|  |
| --- |
| Akademia Górniczo-Hutnicza w Krakowie, wydział Inżynierii Metali i Informatyki przemysłowej |
| Test-Driven Development jako zwinna technika programowania |
| Praca Inżynierska |
|  |
| **Tomasz Piętka** |
| **13 czerwca 2015 roku** |

|  |
| --- |
|  |

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszy projekt inżynierski wykonałem osobiście i samodzielnie i że nie korzystałem ze źródeł innych niż wymienione w pracy.

Kraków, dnia ………….… Podpis dyplomanta ……………………

Spis treści

[1. Wstęp 4](#_Toc421973252)

[2. Część teoretyczna 5](#_Toc421973253)

[2.1. Programowanie zwinne („Agile”) 5](#_Toc421973254)

[2.2. Extreme programming 7](#_Toc421973255)

[2.3. Testy jednostkowe 7](#_Toc421973256)

[2.4. Testy integracyjne 7](#_Toc421973257)

[2.5. Test-Driven development 7](#_Toc421973258)

[2.6. Test-Driven design 7](#_Toc421973259)

[2.7. Planning Poker 7](#_Toc421973260)

[3. Badania własne 8](#_Toc421973261)

[3.1. Charakterystyka aplikacji Planning Poker oraz omówienie jej składowych 8](#_Toc421973262)

[3.1.1. Technologia 8](#_Toc421973263)

[3.1.1.1. C#.Net 8](#_Toc421973264)

[3.1.1.2. Nunit 8](#_Toc421973265)

[3.1.1.3. Fluent Assertions 8](#_Toc421973266)

[3.1.1.4. Linq 8](#_Toc421973267)

[3.1.1.5. WPF 8](#_Toc421973268)

[3.1.2. Struktura testów 8](#_Toc421973269)

[3.1.2.1. Testy jednostkowe 8](#_Toc421973270)

[3.1.2.2. Testy integracyjne 8](#_Toc421973271)

[3.1.3. Struktura aplikacji 8](#_Toc421973272)

[3.1.3.1. Klasa Player 8](#_Toc421973273)

[3.1.3.2. Klasa Story 8](#_Toc421973274)

[3.1.3.3. Klasa PlanningPoker 8](#_Toc421973275)

[3.1.3.4. Klasa Estimation 8](#_Toc421973276)

[3.1.4. Graficzny interfejs użytkownika 8](#_Toc421973277)

[3.2. Przykład tworzenia aplikacji 8](#_Toc421973278)

[3.2.1. Definiowanie testu jednostkowego krok po kroku 8](#_Toc421973279)

[3.2.2. Testy konstruktorów 8](#_Toc421973280)

[3.2.3. Wyjątki 8](#_Toc421973281)

[3.2.4. Test Case 9](#_Toc421973282)

[3.2.5. Refactoring 9](#_Toc421973283)

[4. Wnioski 10](#_Toc421973284)

[5. Bibliografia 11](#_Toc421973285)

# Wstęp

Programowanie zwinne (ang. Agile software development), czyli grupa metodyk wytwarzania oprogramowania oparta na programowaniu iteracyjno-przyrostowym prężnie rozwijają się od kilkudziesięciu lat. Wielokrotnie zostało udowodnione, że projekty, w których stosuje się medotyki zwinne odnoszą większe sukcesy od projektów tradycyjnych.

Pojęcie zwinnego oprogramowania zostało zaprezentowane w tak zwanym Agile Manifesto w 2001 roku. Sygnatarusze manifestu przedkładali ludzi i interakcje ponad procesy i narzędzia, działające oprogramowanie ponad obszerną dokumentację, współpracę z klientem ponad formalne ustalenia oraz reagowanie na zmiany ponad podążanie za planem[[1]](#endnote-1). Oznacza to, iż w przeciwieństwie do dotychczasowego, tradycyjnego podejścia występuje większy nacisk na działające oprogramowanie i satysfakcję klienta.

Manifest z 2001 roku był wypadkową wielu ruchów, które miały swoje początku w latach 90 XX wieku. Ken Schwaber już w 1996 roku zdefiniował i sformalizował najpopularniejszą obecnie metodykę o nazwie Scrum.

