Studium wzorców projektowych w programowaniu aplikacji interaktywnych

(Study of design patterns in interactive applications programming)

Dominik Szczechla

Praca inżynierska

Promotor: dr Wiktor Zychla

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

20 stycznia 2017

Streszczenie

Niniejsza praca inżynierska stanowi studium często wykorzystywanych wzorców projektowych w programowaniu aplikacji interaktywnych. Składa się z przykładów praktycznego zastosowania w formie implementacji gry zręcznościowej DespatBreakout, której kod źródłowy wraz z komentarzami znajduje się w serwisie Gihub. Uzupełnieniem pracy jest krótki wstęp teoretyczny, dokumentacja wzorców, oraz uwagi autora. Celem pracy jest przygotowanie pomocy naukowej dla studentów uczących się wzorców projektowych w czasie kursów akademickich.

This engineering thesis is a study of most commonly used design patterns in interactive application programming. It consists of multiple examples of practical usage in form of an implementation of arcade game DespatBreakout, source code of is available, with comments, on Github. It is supplemented by a short theoretical introduction, documentation of the patterns, and some notes from the author. Puropose of this paper is to prepare a study help for students learning about design patterns on academical courses.

Spis treści

1. Wstęp 5
   1.1. Wykorzystane technologie 6
   1.2. Opis pracy 6
   1.3. Propozycja wykorzystania 7
2. Charakterystyka wzorców projektowych 8
3. Opis zaimplementowanych wzorców 9
   3.1. Wzorce kreacyjne 9
      3.1.1. Fabryka 10
      3.1.2. Prototyp 10
      3.1.3. Singleton 11
   3.2. Wzorce strukturalne 11
      3.2.1. Adapter 12
      3.2.2. Dekorator 13
      3.2.3. Pełnomocnik 14
   3.3. Wzorce czynnościowe 14
      3.3.1. Interpreter 14
      3.3.2. Obserwator 15
      3.3.3. Pamiątka 16
      3.3.4. Polecenie 16
      3.3.5. Stan 17
      3.3.6. Strategia 17
   3.4. Wzorce architektury gier komputerowych 18
      3.4.1. Komponent 18
      3.4.2. Metoda aktualizacji 19
      3.4.3. Pętla gry 19
   3.5. Inne użyte wzorce i metody 20
4. Podsumowanie i wnioski 21
5. Bibliografia 22

1. Wstęp

Praca współczesnego programisty najczęściej wymaga ścisłej koordynacji wykonywanych zadań z zespołem. W celu wspólnej pracy nad dużym projektem autorzy zwykli ustalać i przestrzegać określonych schematów, tak by zapewnić przejrzystość i zrozumiałość kodu – nie tylko dla swoich współpracowników, ale i dla kolejnych osób, które będą bazować na zastanym kodzie źródłowym w przyszłości. Powstała cała gałąź badań nad inżynierią oprogramowania, zajmująca się technikami organizacji pracy i dokumentacji w celu zapewnienia wymaganej niezawodności programów, jak i wydajności pracy programistów.

Jednym z narzędzi proponowanych już od lat 90 XX wieku jest stosowanie wzorców projektowych w celu rozwiązywania najczęściej napotykanych problemów. Okazuje się jednak, że w komercyjnym programowaniu nie zawsze są one szeroko używane, przez co często nawet w obrębie jednego projektu można zauważyć różniące się od siebie, czasem zaskakujące pomysłowością rozwiązania . Niestety, takie podejście zaciemnia kod i powoduje podwyższenie „progu wejścia” w projekt. Osoby niewdrożone wymagają wraz z jego rozrostem co raz więcej czasu na naukę zanim będą w stanie samodzielnie pracować, a w pewnym momencie nawet wśród osób obeznanych z projektem następuje proces specjalizacji w niektórych tylko jego fragmentach – zmiany w innych zaczynają wymagać pomocy drugiego programisty, najczęściej autora danej części programu. Powstające łańcuchy zależności stanowią słaby punkt projektu – zwłaszcza w sytuacji zmian kadrowych, kiedy nagły brak istotnego eksperta może spowodować nawet paraliż pracy zespołu.

Dlaczego zatem wzorce nie są stosowane powszechnie? Okazuje się, że wśród wielu programistów stanowią one bardziej ciekawostkę, niż realny sposób patrzenia na funkcjonalności. Wielu zna tylko kilka wzorców, i nie do końca rozumie gdzie je stosować – choć czasem zupełnie nieświadomie stosują podobne do nich konstrukcje w codziennej pracy. Inni uważają, że często implementacja wzorca jest działaniem na wyrost, i preferują rozwiązanie jak najbardziej bezpośrednie, najprostsze do napisania.

Można zadać kolejne pytanie – dlaczego programiści nie znają wzorców, skoro na przedmiotach typu Inżynieria Oprogramowania czy Projektowanie Obiektowe Oprogramowania są one omawiane. Wydaje się, że ze strony teoretycznej studenci znają poszczególne wzorce, ale nie wykorzystują ich w praktyce – przez co wiedza o nich nie zostaje utrwalona. Podejście czysto teoretyczne do nauki wzorców pozostawia istotne luki – brak punktu zaczepienia w postaci większego, kompletnego projektu sprawia, że ćwiczenia w implementacji wzorców stają się „sztuką dla sztuki” i nie dają wskazówek co do realnego zastosowania danego wzorca. Zdarza się, że zrozumienie tematu jest płytkie – bazuje raczej na pamięciowym opanowaniu kilku definicji. Brakuje także metod rozeznania – kiedy warto wykorzystać wzorzec, jaki wzorzec wykorzystać w niejasnej sytuacji, w końcu jak dostosować go do aktualnych potrzeb? Niniejsza praca stara się odpowiedzieć na te pytania, prezentując kilkanaście popularnych wzorców projektowych w implementacji gry zręcznościowej. Przykład gry wydaje

się dobrze działać na wyobraźnię studentów i zachęca do tematu, dzięki czemu pozwala lepiej zrozumieć i zapamiętać dany wzorzec. Istotą pracy jest kod źródłowy – przygotowany w konsultacji z promotorem, ma za zadanie w czytelny i przejrzysty sposób prezentować opisywane zagadnienie.

