1. Mechanizm ramek

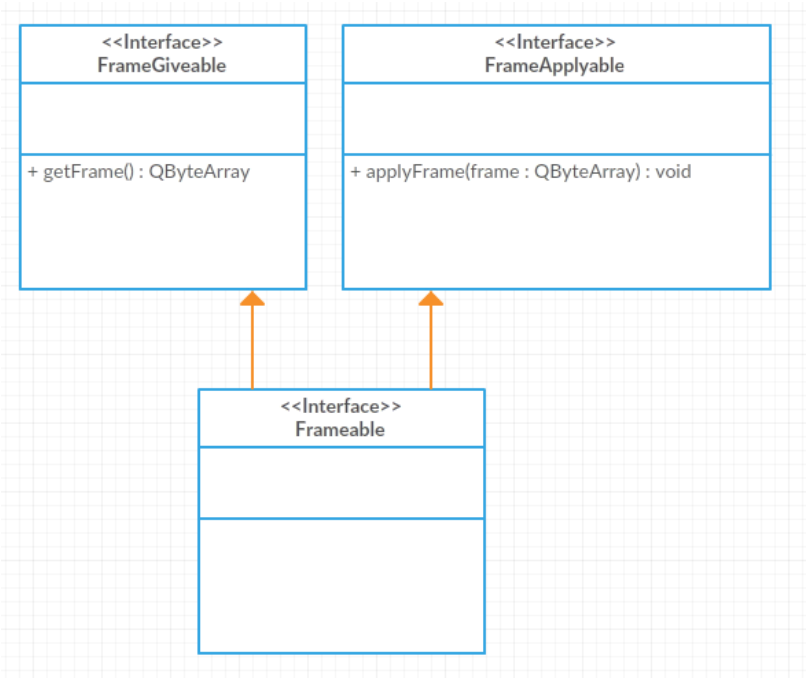
Ramki to po prostu tablica znaków z zakodowaną w pewien sposób informacją. Są one transmitowane przez sieć protokołem TCP. Za obsługę wysyłania bądź odbierania tych ramek odpowiadają klasy TcpClient i TcpServer, które znajdują się kolejno na kliencie i na serwerze.

Wspomniane ramki są reprezentowane jako QByteArray, które opakowuje ciąg znaków i daje przydatny interfejs do manipulowania nimi.

Cała rozpiska ramek znajduje się w odrębnym dokumencie. Ramki zostały zaprojektowane w arkuszu kalkulacyjnym i były automatycznie aktualizowane gdy zachodziła taka potrzeba.

W celu usprawnienia tworzenia ramek bądź ich aplikowaniu na obiekty utworzony został pomocniczy interfejs: Frameable (realizowany w C++ jako klasa abstrakcyjna). Sam Frameable dziedziczy po FrameGiveable (który dostarcza metodę getFrame) i po FrameApplyable (daje metodę applyFrame).

Wygląda to następująco:



Postępowanie jest następujące: gdy wiemy, że klasa będzie dawała coś, co pozwoli utworzyć jakąkolwiek ramkę, to dziedziczymy po FrameGiveable. Natomiast gdy wiemy, że w jakiś sposób ramki mają wpływać na stan obiekty klasy poprzez przypisanie nowych wartości, to dziedziczymy klasę po FrameApplyable.

Klasy, które dziedziczą po wspomnianych interfejsach:

- Player – dziedziczy po Frameable, gdyż musi mieć możliwość utworzenia ramki na podstawie swoich pól (id gracza, jego grupa, ilość punktów, poziom życia, pozycja (x, y), kierunek zwrotu czołgu, imię gracza, czy żyje i jak nie żyje, to przez ile okresów gry). Informacje te są potrzebne do tego, by serwer mógł wysłać ramkę ze stanem planszy co określony interwał

czasu (u nas: 200 ms). Klasa ta potrafi przyjmować ramkę – potrzebne to jest dla klienta, który musi mieć informację o każdym z graczy by odpowiednio namalować planszę i to, co na niej się dzieje.

