Wzorce Projektowe - Projekt

Monika Wielgus

Styczeń 2021

1 Opis projektu

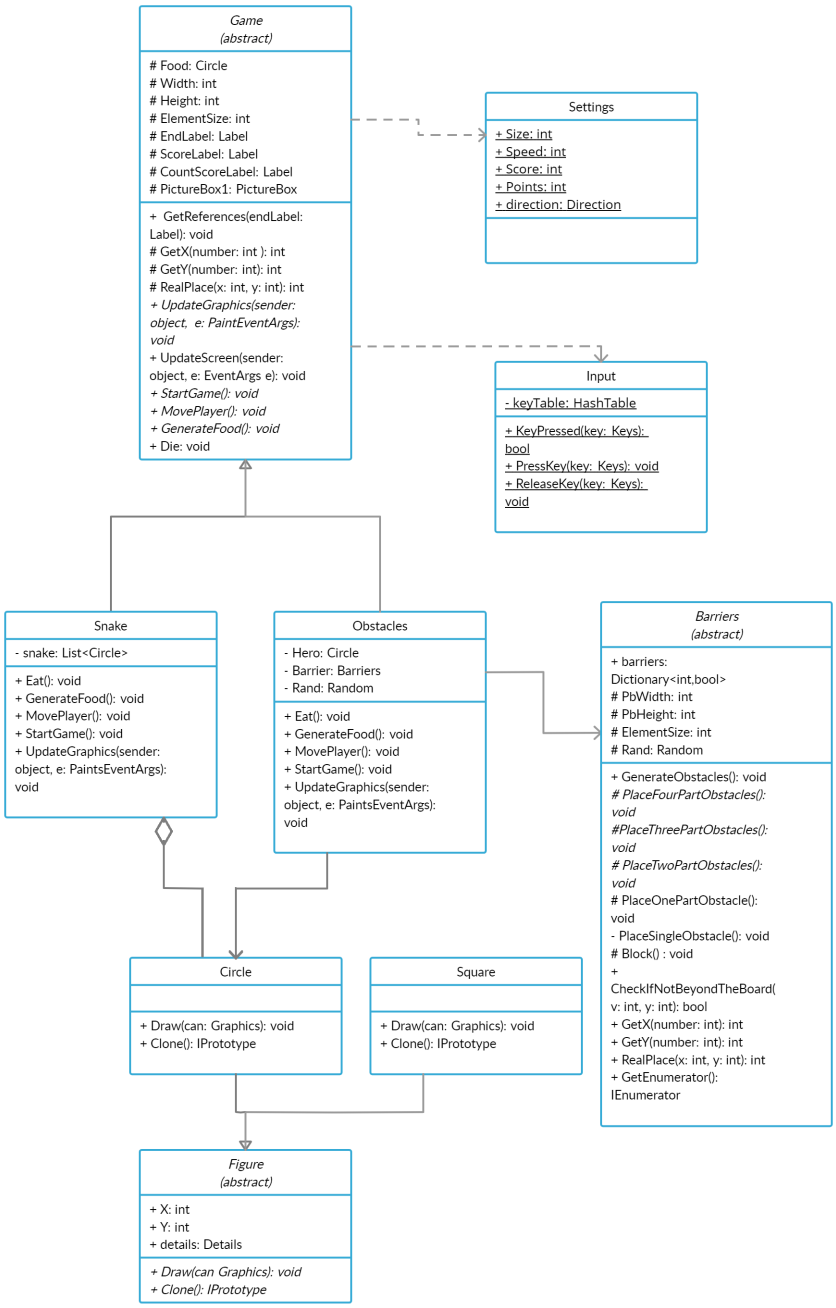
Projekt miał na celu przygotowanie szablonu umożliwiającego pisanie prostych gier polegających na poruszaniu się po planszy i zdobywaniu punktów, zbierając określone elementy. Zostały przedstawione dwie przykładowe implementacje - powszechnie znana gra Snake oraz gra polegająca na omijaniu przeszkód.

