Group 24

Battle Tanks
Software Architecture Document (SAD)

CONTENT OWNERS:
Ryan Russell (V00873387), Luciano De Gianni (V00908223)

Document Number:	Release/Revision:	Release/Revision Date:
1	1.0	6 December 2020

Table of Contents

1 Documentation Roadmap

1.1 Scope and Summary

The architecture of Battle Tanks, built by Group 24, is captured in this document. Battle Tanks is an online multiplayer game where two players compete to destroy the other player's tank.

The GitLab repository is available here: https://gitlab.csc.uvic.ca/courses/2020091/SENG350/teams/group_24/battle-tanks. Battle Tanks is deployed and hosted at: https://battle-tanks-24.herokuapp.com/. Please note that it may take a moment for the server to respond initially as it goes to sleep when not in use.

To begin playing, each player connects to the Battle Tanks URL. Whichever player connected first will get to play first and the other player will be unable to move. Once the first player fires, it becomes the other player's turn and the game continues from there. Once one player has no health left, the game is over.

1.2 How The Documentation is Organized

1. **Documentation Roadmap:** describes the layout of this document and summarizes information about it's components and purpose.
2. **How A View Is Documented:** the structure of a view's documentation is described here.
3. **System Overview:** a broad overview of the system and discusses how users are expected to interact and access the system.
4. **Mapping Between Views:** shows the relationship between elements in different views.
5. **Rationale:** provides background and justification for the software architecture.
6. **Directory:** contains an index, glossary, and list of acronyms used.

1.3 View Overview

The Battle Tanks Software Architecture Document presents four views of the system, separated into three categories: module views, component and connector (C&C) views, and allocation views. The four views are as follows:

1. **Package Decomposition View:** this module view instantiates the Strategy pattern by association to the classes contained in its packages. The element types in this view are strictly packages, and therefore do not have any distinct properties themselves. The **UML** Package Diagram modelling technique was used to construct this view.
2. **Class Decomposition View:** this module view instantiates the Strategy pattern. The element types are classes, there are several relation types including dependencies, aggregations, specializations and associations, and the property types vary greatly depending on the class. The UML Class Diagram modelling technique was used to construct this view.
3. **Client and Server Communication View:** this component & connector view has two element types: components and interfaces, and the relationships are considered connectors. The UML Connector Diagram modelling technique was used to construct this view.
4. **Deployment View**: this allocation view instantiates the client and server architecture pattern. The element type is nodes, which can include devices, environments, and

developers as well. The UML Deployment Diagram modelling technique was used to construct this view.

1.4 How Stakeholders Can Use the Documentation

Developers: begin by reading through sections 2 and 3. Section 2 will give an overview of how the views have been documented. Section 3 will provide context for what the system is trying to achieve. Developers will want to start with Sections 4.1 and 4.2 to introduce themselves to the module and component views of the architecture.

An example scenario for a developer could be the following: a developer would like to know which class the handleBroadcast() function falls under. Therefore, the developer consults Section 4.1.2 to inspect the decomposition of classes in Battle Tanks. Using the primary presentation, the developer discovers that the handleBroadcast() function is a part of the GameManager class.

Project Managers: begin by reading the section 1 the Scope and Summary to get an idea of what the document covers; following section 1, sections 2 and 3 should be read to understand how views are documented and to get an overview of the system. From there, move on to section 4.2, the component and connector views of the system.

An example scenario for a project manager could be the following: a manager would like to understand how system views are broken down in order to explain the structure in a meeting. They navigate to Section 2 to discover how a view is documented. There, they find that all of the views in this document are separated into five template sections. They prepare for their meeting using this information.

End Users: most end users will be interested in sections 1.1 and 3 to get an overview of the system functionality.

An example scenario for an end user could be the following: a user can’t be bothered to read an entire technical document, so they just want to find a summary of Battle Tanks. They decide to read Section 3, System Overview. Here, they gain an understanding of the basic game flow.