1.1. Wykorzystane narzędzia i technologie

- Język C# i środowisko Microsoft Visual Studio 2013

- Platforma .NET 4.5

- Framework Monogame w wersji 3.5 (http://www.monogame.net/)

  Monogame początkowo było rozszerzeniem dla narzędzi XNA oferowanych przez Microsoft do tworzenia gier na ich platformy. Jego istotą było wsparcie dla pozostałych platform (obecnie wspierane: iOS, Android, MacOS, Lunux, wszystkie platformy Windows, OUYA, PS4, PSVita, XBOX. Microsoft przestał już rozwijać XNA, natomiast open-source'owy Monogame jest ciągle rozwijany i aktualizowany, przez co pozostaje solidnym narzędziem do programowania gier w językach C# i .NET.

- FxCop

  FxCop jest wbudowanym w środowisko Visual studio narzędziem analizującym kod wynikowy. Dostarcza ono wskazówek na temat projektowania, zabezpieczeń i szybkości działania aplikacji, proponując zmiany w wypadku łamania przez nią proponowanych przez firmę Microsoft zasad tworzenia szybkiego i łatwego w utrzymaniu kodu.

- StyleCop

  StyleCop jest narzędziem analizującym kod źródłowy. Zapewnia informacje o estetycznych i projektowych błędach w kodzie takich jak niepoprawne nawiasowanie, nieścisłości w nazywaniu zmiennych, braki w komentarzach itd.

- GitHub

  GitHub jest serwisem umożliwiającym dzielenie się repozytoriami zarządzanymi przez popularny system kontroli wersji Git.

1.2. Opis pracy

Niniejsza praca obejmuje implementację gry zręcznościowej typu Arkanoid z wykorzystaniem wzorców projektowych i dobrych praktyk programowania. W implementacji wykorzystano dwanaście uniwersalnych wzorców projektowych, oraz trzy typowe dla programowania gier.

Najważniejszą częścią pracy jest kod źródłowy z komentarzem, znajdujący się w repozytorium GitHub https://github.com/Randil/despat-studio. W repozytorium znajduje się dodatkowo dokumentacja – streszczenie niniejszej pracy oraz pełna treść rozdziałów 2 i 3. Kopia źródeł znajduje się wśród załączników do pracy.

W grze DespatBreakout zaimplementowano następujące elementy:

- menu główne z dostępem do ekranu Osiągnięć i Instrukcji;
- menu wyboru misji – lista dostępnych scenariuszy jest przechowywana w pliku XML, również poszczególne poziomy są przygotowane poprzez napisanie scenariusza w pliku XML i parsowane w trakcie wykonania programu;
- silnik gry – poruszająca się paletka o zmiennych rozmiarach, piłeczki z różnymi modelami kolizji, różne typy klocków, bonusy do zbierania, zapis stanu gry, wyzwania w postaci pomiaru czasu ukończenia poziomu i klasy zbierającej dane o osiągnięciach.

Krótki filmik przedstawiający grę w działaniu można znaleźć pod adresem: https://www.youtube.com/watch?v=DWUjRW9Zmwg.

1.3. Propozycja wykorzystania

Program i dokumentację przygotowano z myślą o wykorzystaniu w zajęciach Projektowanie Obiektowe Oprogramowania, Inżynieria Oprogramowania, Programowanie Obiektowe – i im podobnych, jako dodatkową pomoc naukową. Dokumentacja znajdująca się w serwisie GitHub zawiera treści typowe dla notatki z wykładu na temat wzorców – opis teoretyczny, schemat klas, dodatkowe wskazówki na temat implementacji. W opisie każdego wzorca znajdują się odnośniki do poszczególnych klas implementujących dany wzorzec. W kodzie znajdują się komentarze wyjaśniające trudniejsze momenty i oznaczające które fragmenty należą do implementacji danego wzorca.

2. Charakterystyka wzorców projektowych

Wzorzec projektowy (ang. *design pattern*) – opis uniwersalnego rozwiązania często powtarzającego się w programowaniu obiektowym problemu. Wzorce starają się opisać sposób podejścia do sytuacji, nie definiują jednoznacznie konkretnej implementacji. Stosowanie zunifikowanych wzorców w projektowaniu rozwiązań podobnych problemów ułatwia tworzenie i zarządzanie kodem w projekcie.

Wzorce projektowe zostały spopularyzowane przez książkę „Inżynieria oprogramowania: Wzorce projektowe" autorstwa tzw. „Bandy czterech" – Ericha Gammę, Richarda Helma, Ralpha Johnsona i Johna Vlissidesa. Opisali oni większość popularnych do dzisiaj wzorców projektowych, oraz zaproponowali układ opisu nowych.

By przedstawić prawidłowo definicję danego wzorca projektowego należy przede wszystkim opisać:

- problem – przedstawienie możliwie jednoznacznego sposobu na rozpoznanie sytuacji, do rozwiązania której proponowane jest użycie określonego wzorca. Dobrze zapewnić przykład problemu, jak również opisać sytuacje w których danego wzorca nie należy stosować;
- rozwiązanie – opis metody podejścia do problemu, proponowane klasy i obiekty, ich powiązania i sposób działania. Pomóc może tutaj diagram w UML i przykład kodu źródłowego;
- konsekwencje – wady i zalety rozwiązania, uzasadnienie wyboru akurat tej metody w danej sytuacji, odnośniki do alternatywnych rozwiązań, podobnych wzorców.

Niniejsza praca nie przedstawia dokładnego, teoretycznego opisu prezentowanych wzorców – istnieje fachowa literatura skupiająca się na tym temacie, odnośniki do kilku pozycji znajdują się w bibliografii. Zamiast tego stawia na przedstawienie wzorców w działaniu, dodając również najistotniejsze uwagi z wykorzystanych źródeł, jak również wnioski które nasunęły się autorowi podczas pracy z danym wzorcem.

3. Opis zaimplementowanych wzorców

3.1. Wzorce kreacyjne – opisują sposoby tworzenia i inicjalizacji obiektów.

3.1.1. Fabryka (ang. *Factory*)

Wzorzec którego celem jest zapewnienie interfejsu do wygodnego tworzenia obiektów z jednej rodziny.