2 Lista zaimplementowanych funkcjonalności

Szablon gry umożliwia poruszanie się po planszy przy pomocy strzałek.
Po uruchomieniu gry wyświetlają się dwa przyciski:
- Snake - tworzy nowe okno i uruchamia w nim grę Snake:
Pojawiają się dwa kółka: głowa węża i pokarm. Po prawej stronie mamy licznik punktów. Jeśli położenie głowy węża zrówna się z zielonym kółkiem (pokarm), wąż wydłuża się, a liczba punktów zwiększa o 1. Wyjście poza krawędzie skutkuje końcem gry. Po przegranej na planszy umieszczony zostaje tekst powiadamoiający o końcu gry oraz o liczbie zdobytych punktów. Wywołanie akcji przyciśnięcie entera umożliwia rozpoczęcie nowej gry Snake.
- Przeszkody - tworzy nowe okno i uruchamia w nim grę Przeszkody:
Pojawiają się przeszkody i dwa kółka: bohater i pokarm. Po prawej stronie mamy licznik punktów. Jeśli położenie bohatera zrówna się z pokarmem, liczba punktów zwiększa się o 1. Wyjście poza krawędzie skutkuje końcem gry. Po przegranej na planszy umieszczony zostaje tekst powiadamoiający o końcu gry oraz o liczbie zdobytych punktów. Wywołanie akcji przyciśnięcie entera umożliwia rozpoczęcie nowej gry Przeszkody.

3 Diagram UML systemu

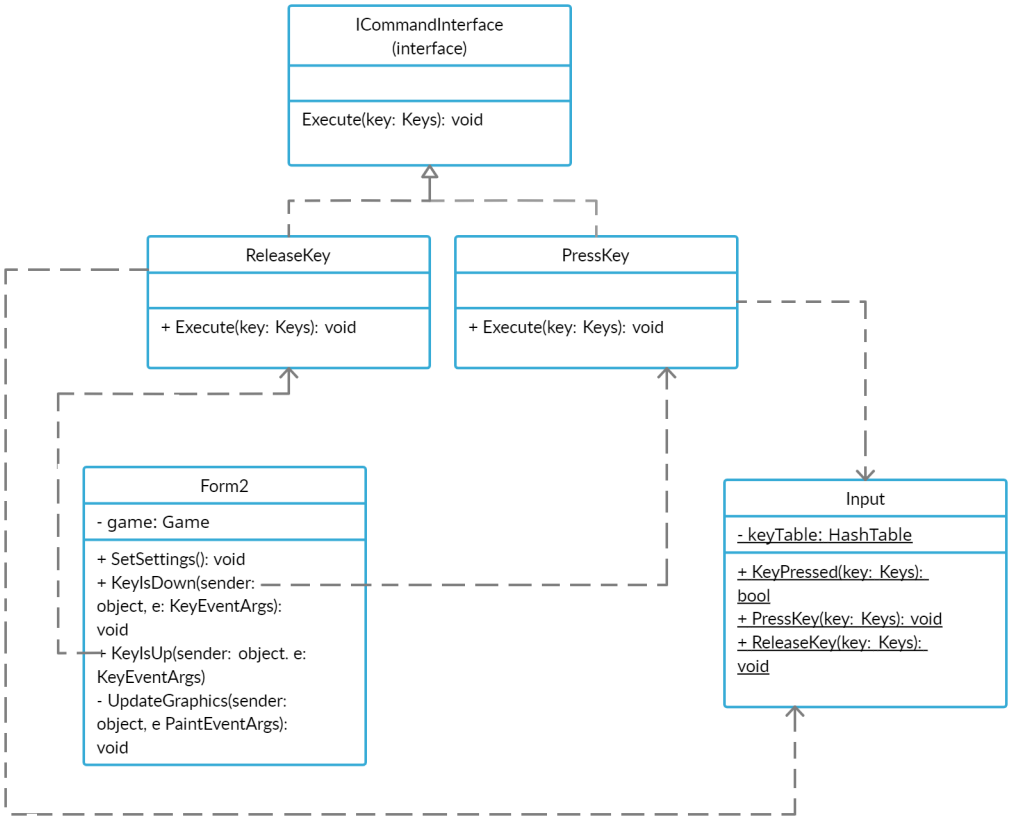
Poniżej przedstawiam diagram UML najważniejszych klas systemu. Bardziej dokładne diagramy będą pokazane przy poszczególnych wzorcach.