2 How A View Is Documented

Each view is documented using the following template:

1. **Primary Presentation:** a UML diagram presenting the elements and relations relevant to the view
2. **Element Catalog:** a list of elements in the Primary Presentation. Each list entry contains and explanation for the elements purpose and the element's properties
3. **Context Diagram:** illustrates how the part of the system being examined in the view is incorporated in the overall architecture
4. **Variability Guide:** describes the variability available in the system and explains how someone can make use of the variability
5. **Rationale:** elaborates on the design decisions made and explains the justification for the decisions

3 System Overview

Battle Tanks is a turn-based action and strategy game where two players compete against each other with the goal of destroying the other player's tank. Both players start with 100 health. When it's a player's turn, they can move their tank around their section of the map to a desired location and fire a shell at the opposing tank. Only one shell can be fired per turn. After the shell is fired, the player's turn ends and their opponent's turn begins.

There are three types of shells available to the user: normal shells with low weight and average damage output, heavy shells with higher weight and higher damage output, and explosive shells that separate into three smaller shells with average weight and low damage, but high bounce distance. The tanks can move left to right and the player can adjust the angle and power of their shot using the mouse or trackpad. Once either the player or opponent's health reaches 0, the game is over and the tank with health remaining is the winner!

To begin playing a game, one user will create a game instance by navigating to the web application hosted on **Heroku**. Once a second player joins the instance, a game will begin automatically. The technologies and services we used to develop Battle Tanks include Heroku for hosting, Node.js, Phaser, Socket.io, SQLite, and Express.

3.1 Quality Attribute Scenarios

1. **Modifiability:** A new tank type and corresponding shell type are added to the application in no more than 1 person-day of effort for the implementation. Both the new tank type and shell type integrate fluidly with the existing game scenario.
2. **Usability:** A new user hears about Battle Tanks and wants to try out the game after remembering enjoying similar tank games as a child. The user follows the instructions to join a game instance and begins playing. The controls are intuitive and enough relevant information is displayed for the user to understand how to play after just one game.
3. **Performance:** A player ends their turn by firing a shell at their opponent, but missing. Their opponent begins the turn by moving left. The corresponding movement appears on the player's client no more than 500ms after the command is input on the opponent's client.

4 Views

This section contains several architectural views of Battle Tanks. They are separated into three categories: module views, component and connector (C&C) views, and allocation views.

4.1 Module Views

The module views shown in the subsection aim to show the way in which the system is decomposed into manageable units of software from two levels of abstraction: the package level and class level.

