JSTanks - Design Document

Jiahao Li	Pavithran Pathmarajah	Viren Patel
LI577	PATHMAP	PATELVH3
001416646	001410729	001419057

November 6, 2016

Contents

1 Revision History

1.1 Revision 0

Table 1: Revision History

Date	Developer	Change
November 6	Jiahao Li	Initial Draft
November 6	Pavithran Pathmarajah	Initial Draft
November 6	Viren Patel	Initial Draft

2 Introduction

Module Decomposition is a good practice when it comes to developing applications involving many different data structures. Following this can benefit the program in many ways including the implementation process, maintenance, testing, and understanding of the program to new entities.

Most programmers use the process of decomposing their program into different modules which represent different aspects of the program. This can allow the programmer to think of one module/aspect of the program at a time. Thus, simplifying the implementing process.

It is easy to maintain a program which has its components implemented in separate modules. For example, the debugging process in case there are errors in the program, can be time consuming if the program has not been decomposed. On the other hand, a program that has been decomposed into modules would have the compiler point at the module in which the error occurs and therefore, the programmer only has to look through that module ignoring, but not completely, the code in the other modules.

Testing is one of the most vital phases in the development process. A program which shows no module decomposition can make the testing process significantly tough for the programmer. This is because although he can test different functions of the program, he cannot tell if specific aspects of the program are working fine or not. In contrast, in a well decomposed program, the programmer can test out different modules and know for sure if they are working fine or not. He does not have to test the whole code in order to check a single module/aspect of the program.

Module Decomposition makes it easier for stakeholders other than the developers, to understand the program and know how it accomplishes different tasks. They can learn about each module and connect them together to know how the

program functions as one.

This document is a Module Guide for JSTanks. JSTanks has followed the process the Module Decomposition greatly and can be seen in this doument.This document goes in to detail to show how the things mentioned in the Software Requirements Specifications document are accomplished in the program. We plan to use this document for all the purposes discussed above.

This document has been divided into the following sections: Anticipated and Unlikely Changes of the software requirements, Module Hierarchy, Connection between Requirements and Design, Module Decomposition, Traceability Matrix, and Use Hierarchy between Modules.

3 Anticipated and Unlikely Changes

This sections contains possible changes to the project which may occur. The changes are placed into two categories anticipated and unlikely, depending on how likely a change is to occur.

3.1 Anticipated Changes

Anticipated changes are sources within modules which may be modified throughsout the products life time. The changes will be made to specific modules in such a way that it will not affect how modules interact, or respond, but how they work.

AC1

Each tile pre-render's it's own image when it is created. Will be changed such that shared images are pre-rendered on startup

AC2

Currently a single thread runs procedurally. Updated game will have a sole update and render thread in the background. Preventing browser lock up.

AC3

Level and map specifications may be moved to an external text or Html file to be parsed by the game dynamically. Allowing for the addition of new maps without having to hardcode scripts.

AC4

Artificial Intelligence of bots, to be replaced with an algorithm to try and attack the player tank and the home-base depending on what is closer.

AC5

The strength of projectiles to go down depending on number of grid spots away from the tank they have travelled.

AC6

Only re-rendering changed entities instead of the current system where all entities are re-rendered each frame.

3.2 Unlikely Changes

Unlikely changes are the changes that will most likely not occur, for the affects of these changes would require many modifications to many modules for it would have an effect on how modules interact and respond.

UC1

The data that tile-entities store and provide will remain at a minimum of health information and graphics renders.

UC2

The game will continue to be purely, HTML, CSS, an JavaScript such that it modules will not have to be ported to another language or ecosystem

UC3

All interfacing between entities and the user will continue to propagate through the gameBoard class.

UC4

Game graphics will not be changed from a three level render stack to, a two level stack. Since it would require changes across all modules.

4 Module Hierarchy

This section provides a simple overview of the module design. The modules are hierarchically labelled M1 through M13 below and are decomposed in the table below that.
Hierarchy. These are the modules which will be implemented.