W projekcie: Klasa MissionParser jest fabryką przygotowującą zestaw klocków na potrzeby obiektu MissionScreen. Parser pobiera wszystkie dane do inicjalizacji wytwarzanych obiektów z pliku XML, tworzy klocki odpowiednich typów, dodaje je do listy, i zwraca gotowy obiekt typu BrickWall.

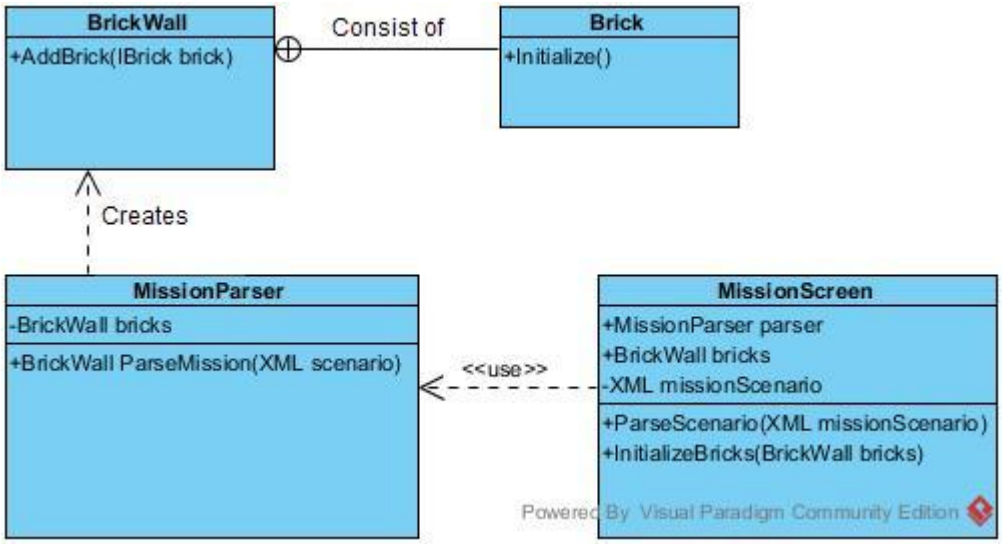

Diagram 1: Fabryka

Uwagi:

- Utworzenie oddzielnej klasy – fabryki jest szczególnie wygodne, gdy tworzenie danego obiektu jest skomplikowane, np. wymaga stworzenia wielu innych obiektów.
- Fabryka może współdziałać z Prototypem, tworząc nowe obiekty na podstawie gotowych instancji.
- Stosowanie fabryki utrudnia rozszerzanie rodziny obiektów, każde dziedziczenie, dekorator czy nowy interfejs muszą być w niej uwzględniane. Należy rozważyć, czy w danej sytuacji fabryka będzie najlepszym podejściem.

3.1.2. Prototyp (ang. *Prototype*)

Wzorzec umożliwiający tworzenie nowych instancji danej klasy dzięki kopiowaniu już istniejących obiektów.

W projekcie: Aktywacja bonusu BonusCommandMultiball powoduje wywołanie metody Duplicate() dla każdej piłeczki z listy balls. Klasa Ball zwraca dokładne kopie piłeczek, następnie BonusCommandMultiball nieco zmienia trajektorię ich lotu, i dodaje je do listy w klasie MissionScreen.

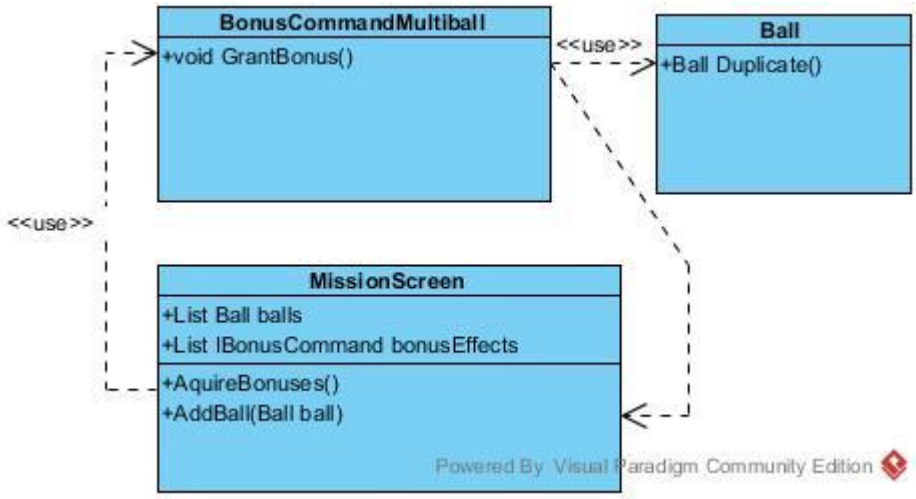

Diagram 2: Prototyp

Uwagi:

- Wzorzec nie specyfikuje metody implementacji kopiowania. Klasa duplikowana może być – jak w przykładzie - czynna (instancja „kopiuje się sama”), lub bierna („daje się skopiować”).
- Prototyp jest bardzo przydatny w sytuacjach gdy tworzonych jest wiele skomplikowanych, ale podobnych do siebie elementów. Można go połączyć np. z fabryką (fabryka przechowuje kilka prototypów obiektów i serwuje ich kopie).
- Należy pamiętać, które pola w duplikacie mają być wskaźnikami do TYCH SAMYCH obiektów, a które tylko do TAKICH SAMYCH (operator New).

3.1.3. Singleton (ang. *Singleton*)

Wzorzec ograniczający ilość instancji obiektów danej klasy do jednej, oraz zapewniający do niej globalny dostęp.

W projekcie: Singletonem jest klasa DespatBreakout. To główna klasa gry, agregująca wszystkie pozostałe elementy, odpowiedzialna za obsługę silnika Monogame. Do jej publicznych pól mają dostęp wszystkie pozostałe klasy.