4.1.1 Package Decomposition View

4.1.1.1 Primary Presentation

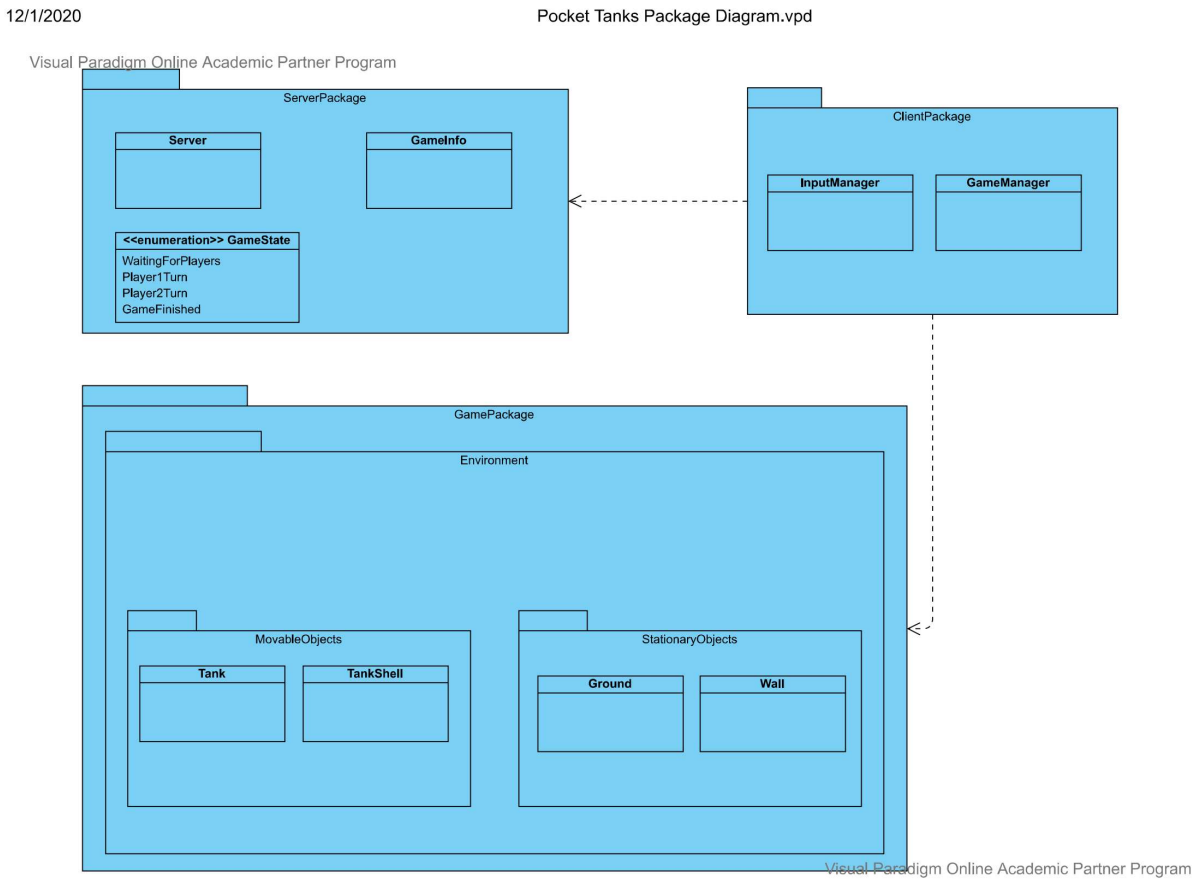

Figure 1: Battle Tanks UML Package Diagram

4.1.1.2 Element Catalog

1. **ServerPackage:** this package contains the application server and related packages that store information integral to game instances.
2. **Server:** this module stores the software required for the Battle Tanks server implementation, which is somewhat analogous to the server.js file in the code base.
3. **GameInfo:** this package stores information about a game instance like the objects and physics present on the map.
4. **GameState:** this enumeration is used so the server knows (and therefore the clients know) what state the game instance is in. There is an analogous GameState in classes.js.

5. **ClientPackage**: this package contains the relevant software for the client implementation like the management of user input and the game from a player's perspective.
6. **InputManager:** this purpose of this package is to perform the input handling from a player. This includes moving a tank, firing a shell, and switching the shell type. This is analogous to the handleInput() and handleBroadcast() functions in classes.js.
7. **GameManager:** the purpose of this package is to manage a game instance from a player's perspective; what they see, when they see it, and how they operate their tank.
8. **GamePackage:** contains everything visibly located in the game environment, including the environment.
9. **Environment:** the game environment (analogous to the scene in the code), which includes all of the objects, the background and the game physics. The code for the environment is implemented across game.js and classes.js.
10. **MovableObjects:** these objects are capable of motion, and may be player-controlled.
11. **Tank:** the cornerstone of Battle Tanks, the tank package contains the object along with the relevant controls for a player-controlled tank.
12. **TankShell:** this package contains the objects and operations for the three types of tank shells in Battle Tanks.
13. **StationaryObjects:** these objects are not capable of motion and cannot leave the environment once instantiated.
14. **Ground:** the ground of the map, which the tanks can move across.
15. **Wall:** walls can rebound shells and block tanks from moving past them.

4.1.1.3 Context Diagram

A context diagram is not required for this decomposition view as the packages in the primary presentation make up the entire application.

4.1.1.4 Variability Guide

There are no variation points to be exercised from the package decomposition architectural view, although there are variation points contained within some of the lowest-level packages in the UML diagram. Those variation points will be discussed in the class decomposition view.

4.1.1.5 Rationale

The package decomposition design shown in Section 4.1.1.1 was produced primarily with the notion of a client-server architecture in mind. This is why the UML package diagram distinctly separates the ServerPackage and ClientPackage. Additionally, the Strategy design pattern was selected as the primary design pattern for the development of Battle Tanks due to high priority objectives for optimizing the Modifiability quality attribute. While the strategy pattern was not explicitly selected to solve an architectural problem that is relevant to this view, the most relevant context for understanding the usage of the pattern is within the subpackages of GamePackage. Namely, the Tank and TankShell packages.

One of the primary reasons for the usage of the Strategy pattern was that we want to be able to easily introduce variety into our game in the form of different shell types and tank types. The usage of the Strategy pattern to solve this problem will be explained further in Section 4.1.2.