Game
(abstract)
Food: Circle
Width: int
Height: int
ElementSize: int
EndLabel: Label
ScoreLabel: Label
CountScoreLabel: Label
PictureBox1: PictureBox
+ GetReferences(endLabel: Label): void
# GetX(number: int): int
# GetY(number: int): int
# RealPlace(x: int, y: int): int
+ UpdateGraphics(sender: object, e: PaintEventArgs): void
+ UpdateScreen(sender: object, e: EventArgs e): void
+ StartGame(): void
+ MovePlayer(): void
+ GenerateFood(): void
+ Die: void
Settings
+ Size: int
+ Speed: int
+ Score: int
+ Points: int
+ direction: Direction
Input
- keyTable: HashTable
+ KeyPressed(key: Keys): bool
+ PressKey(key: Keys): void
+ ReleaseKey(key: Keys): void
Snake
- snake: List<Circle>
+ Eat(): void
+ GenerateFood(): void
+ MovePlayer(): void
+ StartGame(): void
+ UpdateGraphics(sender: object, e: PaintsEventArgs): void
Obstacles
- Hero: Circle
- Barrier: Barriers
- Rand: Random
+ Eat(): void
+ GenerateFood(): void
+ MovePlayer(): void
+ StartGame(): void
+ UpdateGraphics(sender: object, e: PaintsEventArgs): void
Barriers
(abstract)
+ barriers: Dictionary<int,bool>
# PbWidth: int
# PbHeight: int
# ElementSize: int
# Rand: Random
+ GenerateObstacles(): void
# PlaceFourPartObstacles(): void
#PlaceThreePartObstacles(): void
PlaceTwoPartObstacles(): void
PlaceOnePartObstacle(): void
- PlaceSingleObstacle(): void
Block() : void
+ ChecklfNotBeyondTheBoard(v: int, y: int): bool
+ GetX(number: int): int
+ GetY(number: int): int
+ RealPlace(x: int, y: int): int
+ GetEnumerator(): IEnumerator
Circle
+ Draw(can: Graphics): void
+ Clone(): IPrototype
Square
+ Draw(can: Graphics): void
+ Clone(): IPrototype
Figure
(abstract)
+ X: int
+ Y: int
+ details: Details
+ Draw(can Graphics): void
+ Clone(): IPrototype

