

TESTOWANIE OPROGRAMOWANIA

Projekt Zaliczeniowy - RAPORT

Data rozpoczęcia prac	28.04.2017
Data oddania raportu	30.05.2017

Nr grupy	4
Nazwisko i imię prowadzącego grupę	Dominik Albiniak

Członkowie:

Lp.	Nazwisko	Imię
1	Albiniak	Dominik

Testowana aplikacja - "Wiedźmin 3 – Dziki Gon"

Aplikacją testowaną będzie gra komputerowa "Wiedźmin 3 – Dziki Gon" w wersji 1.31 wydana na komputery stacjonarne (PC). Gra stworzona przez producenta i wydawcę: CD Projekt RED w reżyserii Konrada Tomaszkiewicza.

Twórcy tej gry akcji stworzyli do tego autorski silnik graficzny *REDengine 3* napisany za pomocą języka C++. Do ważnych cech tej technologii należy wsparcie dla 64-bitowego HDR (lighting calculations), teselacja otoczenia (wdrażana w grach obsługujących Direct X11), oraz wszelkie innowatorskie algorytmy umożliwiające renderowanie przeszkołd (np. drzewa, krzaki, beczki) oraz umożliwiające oglądanie świata gry w wysokiej rozdzielczości (dynamiczne cienie, wysoki dystans rysowania tekstur, PBR itp.). *Redengine 3* i *Wiedźmin 3* różnią się od swoich poprzednich wersji tym, że gracz otrzymał nowe formy wspinaczki po górach oraz możliwość nurkowania postacią, którą gra.

Testowanie gry serii *Wiedźmin* ma dużo wad i zalet

Wady	Zalety
Testowanie ręczne Długość kodów Wysoka ilość zmiennych Zgodność gry wedle fabuły Testowanie polega na graniu i znalezieniu 'odpowiedniego' momentu w grze Niektóre błędy mogą wynikać z silnika gry, a nie z samego programu Gra jest nieliniowa Testując program należy wykonać różne wariacje aby dojść do oczekiwanej sytuacji Niektóre łączenia klas podzbioru mogą doprowadzić do eksplozji kombinatorycznej	Wysoka jakość kodu Łatwo rozpoznać przyczynę błędu Brak skomplikowanych miejsc w kodzie Łatwo o przypadkową sytuację, która pomoże nam w wykryciu błędów Gra stawia na dowolność, dlatego łatwo o znalezienie każdej możliwej sytuacji

Ciągłe zmiany w grze (pogoda, ilość światła, pora dnia) mogą utrudniać testowanie

Wiedźmin 3 ze względu na swoją budowę musi być przetestowany ręcznie, ponieważ stworzenie programu, który miałby za nas pracować byłoby zbyt kosztowne. W związku z tym musimy samodzielnie zatroszczyć się o stworzenie testów pokrywających ilość testowanego kodu.

Tworzony przez nas test musi być dokładnie zaplanowany, ponieważ wpływ na pewien czynnik może być poprzednikiem wielu różnych kombinacji, a precyzyjne stworzenie wszystkich kombinacji jest wręcz niemożliwe do wykonania. Wiedząc to, możemy dojść do wniosku, że kontrolowalność naszego testu jest ciężkie do wykonania. Z tego powodu, posłużymy się testowaniem dynamicznym. Problemem może być również długość kodu i ilość zmiennych, która utrudnia nam pracę przy znalezieniu wszystkich błędów i zwiększa ilość czasu, którego potrzebujemy na dokładną analizę kodu oraz możliwe rozmowy z developerami i moderatorami podczas inspekcji.

Dodatkowym utrudnieniem jest fakt, że program może działać inaczej na różnych systemach oraz z różnymi specyfikacjami. Warto więc sprawdzić czy każdy system poradzi sobie tak samo.

Testowana gra może być dosyć trudnym problemem dla testera, ponieważ nie można mieć pewności, czy dany problem został już rozwiązany. W takim przypadku tester musi polegać na własnej wiedzy i umiejętnościach.

Ze względu na złożoność aplikacji, tester powinien sprawdzić, czy są niezgodności w podprogramach, które budują tę grę. Może się więc okazać, że problemy leżą już w samej architekturze aplikacji. To właśnie tam można by doszukiwać się błędu, jeżeli problem polegałby na modelu otwartego świata lub innej struktury tworzącej grę.

Warto jednak wspomnieć o tym, że *Wiedźmin 3* stawia na dobrowolność w wykonywaniu działań, więc należy dokładnie sprawdzić, czy klient będzie mógł operować tak, jak mu to zaoferowano. Z tego względu należy sprawdzić, czy wszystkie możliwe kombinacje zostały sprawdzone i zaakceptowane.

Gra w dalszej części użytkowania, będzie często zmieniać parametry i zazwyczaj będzie bardziej złożona, dlatego można by posłużyć się testowaniem opartym na jednoczesnym uczeniu się i obserwowaniu oprogramowania i przepływu danych.

Analiza podstawy testów

Do analizy całego programu posłużymy się kodem zapisanym w plikach głównych Wiedźmina. Wystarczającym plikiem do podglądu i edycji plików może być tutaj program [Atom](#) lub notatnik. Analizą funkcjonalności i sposobem działania zajmie się sam program. Kwestią braku testów jednostkowych i wyroczni testowej może się okazać tutaj sporym problemem, ponieważ nie możemy być pewni, czy testowana przez nas aplikacja będzie pokrywać się z oczekiwany wynikiem.

Głównym powodem, dla którego nie będziemy mogli skorzystać z testów jednostkowych jest fakt, że gra nie jest przystosowana do takiej metody, a jedynym wyjściem jest włączenie aplikacji i samodzielne testowanie programu krok po kroku.

Nie istnieje również żadna wyrocznia testowa do wykorzystania. Powodem, dla którego wyrocznia nie istnieje, to brak możliwości pełnego przejścia z różnych faz, eksplozja kombinatoryczna warunków w świecie gry, wszechzepienie "sztucznej inteligencji" do świata gry oraz ciągłe zmiany w grze, które mogą być spowodowane naszym testowaniem oraz losowością gry.

Jedyną szansą na stworzenie powiązań z innymi plikami może być oddzielne stworzenie własnego kodu i "wszechzepienie" go do programu. Mechanizm gry sprowadza się do wykorzystania techniki działania z oryginalnym kodem.

Przykładowy test aplikacji

1. Testowanie programu - znalezienie błędów
2. Znalezienie danego błędu w kodzie gry
3. Naprawa błędu
4. Ponowne przetestowanie i analiza do momentu, aż program będzie działać wedle ustalonego schematu

Może to polegać na przetestowaniu aplikacji i znalezieniu przez nas błędów, a dopiero później znalezieniu miejsca występowania tego błędu w kodzie.

Przetestowanie samego kodu będzie w tym wypadku polegało na testowaniu statycznym, tzn. Ręcznym szukaniu niejasności podczas przeglądu lub analizy statycznej. Kod będzie sprawdzany np.. Dzięki wyżej wymienionych programom.