- Shot – dziedziczy po Frameable, bo musi mieć możliwość utworzenia ramki ze swoich pól (czyli o informacjach o pocisku: id pocisku, id przypisanego gracza, pozycja startowa (x, y), kierunek lotu, czas trwania lotu i siła uderzenia). Podobnie jak dla klasy Player, informacje są potrzebne do wysłania przez serwer stanu planszy. Klasa potrafi przyjąć ramkę i zaaplikować ją na wspomniane pola bo klient musi posiadać informacje o wystrzałach by poprawnie je namalować.
- Board – również typu Frameable. Klasa zawiera tablicę 2D (a dokładnie QVector<QVector<BoardElement> >) z polami planszy, które można w dużym uproszczeniu można traktować jak tablicę z polami zawierającymi id elementu planszy. Dziedziczenie po Frameable jest konieczne, bo serwer musi rozsyłać ramki z tymi informacjami co 200ms (getFrame()) i klient musi je odbierać i interpretować (applyFrame()).
- Model – dziedziczy po Frameable. Jest to klasa zbiorcza na planszę, graczy i pociski. Ona jest pośrednikiem do Player, Shot i Board. Przyjmuje ramki i je produkuje poprzez wywołanie applyFrame() i getFrame() na Board, Player i Shot.
- PlayerAction – typu Frameable. Dziedziczenie po tym interfejsie jest konieczne, gdyż informacje o akcjach gracza (kierunek poruszenia się gracza i informacja o tym, czy pocisk został wystrzelony) muszą zostać wysłane z klienta na serwer, czyli klient musi utworzyć ramkę poprzez getFrame() i serwer musi jak zaaplikować poprzez applyFrame().
- Game – typu FrameApplyable ze względu na to, by klient mógł odebrać ramkę i od serwera i ją przetworzyć (m. in poprzez wywołanie applyFrame() na jednym z modelu). Klasa Game zawiera się w kliencie i zbiera wszystkie informacje o stanie gry. Posiada dwa egzemplarze klasy Model: model1 i model2. Jest to konieczne do płynnej animacji – chcemy mieć możliwość płynnie animować obiekty niezależnie od tego, jak często przyjdzie ramka z serwera. Do tego płynnego przejście potrzebne są 2 modele.
- NetworkManager – klasa, która przyjmuje wszystkie ramki od serwera w kliencie. Dlatego też dziedziczy po FrameApplyable i nadpisuje metodę applyFrame().

2. Organizacja klas klienta.

Podstawową klasą, która zbiera akcje użytkownika jest MainWindow. Dziedziczy ona po QWidget i jest powiązana z plikiem mainwindow.ui, który zawiera interfejs użytkownika.

Klasa ta trzyma jako swoje pole obiekty typu Canvas, NetworkManager i Game. Canvas to obiekt klasy QWidget, na którym odbywa się malowanie. NetworkManager zarządza ramkami i komunikacją z serwerem. Game jest odpowiedzialny za zbieranie informacji o grze.

Klasa Canvas posiada jako właściwość klasę Drawer, która jest wykorzystywana do malowania planszy (na płótnie Canvas). Klasa Drawer zawiera klasę pomocniczą (helper) Sprites, która dostarcza interfejs do obrazków używanych do namalowania planszy.

NetworkManager posiada klasę TcpClient. Zastosowany został wzorzec projektowy o nazwie Fasada, gdyż ukrywany jest dostęp do TcpClient, który zawiera funkcjonalność nieco niższego poziomu obsługi naszej aplikacji.

Klasa Game posiada wszystkie informacje na temat gry. Zawiera m. in. 2 obiekty typu Model w celu przechowania aktualnego stanu gry i 1 obiekt typu PlayerAction w celu przechowania akcji wykonywanych przez użytkownika.

