Implementacja z listingu 3 zagwarantuje wzrost jakości w przyjętym kontekście.

6. Podsumowanie

W pracy zostały przedstawione wybrane zagadnienia związane z implementacją wzorców projektowych przez małe zespoły wytwórcze, pracujące według metodyk zwinnych. Zostały przedstawione również trudności związane z implementacją wzorców projektowych oraz zastosowanie autorskiej metody, która wspomaga rozwiązanie tych trudności. Zaletą metody jest wykorzystanie elementów bazujących bazowych programowania obiektowego oraz wyrażenie jakości implementacji wzorców projektowych w postaci liczbowej.

Dalsze badania przewidują doszczegółowienie metryk dla wybranych wzorców projektowych, a następnie przeprowadzenie praktycznego eksperymentu z udziałem zespołu programistów pracującego według metodyki Scrum.

Bibliografia

1. Alexander C. (2008) *Język wzorców. Miasta, budynki, konstrukcja*. GWP, Gdańsk.
2. Bielawa A. (2011) *Postrzeganie i rozumienie jakości – przegląd definicji jakości*, w: Studia i Prace Wydziału Nauk Ekonomicznych i Zarządzania, nr 21.
3. Czyczyn-Egird D., Wojszczyk R. (2016) *Determining the popularity of design patterns used by programmers based on the analysis of questions and answers on stackoverflow.com Social Network*, w: Communications in Computer and Information Science, vol. 608, s. 421–433.
4. Durczak K. (2011) *System oceny jakości maszyn rolniczych*, Rozprawa naukowa nr 418, Wydawnictwo Uniwersytetu Przyrodniczego w Poznaniu.
5. Fowler M. i inni (2005) *Architektura systemów zarządzania przedsiębiorstwem – Wzorce projektowe*. Helion, Gliwice.
6. Fowler M. i inni (2011) *Refaktoryzacja. Ulepszanie struktury istniejącego kodu*. Helion, Gliwice.
7. Gamma E. i inni (1995) *WZORCE PROJEKTOWE Elementy oprogramowania wielokrotnego użytku*. Helion, Gliwice.
8. Grzanek K. (2008) *Realizacja systemu wyszukiwania wystąpień wzorców projektowych w oprogramowaniu przy zastosowaniu metod analizy statycznej kodu źródłowego*, Dysertacja doktorska, Politechnika Częstochowska, Łódź.
9. Hernandez J. i inni (2011) *Selection of Metrics for Predicting the Appropriate Application of design patterns*, 2nd Asian Conference on Pattern Languages of Programs.
10. Hołodnik-Janczura G. (2007) *Badanie jakości produktu informatycznego metodą wartościowania*. Badania Operacyjne i Decyzje, Oficyna Wydawnicza Politechniki Wrocławskiej, s. 55–69.
11. Kan S. H. (2006) *Metryki i modele w inżynierii jakości oprogramowania*. PWN SA, Warszawa.
12. Kerievsky J. (2005) *Refaktoryzacja do wzorców projektowych*. Helion, Gliwice.
13. Khaer Md. A. i in. (2007) *An Empirical Analysis of Software Systems for Measurement of Design Quality Level Based on Design Patterns, Computer and Information Technology,* 10th international conference on Computer and Information Technology, p. 1–6, Dhaka.

14. Kobyliński A. (2003) *ISO/IEC 9126 – analiza modelu jakości produktów programowych*, w: Systemy Wspomagania Organizacji 2003.
15. Martin R., Martin M (2008): *Agile. Programowanie zwinne: zasady, wzorce i praktyki zwinnego wytwarzania oprogramowania w C#.* Helion, Gliwice.
16. Mazur Zygmunt i inni (2010) *Kontrola jakości wewnętrznej i kwalifikacje zespołu projektowego a jakość systemu baz danych*, w: Studia Informatica, vol. 31, nr 2B.
17. Mehlitz P.C., Penix J. (2003) *Design for Verification Using Design Patterns to Build Reliable Systems*, 6th Workshop on Component-Based Software Eng.
18. McConnell S. (2010) *Kod Doskonały*. Helion, Gliwice.
19. Nicholson J. et al. (2013) *Automated verification of desing patterns: A case study* [w:] Science of Computer Programing, Vol. 80, Part B, p. 211–222, Elsevier
20. Pressman R.S. (2004) *Inżynieria oprogramowania, praktyczne podejście do inżynierii oprogramowania*. WNT, Warszawa.
21. Rasool G. (2010) *Customizable Feature based Design Pattern Recognition Integrating Multiple Techniques*, Dysertacja Doktorska, Technische Universitat Ilmenau, Ilmenau.
22. Roman A. (2015) *Testowanie i jakość oprogramowania. Metody, narzędzia, techniki*. Wydawnictwo Naukowe PWN, Warszawa.
23. Tsantalis N. et al. (2006) *Design Pattern Detection Using Similarity Scoring* w IEEE Transactions on Software Engineering, vol. 32, Issue: 11, s. 896–908.
24. Wojszczyk R. (2013) *Zestawienie metryk oprogramowania obiektowego opartych na statycznej analizie kodu źródłowego*, w: Zarządzanie projektami i modelowanie procesów, s. 95–107.
25. Wojszczyk R., Khadzhynov W. (2017) *The Process of Verifying the Implementation of Design Patterns – Used Data Models*, w: Advances in Intelligent Systems and Computing, vol. 521, s. 103–116.

DIFFICULTIES IN IMPLEMENTATION OF DESIGN PATTERN IN SMALL DEVELOPERS TEAM

Abstract

There are many small businesses create their own, original software or performing services for third parties, in IT market. Small teams often work by the so-called. agile methodologies, which are limited by the amount of project documentation, and most of the procedures considerably simplified. However, small teams not reject good practice, including design patterns.

The aim of the article is to present some difficulties faced by small development teams in the implementation of design patterns, and to present the author's method of supporting an increase in the quality of implementation of design patterns.

Key words: Agile Manifesto, SCRUM, software development, design patterns.