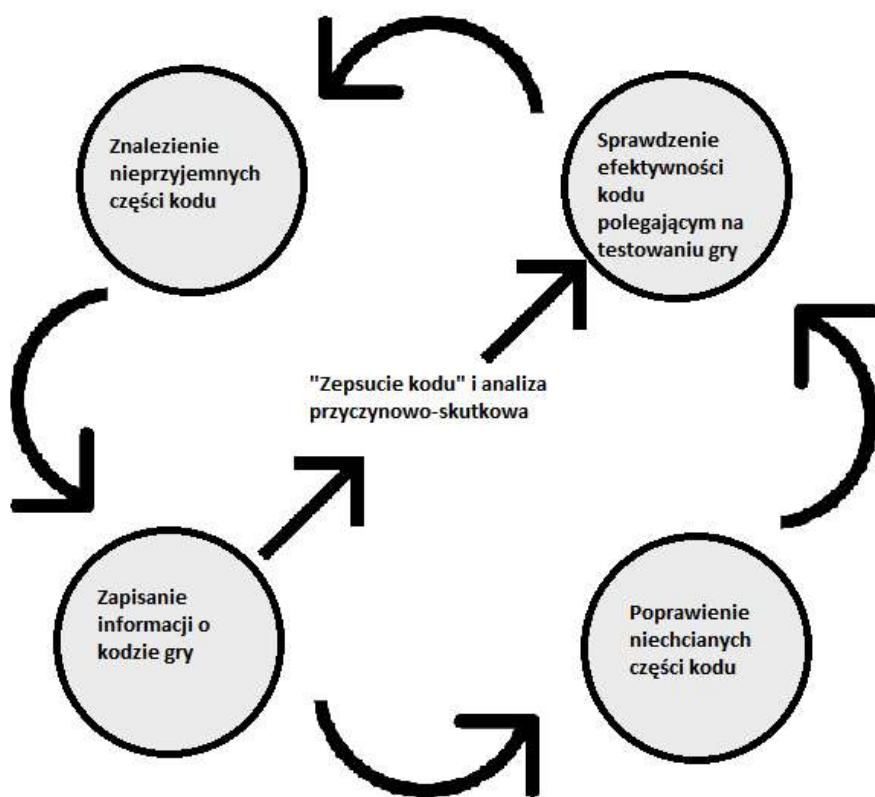
Dodatkowo jeszcze warto napisać, dlaczego nie zostaną użyte inne metody testowania - testowanie technikami czarnoskrzynkowymi lub testowanie za pomocą "use-caseów". Wyjaśnił to tym Przemysław Wójcik - Game QA Lead CD Projekt RED podczas swojego wykładu na *Testing Cup 2016* ([link](#)). Mówił on o tym, że wyżej wymienione techniki nie sprawdzają się w testowaniu gry oraz ubolewa on nad tym, że nie ma dobrej techniki testowania oprogramowania, która mogłaby być wprowadzona do testowania gier komputerowych.

Plan Testów

Wybór podejścia do testowania (strategii testowej) wraz z uzasadnieniem

Do testowania świata gry stworzonego przez CDP RED posłuży mi plik *npc* zawierający najważniejsze informacje na ten temat. Pragnę przetestować plik wraz z grą na dwa sposoby: po pierwsze, to sprawdzić jakość kodu oraz zastanowię się nad ulepszeniem go. Po drugie, spróbuję przetestować kod za pomocą "zniszczenia go". W skrócie zostaną zaprezentowane dwie metody: statyczna i dynamiczna. Do przetestowania pliku wraz z grą posłuży mi ów kod *npc*. Z punktu widzenia testowania oprogramowania wskazana metoda może być postrzegana jako bardzo niestaranna oraz nieefektywna. Jednak patrząc na wszystkie sposoby przetestowania aplikacji, to ten sposób może być uznany za najlepszy pod kątem przetestowania gry.

A więc szczegółowy plan wyglądać będzie następująco:



Pierwsze co powinniśmy zrobić to znalezienie odpowiedniego miejsca w programie, gdzie gra może działać źle lub kod można poprawić tak, aby był bardziej wydajny. Wydajność kodu może być tłumaczena jako poprawne, szybkie pod względem czasu wykonania według ustalonej wcześniej prędkości działanie algorytmów oraz napisanie kodu w sposób czytelny i szybki do zrozumienia dla reszty zespołu lub osób trzecich.

Po wykonaniu analizy kodu, wszystkie informacje o możliwych zmianach powinny być zapisane. Po tej czynności możemy przejść do dwóch czynności w zależności od potrzeby.

Mogliśmy wcześniej wymienione problemy spróbować naprawić lub dodatkowo przetestować grę, "psując" ją. Zmieniając kod gry na niestabilny lub nieprawidłowy, będziemy mogli zbadać krok po kroku, czy nie występują po nim inne niechciane sytuacje, które mogły być wcześniej

niezauważone. Dzięki tej metodzie możemy znacznie zmniejszyć czas na testowanie, ponieważ możemy w ten sposób dotrzeć do wielu innych błędów, które mogłyby pokazać się dopiero w chwili, gdy do gry dodawane były nowe zawartości (dodatki, patche, mody etc.). Jest to *tanie* i szybkie przeanalizowanie kodu, dzięki któremu zmniejszymy przede wszystkim koszt produkcji oraz zagwarantujemy sobie, że w przyszłości może być mniej problemów, np. tuż przed premierą. Po takiej analizie warto już rozpocząć testowanie gry oraz przeanalizowanie zdarzeń i zachowania otoczenia. Dopóki nie uznamy pracy za skończonej, powinniśmy powtarzać ten proces.

Zaplanowane poziomy testowania wraz z uzasadnieniem

Aby dokładnie użyć wspomnianej wcześniej strategii oczywistym wyborem będzie testowanie akceptacyjne, polegające na nabraniu zaufania do programu i stworzeniu dobrej współpracy z użytkownikiem. Będziemy mogli przez to ocenić poziom użyteczności gry i ocenić, czy nadaje się ona do użycia.

Podstawa testów

- wymagania użytkownika
- wymagania systemowe
- przypadki użycia
- procesy biznesowe
- raporty z analizy ryzyka

Typowe obiekty testów

- proces biznesowy na systemie w pełni zintegrowanym
- procesy utrzymania i obsługi
- procedury pracy użytkowników
- dane konfiguracyjne.

Dodatkowym uzasadnieniem, dlaczego wybrane zostanie testowanie akceptacyjne jest fakt, że produkt jest już działający, a testy akceptacyjne najlepiej pasują do tego typu produktów. Jednocześnie chciałbym wskazać, że nie ma tu potrzeby badania gry na poziomie niskopoziomowym (modułowy), a wszystkie testy związane z wymaganiami zostały już dawno wykonane i opublikowane.

Jednakże dodatkowymi testami, które wykonam to testy funkcjonalności *Wiedźmina* na moim komputerze. Zostaną podane dane techniczne oraz zostanie sprawdzone, jak gra radzi sobie na wybranym sprzęcie.

Zaplanowane typy testów wraz z uzasadnieniem

Zaplanowane będzie stworzenie dokumentacji zawierającej procesy biznesowe oraz stworzenie testów regresyjnych i retestów. Kiedy defekt zostanie znaleziony i naprawiony, program powinien zostać ponownie przetestowany. Zostanie podjęta ocena ryzyka co do wykonywanych działań. O ile będzie taka potrzeba mogą być wykorzystane *smoke testy* polegające na sprawdzeniu bezproblemowości programu. Ma to na celu znalezienie defektu, jeżeli taki się znajdzie, oraz przetestowanie funkcjonalności, które gra komputera powinna spełniać.

Metryki wykorzystane w procesie testowym

Posłużę się metrykami podanymi na stronie ([link](#))

Każda informacja podczas tworzenia metryk będzie udokumentowana z konkretną informacją. Informacje zawarte w metrykach testowych będą zawierać najważniejsze dane, takie jak koszty testowania związane z pracą testera oraz uwzględniony średni czas na naprawę w zależności od priorytetu, pokrycie wymagań, ryzyka kodu, ilość testów oraz procent pokrycia każdego z testów.

Do stworzenia jasnej i przejrzystej metryki, każda informacja będzie zawierać konkretną datę, a każdy defekt odpowiedni opis potrzebny do zrozumienia przyczyny błędu. Dany defekt będzie można naprawić, jeżeli jest to opłacalne z ekonomicznego punktu widzenia.

Rozsądne będzie jeszcze napisać o tym, jak wygląda stopień grywalności i czy znalezione problemy będą tej "grywalności" przeszkadzać.

W podsumowaniu metryk testowych będą przeanalizowane wszystkie informacje oraz przedstawione decyzje wraz ze stosownym uzasadnieniem.