4 Użyte wzorce projektowe

4.1 Polecenie

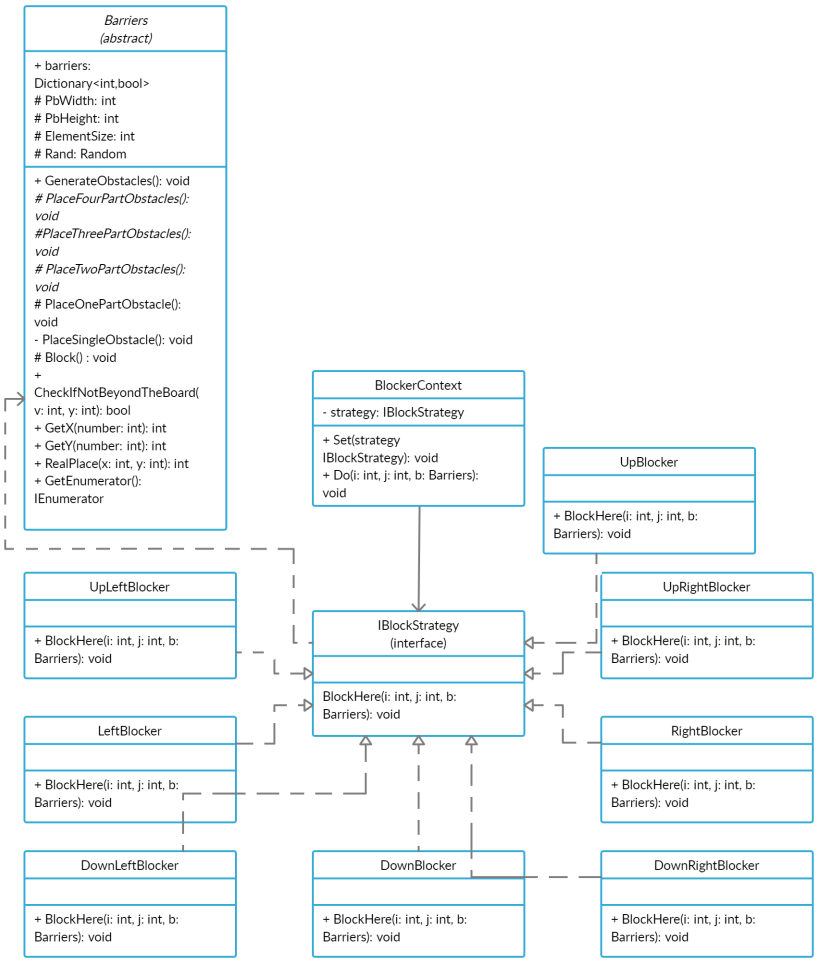
Polecenie - wzorzec behawioralny, zmieniający żądanie w samodzielny obiekt. Usuwa powiązanie pomiędzy obiektem, który wywołuje operację i obiektem, który wie, jak ją wykonać.
W moim przypadku oddziela klasę Input od Form2. Metoda KeyIsDown w Form2 wykorzystuje klasę PressKey, implementującą ICommandInterface - interfejs polecenia, natomiast Metoda KeyIsUp wykorzystuje ReleaseKey. Pozwalają one zmieniać przyciśnięte klawisze w klasie Input.



4.2 Strategia

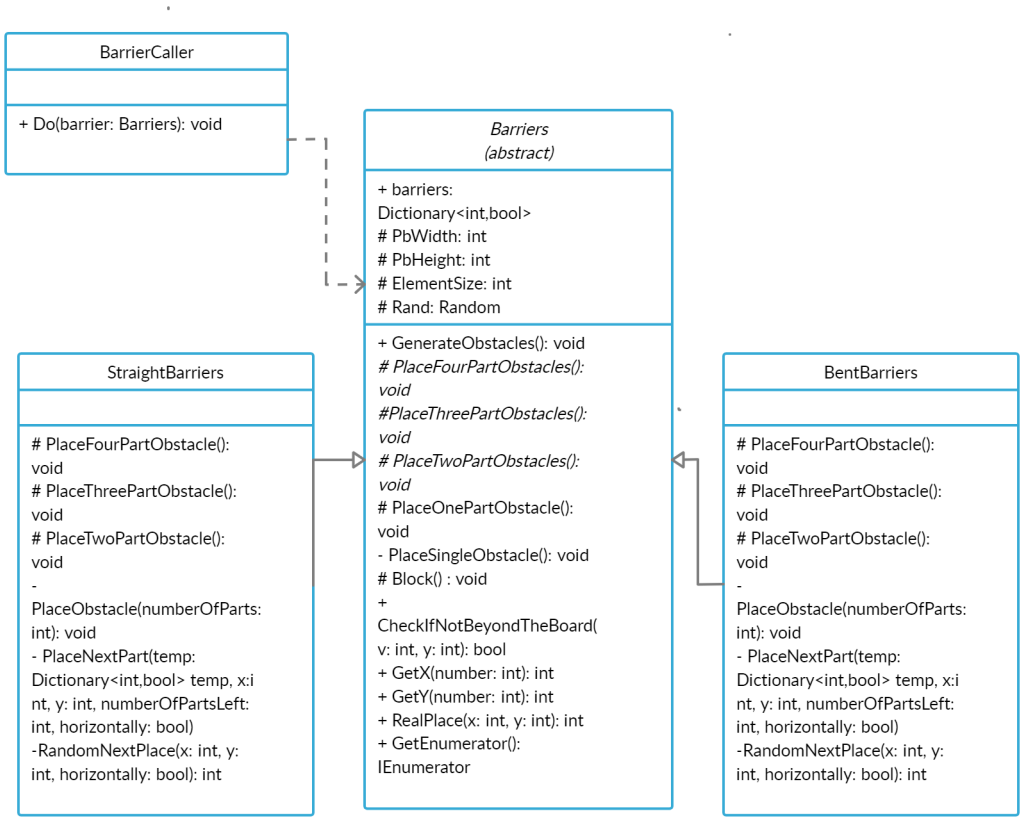
Strategia - wzorzec behawioralny, pozwalający zdefiniować rodzinę algorytmów, umieścić je w osobnych klasach jednocześnie czyniąc obiekty tych klas wymienialnymi.