M1: Game Module
M2: GameBoard Module
M3: Projectile Queue Module
M4: Create Game Module

M5: Player Module
M6: Bot Module
M7: Tank Module
M8: Empty Tile Module
M9: Projectile Module
M10: Home-Base Module
M11: Wall-Module
M12: Steel-Wall Module
M13: Overlay Module

Level 1	Level 2
Game Module	- Set graphics area - Create graphics context - start main game loop - interface between key presses, Overlay menu and GameBoard - GameBoard Module - Overlay menu
GameBoard Module	- Interface between user and game - update board graphics - update projectiles - move entities on request - Create Game Module - Projectile QueueModule - Player Module - Bot Module - Wall Module - Steel-Wall Module - Home-Base Module

	- Empty Tile Module
Projectile Queue Module	- Array of Projectiles - draw projectiles on board - Projectile Module
Create Game Module	- Place tiles on board - Player Module - Bot Module - Wall Module - Steel-Wall Module - Home-Base Module
Player Module	- Keyboard interfacing with tank - Tank Module
Bot Module	- Artificial Intelligence interfacing with tank - Tank Module
Tank Module	- tank movement - fire projectiles - tracks tank health
Empty Tile Module	- No strength factor - Can be moved to - replaces destroyed entities
Projectile Module	- Damage tile entities - identify tile hit
Home-Base Module	- Player' home base will trigger end game on destroy - tracks base health
Wall Module	- A weak wall, that can be destroyed

	- tracks wall health
Steel-Wall Module	- A weak wall, that can be destroyed - tracks steel-wall health
Overlayl Module	- Pauses game loop - overlay's pause menu

5 Connection between Requirements and Design

The requirements included in the Software Requirements Specifications document are implemented into the following modules:

JSTanks.html
Tanks.css
Overlay.js
Game.js
GameBoard.js
createBoard.js
Tank.js
Player.js
Bot.js
barrier.js
Projectile.js

The specific objects involved in these modules are listed in the Module Hierarchy section of this document. The corresponding modules of all function and non-functional requirements can be found in Table 4.

6 Module Decomposition

Modules are decomposed according to the principles of "information hiding" proposed by Parnas ed el (1984). We have made the design decision to not implement traditional information hiding, such that the project may be used by others whom are starting out in game design or programming, whom will be able to easily decipher our project and build their own web based game. By hiding how our key modules function internally, these new-comers will not be able to fully understand how to develop a basic HTML, CSS, JavaScript game; for this reason JSTanks has opted out from information hiding. All module decomposition can be found in the Module Interface Specification.

7 Traceability Matrix

Requirement	Modules
	Functional Requirements
FR1	JSTanks.html
FR2	JSTanks.html
FR3	JSTanks.html, Tank.css, Game.js, GameBoard.js, barrier.js, createBoard.js, Tank.js, Bot.js, Overlay.js
FR4	Overlay.js
FR5	Overlay.js
FR6	Overlay.js
FR7	Overlay.js
FR8	Overlay.js
FR9	Overlay.js
FR10	Overlay.js
FR11	Overlay.js
FR12	Overlay.js
FR13	Bot.js, Projectile.js, Tank.js
FR14	Bot.js
FR15	Bot.js
FR16	Overlay.js
FR17	Player.js, Tank.js
FR18	Projectile.js
FR19	Projectile.js,Tank.js, Bot.js
FR20	GameBoard.js, barriers.js, Projectile.js
FR21	GameBoard.js, barriers.js, Projectile.js
FR22	GameBoard.js, barriers.js, Projectile.js
FR23	Projectile.js,Tank.js,Player.js
FR24	EndGame.js
FR25	EndGame.js
	Non-functional Requirements
NF1.1	JSTanks.html, Tank.css, Game.js, GameBoard.js, barriers.js, createBoard.js, Tank.js, Bot.js
NF1.2	JSTanks.html, Tank.css, Game.js, GameBoard.js, Tank.js, Bot.js, Overlay.js
NF2.1	Tank.js, Player.js, Projectile.js
NF2.2	–
NF2.3	JSTanks.html
NF2.4	Overlay.js
NF2.5	–
NF3	–
NF4	JSTanks.html, Tank.css
NF5	Game.js, GameBoard.js, Tank.js, Player.js, Bot.js, Overlay.js, barrier.js, createBoard.js, Projectile.js
NF6	–
NF7	Overlay.js
NF8	–

Table 4: Trace Between Requirements and Modules

8 Use Relations

Module		Use Relation
JSTanks.html	uses	Tanks.css, Overlay.js, Game.js, GameBoard.js, createBoard.js, Tank.js, Player.js, Bot.js, Barrier.js, and Projectile.js
Tanks.css	uses	–
Overlay.js	uses	–
Game.js	uses	GameBoard.js
GameBoard.js	uses	Player.js, Bot.js, barrier.js, and Projectile.js, and createBoard.js
createBoard.js	uses	barrier.js
Tank.js	uses	–
Player.js	uses	Tank.js
Bot.js	uses	Tank.js
barrier.js	uses	–
Projectile.js	uses	–

Table 5: Uses Relationship between Modules

List of Tables

List of Figures