Obszary aplikacji podlegające testom

Obszarem aplikacji podlegającym testom będzie plik **npc.ws** napisany w języku c++. Ów plik jest napisany do sterowania postaciami i przeciwnikami w grze, dodawaniem zdolności do wybranych postaci oraz zbiorem kodu odpowiadającym za to, co się wydarzy po zakończeniu akcji w danym fragmencie gry.

Plik **npc.ws** może być otwarty za pomocą nawet najwyklejszego programu do edycji plików, jakim jest np. notatnik lub wordpad. Ja posłużę się jednak programem Atom. Kod zapisany w tym pliku łącznie ze znakami nowej linii zawiera prawie 500 linii kodu. Obszar podlegający testom będzie zawierał wszystkie zmienne, warunki, komendy oraz przejścia w tym pliku.

Obszary aplikacji nie testowane

Obszarem aplikacji nie podlegającym testom będą: inne pliki powiązane z kodem, pliki niepowiązane, ale obecne w programie, import danych z programu.

Napisane wyżej obszary nie będą wykorzystywane w testowaniu ze względu na złożoność programu. Ponieważ kod npc zawiera już wystarczająco dużo informacji, opisywanie innych kodów zmuszałoby do napisania bardzo długiego raportu lub zmuszałoby do napisania wielu raportów, gdzie każdy opisywałby osobny plik.

Opis środowiska testowego, niezbędnych uprzęży testowych, osprzętu itd.

Opis systemu sprzętowego:

- Windows 10 Education 64-bit
- Computer type: Desktop
- .NET Framework installed v4.6 FULL, CLIENT
- Intel Core i5 6600K 3.50 Ghz 4 cores
- 2 x 8.0 GB RAM DDR4-2132 (1066 MHz)
- Motherboard Gigabyte Technology Z170-Gaming K3 (U3E1)
- NVIDIA GeForce GTX 960
- 45GB of free space
- Direct X11
- Microsoft Visual c++ 2012

Opis aplikacji:

- Version 1.31 The Witcher 3: Wild Hunt
- DLCs Blood and Wine, Free DLC program (16 DLC), Hearts of Stone
- Downloaded from GOG Galaxy([link](#))

Do testowania aplikacji są potrzebne :

- Zwykły program do edycji tekstu,
- Pełna wersja gry wraz ze wszystkimi plikami niezbędnymi do komplikacji.

Analiza ryzyka, priorytetyzacja testów

Ryzyko związane z wykryciem defektu może być oszacowane w oparciu o długość kodu, ponieważ im większy kod, tym proporcjonalnie rośnie szansa na znalezienie defektu.

Co więcej, na znalezienie problemu wpływa liczba osób pracujących nad programem oraz ich pochodzenie, doświadczenie deweloperów pracujących nad grą oraz ilość czasu, która została do czasu wydania gry. Gra jest stworzona w doświadczenym zespole, którego część tworzyła poprzednie odstony serii "Wiedźmin" lub współpracowała przy tworzeniu innych produkcji.

Dodatkowo, dobrze wyglądający na pierwszy rzut oka kod pliku npc.ws może świadczyć o tym, że kod był starannie tworzony i/lub były dokonywane prace służące naprawieniu błędów lub poprawy jakości kodu.

Z drugiej strony to "grywalność" jest czynnikiem, który klient uważa za priorytetowy. Nie jest on jednak możliwy do wyliczenia, dlatego deweloperzy nie mogą posługiwać się tą miarą, ponieważ to już zazwyczaj zależy od innej części zespołu.

Postługując się tymi czynnikami, możemy stwierdzić, że są szanse na znalezienie defektów lub nieścisłości, ale mogą one dotyczyć "grywalności" programu lub mogą to być drobne pomyłki przy budowaniu gry.

Wiedząc już o tych wszystkich czynnikach, można dojść do wniosku, że tworzone zostaną tutaj testy akceptacyjne związane ze stworzeniem wrażenia niezawodności oraz testy oparte na przeglądzie kodu, które pomogą w znalezieniu możliwych nieścisłości.

Harmonogram prac

W czasie realizacji raportu gry "Wiedźmin" będą realizowane kolejne etapy testowania, szeroko opisane w poprzednich punktach (Patrz *plan testów*). W tabeli poniżej zostały opisane poszczególne kroki. Całe testowanie ma na celu podniesienie niezawodności oprogramowania oraz usprawnić kod pod względem rozgrywki, która ma zostać zaoferowana klientom.

Wiedźmin 3: Dziki Gon	Data: 22. 05. 2017r.
1.	Testowanie statyczne. Analiza kodu.
2.	Zapisanie najistotniejszych informacji z punktu pierwszego.
3.	Klasyfikacja zagadnień.
4.	Testy akceptacyjne.
5.	Stworzenie metryk.
6.	Podsumowanie testu. Zalecenia co do programu.

Inne uwagi

Chciałbym zaznaczyć, że plan, według którego będzie testowana aplikacja, może być pomysłem wcześniej nieopublikowanym. Należy pamiętać, że nie zostały jeszcze stworzone żadne metody do wydajnego testowania gier komputerowych.

Z tego powodu, technika przeze mnie stosowana, może się okazać nieefektywna dla wielu innych programów. Dodatkowo testowaniem akceptacyjnym gier zajmują się głównie testerzy w wersji alfa i beta.

Projektowanie testów

Niniejszy plan obejmuje zakres testów przedstawionych wyżej.

Dodatkowo przeprowadzane będą testy biało-skrzynkowe, które zawierają następujące cechy:

Dokładność wyrażeń logicznych,

Pokrycie ścieżek liniowo niezależnych wyszczególnionych funkcji,

Pokrycie decyzji/warunków poszczególnych funkcji,

Pokrycie kodu.

Do testów akceptacyjnych posłużą

Sprawdzenie przypadków użycia,

Uruchamianie systemu po zmianach w pliku po testowaniu statycznym,

Testy zgodności z informacjami zawartymi w kodzie,

Ładowanie danych po zmianach, w instrukcjach pliku.

Dokładność wyrażeń logicznych

Wyrażenie to obejmuje ilość przypadków użycia w funkcjach zmiennych wartości logicznych. Obejmuje to również zakres przypadków testowych oraz ich liczbę.

Operatory logiczne :

NOT negacja

OR alternatywa

AND koniunkcja

XOR alternatywa wykluczająca

IMP implikacja

EQV równoważność

Generacja testów wykorzystana do sprawdzenia poprawności logicznych funkcji.

Pokrycie ścieżek liniowo niezależnych

Ilość ścieżek pierwszych, które mogą pojawić się przy uruchamianiu odpowiednich opcji w programie. Sprawdzenie polega na analizie przepływu danych, tak, aby każdy możliwy przypadek nie zawierał w sobie defektów

Pokrycie decyzji/warunków

Testy opierające się na liczbie decyzji i warunków w programie. Testowanie polega na stworzeniu wszystkich możliwych testów subsumujące kryterium pokrycia jednocześnie warunków oraz decyzji.

Pokrycie kodu

Zazwyczaj podana w procentach; miara wydajności programu opisująca jak dużą część funkcji można pokryć wybranymi testami. Dodatkowo obliczana jest miara ilości całego kodu do ilości kodu, który jest w pełni wykorzystywany.