Kontekst nie jest odpowiedzialny za wybór stosownego algorytmu dla danego zadania. To klient przekazuje żądaną strategię kontekstowi. W ten sposób możemy dodawać kolejne algorytmy bez zmieniania kodu kontekstu i kodu innych strategii.
W moim programie wykorzystuję ten wzorzec, aby zablokować miejsca otaczające przeszkody przy ich tworzeniu. Przeszkody działają mniej więcej tak jak statki w popularnej grze, czyli na bezpośrednio sąsiadujących polach nie może stać inna przeszkoda. Z tego powodu mamy osiem Blockerów: UpBlocker,..., UpLeftBlocker, blokujących określone miejsca. Gdyby Strategia nie została tutaj użyta, metoda Block() w klasie Barriers zawierałaby duży blok operatorów warunkowych, mających wybrać odpowiednią wersję tego samego algorytmu, co byłoby nieczytelne.



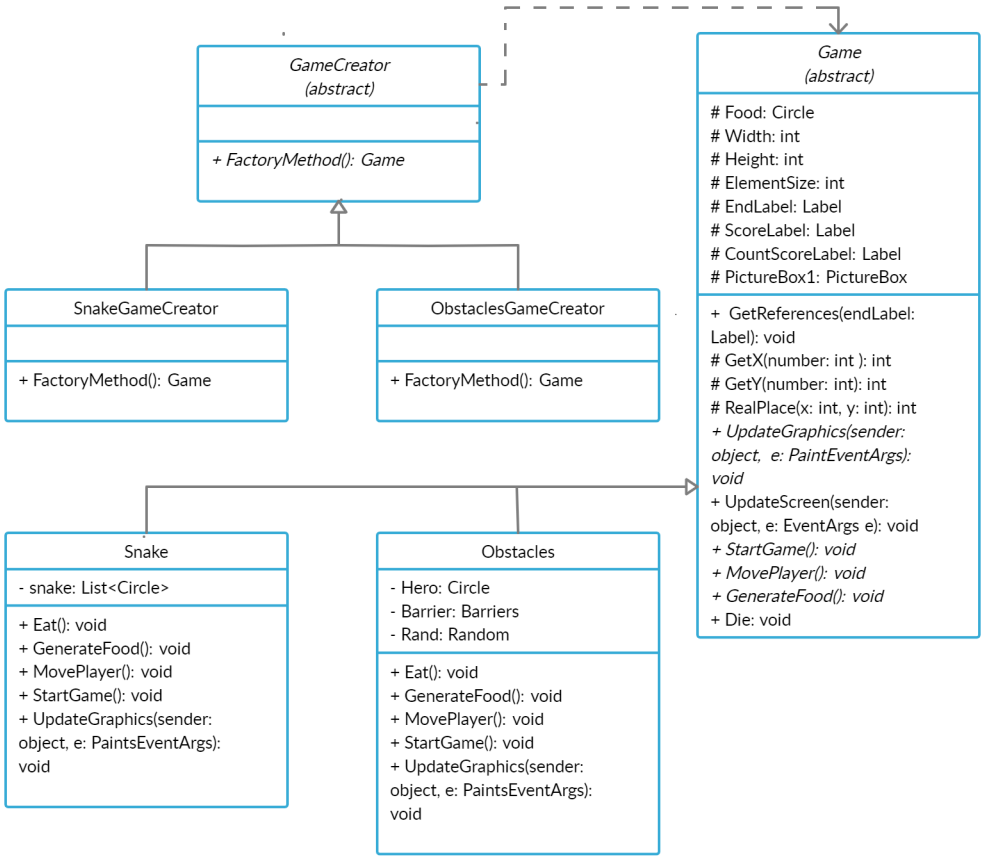
4.3 Metoda szablonowa

Metoda szablonowa - wzorzec behawioralny, definiujący szkielet algorytmu w klasie bazowej, ale pozwalający podklasom nadpisać pewne etapy tego algorytmu bez konieczności zmiany jego struktury. Przydaje się, kiedy chcemy przeprowadzić dokładnie te same kroki algorytmu, ale w inny sposób.