4.1.2 Class Decomposition View

4.1.2.1 Primary Presentation

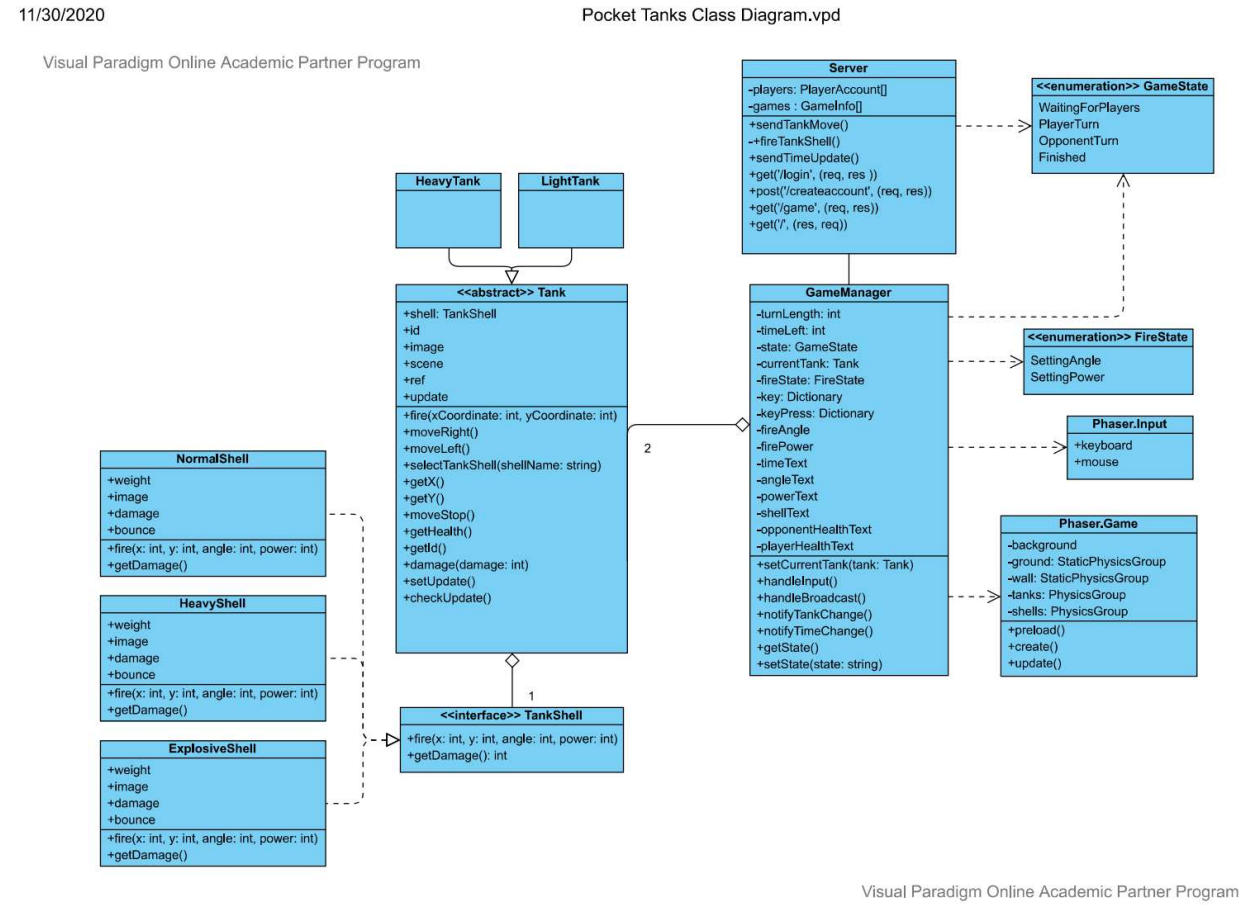

Figure 2: Battle Tanks UML Class Diagram

4.1.2.2 Element Catalog

1. **GameManager:** the purpose of this class is to manage a game instance from a player's perspective; what they see, when they see it, and how they operate their tank. Includes a variety of attributes, input handling, and broadcast handling between players.
2. **Server:** composes the software required for the Battle Tanks server implementation, which is somewhat analogous to the server.js file in the code base.
3. **GameState:** this enumeration is used so the server knows (and therefore the clients know) what state the game instance is in. There is an analogous GameState in classes.js.