Przypadki testowe:

Nazwa	Warunki wstępne	Kroki wykonania	Oczekiwany rezultat	Błędy, które mogą wystąpić
Zdefiniowane wartości początkowe	-	Przypisz początkowe wartości dla zmiennych	Wszystkie zmienne będą zawierać zgodne wartości dla programu	Przypisanie zmiennym, które nie spełniają żadnej funkcji w kodzie, wartości początkowe. Złe wartości początkowe
Przygotowywanie ataku	-	Pobierz dane W zależności od potrzeb aktualizuj parametry, zwróć odpowiedni wynik	Funkcja wykona odpowiednie kroki dla parametrów wejściowych	Podanie niewłaściwych danych Działanie niepożądane
Reakcja na atak	-	Zwróć oczekiwana akcję dla	Funkcja oblicza zadane obrażenia	W pewnych sytuacjach używanie liczb zmiennoprzecin

		<i>danego zdarzenia</i>	<i>typu float i procent punktów</i>	<i>kowych może wiązać się z ryzykiem. Czy program jest zabezpieczony przed dzieleniem przez zero?</i>
<i>Update postaci</i>	-	<i>Przypisz nowe zmienne</i>	<i>Kod rozpoznaje co gracz z robił i na tej podstawie go nagradza</i>	<i>Czy wartość zdobytych punktów może wynosić mniej niż zero?</i>
<i>Aktualizuj Bestiariusz</i>	-	<i>Dokonaj utworzenia nowego okienka z informacją</i>	<i>Zwrócenie informacji o procesie i aktualizacja postaci</i>	<i>Funkcja zwróci błędne dane lub nie wykona się wcale</i>
<i>Kalkulator doświadczenia</i>	If(ShouldGiveExp(damageAction.attacker)) jest prawdziwe	<i>Oblicz liczbę doświadczenia</i>	<i>Zwróć wartość otrzymanego doświadczenia : float</i>	<i>Czy wszystkie przypadki testowe są wykorzystywane? Czy kalkulator może zwrócić liczbę mniejszą od zera?</i>
<i>Funkcje typu 'timer'</i>	-	<i>Zwraca różne zmienne</i>	<i>Zwraca w zależności od rodzaju wartości, klasy itp..</i>	<i>Czy funkcje są poprawne? Czy jakość przedstawionego kodu nie budzi zastreżeń?</i>
<i>Funkcje typu 'event'</i>	-	<i>Przenosi do innych funkcji, rozpoczynające akcje</i>	<i>Czyta zmienne dostarczone do funkcji i odgrywa odpowiednią scenę</i>	<i>Czy funkcje działają poprawnie? Czy niczego nie brakuje?</i>

Technika projektowania testów wraz z uzasadnieniem:

Zdefiniowanie wartości początkowych opiera się na ręcznym znajdywaniu problemów związanych z tworzeniem zmiennych. Testy polegają na analizie kodu(zalecane użycie komendy GREP).

- Niepoprawne użycie zmiennej
- Użycie złej wartości
- Brak późniejszego użycia

- Niezdefiniowanie zmiennej; to samo dla funkcji

Dzięki tej analizie możemy zwiększyć pokrycie kodu oraz zapewnić sobie, że funkcje, które będą się opierać na tych zmiennych otrzymają odpowiednie wartości.

Następne przypadki testowe będą się również opierać na analizie kodu, a co więcej, wykorzystamy również testy białośkrzynkowe. Oparcie się o dokładność wyrażeń logicznych, decyzji i warunków może sprawdzić funkcjonalność kodu. Oparte jest to na kodzie npc, ponieważ znaczna jego część to właśnie wyrażenia "if ()" lub "if () else ". Skonstruowane dzięki tym metodom testy, powinny dać dużą efektywność.

Suita testowa winna wyglądać następująco: jest to suita suity decyzji/warunków i testów wartości logicznych.

Zdolność do wykrywania awarii takich testów powinna być w takim wypadku duża, ponieważ mogą zostać sprawdzone wszystkie wartości, jakie mogą wystąpić. Stosując tabelę wartości wyników oczekiwanych i rzeczywistych możemy mieć wgląd w stopień awaryjności programu.

Zaprojektowane testy – harmonogram działania:

1. Analizuj kod, stwórz pierwsze klauzule dla testów logicznych.
2. Stwórz tabelę wartości i wyników dla testów decyzji/warunków.
3. Usuń powtórzenia, które wystąpią pomiędzy dwoma wynikami.
4. Skonstruuj na tej podstawie tabele oczekiwanych wyników i wyników rzeczywistych.

Identyfikacja warunków testowych:

Informacja o tym, jak oprogramowanie jest skonstruowane(kod, struktura), która jest używana do tworzenia przypadków testowych

*Rozmiar pokrycia oprogramowania mierzony w stosunku **cały kod / rozmiar funkcji**. Dzięki dalszemu tworzeniu przypadków testowych można systematycznie zwiększać pokrycie kodu.*

<i>Analiza pokrycia poleceń kodu</i>
<i>Analiza pokrycia decyzji</i>
<i>Analiza pokrycia warunków</i>
<i>Analiza przedstawienia kodu</i>
<i>Oparcie wszystkich informacji na znalezionych defektach.</i>

Wykonanie testów, raport z przebiegu testów

Uwaga na "magic numbers" - zmienne często występują w podobnych nazwach, nieostrożny developer może doprowadzić do awarii lub błędu.

Nazwa, ID	Statemachine import class CNewNPC extends CActor, 01.
Kod, konkretne miejsce	23 - 231
Data	23. 05. 2017r. 19:30
Status, Poziom istotności	Minor, Niski
Zaprojektowane testy	Analiza statyczna kodu - przegląd
Liczba znalezionych defektów	<p>1. <code>: isMiniBossLevel, :suppressBroadcastingReactions, :</code> <code>dontUseReactionOneLiners, : previousStance, : shieldDebris, :</code> <code>lastMealTime, : packName, :isPackLeader, :wasNGPPlusLevelAdded,</code> 2.</p> <p>Problem: "Magic numbers" - zmienne zaimplementowane, nie używany w tym pliku. Brak konkretnej informacji o zmiennych.</p> <p>Możliwość rozwiązania: usunięcie implementacji zmiennych, jeżeli są potrzebne w innych plikach, to tam je zaimplementować.</p> <p>Skutek naprawy: Poprawa jakości kodu, zwiększenie pokrycia kodu, poprawa czytelności.</p> <p>Razem: 2</p>
Pokrycie funkcjonalności	Funkcjonalność kodu zmaksymalizowana
Pokrycie ryzyka	Zmniejsza łatwość popełnienia błędu w przyszłości
Pokrycie kodu	208 / 4810 ~ 4,32% całości kodu
Czas projektowania	-
Czas implementacji	5 minut
Czas wykonania testów	5 minut

Nazwa, ID	Public function GetBloodType() : EBloodType, 02.
Kod, konkretne miejsce	238 - 276
Data	23. 05. 2017r. 20:00
Status, Poziom istotności	Minor, Niski
Zaprojektowane testy	Analiza statyczna kodu - przegląd
Liczba znalezionych defektów	<p>Użycie pustych 9 pustych 'case'</p> <p>Razem: 1</p> <p>Możliwość rozwiązania: usunięcie pustych 'case'.</p> <p>Skutek naprawy: Poprawa jakości kodu, zwiększenie pokrycia kodu, poprawa czytelności.</p>
Pokrycie funkcjonalności	Funkcjonalność linii kodu zmaksymalizowana
Pokrycie ryzyka	-
Pokrycie kodu	38 / 4810 ~ ok. 0.79% całości
Czas projektowania	-
Czas implementacji	10 minut
Czas wykonania testów	10 minut