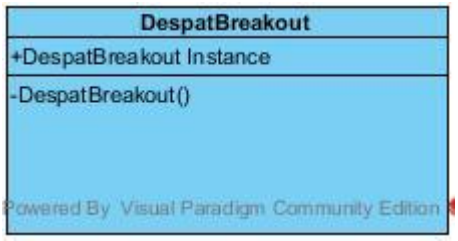

Diagram 3: Singleton

Uwagi:

- Nie należy stosować singletonów wszędzie tam, gdzie przewiduje się istnienie tylko jednej instancji danej klasy, należy także uważać na globalny dostęp do pól. Nadużywanie tego wzorca jest błędem, uważanym za „antywzorzec projektowy".
- Szczególnej uwagi wymaga stosowanie singletonów w systemach wielowątkowych - równoczesne zażądanie instancji przez dwa wątki może spowodować utworzenie dwóch instancji single tonu.

3.2. Wzorce strukturalne – opisują struktury współpracujących ze sobą obiektów.

3.2.1. Adapter (ang. *Adapter*)

Adapter to wzorzec stosowany w wypadku importowania do projektu klas z zewnątrz, dla zapewnienia zgodności interfejsów z istniejącym kodem.

W projekcie: Na potrzeby prezentacji wzorca została skopiowana klasa BrickImported z open-source'owej implementacji Arkanoida (można znaleźć ją pod adresem: http://xnarkanoid.codeplex.com/SourceControl/latest#XNArkanoid_2010.
Klasa BrickImportedAdapter zawiera instancję BrickImported, i korzysta z jej metod w celu implementacji interfejsu IBrick.

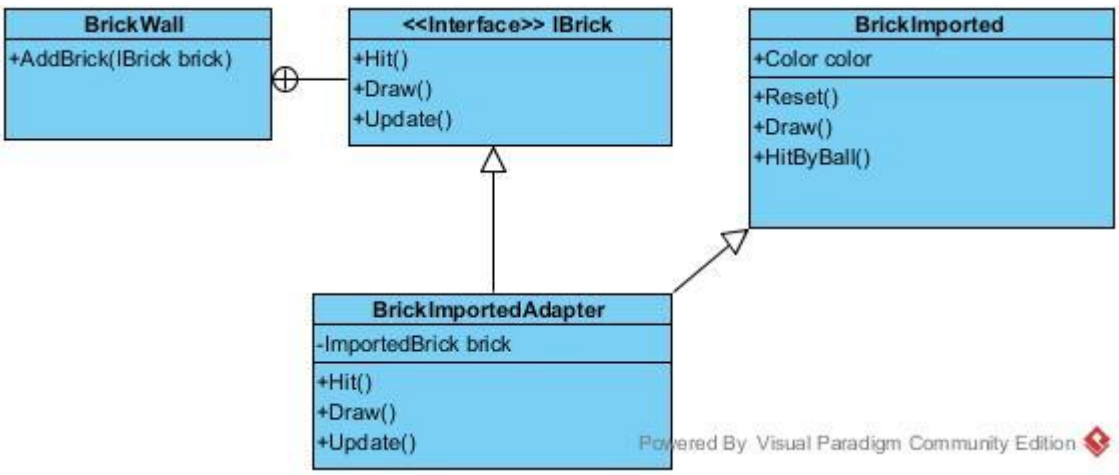

Diagram 4: Adapter

Uwagi:
- Wzorzec znajduje zastosowanie jedynie w celu uzgadniania interfejsów.

3.2.2. Dekorator (ang. *Decorator*)

Wzorzec, którego celem jest elastyczne rozszerzanie odpowiedzialności obiektu, również w czasie działania programu. Alternatywa dla dziedziczenia.

W projekcie: Różne rodzaje klocków są reprezentowane przez „bazową" klasę Brick (i równorzędne ImportedBrickAdapter), oraz zestaw dekoratorów. Instancje dekoratora zawierają wskaźnik na „dekorowany" obiekt, i korzystają z jego metod w celu realizacji interfejsu IBrick, dodając przy tym nowe funkcje. Dekoratory klocków można stosować w dowolnej kolejności i kombinacjach, tworząc np. niewidzialny, wytrzymujący kilka ataków klocek, który po zbiciu spowoduje przyznanie bonusu.

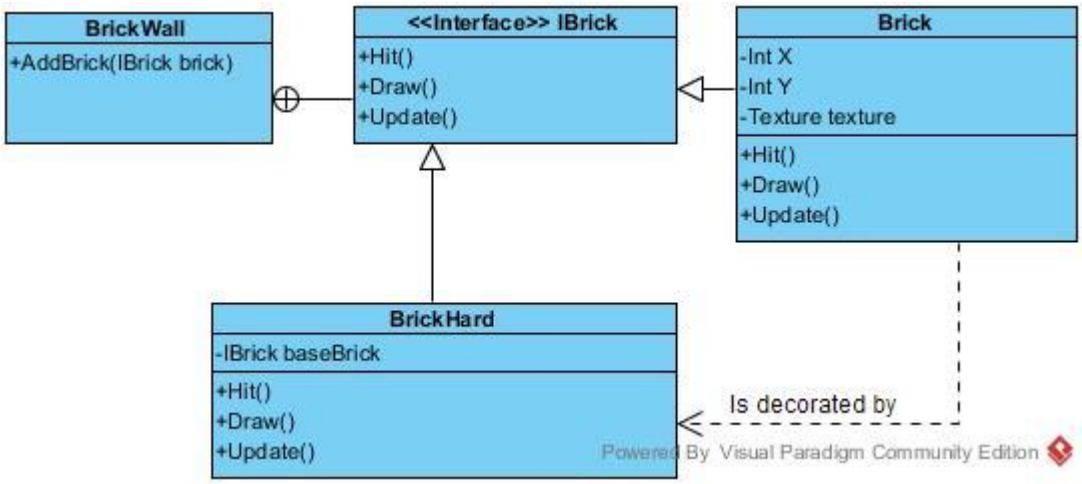

Diagram 5: Dekorator

Uwagi:
- Dekoratory są bardziej elastyczne niż dziedziczenie, dzięki możliwości zastosowania kilku na raz.
- Przy tworzeniu hierarchii klas należy rozważać różne podejścia. Czasami może okazać się, że najlepiej będzie nie dziedziczyć i nie stosować dekoratorów – pojedyncza, rozbudowana klasa bazowa to nie zawsze błąd!