4. **FireState:** similar to the GameState, but simply defines whether the player is currently setting the angle or power of the shell firing mechanism.
5. **Phaser.Input:** this class handles the user input by two I/O devices: keyboard and mouse.
6. **Phaser.Game:** contains the physics required for Battle Tanks to function along with the three inherent phaser functions: preload, create, and update.
7. **Tank:** the cornerstone of Battle Tanks, the tank class defines the relevant controls and attributes for a player-controlled tank.
8. **HeavyTank:** a specialization of the Tank class for a tank type with higher armor and weight.
9. **LightTank:** a specialization of the Tank class for a tank type with lower armor and higher speed.
10. **TankShell:** this class defines the single attribute and operation for the base tank shell.
11. **NormalShell:** a specialization of the TankShell class for a shell type with low weight and average damage output.
12. **HeavyShell:** a specialization of the TankShell class for a shell type with high weight and high damage output.
13. **ExplosiveShell:** a specialization of the TankShell class for a shell type that separates into three smaller shells with average weight and low damage, but high bounce distance.

4.1.2.3 Context Diagram

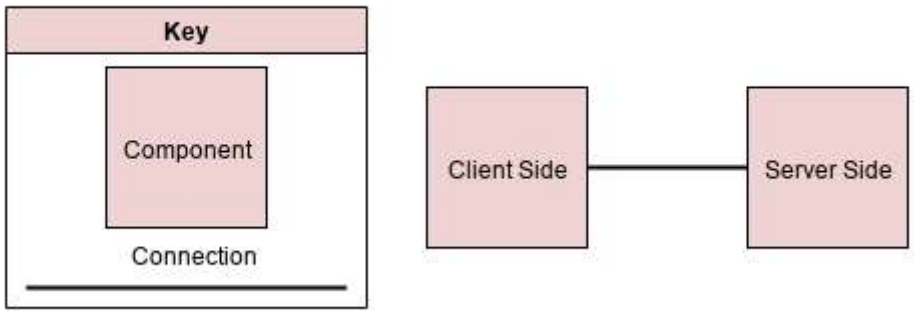

Figure 3: Client and Server Context Diagram For Class Decomposition View

The context diagram shown in Figure 3 displays the separation between client-side classes and server-side classes for the class decomposition view. Nearly all of the classes shown in Section 4.1.2.1 fall under the client side. The only exceptions are the Server and GameState classes.

4.1.2.4 Variability Guide

The main variability points displayed in this view are the options between two types of tanks and three types of tank shells. The two types of tanks are implemented by the LightTank and HeavyTank classes. The variability between these two options allows for slightly different playstyles: quicker movement but takes more damage, or slower movement but with increased armor.

The three types of tank shells are implemented by the NormalShell, HeavyShell, and ExplosiveShell classes. Once again, this variation point supports differing playstyles, but this time can be varied on a turn-by-turn basis. This allows users to select the best shell option for a given turn. For example, if a player has locked onto their target's location and knows the exact angle and power required to hit them, a heavy shell would be a good option as it inflicts the most damage. On the other hand, the explosive shell, which splits into three lower damage shells, could be a good play on the first turn of a game as when the player does not yet know the required angle to hit their opponent.

4.1.2.5 Rationale

The Strategy design pattern was selected as the primary design pattern for the development of Battle Tanks due to high priority objectives for optimizing the Modifiability quality attribute. The usage of the Strategy pattern is arguably most apparent in the class decomposition view. To be exact, one of the primary reasons for the usage of the Strategy pattern was that we want to be able to easily introduce variety into our game in the form of different shell types and tank types.

TankShell instances are created before firing and destroyed after they collide. Using the Strategy pattern lets us take advantage of this already present behaviour because it focuses on using a reference to an instance of a class that realizes an interface. We can destroy the 'loaded' shell and

instantiate a new one if the player switches shells. Using an interface helps us achieve encapsulation and we can defer binding to runtime because we know each instantiated TankShell will implement the TankShell interface.

We can use an abstract class for our Tank to provide means for creating different types of tanks. Tanks will be created once at the start of the game so we won't need to worry about the frequent initialization and destruction of Tank instances.