Nazwa, ID	Event OnSpawned(spawnData : SEntitySpawnData), 03.
<i>Kod, konkretne miejsce</i>	370- 606
<i>Data</i>	23. 05. 2017r. 20:05
<i>Status, Poziom istotności</i>	Minor, Niski
<i>Zaprojektowane testy</i>	Analiza statyczna kodu - przegląd
<i>Liczba znalezionych defektów</i>	<p>1.</p> <p>Użycie zbyt długich nazw zmiennych jak: <i>"CMovingPhysicalAgentComponent"</i>, <i>"SetOriginalInteractionPriority"</i>, <i>"RestoreOriginalInteractionPriority"</i>, <i>"SignalGameplayEventParamFloat"</i>, <i>"levelBonusesComputedAtPlayerLevel"</i>, <i>"SetFocusModeSoundEffectType"</i></p> <p>2.</p> <p>Duża część kodu przeznaczona na ciągłe wywołanie funkcji <i>AddAnimEventCallback()</i></p> <p>Razem: 2</p> <p>Możliwość rozwiązania: Poprawienie nazw zmiennych na krótsze. Wywoływanie funkcji <i>AddAnimEventCallback</i> jednorazowo dla danego fragmentu.</p> <p>Skutek naprawy: Poprawa jakości kodu, zwiększenie pokrycia kodu, poprawa czytelności.</p>
<i>Pokrycie funkcjonalności</i>	Znacząca poprawa jakości kodu. Kod krótszy, bardziej czytelny.
<i>Pokrycie ryzyka</i>	Zmniejsza ryzyko popełnienia błędu.
<i>Pokrycie kodu</i>	236 / 4810 ~ ok. 4.91% całości
<i>Czas projektowania</i>	-
<i>Czas implementacji</i>	15 minut
<i>Czas wykonania testów</i>	5 minut

Nazwa, ID	Protected function PerformParryCheck(parryInfo : SParryInfo) : bool, 04.
<i>Kod, konkretne miejsce</i>	948- 1016
<i>Data</i>	24. 05. 2017r. 20:30
<i>Status, Poziom istotności</i>	Defect, Średni
<i>Zaprojektowane testy</i>	Analiza statyczna kodu - przegląd, testy wyrażeń logicznych
<i>Liczba znalezionych defektów</i>	<p>1.</p> <p>Kod: Postać 'Imlerith' występuje tylko jeden raz jako główny przeciwnik.</p> <p>Możliwość rozwiązania: zastosować wartości dla postaci, które nie będą musiały później korzystać z dodatkowych warunków.</p> <p>Skutek naprawy: Zmniejszenie ilość warunków, zwiększenie pokrycia kodu.</p> <p>TESTOWANIE PAROWANIA CIOSÓW</p> <p>Załóżmy, że nastąpiła niepożądana sytuacja, po której <i>parryInfo.targetToAttackerAngleAbs < 0</i> np.. -1 lub -1000000 oraz <i>!npcTarget.HasShieldedAbility() == true</i>, wtedy warunek dla if jest równy true. Może to być wtedy sytuacja bardzo niepożądana</p>

	<p><i>Możliwość naprawy: Dodatkowe uwarunkowanie, które będzie mówić o liczbie nie mniejszej od zera</i></p> <p><i>Skutek: Program staje się bardziej niezawodny</i></p> <p><i>Razem: 2</i></p>
<i>Pokrycie funkcjonalności</i>	<i>Znacząca poprawa jakości kodu. Kod krótszy, bardziej czytelny.</i>
<i>Pokrycie ryzyka</i>	<i>Zmniejszenie ilości błędów</i>
<i>Pokrycie kodu</i>	<i>68 / 4810 ~ ok. 1.41% całości</i>
<i>Czas projektowania</i>	<i>-</i>
<i>Czas implementacji</i>	<i>20 minut</i>
<i>Czas wykonania testów</i>	<i>5 minut</i>

Nazwa, ID	Timer function AddLevelBonuses(), 05.
<i>Kod, konkretne miejsce</i>	<i>1111- 1307</i>
<i>Data</i>	<i>24. 05. 2017r. 21:00</i>
<i>Status, Poziom istotności</i>	<i>Minor, Niski</i>
<i>Zaprojektowane testy</i>	<i>Analiza statyczna kodu - przegląd, testy wyrażeń logicznych</i>
<i>Liczba znalezionych defektów</i>	<p>1. <i>Kod: Pusty else if()</i></p> <p><i>Kod: if(), który nic nie robi</i></p> <p><i>Możliwość naprawy: Wykorzystanie bądź usunięcie.</i></p> <p><i>Skutek: Kod jest czytelniejszy</i></p> <p>2.</p> <p><i>Kod jest skopiowany i wklejony w miejsce obok</i></p> <p><i>Możliwość naprawy: zastosowanie funkcji, zamiast kopiować ten sam kod</i></p> <p><i>Skutek: Kod jest czytelniejszy;</i></p> <p><i>Razem: 2</i></p>
<i>Pokrycie funkcjonalności</i>	<i>Znacząca poprawa jakości kodu. Kod krótszy, bardziej czytelny.</i>
<i>Pokrycie ryzyka</i>	<i>-</i>
<i>Pokrycie kodu</i>	<i>196 / 4810 ~ ok. 4.07% całości</i>
<i>Czas projektowania</i>	<i>-</i>
<i>Czas implementacji</i>	<i>5 minut</i>
<i>Czas wykonania testów</i>	<i>5 minut</i>

Nazwa, ID	Private function ApplyFistFightLevelDiff(), 06.
<i>Kod, konkretne miejsce</i>	1478- 1548
<i>Data</i>	24. 05. 2017r. 21:15
<i>Status, Poziom istotności</i>	Minor, Niski
<i>Zaprojektowane testy</i>	Analiza statyczna kodu - przegląd, testy wyrażeń logicznych
<i>Liczba znalezionych defektów</i>	<p>1. Analiza wartości kończącej() w pętli na $i < 5$. Dokładna ocena, dlaczego akurat ta wartość znajduje się w kodzie</p> <p>Możliwość rozwiązywania: Możliwość dodania zmiennej zdefiniowanej maksymalnej dla funkcji</p> <p>2. Powtarzająca się pętla()</p> <p>Możliwość rozwiązywania: Usunięcie pętli</p> <p>Razem: 2</p>
<i>Pokrycie funkcjonalności</i>	Zwiększone
<i>Pokrycie ryzyka</i>	Brak możliwego zapętlenia
<i>Pokrycie kodu</i>	70 / 4810 ~ ok. 1.45% całości
<i>Czas projektowania</i>	-
<i>Czas implementacji</i>	5 minut
<i>Czas wykonania testów</i>	5 minut

Nazwa, ID	Public function CalculateExperiencePoints(optional skipLog : bool) : int, 07.
<i>Kod, konkretne miejsce</i>	2225- 2362
<i>Data</i>	24. 05. 2017r. 21:35
<i>Status, Poziom istotności</i>	Minor, Niski
<i>Zaprojektowane testy</i>	Analiza statyczna kodu - przegląd, analiza warunków/decyzji
<i>Liczba znalezionych defektów</i>	<p>1. Zbyt częste użycie if() do obliczenia modOther Możliwość rozwiązywania: Podzielenie odpowiednich przejść na grupy, które zwracają tę samą wartość. Razem: 1</p>
<i>Pokrycie funkcjonalności</i>	Zwiększone
<i>Pokrycie ryzyka</i>	
<i>Pokrycie kodu</i>	137 / 4810 ~ ok. 2.84% całości
<i>Czas projektowania</i>	-
<i>Czas implementacji</i>	10 minut
<i>Czas wykonania testów</i>	10 minut

Nazwa, ID	Public function ReactToBeingHit(damageAction : W3DamageAction, optional buffNotApplied : bool) : bool, 08.
Kod, konkretne miejsce	1801- 1905
Data	25. 05. 2017r. 00:00
Status, Poziom istotności	Defect, Medium
Zaprojektowane testy	Analiza statyczna kodu - przegląd, analiza warunków/decyzji
Liczba znalezionych defektów	<p>ANALIZA WALKI:</p> <p>1.</p> <p>(3635) Dzieląc damageValue przez totalHealth nie jest założony przypadek, że totalHealth może wynosić zero. Kiedy nastąpi taka sytuacja, program może ulec awarii.</p> <p>Możliwość naprawy: uniknięcie sytuacji dzielenia przez zero</p> <p>Skutek: Zmniejszona ilość sytuacji, w których nastąpi błąd wykonania gry</p> <p>2.</p> <p>Ciężki do zrozumienia warunek(1820)</p> <p>Dodatkowo, po pierwszym AND GetCurrentStance() == NS_Fly nie subsumuje NOT(damageAction.IsDoItDamage() && !damageAction.DealsAnyDamage). Oznacza to, że podczas NS_Fly nie muszą zostać zadane obrażenia, aby został spełniony warunek, gdzie dalej ta opcja musi już być uwzględniona. Rozumie się przez to, że może nastąpić nieoczekiwane działanie programu.</p>
Pokrycie funkcjonalności	Zwiększone
Pokrycie ryzyka	Poprawa niezawodności programu.
Pokrycie kodu	104/ 4810 ~ ok. 2.16% całości
Czas projektowania	-
Czas implementacji	15 minut
Czas wykonania testów	15 minut