Ja użyłam tego wzorca projektowego do generowania prostych i zakrzywionych przeszkód. Metoda GenerateObstacles() w klasie Barriers pokazuje kroki tworzenia zestawu przeszkód do gry, jednak to klasy dziedziczące ustalają w jaki sposób daną przeszkodę stworzyć.
Metoda szablonowa jest tutaj potrzebna, bo gdyby nie ona miałabym dwie klasy o w większości identycznym kodzie. Edytowanie go byłoby konieczne w obu tych klasach. Dzięki zastosowaniu wzorca, jest to o wiele łatwiejsze.
Klasa BarrierCaller wywołuje metodę GenerateObstacles() od odpowiedniego obiektu klas dziedziczących po Barriers.



4.4 Metoda wytwórcza

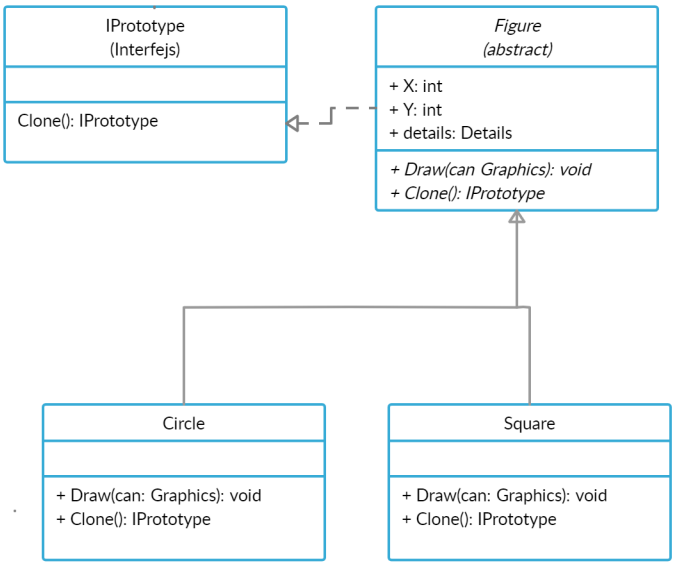
Metoda wytwórcza - wzorzec kreacyjny, który udostępnia interfejs do tworzenia obiektów w ramach klasy bazowej, ale pozwala podklasom zmieniać typ tworzonych obiektów.
Metoda wytwórcza oddziela kod konstruujący produkty od kodu który faktycznie z tych produktów korzysta.
Zastosowałam metodę wytwórczą, bo chcę mieć łatwą możliwość zaimplementowania innych gier. Kiedy będę chciała dopisać kolejną grę, wystarczy, że utworzę podklasę kreacyjną i napiszę jej metodę wytwórczą, bez ingerencji w resztę kodu.