Wymieniając kolejne metodyki należy wspomnieć o KanBan, Lean Software development, Feature Driven Development czy najważniejszym w kontekście tego oprogramowania Programowaniu Ekstremalnym (ang Extreme Programming).

Programowanie Ekstremalne– paradygmat zdefioniowany przez Kenta Becka w 1999 w książce *Extreme Programming Explained[[2]](#endnote-2).* Extreme programing to tak naprawdę zbiór technik, jedną z nich jest Test-Driven Developement (w skrócie TDD).

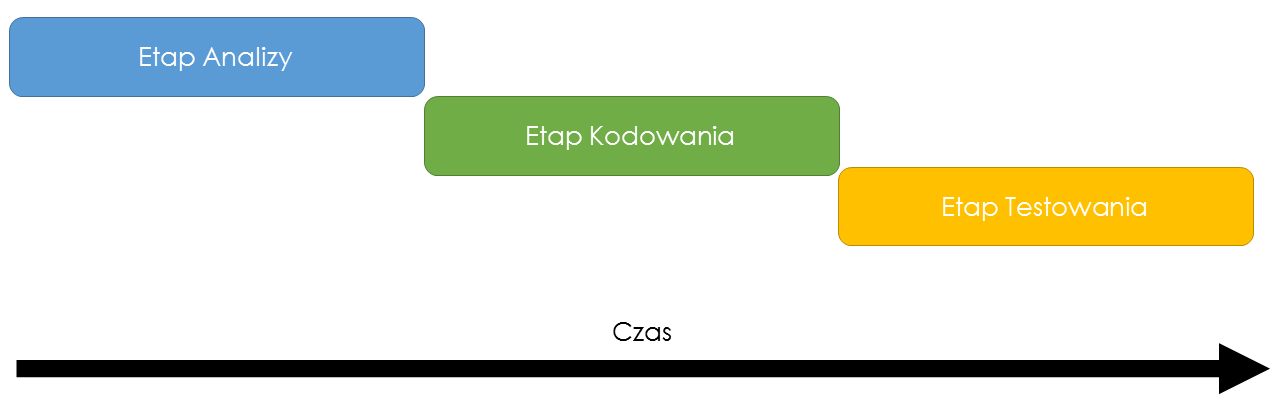
Mimo tego, iż TDD jest znane od wielu lat, nie jest ono używane w wystarczającym stopniu. Głównym powodem jest to, iż założone jest, że programiści radykalnie zmienią swój sposób pracy oraz jej filozofię – tak zwane Kaikaku (jap radykalna zmiana). Pisanie testów przed napisaniem działającego kodu jest najczęściej trudne do zrozumienia oraz zastosowania. Kaikaku to koncepcja przeciwna Kaizenowi (który zakładał ciągłą zmianę w procesach produkcyjncych), skupiająca się na małych, radykalnych, inkrementacyjnych zmianach. Kaikaku idealnie opisuje to, co musi zajść w sposobie myślenia programisty używającego tradycyjne metody programowania, aby mógł korzystać ze wszystkiego co daje poprawne stosowanie TDD.

Jako entuzjasta i praktyk zwinnych metodyk oprogramowania, w tej pracy postaram się udowodnić, iż programowanie w technice TDD jest dużo bardziej wydajne niż programowanie tradycyjne. TDD skupia się także na jakości oprogramowania. Aplikacja, która jest częścią tego projektu umożliwia przeprowadzanie sesji tzw Planning Poker – kolejnej techniki programowania zwinnego polegającej na ułatwieniu procesu estymacji pracy.[[3]](#endnote-3)