4.2 Component & Connector Views

4.2.1 Client and Server Communication View

4.2.1.1 Primary Presentation

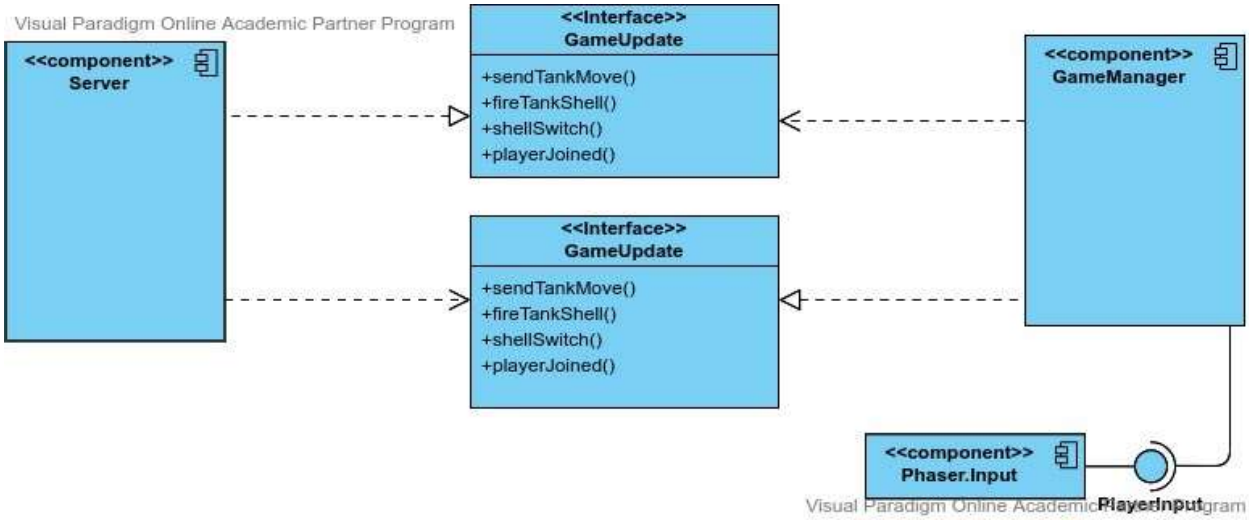

Figure 4: Client and Server Communication Diagram

4.2.1.2 Element Catalog

4.2.1.2.1 Elements And Their Properties

1. **Server:** contains all of the software artifacts necessary to run the server portion of Battle Tanks. Broadcasts and receives information using the GameUpdate interface.
2. **GameUpdate:** interface used for communication between the Server and GameManager. GameUpdate is described in further detail in the next section.

3. **GameManager:** runs on the client side. Responsibilities include: handling server input, keyboard and mouse input, game physics, and game logic.
4. **Phaser.Input:** provides an interface for the GameManager to receive keyboard and mouse input from the player.

4.2.1.2.2 Element Interfaces

1. **GameUpdate:** contains four methods: sendTankMove, fireTankShell, shellSwitch, playerJoined. The playerJoined method handles connecting to players and determines who goes first. The fireTankShell method broadcasts that a shell has been fired and the data related to a shell firing. sendTankMove communicates tank movement between clients. Finally, shellSwitch communicates what shell a player has equipped currently.

4.2.1.3 Context Diagram

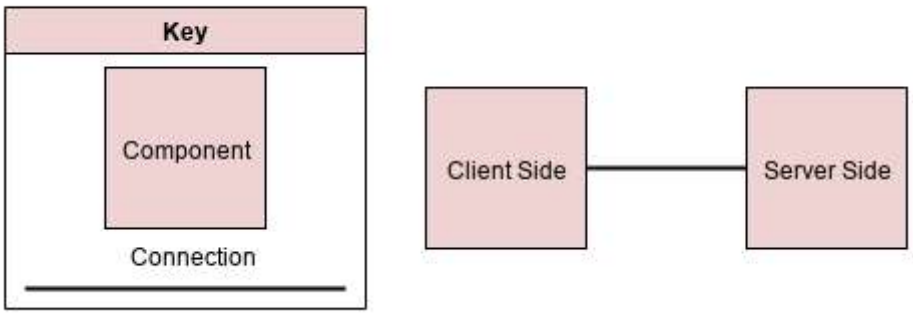

Figure 5: Client and Server Context Diagram

The server component is executing on the server side and the GameManager component is executing on the client-side component. The GameUpdate is used to communicate across the connection between the Client Side component and the Server Side component.

4.2.1.4 Variability Guide

There are no variability points available for this view.