3.2.3. Pełnomocnik (ang. *Proxy*)

Proxy zakłada utworzenie obiektu zastępującego inny obiekt. Wyróżniamy kilka odmian tego wzorca, w zależności od celu jego zastosowania:

- wirtualne – steruje dostępem do kosztownych obiektów („leniwe" tworzenie obiektu);
- zabezpieczające – sprawdza, czy klasa ma prawo wywołać dany obiekt, implementuje uwierzytelnianie itd.;
- zdalne – reprezentuje obiekty, które znajdują się poza lokalną maszyną;
- logujące – odnotowuje dostęp do obiektu, tworząc logi, czasem może wykonywać inne akcje podczas dostępu do obiektu.

W projekcie: Klasa MissionScreen wywołuje MissionFinishedProxy kiedy zostaną spełnione warunki końca misji. Jest to Proxy typu logującego – informuje o zakończeniu misji klasę AchievementsManager, oraz inicjalizuje obiekt FinishScreen.

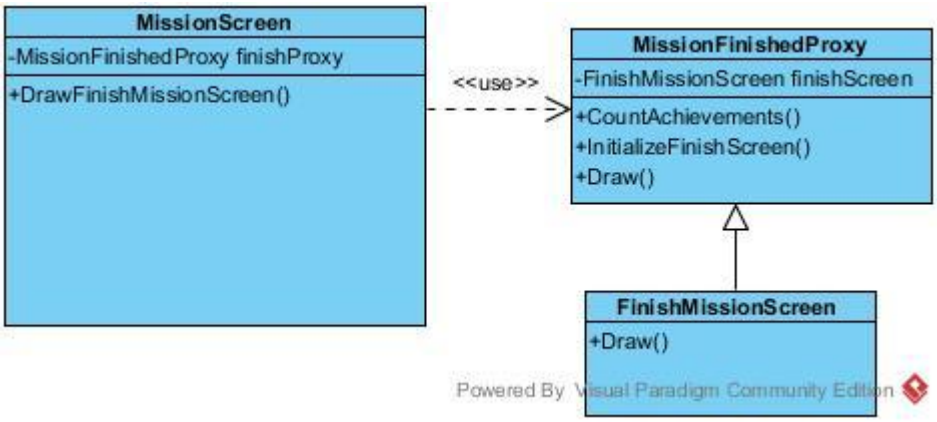

Diagram 6: Pełnomocnik

Uwagi:

- Proxy logujące można często zastąpić wzorcem Obserwatora.
- Proxy wirtualne i zdalne są dobre do optymalizacji działania programu – jego interfejs można zaprojektować tak, by niektóre metody nie wymagały faktycznego tworzenia czy pobierania obiektu na którym działamy – np. w czasie gry poprzez sieć program działa na lokalnym pełnomocniku, a synchronizację ze zdalnym obiektem wykonuje tylko co jakiś czas, dzięki czemu działa płynniej, zmniejszając wpływ pingu na szybkość reakcji.

3.3. Wzorce czynnościowe – opisują zachowanie i podział odpowiedzialności przy współpracujących ze sobą obiektach.

3.3.1. Interpreter (ang. *Interpreter*)

Wzorzec zakładający rozwiązanie problemu jako stworzenie opisu gramatyki pewnego interpretowalnego języka, i zaprojektowanie interpretera, parsera tego języka.

W projekcie: Dla zapewnienia elastyczności i łatwości projektowania nowych poziomów – opis danej misji, np. początkowe ustawienie klocków – są przechowywane w plikach XML o określonej strukturze. Klasa MissionParser zawiera zasady interpretacji tych plików, i jest jednocześnie fabryką obiektów które opisuje scenariusz misji.

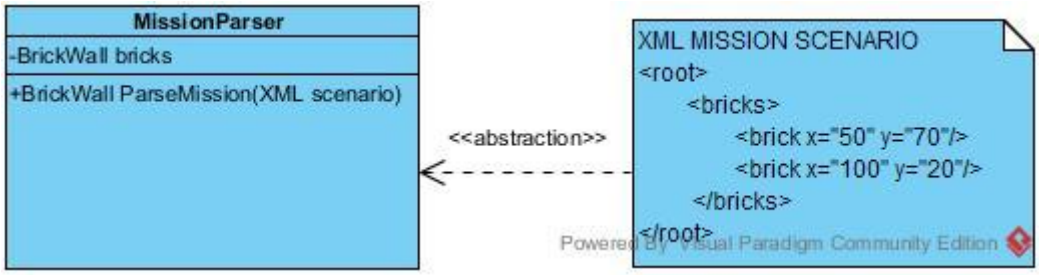

Diagram 7: Interpreter

Uwagi:

- Wzorzec powszechnie wykorzystywany w kompilatorach.
- Znajduje zastosowanie wszędzie, gdzie instancję problemu możemy opisać jako pewne wyrażenie o nieskomplikowanej strukturze.
- XML jest popularnym językiem zapisywania i przechowywania np. konfiguracji programu, większość języków programowania posiada wygodne mechanizmy przeglądania i modyfikacji plików tego typu.

3.3.2. Obserwator (ang. *Observer*)

Wzorzec opisujący sposób informowania obiektów o zmianie stanu innego obiektu.

W projekcie: O zniszczeniu obserwowanego klocka (IBrick) należy poinformować instancję BrickWall która usunie dany klocek z listy, oraz klasę AchievementsManager służącą do zbierania informacji o osiągnięciach gracza. Obie te klasy implementują interfejs IBrickObserver, dzięki czemu klocek może przechowywać wskaźniki do zainteresowanych nim obiektów w liście, i bez wiedzy o szczegółach ich implementacji poinformować o swoim stanie.