# Część teoretyczna

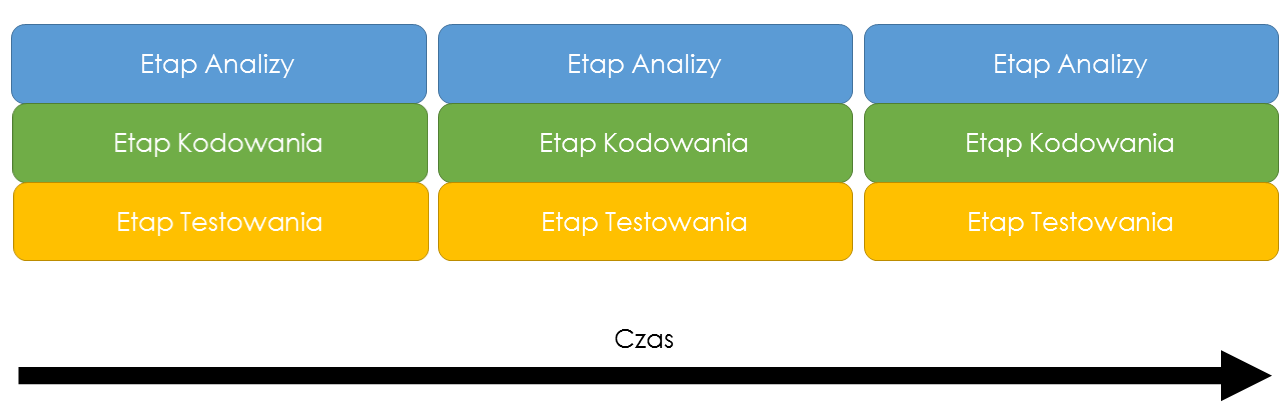
## Programowanie zwinne („Agile”)

Agile to sposób programowania, który przypomina nam, że mimo tego, że to komputery uruchamiają kod to ludzie go tworzą i utrzymują[[4]](#endnote-4). Aby zrozumieć to co dzieje się od kilkunastu lat należy cofnąć się do połowy XX wieku. Inżynieria oprogramowania jako dziedzina biznesu powstała około 40 lat temu. Wtedy urodziło się tradycyjne podejście do projektów, a z racji tego, że programowanie do dziedzina bardzo specyficzna, inżynierowie byli zmuszeni zapożyczać techniki z istniejących dziedziń biznesu. Stąd zapożyczenia m.in. z budownictwa widoczne do dzisiaj – nazywamy siebie inzynierami, mamy „buildy” (budynki) itp. Zapożyczone zostały również techniki prowadzenia projektów. Do niedawna dominowało przeświadczenie iż każdą fazę tworzenia oprogramowania należy wykonywać po sobie, sekwencyjnie. Tradycyjne podejście może być przedstawione na poniższym rysunku.



Rysunek 1. Tradycyjne podejście do projektów programistycznych, tzw „wodospad”, opracowanie własne

Takie podejście nazywamy obecnie wodospadem (ang waterfall). Każdy z etapów następuje po sobie, jest od siebie zależny. Głównym problemem takiego podejścia jest problem sprzężenia zwrotnego (ang feedback loop). Każdy etap wymaga osobnego zespołu. Zespoły w jednym momencie czasu pracują nad różnymi projektami. Gdy problem z wymaganiami – redefinicja wymagań, niedostateczne ich zrozumienie itp – zostaje zidentyfikowany na jednym z ostatnich etapów (najcześciej testy) caly proces musi odbyć się jeszcze raz. Tego typu podejście przez lata nastręczało wiele problemów – stąd rozwój metodyk zwinnych, których głównym zadaniem jest skrócenie czasu reakcji w sprzężeniu zwrotnym. Poniższy rysunek obrazuje iteracyjne podejście do prowadzenia projektu.



Rysunek 2. Zwinne podejście do projektów programistycznych, tzw „Agile”, opracowanie własne

Tutaj etapy projektu prowadzone są równocześnie, w krótkich iteracjach (najczęsciej 1-4 tygodni). Metodyki Agile skupiają się najczęściej na dostarczeniu małej części działającego oprogramowania (z metodyki Scrum, „przyrost”[[5]](#endnote-5)). Dzięki temu reakcja na sprzężenie zwrotne jest krótsza – zespół może modyfikować założenia w trakcie pisania projektu. Dzięki temu, iż zespoły są odpowiedzialne za całość działającego oprogramowania (a nie za poszczególne etapy) występuje lepsze zrozumienie tego co aktualnie się dzieje[[6]](#endnote-6). Wspomniany wcześniej Manifest Agile wygląda następująco:

|  |  |
| --- | --- |
| Manifest Zwinnego Tworzenia Oprogramowania | |
| Wytwarzając oprogramowanie i pomagając innym w tym zakresie,  odkrywamy lepsze sposoby wykonywania tej pracy.  W wyniku tych doświadczeń przedkładamy: | |
| **Ludzi i interakcje** ponad procesy i narzędzia. | |
| **Działające oprogramowanie** ponad obszerną dokumentację. | |
| **Współpracę z klientem** ponad formalne ustalenia. | |
| **Reagowanie na zmiany** ponad podążanie za planem. | |
| Doceniamy to, co wymieniono po prawej stronie,  jednak bardziej cenimy to, co po lewej. | |
| Kent Beck  Mike Beedle  Arie van Bennekum  Alistair Cockburn  Ward Cunningham  Martin Fowler  James Grenning  Jim Highsmith  Andrew Hunt | Ron Jeffries  Jon Kern  Brian Marick  Robert C. Martin  Steve Mellor  Ken Schwaber  Jeff Sutherland  Dave Thomas |