Nazwa, ID	Event OnDeath(damageAction : W3DamageAction) 09.
Kod, konkretne miejsce	2370- 2871
Data	25. 05. 2017r. 00:40
Status, Poziom istotności	Minor, Niski
Zaprojektowane testy	Analiza statyczna kodu - przegląd, analiza warunków/decyzji
Liczba znalezionych defektów	<p>1.</p> <p>Zbyt długie nazwy funkcji i zmiennych jak: "GetMonsterParamsForActor()", "CMovingPhysicalAgentComponent", "GetBehTreeReactionManager", "SingleGameplayEventParamObject", "FindGameplayEntitiesInRange", "OnChangeDyingInteractionPriorityIfNeeded()</p> <p>Zalecane: zmiana nazw</p>

	<p><i>Skutkuje poprawą jakości kodu</i></p> <p>2.</p> <p><i>if() niedokładne przyjęcie kiedy spełniony jest if(), program może nie działać poprawnie</i></p> <p><i>Zalecane: Podanie np. przybliżonej odległości od krawędzi, od której akcja "FallingDamage" ma zostać aktywowana przez funkcję IncStart()</i></p> <p><i>Skutek: Zmniejszenie szansy na "niebezpieczne" zachowanie się gry.</i></p> <p>3.</p> <p><i>"MagicNumbers" - pojawienie się takich zmiennych jak "vfxEnt", "fxEnt", "CreateFXEntityAtPelvis", których nazwy nie są łatwe do zrozumienia</i></p> <p><i>Zalecane: zmiana nazw;</i></p> <p><i>Skutek: Poprawa jakości kodu</i></p> <p>4.</p> <p><i>Zamiast ustawiać minDist = 10000 lepiej ustawić na pierwszy element VecDistance2D(this.GetWorldPosition(), entities[0].GetWorldPosition()) dla każdego i > 0. Dodatkowo można zacząć wtedy pętle od i = 1. Wtedy pętla wykona się (n - 1) razy, n = entities.Size()</i></p> <p><i>Zalecane: zmiana minDist na początkową wartość.</i></p> <p><i>Skutek: Poprawa wydajności oraz stylu kodu.</i></p> <p><i>Razem:4</i></p>
<i>Pokrycie funkcjonalności</i>	<i>Zwiększone</i>
<i>Pokrycie ryzyka</i>	<i>Poprawa stabilności</i>
<i>Pokrycie kodu</i>	<i>501 / 4810 ~ ok. 10.42% całości</i>
<i>Czas projektowania</i>	<i>-</i>
<i>Czas implementacji</i>	<i>15 minut</i>
<i>Czas wykonania testów</i>	<i>15 minut</i>

Nazwa, ID	Event OnInteraction(actionName : string, activator : CEntity), 10.
<i>Kod, konkretne miejsce</i>	<i>3910- 3956</i>
<i>Data</i>	<i>25. 05. 2017r. 01:30</i>
<i>Status, Poziom istotności</i>	<i>Minor, Niski</i>
<i>Zaprojektowane testy</i>	<i>Analiza statyczna kodu - przegląd, analiza warunków/decyzji</i>
<i>Liczba znalezionych defektów</i>	<p>1. <i>Puste if'y ()</i></p> <p><i>Zalecane: Usunięcie zbędnego kodu</i></p> <p><i>Skutek: Zwiększenie pokrycia kodu.</i></p> <p><i>Razem:1</i></p>

<i>Pokrycie funkcjonalności</i>	Zwiększone
<i>Pokrycie ryzyka</i>	-
<i>Pokrycie kodu</i>	46 / 4810 ~ ok. 1.16% całości
<i>Czas projektowania</i>	-
<i>Czas implementacji</i>	5 minut
<i>Czas wykonania testów</i>	5 minut

Nazwa, ID	Event OnAnimEvent_ActivateUp(animEventName : name, ...), 11.
<i>Kod, konkretne miejsce</i>	3760- 3784
<i>Data</i>	26. 05. 2017r. 11:00
<i>Status, Poziom istotności</i>	Minor, Niski
<i>Zaprojektowane testy</i>	Analiza statyczna kodu - przegląd
<i>Liczba znalezionych defektów</i>	<p>1.</p> <p>Użycie niepotrzebnego mnożenia 0.0 * VecNormalize(spawnRot.Yaw + 180) w wyliczaniu spawnPos</p> <p>Zalecane: Usunięcie dodatkowego mnożenia</p> <p>Skutek: Poprawa wydajności</p> <p>Razem: 1</p>
<i>Pokrycie funkcjonalności</i>	Zwiększoną wydajność
<i>Pokrycie ryzyka</i>	-
<i>Pokrycie kodu</i>	24 / 4810 ~ ok. 0.50% całości
<i>Czas projektowania</i>	-
<i>Czas implementacji</i>	5 minut
<i>Czas wykonania testów</i>	5 minut

Nazwa, ID	public final function TryToHideAllHorseItems() : bool, 12.
<i>Kod, konkretne miejsce</i>	4710- 4764
<i>Data</i>	26. 05. 2017r. 11:30
<i>Status, Poziom istotności</i>	Minor, Niski
<i>Zaprojektowane testy</i>	Analiza statyczna kodu - przegląd
<i>Liczba znalezionych defektów</i>	<p>1.</p> <p>Użycie zmiennych do funkcji int i, k zamiast int i, j</p> <p>Możliwość naprawy: zmiana nazwy</p> <p>Razem: 1</p>
<i>Pokrycie funkcjonalności</i>	Kod bardziej estetyczny
<i>Pokrycie ryzyka</i>	-
<i>Pokrycie kodu</i>	54 / 4810 ~ ok. 1.12% całości
<i>Czas projektowania</i>	-
<i>Czas implementacji</i>	1 minuta
<i>Czas wykonania testów</i>	-

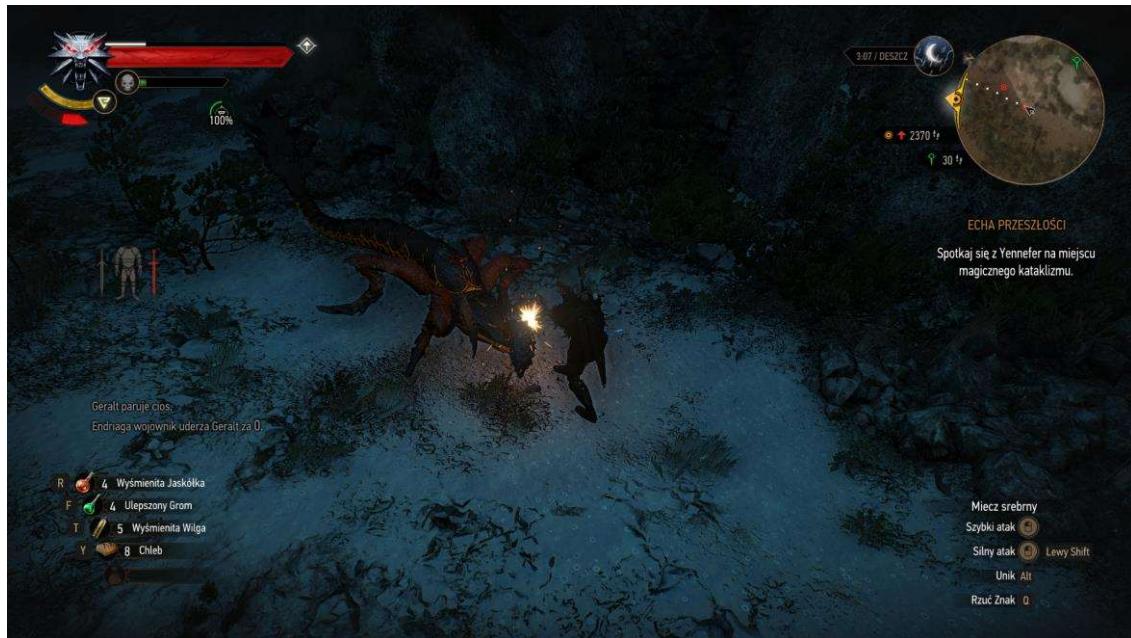
Uwaga: W pliku npc.ws znajdują się puste funkcje takie jak np.. timer funtion Tick(deltaTime : float, id : int), event OnScriptReloaded(), function UpdateVisualDebug(), eventOnInAirStarted() i wiele więcej!