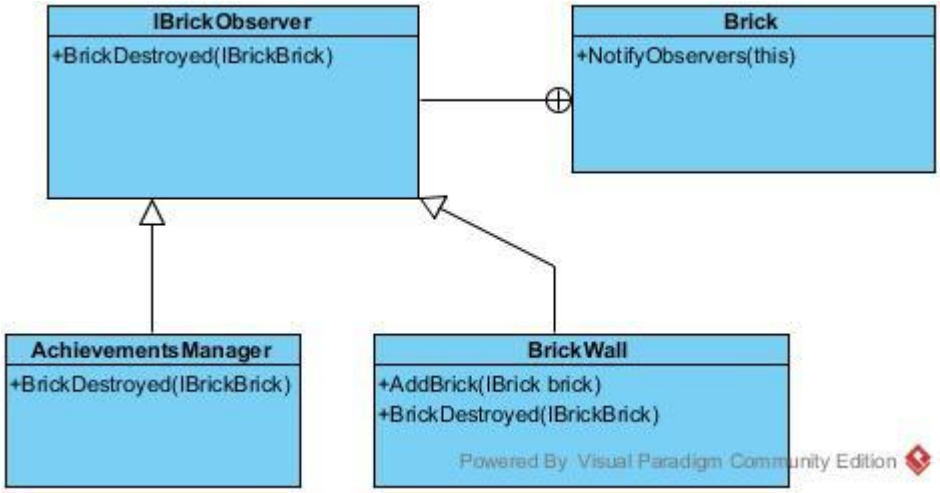

Diagram 8: Obserwator

Uwagi:

- Przydatny wzorzec w skomplikowanych systemach, zmniejsza powiązanie między Obserwowanym a Obserwatorem, ułatwiając modyfikacje.
- Podobny wzorzec: Event Aggregator – rozszerza obserwatora, kierując informowaniem o określonych wydarzeniach dużą liczbę „obserwujących" - i posiadając wiele „obserwowanych" obiektów.

3.3.3. Pamiątka (ang. *Memento*)

Ten wzorzec służy do zapamiętywania stanu danego obiektu czy obiektów. Tworzone obiekty – pamiątki nie muszą bezpośrednio odwzorowywać stanu zapamiętywanego obiektu, w szczególności duże obiekty mogą tworzyć małe, przyrostowe pamiątki.

W projekcie: W wielu grach naturalnym wykorzystaniem Memento jest funkcja zapisu stanu gry. W DespatBreakout znajdziemy klasę MissionSave, która przechowuje stan gry, jeśli w jej czasie wciśniemy przenoszący do menu głównego klawisz Escape. MissionSave nie kopiuje całego stanu rozgrywki, a jedynie dane wystarczające do jego odtworzenia. Istnienie zapisanego stanu gry powoduje dodanie nowej kontrolki do menu wyboru misji – taki poziom jest inicjalizowany obiektem MissionSave, a nie jak większość – scenariuszem w formie XML.

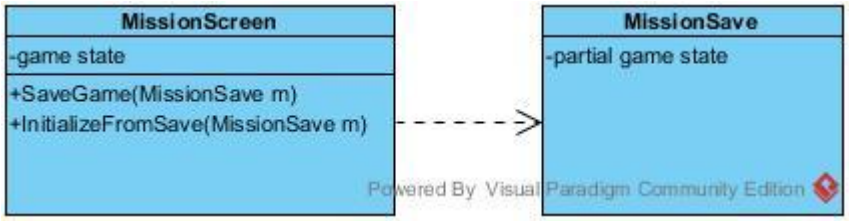

Diagram 9: Pamiątka

Uwagi:

- W implementacji Memento należy szukać balansu między ograniczeniem zużycia pamięci - a wygodą tworzenia pamiątki. Pamiątki małych obiektów mogą być po prostu ich kopiami.
- Wzorzec wykorzystywany np. do opcji undo/redo w różnych edytorach.
- Pamiątka nie musi zakładać serializacji danych do pliku – np. zawartość schowka w większości edytorów tekstu znika po zamknięciu programu.

3.3.4. Polecenie (ang. *Command*)

Wzorzec Polecenia zakłada traktowanie wywołań pewnych funkcji jako obiektów, dzięki czemu można je np. kolejkować, łączyć w listy itd.

W projekcie: Zebranie spadającej ikony bonusu nie powoduje jego wywołania od razu, a dodaje obiekt typu IBonusCommand do listy w instancji MissionScreen. Wszystkie bonusy zostają uruchamiane w metodzie Update(). Dzięki takiemu podejściu możliwa jest np. łatwa inicjalizacja misji z określonym zestawem bonusów na start, bez potrzeby przerabiania większej ilości kodu.

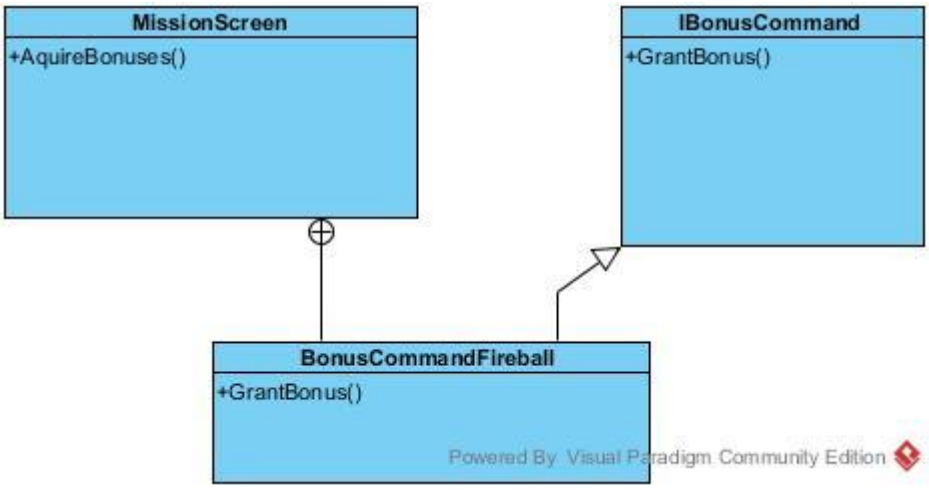

Diagram 10: Polecenie

Uwagi:

- Uelastycznia obsługę różnych wydarzeń. Przykładowo: łatwo zmodyfikować BonusCommand tak, by po upłynięciu określonego czasu cofał przyznany bonus.

3.3.5. Stan (ang. *State*)

Wzorzec zakłada zmianę zachowania obiektu w momencie zmiany informacji o jego „stanie”.