Tabela 1. Manifest Zwinnego Tworzenia Oprogramowania, http://agilemanifesto.org/iso/pl/

## Extreme programming

Programowanie ekstremalne (ang. Extreme Programming) to zbiór praktyk programowania, które skupiają się na ulepszaniu jakości oprogramowania i zarządzania reakcją na zmieniające się wymagania klientów[[7]](#endnote-7). Zalecenia XP:

- **Iteracyjność** – program tworzy się w iteracjach i planuje się tylko kolejną iterację.

- **Nie projektować z góry** – architektura systemu powinna być tworzona na bierząco. Nie można z góry przewidzieć jaka architektura będzie najlepsza przed rozpoczęciem projektu.

- **Testy jednostkowe –** należy je pisać przed pisaniem kodu. Jest to definicja techniki Test-Driven Development. Niniejsze opracowanie w dalszej części skupia się na tej technice.

- **Ciągłe modyfikacje architektury** – architektura rozwija się wraz z powstawaniem oprogoramowania.

- **Programowanie parami** – kod należy pisać parami – jedna osoba pracuje przy klawiaturze, druga jest obserwatorem.

- **Stały kontakt z klientem** – klient musi wiedzieć co dzieje się w projekcie oraz przekazywać informacje o zmieniających się wymaganiach.[[8]](#endnote-8)

## Testy jednostkowe

Testy jednostkowe jako pierwsze zostały zdefiniowane w latach 70 XX wieku i przedstawione jako koncepcja wraz z językiem Smalltalk przez Kenta Becka. Testy jednostkowe w programowaniu obiektowym to jeden z najlepszych sposobów dzięki któremu programista może poprawić jakość swojego kodu zyskując dogłębne zrozumienie wymagań funkcjonalnych tworzonej klasy lubi modelu. Wikipedia definiuje unit test jako (tłumaczenie własne):

**Definicja 1:** Test jednostkowy to kawałek kodu (najczęściej metoda) który wywołuje inny kawałek kodu i sprawdza jego poprawność zgodnie z pewnymi założeniami. Jeśli założenia okazują się błędne, test jednostkowy zawodzi (fails).

System, do którego pisze się testy jednostkowe nazywany jest Systemem do testowania (ang System Under Test – SUT).

Bardziej dokładną definicję proponuje Roy Osherove w książce The Art of Unit Testing. Aby ją zrozumieć, należy wcześniej zdefiniować jednostkę pracy (ang unit of work). Za Osherove (tłumaczenie własne):

**Definicja 2**: Jednostka pracy jest sumą akcji, które mają miejsce przed wywołaniem metody publicznej w systemie i pojedynczy, zauważalny rezultat końcowy testu tego systemu. Zauważalny rezultat końcowy może być zaobserwowany bez patrzenia na stan systemu i tylko poprzez publiczne API i zachowania. Rezultat końcowy to jedno z poniższych:

- Wywołana publiczna metoda która zwraca wartość (funkcja nie jest typu void)

- Istnieje zauważalna zmiana stanu lub zachowania systemu przed i po wywołaniu, która może być określona bez sprawdzania stanu prywatnego (na przykład system może zalogować wcześniej nie istniejącego użytkownika (...))

- Istnieje wywołanie do innego systemu nad którym test nie ma kontroli oraz ten inny system nie zwraca żadnej wartości, lub wartość zwracana z tego systemu jest ignorowana.