4.2.1.5 Rationale

The GameUpdate interface contributes to improving the performance of the system. Defining methods for the interface helps to minimize the data flow through the server. For instance, sendTankMove is called frequently throughout a game of Battle Tanks; by creating a separate method for communicating the movement instead of sending all of the update information at once, the amount of data that has to be received and broadcast by the server is reduced.

4.3 Allocation Views

The allocation view described below describes how different software units were allocated to various nodes. Nodes can include devices, environments, and developers as well.

4.3.1 Deployment View

4.3.1.1 Primary Presentation

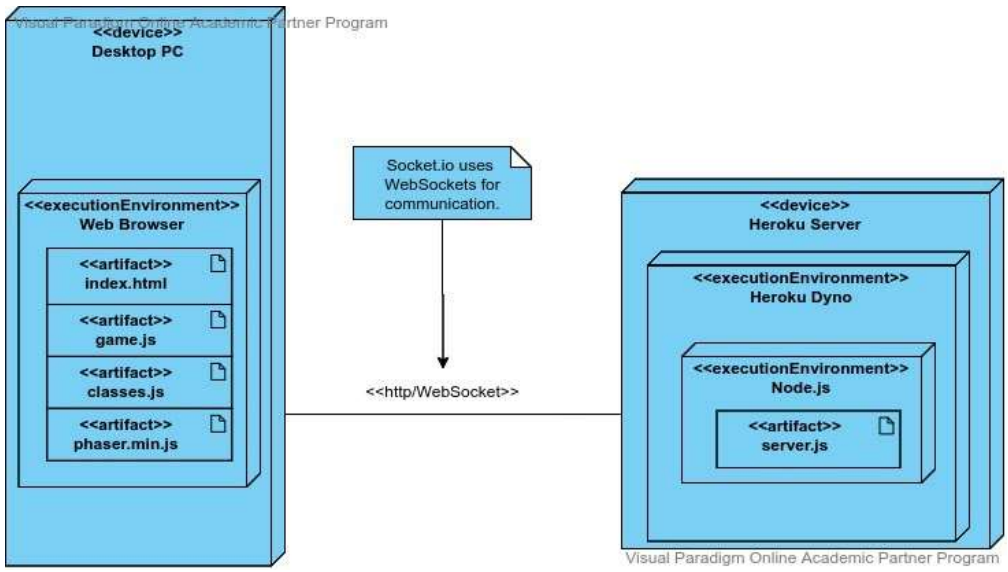

Figure 6: Battle Tanks Deployment Diagram

4.3.1.2 Element Catalog

1. **Desktop PC:** the Desktop PC represents the user's machine.
2. **Web Browser:** represents the Web Browser running on the user's machine. Battle Tanks is a web application so the client portion will always be run in a Web Browser.

3. **index.html:** the html file that is run in the user's browser. It contains the Controls and Rules information and includes links to the game.js, classes.js, and phaser.min.js artifacts. Battle Tanks runs inside of a <canvas> element inside of index.html.
4. **game.js:** contains the asset loading, initialization, and main game loop for Battle Tanks.
5. **classes.js:** contains the classes and enumerated types used in game.js.
6. **phaser.min.js:** contains all of the classes and methods necessary to use the phaser engine.
7. **Heroku Server:** represents the Heroku Server where the Dyno running Battle Tanks is hosted.
8. **Heroku Dyno:** a Heroku Dyno is a linux container in which the execution environment for Battle Tanks is contained. All server-side computation is performed inside of the Heroku Dyno.
9. **Node.js:** the Node.js runtime environment is used for executing Javascript without a web browser.
10. **server.js:** contains all server side functionality. Is responsible for giving the connecting user the index.html file and communication between clients.

4.3.1.3 Context Diagram

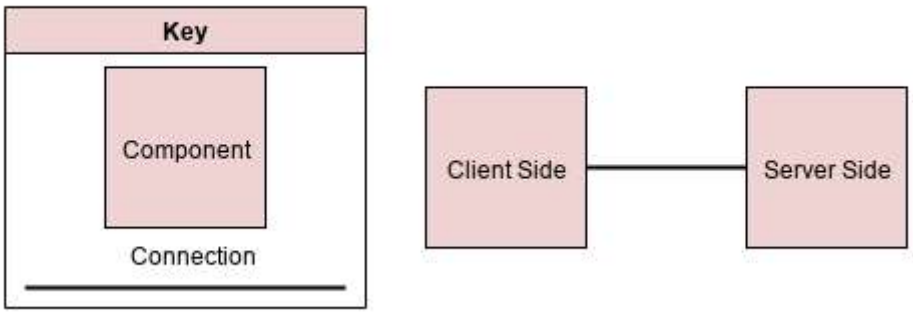

Figure 7: Client and Server Context Diagram for Deployment View

All of the contents in the Desktop PC node can be found on the client-side component. The server-side component contains all items on the Heroku Server node in the primary presentation.