4.5 Prototyp

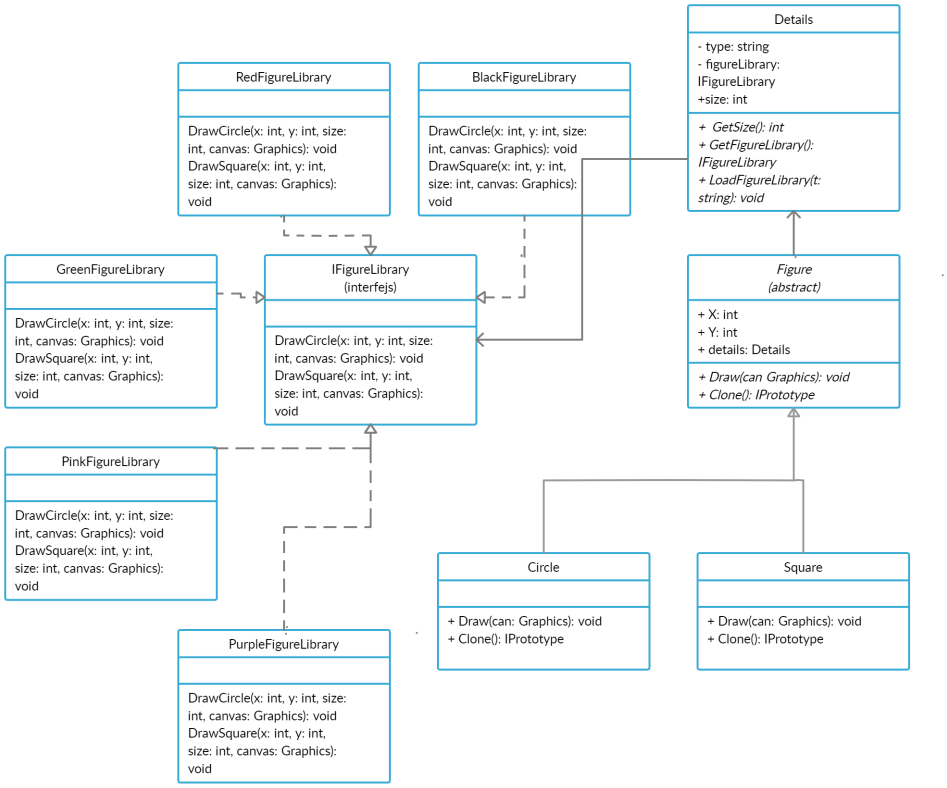
Prototyp - wzorzec kreacyjny, który umożliwia kopiowanie już istniejących obiektów bez tworzenia zależności pomiędzy kodem, a klasami obiektów.

Interfejs IPrototype ma pojedynczą metodę: Clone(). Ten interfejs jest wspólny dla wszystkich obiektów wspierających klonowanie. Jest to na przykład sposób na kopiowanie pól prywatnych, bo sam obiekt to kopiowanie wykonuje.
Ja samo klonowanie wykorzystuję głównie w metodzie Eat() w klasie Snake. Dzięki temu kopiuję ostatnie kółko z listy elementów snake'a.