Idea jednostki pracy, że jednostka to pojedyncza, atomiczna metoda i kilka klas i funkcji, które pomagają osiągnąć jej cel. Dzięki zrozumieniu powyższego konceptu można zastosować następującą definicję testu jednostkowego:

**Definicja 3**: Test jednostowy to kawałek kodu (najczęściej metoda) który wywołuje jednostkę pracy i sprawdza jeden specyficzny rezultat końcowy tej jednostki pracy. Jeśli założenie rezultatu końcowego okaże się złe, test jednostkowy zawodzi (fails). Zakres testu jednostkowego ma tak małą rozpiętość jak metoda lub tak dużą jak wiele klas[[9]](#endnote-9).

## Testy integracyjne

Prostą definicję testu integracyjnego proponuje Microsoft (tłumaczenie własne):

**Definicja 4**: Test integracyjny testuje różne componenty rozwiązania, które składają się na jedną, większą całość[[10]](#endnote-10).

Bardziej kompleksowa definicja opierająca się na wcześniej zdefiniowanej jednostce pracy to (tłumaczenie własne):

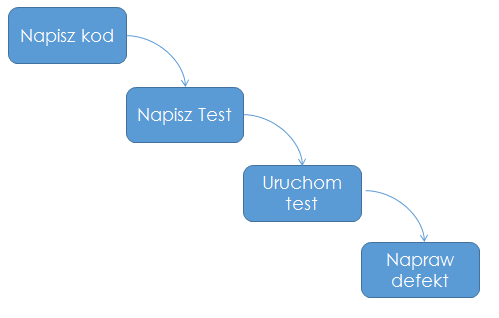
**Definicja 5**: Testy ingegracyjne testują jednostkę pracy bez posiadania pełnej kontroli nad nią i używają jedną lub więcej jej zależności, takich jak czas, sieć, baza danych, wątki itp[[11]](#endnote-11).

Czyli jak łatwo zauważyć, testy integracyjne dopełniają testy jednostkowe a celem obu jest stworzenie oprogramowania jak najlepszej jakości.

## Test-Driven development

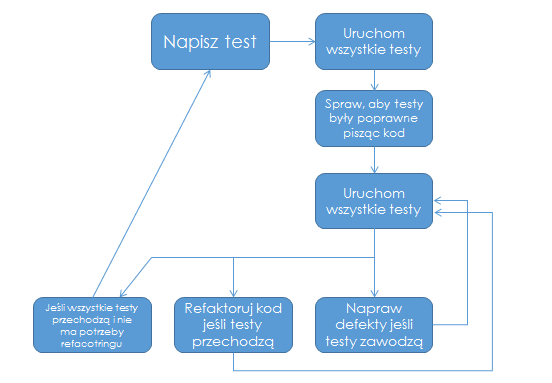
W tym momencie zdefiniowane zostały rodzaje testów, jednak pozostaje pytanie kiedy je napisać. Głownym założeniem techniki Test-Driven Development (w skrócie TDD) jest pisanie testów jednostkowych i integracyjnych ***przed*** napisaniem kodu aplikacji.

Poniższe rysunki przedstawiają różnicę pomiędzy tradycyjnym podejściem do pisania testów jednostkowych a podejściem TDD.



Rysunek 3. Tradycyjne podejście do pisania testów, opracowanie własne za Roy Osherove

Jak widać na powyższym rysunku, w tradycyjnym podejściu najpierw powstaje aplikacja, metoda lub inna część kodu a poźniej test. Głownym założeniem techniki Test-Driven Development (w skrócie TDD) jest pisanie testów jednostkowych i integracyjnych ***przed*** napisaniem kodu aplikacji.



Rysunek . Podejście TDD, opracowanie własne za Roy Osherove

TDD to przede wszystkim technika programowania – testy są tutaj ważna, jednak najważniejszą rzeczą pozostaje działające oprogramowanie. Mamy do czynienia ze swoistą mantrą TDD, którą można opisać w 3 prostych krokach:

- Krok Czerwony – napisz test, który zawodzi ponieważ nie ma jeszcze działającego kodu

- Krok Zielony – napisz kod, tak aby wcześniej napisany test się powiódł.

- Krok Refactor – gdy test przechodzi, można pisać kolejny test lub zmienić implementacje kodu aby był bardziej czytelny[[12]](#endnote-12).

Należy pamiętać, że w praktyce uruchamia się testy wielokrotnie. Wymaga to dyscypliny.