W projekcie: Klasa będąca punktem centralnym programu – DespatBreakout – zawiera zmienną currentGameState, reprezentującą aktualny stan programu. Możliwe stany to MainMenu, Mission, Achievements, MissionChoice, Tutorial. Każdy stan odpowiada jednemu z ekranów gry. Metody Update() i Draw() tej klasy zawierają różne zachowania, zależne od stanu zmiennej, odpowiednio aktualizując i rysując odpowiedni ekran.

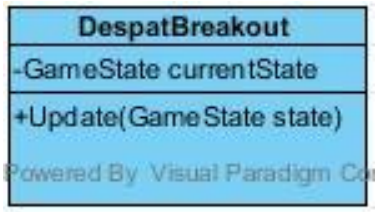

Diagram 11: Stan

Uwagi:

- Można łączyć ze wzorcem Strategia, ładując odpowiedni obiekt Strategii w zależności od aktualnego Stanu.
- Maszyna stanowa to popularny wzorzec przy tworzeniu interfejsów użytkownika – przykładowe stany elementów UI to isHoovered, isVisible, isClicked – i tak dalej.

3.3.6. Strategia (ang. *Strategy*)

Kapsułkuje algorytm działania w formie obiektu. Pozwala podawać obiektowi różne strategie działania w zależności od potrzeb.

W projekcie: Algorytm wykrywania kolizji został opakowany w klasy implementujące interfejs ICollisionStrategy. Każda piłeczka posiada strategię kolizji, która w czasie gry może się zmienić. Zaimplementowano zwykłą strategię, StrategyFireball polegające na natychmiastowym niszczeniu klocków bez odbijania, oraz StrategyFlyBy – testowa strategia, bez kolizji z klockami i bez możliwości spadnięcia piłeczki.

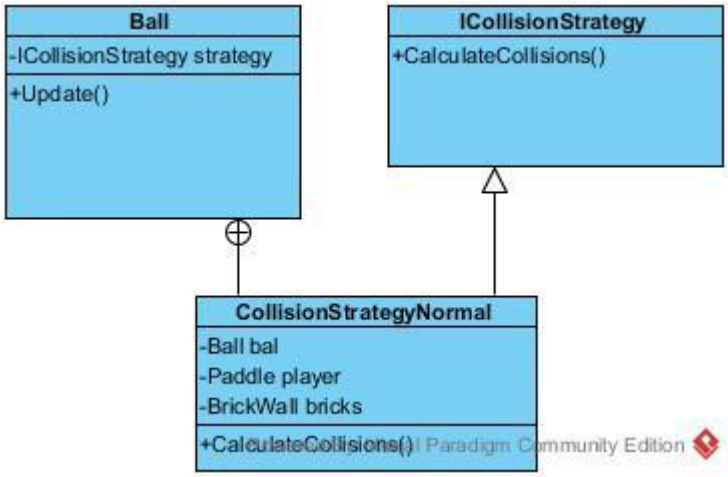

Diagram 12: Strategia

Uwagi:

- Znajduje zastosowanie wszędzie, gdzie oczekiwane są różne działania w zależności od sytuacji, a różnice nie są one trywialne (patrz: wzorzec Stan).
- Naturalne rozwiązanie przy projektowaniu algorytmów sztucznej inteligencji.

3.4. Wzorce architektury gier komputerowych

– niektóre wzorce projektowe są typowe dla programowania gier, i tam znajdują najszersze zastosowanie. Mogą one należeć do jednego z trzech przedstawionych wcześniej typów lub definiować zupełnie nowe. Na potrzeby pracy zostały przedstawione oddzielnie, ponieważ z reguły nie są używane poza programowaniem gier.

3.4.1. Komponent (ang. *Component*)

Zakłada podział abstrakcyjnych obiektów na mniejsze komponenty realizujące różne ich odpowiedzialności.

W projekcie: Kilka elementów zostało rozdzielonych na mniejsze części podczas realizacji innych wzorców – przykładowo silnik odbijania się piłeczki. Wzorcowa prezentacja to np. sposób przedstawienia paletki gracza, która składa się z dwóch komponentów – Paddle odpowiedzialna za wyświetlanie i aktualizację pozycji paletki, oraz PaddleSteering, odpowiedzialna za obsługę I/O.

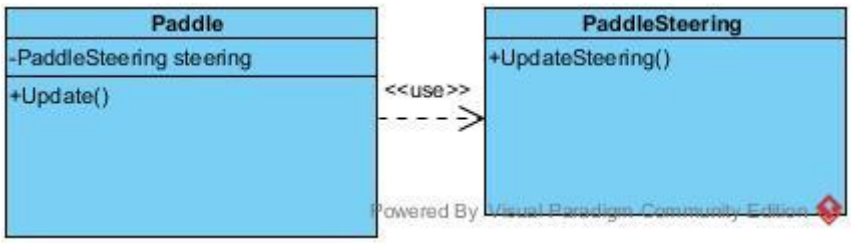

Diagram 13: Komponent

Uwagi:
- Warto podział przeprowadzić zgodnie z odpowiedzialnością poszczególnych części – osobno obsługę dźwięków, grafiki, sterowania, fizyki itd., w zależności od stopnia złożoności obiektu. Dzięki temu osoba modyfikująca jeden z nich nie musi mieć szczegółowej wiedzy o pozostałych elementach.
- Więcej o wzorcu można przeczytać na stronie internetowej: http://gameprogrammingpatterns.com/component.html.

3.4.2. Metoda aktualizacji (*ang. Update method*)
Symuluje działanie obiektów poprzez aktualizowanie ich po jednej klatce.

W projekcie: Framework Monogame domyślnie obsługuje metodę Update(). Aktualizacja obiektów jest wykonywana tak wiele razy na sekundę, jak tylko pozwala na to prędkość procesora. Główna klasa DespatBreakout wywołuje aktualizację dla elementów gry zgodnie ze swoim Stanem.