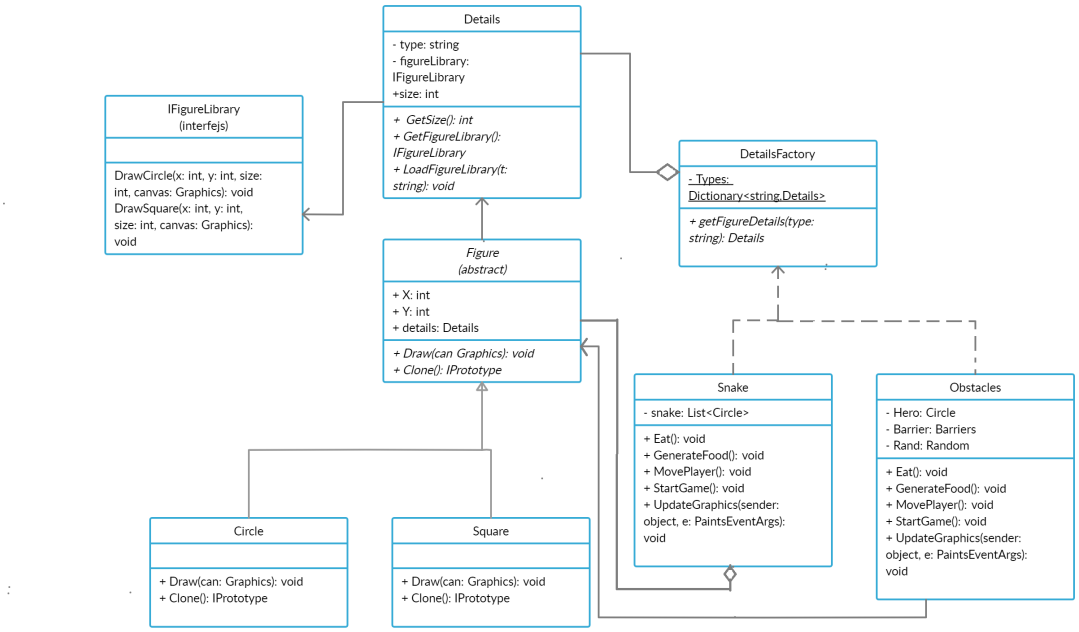
4.6 Most

Most - wzorzec strukturalny, pozwalający na rozdzielenie danej klasy lub zestawu spokrewnionych klas na dwie hierarchie - abstrakcję oraz implementację. Pozwala to na niezależną pracę nad nimi.
W moim projekcie jest to potrzebne, aby nie powielać figur o różnych kolorach. Gdyby nie most, miałabym podklasy Square, we wszystkich pięciu potrzebnych mi kolorach oraz podklasy Circle, również w tych pięciu kolorach. Dodanie jakiegokolwiek nowego koloru wiązałoby się z dodaniem kolejnych dwóch podklas. To nie brzmi aż tak źle, natomiast dodanie nowej figury, powiedzmy Triangle, sprawi, że wystąpi konieczność dodania tylu podklas, ile kolorów tego trójkąta nam potrzeba
Abstrakcją jest u mnie klasa Figure i klasy pochodne Square i Circle, a implementacją interfejs IFigureLibrary i klasy go implementujące.



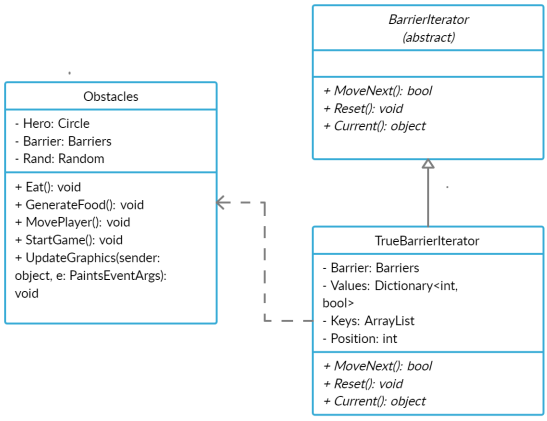
4.7 Pyłek

Pyłek - wzorzec strukturalny, pozwalający zmieścić więcej obiektów w danej przestrzeni pamięci RAM. Stosujemy go, gdy nasz program pracuje z dużą ilością obiektów o podobnych parametrach.
Wykorzystałam pyłek do przechowywania detali figur: klasy rysującej i rozmiaru. Dzięki temu zajmuje to mniej miejsca w pamięci, bo dzięki DetailsFactory, trzymam tylko pojedyncze obiekty tych elementów.



4.8 Iterator

Iterator - wzorzec behawioralny, pozwalający przechodzić od elementu do elementu jakiegoś zbioru bez konieczności eksponowania jego formy.
W moim projekcie iterator przydał się do rysowania przeszkód, ponieważ w Barriers.barriers trzymam informacje zarówno o polach, na których faktycznie znajdują się przeszkody, jak i o polach zablokowanych (co opisałam wyżej na zasadzie analogii do gry w statki). Dzięki zaimplementowaniu własnego iteratora, mogę przechodzić od razu po tych elementach słownika barriers, które mnie interesują.


BarrierIterator
(abstract)
+ MoveNext(): bool
+ Reset(): void
+ Current(): object
Obstacles
- Hero: Circle
- Barrier: Barriers
- Rand: Random
+ Eat(): void
+ GenerateFood(): void
+ MovePlayer(): void
+ StartGame(): void
+ UpdateGraphics(sender: object, e: PaintsEventArgs): void
TrueBarrierIterator
- Barrier: Barriers
- Values: Dictionary<int, bool>
- Keys: ArrayList
- Position: int
+ MoveNext(): bool
+ Reset(): void
+ Current(): object