Zaleca się usunięcie wskazanego problemu w celu nie popełnienia przyszłych pomyłek z wykorzystaniem pustych funkcji.

Raport o incydentach/awariach/defektach

Raport o testowaniu parowania ciosów

Wstępnie zostaje sprawdzony warunek, czy this.IsHuman() da wartość false dla postaci, które nie są ludźmi. Jeżeli IsProtecteByQuen ma wartość true, to faktycznie funkcja zwraca false. Jeżeli IsHuman da false, wartość zwraca true.



Test kontrolny nr. 1: IsProtectedByQuen = true; IsHuman = false; wynik = true;

Dla npcTarget, gdzie npcTarget.isShielded da true, funkcja zawsze zwraca true. Oznacza to poprawność funkcji dla tej zmiennej

Test kontrolny nr. 2: IsProtectedByQuen = false; IsHuman = true; IsShielded = true; wynik = true;



Testy dla zwykłych zwierząt i ludzi też pokazują dokładność funkcji.

Test kontrolny nr. 3: IsProtectedByQuen = false; IsHuman = false; isShielded = false; HasShieldedAbility = true; TargetToAttackerAngleAbs < 90 = true; wynik = true;

Test kontrolny nr. 4: IsProtectedByQuen = false; IsHuman = true; isShielded = false; HasShieldedAbility = true; TargetToAttackerAngleAbs < 90 = true; wynik = true;

Dla reszty zmiennych funkcja zwracała następujące wartości. Wszystko opisuje podana tabela:

Pokrycie warunków/decyzji

R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	Ans
True	True	True	True	True	True	False							
False	False	False	False	False	False	True							

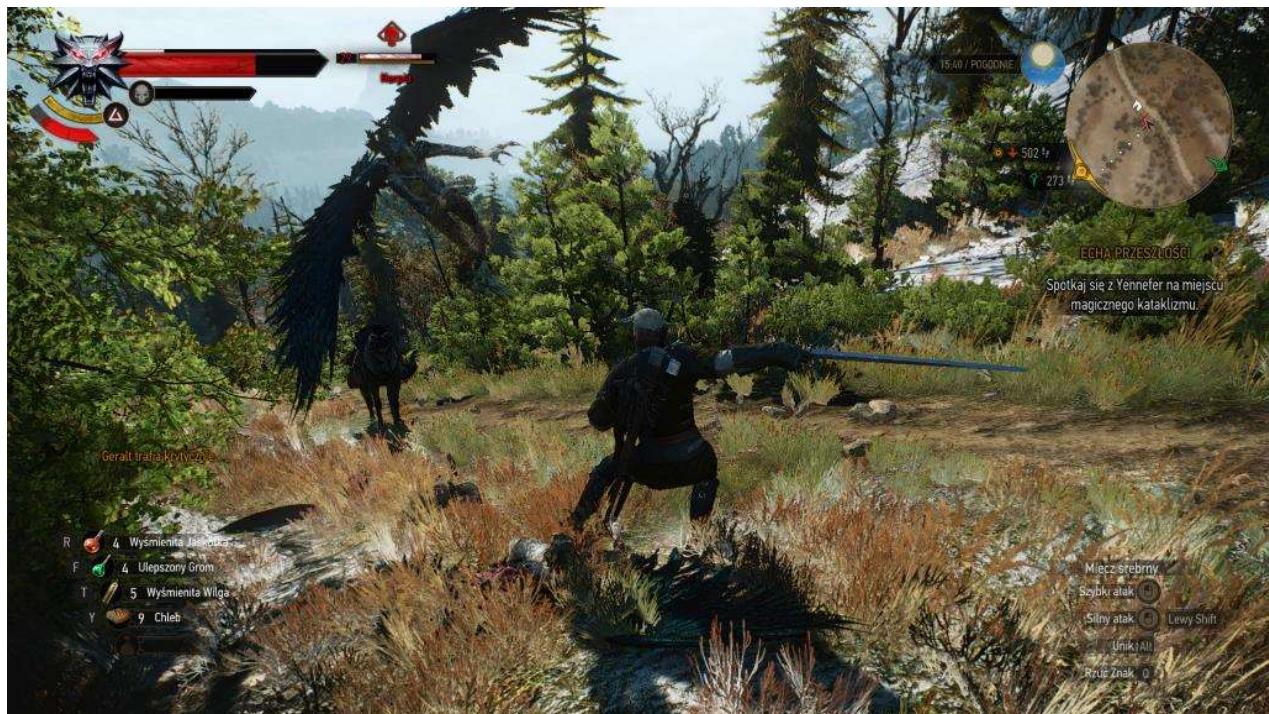
R0 – NOT(canBeParried), R1 – IsHuman, R2 – IsProtectedByQuen, R3 – NOT(CanParryAttack), R4 – NOT(FistFightCheck), R5 – IsInHitAnim, R6 – IsShielded, R7 – isHeavy, R8 – HasStaminaToParry, R9 – HasAbility(), R10 – attacker == thePlayer, R11 – attacker.IsWeaponHeld(), R12 – target.IsWeaponHeld, Ans – wynik;

Oprócz dwóch testów dla wartości/decyzji zostały zrobione 4 testy kontrolne. Wszystkie wykazały poprawność funkcji. Brak wykazanych błędów, awarii i defektów.

Testy nie wykazały aby zmienna parryInfo.targetToAttackerAngleAbs wykazywała niepoprawne działanie, choć może nastąpić szansa na pojawienie się błędu.

Raport z analizy walki:

If(1820) mimo swojego skomplikowania pokazuje, że dla sytuacji na terenach nie plażowych funkcja działa poprawnie, jednak w sytuacji gdy znajdujemy się na płytce mieliźnie podczas wypchnięcia npc do wody GetCurrentStance nie jest równe NS_Swim lub GetMovingAgentComponent nie jest równe true, a funkcja przestaje działać. Wykazuje to brak zmiany dla postaci, która przez czynnik trzeci pojawia się w wodzie. Reszta sytuacji wykazuje poprawne wyniki i pojawienie się AddEffectInfo(EET_Knockdown);



Testy:

	R1	R2	R3	R4	R5	R6	result
Walka w górach z harpią	true	true	true	true	true	true	True
Walka z nekkerem w lesie – uderzenie "aard"	true	false	true	true	true	true	True
Walka z utopcem na nabrzeżu	true	false	true	?	?	true	TRUE!
Walka z nekkerem w lesie – bez "aard"	true	false	true	true	true	false	false

R1 – percentageLoss >= healthLossToForceLand_perc.valueBase, R2 – GetCurrentStance() == NS_Fly,
R3 – NOT(IsUsingVehicle()), R4 – GetCurrentStance() NOT EQUAL NS_Swim,

R5 – NOT(((CMovingPhysicalAgentComponent) GetMovingAgentComponent()).IsOnGround()))),

R6 - NOT(damageAction.IsDoTDamage() AND NOT(damageAction.DealsAnyDamage()))

Dwa testy wykonane zostaną na stworzenie mutantów. Dla pierwszego mutanta R5 ustawimy na
NOT R5, czyli (CMovingPhysicalAgentComponent) GetMovingAgentComponent().IsOnGround()

Po pojawienniu się mutanta, wynik był odwrotny od zamierzonego: każde użycie funkcji wykonywało
wynik true dla funkcji.