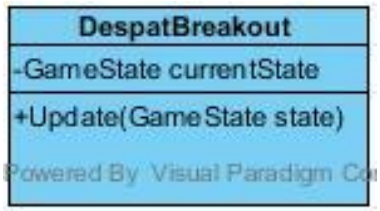

Diagram 14: Metoda aktualizacji

Uwagi:
- Przy projektowaniu metody Update należy zwracać uwagę na odpowiednie przeliczanie zmian w zależności od kwantu czasu jaki minął od poprzedniej klatki – w przeciwnym razie prędkość gry będzie zależeć od prędkości procesora.
- Więcej o wzorcu można przeczytać na stronie internetowej: http://gameprogrammingpatterns.com/update-method.html.

3.4.3. Pętla gry (*ang. Game loop*)
Sposób działania gry polegający na zapętleniu obsługi wejścia/wyjścia, aktualizacji stanu i renderowania gry.

W projekcie: Pętla gry jest zintegrowaną częścią frameworku Monogame. Metoda Update() – patrz poprzedni wzorzec – wykonuje się najczęściej jak to możliwe, zaś Draw() domyślnie 30 lub 60 razy na sekundę. Wewnątrz głównej pętli gry uruchamiana jest metoda Update() i Draw() dla każdego elementu gry który należy przeliczyć lub narysować na ekranie.

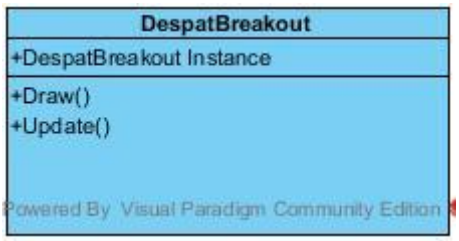

Diagram 15: Pętla gry

Uwagi:

- Podobnie jak w Metodzie Aktualizacji – należy pamiętać o odpowiedniej obsłudze czasu.
- Więcej o wzorcu można przeczytać na stronie internetowej: http://gameprogrammingpatterns.com/game-loop.html.

3.5. Inne użyte wzorce i metody

Poza wzorcami projektowymi, w kodzie gry szeroko wykorzystywane są paradygmaty programowania obiektowego – dziedziczenie, interfejsy, polimorfizm.

4. Podsumowania i wnioski

W toku tworzenia gry okazało się, że faktycznie wiele spośród wzorców idealnie wpasowuje się w projektowane funkcjonalności, niektóre są wręcz naturalnym sposobem rozwiązania danego problemu. Wydaje się to logiczne – często najprostsze wyjście jest właśnie tym pożądanym.

W kilku miejscach specjalnie nagięto lub rozszerzono strukturę programu, w celu zaprezentowania przykładu działania określonego wzorca projektowego. Choć czasem zastosowanie go w konkretnej sytuacji może być przez to kontrowersyjne – kiedy istnieją prostsze rozwiązania – ale przypadki tego rodzaju są prawidłową implementacją, pokazują, jak należy oprogramować wzorzec, budują odpowiednią intuicję – co było celem niniejszej pracy. Miejsca takie opatrzone zostały odpowiednim komentarzem, ze wskazówkami co do wykorzystania danego wzorca w różnych sytuacjach.

Na etapie zbierania informacji autor zapoznał się z wieloma popularnymi wzorcami, jednak nie było możliwe zaprezentowanie wszystkich w tak niewielkim projekcie – i nie było to celem pracy. Przy korzystaniu z wzorców ukazały się ich wady i zalety – jak okazuje się, ich wykorzystanie wymaga dyscypliny, a także zrozumienia danego wzorca. Choć na początku wymagało to skupienia i czasem przeróbek działającego kodu – z czasem systematyczne podejście nagradzało łatwością rozszerzania programu o kolejne funkcjonalności i odmiany istniejących.

Po kilku tygodniach pracy nad projektem zakładającym szerokie wykorzystanie wzorców projektowych nasuwają się wnioski, że warto się ich uczyć i z nich korzystać. Choć nie są niezbędnym elementem warsztatu informatyka, stanowią z pewnością przydatne narzędzie – tym lepsze, im bardziej złożony jest projekt, co czyni je szczególnie dobrym dla programistów dużych, korporacyjnych projektów.

Pozostaje odpowiedzieć na pytanie, czy projekt spełnił założenia i jest dobrą pomocą naukową dla osób uczących się wzorców projektowych. W czasie jego tworzenia nie korzystano – z braku źródeł – z żadnego podobnego opracowania, wzorowano się na podręcznikowych przykładach podanych w literaturze i dostępnych w Sieci – nie zawsze były one dla jasne, z czego wynikała konieczność samodzielnej pracy nad zrozumieniem danego wzorca. Gotowa praca zawiera liczne podpowiedzi i wyjaśnienia wątpliwości, które nurtowały autora w czasie programowania – tym samym stanowiąc dobrą pomoc naukową, choć niekoniecznie wystarczająca do nauki. Zgłębienie tematu wymaga również dobrego poznania teorii – której niniejsza praca zawiera jedynie niezbędne minimum – oraz przede wszystkim długotrwałych ćwiczeń w używanie wzorców w prawdziwych projektach.

5. Wykorzystane materiały, bibliografia

W programie wykorzystane zostały grafiki i fragmenty kodu publikowane na licencji Public Domain:

1. assets.png - http://opengameart.org/content/arkanoid-assets
2. greySheet.png, greySheet.xml - http://opengameart.org/content/ui-pack
3. BrickNormal.png, BrickShadow.png, BrickStone.png, Brick.cs (jako BrickImported) - http://xnarkanoid.codeplex.com/SourceControl/latest#XNArkanoid_2010/XNArkanoid/

Przy tworzeniu kodu i pracy korzystano z następujących źródeł:

1. E.Gamma, R.Helm, R.Johnson, J. M. Vlissides, *Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku,* Helion, Gliwice, 2010.
2. C.Larman, *UML i wzorce projektowe. Analiza i projektowanie obiektowe oraz iteracyjny model wytwarzania aplikacji. Wydanie III,* Helion, Gliwice, 2011.
3. Dokumentacja frameworka Monogame: http://www.monogame.net/documentation/?page=main
4. Dokumentacja FxCop i narzędzi XNA: https://msdn.microsoft.com/en-us/library/
5. Dokumentacja StyleCop: https://stylecop.codeplex.com/
6. Portal Gameprogrammingpatterns: http://gameprogrammingpatterns.com/