Przeanalizujmy jak wygląda pisanie protego testu i implementacji kodu w technice TDD. Przypuśćmy, że implementujemy prosty kalkulator. Zaczniemy więc od napisania testu sprawdzającego poprawność operacji dodawania dwóch liczb typu integer.

**Przykład kodu 1:**

[Test]

public void Dodaj\_DodajeDwieLiczby\_Obliczone()

{

//Arrange

    var calc = new Kalkulator();

//Act

    int suma = calc.Dodaj(2, 2);

//Assert

    Assert.AreEqual(4, suma);

}

Analizując powyższy kod od początku widzimy iż:

- Nasz kod jest metodą nie zwracającą żadnej wartości.

- Nazwa metody jest dodana w konwecji (Nazwa testowanej metody, wykonana akcja, efekt końcowy)

- W sekcji oznaczonej Arrange (zaaranżuj) tworzy się nową instancję klasy Kalkulator.

- W sekcji oznaczonej Act (działaj) deklarujemy zmienną suma i wywołujemy metodę Dodaj z argumentami 2, 2. Metoda powinna dodać te dwie liczby do siebie.

- W sekcji oznazonej Assert (zapewnij) sprawdzamy, czy wynik dodawania to 4.

Oczywiście w tym momencie gdy uruchomimy ten test, test zawiedzie. Nie ma jeszcze klasy kalkulator oraz jej metod. Jest to etap czerwony (ponieważ w więkoszości aplikacji test będzie zaznaczony na czerwono). Kolejnym krokiem jest napisanie logiki klasy Kalkulator tak, aby test się powiódł

**Przykład kodu 2:**

public class Kalkulator

{

    public int Dodaj(int a, int b)

    {

        return a + b

    }

}

Po napisaniu powyższej klasy test napisany wcześniej się powiedzie i programista może skupić się na pisaniu kolejnego testu lub refaktoryzacji kodu.[[13]](#endnote-13)

## Dane statystyczne dotyczące TDD

## Planning Poker

# Badania własne

## Charakterystyka aplikacji Planning Poker oraz omówienie jej składowych

### Technologia

### C#.Net

### Nunit

### Fluent Assertions

### Linq

### WPF

### Struktura testów

### Testy jednostkowe

### Testy integracyjne

### Struktura aplikacji

### Klasa Player

### Klasa Story

### Klasa PlanningPoker

### Klasa Estimation

### Graficzny interfejs użytkownika

### GIT

## Przykład tworzenia aplikacji

### Definiowanie testu jednostkowego krok po kroku

### Testy konstruktorów

### Wyjątki

### Test Case

### Refactoring

# Wnioski

# Bibliografia

1. http://agilemanifesto.org/ [↑](#endnote-ref-1)
2. Extreme Programming Explained: Embrace Change, 2nd Edition, Kent Beck, Cynthia Andres – 26 Listopad 2004 [↑](#endnote-ref-2)
3. http://wikipedia.org/ [↑](#endnote-ref-3)
4. The Agile Samurai – How Agile Masters Deliver Great Software, Jonathan Rasmusson, The Pragmatic Bookshelf, Wrzesień 2010, s 12 [↑](#endnote-ref-4)
5. Scrum Guide, Przewodnik po Scrumie: Reguły gry, Ken Schweber, Jeff Sutherland, Lipiec 2013 [↑](#endnote-ref-5)
6. Agile Principles as Software Engineering Principles: An Analysis, Normand S´eguin, Guy Tremblay, and Houda Bagane D´epartement d’informatique, Universit´e du Qu´ebec `a Montr´eal, Mardzec 2012 [↑](#endnote-ref-6)
7. http://www.extremeprogramming.org/ [↑](#endnote-ref-7)
8. Extreme Programming Explained: Embrace Change, 2nd Edition, Kent Beck, Cynthia Andres, 26 Listopad 2004 [↑](#endnote-ref-8)
9. The art of unit testing with examples in C#, Second Edition, Roy Osherove, Manning, 2014 [↑](#endnote-ref-9)
10. Better Unit Testing with Microsoft® Fakes, Microsoft Corporation, 2013 [↑](#endnote-ref-10)
11. The art of unit testing…, Osherove [↑](#endnote-ref-11)
12. The art of unit testing…, Osherove [↑](#endnote-ref-12)
13. http://dariuszwozniak.net/ [↑](#endnote-ref-13)