4.3.1.4 Variability Guide

Using a Service Other Than Heroku: to use another hosting service or to host Battle Tanks on your own, only the Node.js runtime environment is required to be installed.

4.3.1.5 Rationale

The architecture we used is a client-server architecture; client-server architecture was chosen for its affinity for scalability and modifiability. By having users connect to the web server to download the game's resources, updates can be applied to the server architecture without having to worry about external clients being maintained. A client-server architecture where the server is available over the internet also improves availability, as any device with an internet connection and a web browser can load and play Battle Tanks.

5 Mapping Between Views

The items in the first column of the table are associated with the association described in the association column to the items in the second column.

5.1 Client & Server Communication View Mapped to Deployment View

Table 1: Mapping Between Client and Server Communication View and Deployment View

4.2.1 Client and Server Communication View Element	**4.3.1 Deployment View Element**	**Association**
GameManager	Desktop PC	deployed-on
Server	Heroku Server	deployed-on

5.2 Package Decomposition View Mapped to Deployment View

Table 2: Mapping Between Package Decomposition View and Deployment View

4.1.1 Package Decomposition View Element	**4.3.1 Deployment View Element**	**Association**
ServerPackage	server.js	implemented-by
ClientPackage	classes.js	implemented-by
GamePackage	game.js	implemented-by

6 Rationale

During design of the Battle Tanks application, there were three primary objectives that were considered when decisions of system-wide import were made. These three objectives were centered on three respective quality attributes: modifiability, usability, and performance.

The first objective is focused on modifiability: to add variety to our game, we want to be able to provide a range of Tank Shells and Tanks to choose from. For example, some shells may be heavier than others, dealing more damage while travelling a shorter distance. Certain tanks could have more armor but move slower, or they could be smaller and harder to hit. Players should be able to swap shells during their turn, so the selected shell needs to be able to change at runtime. Each shell should have a unique sprite, may need unique physics, and unique animations. Players should be able to choose a tank at the start of the game, so the Tank needs to be modifiable at runtime. Each Tank should have it’s own physics, sprite, and unique animations.

The second objective is focused on usability: ensuring that users are confident in our game running properly and that users are satisfied with the experience is of great importance for the design process. The methods of increasing this confidence and satisfaction should be measurable with distinct response measures like the ratio of successful operations to total operations and user satisfaction surveys. In order to produce insights into increasing user satisfaction, concrete usability tests involving users playing the game for a restricted time period could be performed. From those usability tests, response measures can be retrieved and used for further iterations of increasing user satisfaction.

The third objective is focused on performance: we want to minimize the amount of memory and processing needed to run a game instance so we can have multiple game instances running at one time. Battle Tanks uses Heroku for hosting the server.js artifact, and the free version provides 512MB of RAM and 550-1000 dyno hours per month which corresponds to hours of runtime.

As previously mentioned, all three of the aforementioned objectives were considered not only for design decisions that pertain to the four views described in Section 4, but also for decisions that extend beyond views. For example, the simple method for starting a game where one user connects to the server and creates a game instance automatically was selected with the usability objective in mind, along with the project time constraints. These project time constraints were also a major factor when making architectural decisions, as making the design too complex could result in an unpolished, or even unfinished product.

Regarding fundamental architectural patterns, the client-server architecture is of most import (though the usage of the Strategy pattern described in Section 4.1.2.5 was also integral to the design). As mentioned in Section 4.3.1, the client-server architecture was chosen for its affinity for scalability and modifiability. By having users connect to the web server to download the game's resources, updates can be applied to the server architecture without having to worry about external clients being maintained. A client-server architecture where the server is available over the internet also improves availability, as any device with an internet connection and a web browser can load and play Battle Tanks.

7 Directory

7.1 Index

7.2 Glossary

Heroku - an online web application hosting service (6)

UML - an acronym that stands for the Unified Modelling Language (4)