Dla drugiego mutanta przyjęto NOT R4 GetCurrentStance() EQUAL NS_Swim daje ten sam efekt

Oznaczać to może, że funkcja nie zwraca uwagi na odepchnięcia do wody.

Wynik = R1 AND (R2 || (R3 AND R4 AND R5 AND R6));

TEZA:

Dla R1 = false IMP wynik = false;

Dla R2 = true IMP (R2 || (R3 AND R4 AND R5 AND R6)) = true;

Dla R2 = false IMP (R2 || (R3 AND R4 AND R5 AND R6)) = false;

R1	R2	R3	R4	R5	R6	result
true	true	true	true	true	true	True
false	false	false	false	false	false	false

W żadnym teście nie spotkano postaci, gdzie totalHealth = 0, co może wykonać dzielenie przez
zero(kod: 3635). Nie sprawia to jednak, że kodu nie należy zabezpieczyć przed niemiłą sytuacją.

Reszta kodu zadziała poprawnie, przy wykonywaniu testów.

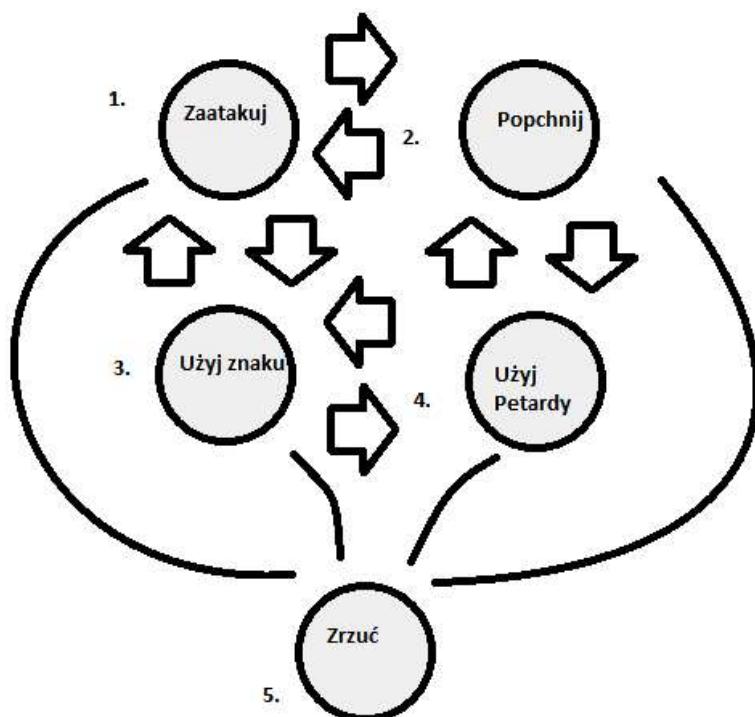
Raport analiza funkcji OnDeath

Funkcja ma złożone działanie, gdzie każdy warunek jest bardzo różny od innego. Po 10 wykonanych testach funkcja spełnia swoje założenia. Jednak dla pewnych testów jest wiele niedociągnięć.

Testy były wykonane również dzięki zastosowaniu przejść pierwzych. Mamy następujące możliwości:

Zaatakuj, popchnij, rzuć czar "aard", użyj petardy, zrzuć.

Punkty 1, 2, 3, 4 mogą się również przecinać na skos



Mając wejścia pierwsze, które kończą się zepchnięciem mamy:

1 2 3 4 5 , 1 2 4 3 5 , 1 3 2 4 5 , 1 3 4 2 5 , 1 4 2 3 5 , 1 4 3 2 5 , 2 1 3 4 5 , 2 1 4 3 5 , 2 3 1 4 5 ,
, 2 3 4 1 5 , 2 4 1 3 5 , 2 4 3 1 5 , 3 1 2 4 5 , 3 1 4 2 5 , 3 2 1 4 5 , 3 2 4 1 5 , 3 4 1 2 5 , 3 4 2 1 5 ,
2 1 5 , 4 1 2 3 5 , 4 1 3 2 5 , 4 2 1 3 5 , 4 2 3 1 5 , 4 3 1 2 5 , 4 3 2 1 5

1. Dla pewnej odległości funkcja if (aardedFlight && damageAction.GetBuffSourceName() == "FallingDamage") działa i odrzuca obiekty na pewną krótką odległość. Do szukania błędu został stworzony mutant if (aardedFlight || damageAction.GetBuffSourceName() == "FallingDamage") nic względnie nie zostaje zmienione. Zmniejsza się czas pewnych operacji związanych z dźwiękiem.
2. Nie jest dodawana ilość doświadczenia za każdym razem, gdy :
 - A) Potwór np. Latająca wiwerna spadnie przez nas do wody po odepchnięciu.
 - B) Postacie po odepchnięciu ze skarpy również nie zostają zapisane jako pokonane
 - C) Inny przeciwnik przypadkiem zaatakuje innego przeciwnika

Testy nie wskazały na uchybienia w pozostałej części funkcji.

Nie możemy znaleźć tutaj pokrycia ścieżek liniowo niezależnych, ponieważ warunki do przetestowania gry i wyznaczenia pokrycia będzie się sprawdzać wyłącznie na papierze.

Metryki:

Procent całego kodu, który zawiera defekty: 35.15%

Defekty o niskim stopniu: 17(tabela), + puste funkcje

Defekty o średnim stopniu: 4(tabela) + 3(raport)

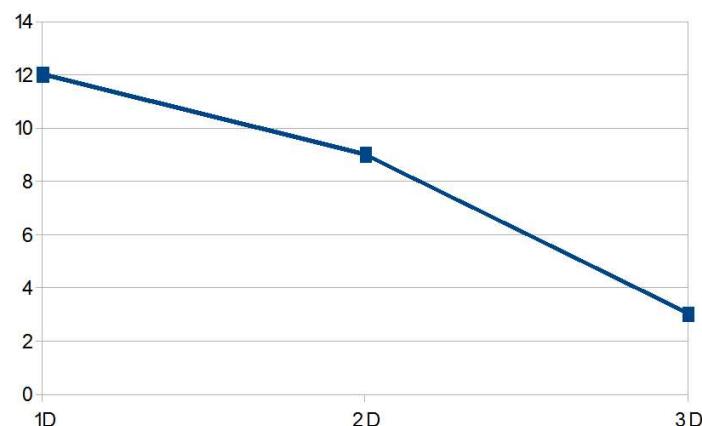
Jeżeli licząc, że każdy defekt o niskim stopniu może być naprawiony od 5 do 30 minut, a średni od 30 do 60 minut przez jednego członka zespołu. Możemy uprościć, że wszystkie naprawy powinny zająć od ok. 295 osoby/minut do górnej granicy 930 osoby/minut.

Problemy przedstawione wyżej nie wpływają zbyt mocno na problem z grywalnością. Oznacza to, że poza naprawieniem kodu gry, nie będzie potrzeby poprawy kwestii jakości i przyjemności z używania programu.

Łączna ilość testów: 18

Testy, które znalazły defekt

Ilość znajdywanych defektów w ciągu 3 dni testowania:



Uwagi:

Program testowany przez 3 dni po 6 godzin.