3. Organizacja klas serwera.

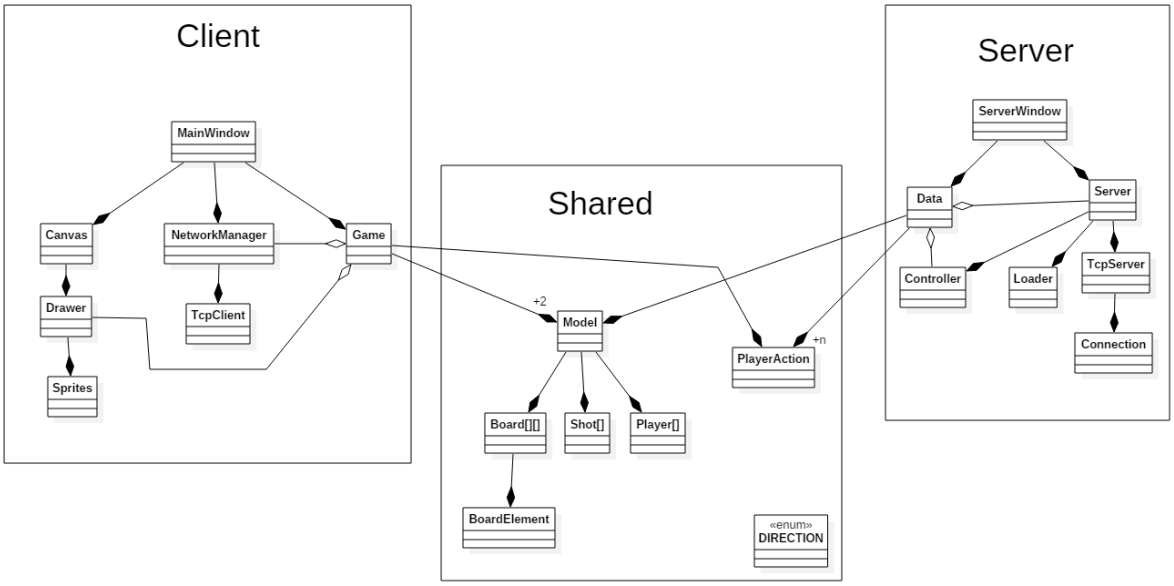
Główną klasą na serwerze jest ServerWindow, która jest typu QWidget. Powiązana jest ona z zaprojektowanym interfejsem gui w programie QtCreator. ServerWindow zawiera w sobie obiekt typu Data w celu składowania danych na serwerze (model + akcje użytkownika) i klasę Server, która jest menagerem sterującym wszystkim co dzieje się w programie.

Klasa Server opakowuje klasę TcpServer służącą do realizacji serwera TCP. Zawiera także klasę służącą do Ładowania plików z mapami – Loader i klasę, która steruje logiką całej gry – Controller.

4. Klasy wspólne

Klasami wspólnymi dla klienta i serwera jest Model (zawiera informacje o planszy, pociskach i czołgach), PlayerAction (przechowuje ruchy gracza). Model zawiera Board, Shot, Player i BoardElement, które też są współdzielone. Dodatkowo współdzielony jest typ wyliczeniowy DIRECTION, który przechowuje kierunek lub jego brak.

5. Diagram klas (same połączenia)



6. Sygnały i sloty

Qt posiada mechanizm sygnałów i slotów w celu komunikowania się obiektów między sobą. Sygnał i slot deklarowane są tak samo jak funkcje składowe klasy, przy czym sygnału się nie implementuje. Gdy sygnał jest przypięty po slot to wtedy jeśli wyemitowany zostanie sygnał, to slot go odbierze i się wykona. Ważne jest to, by deklaracje sygnału i slotu były ze sobą zgodne.

W aplikacji znajdują się następujące połączenia:

Sygnały w klasie Game (klient):

- Sygnał pojawienia się błędu i jego obsługa w slocie w MainWindow
- Sygnał o tym, że model został zaktualizowany i obsługa w MainWindow
- Sygnał zmiany stanu gry obsługiwany w MainWindow i Canvas. Dostępne stany aplikacji: NO_PLAYING, CONNECTING, SENDING_HELLO, WAITING_FOR_PLAYER, PLAYING, GAME_OVER, SENDING_GOODBYE, DISCONNECTING

Sygnały w klasie Server:

- Sygnał zapisania stanu aplikacji w klasie Server do dziennika (audyt) i obsługa w ServerWindow
- Sygnał o tym, że nowy klient został dodany i obsługa w ServerWindow
- Sygnał o tym, że klient został usunięty i obsługa w ServerWindow
- Sygnał o rozłączeniu i obsługa w ServerWindow