SOFTWARE ARCHITECTURE DOCUMENT

Project Bomberman: Reloaded

MARCH 2, 2014

TEAM RETRO REVOLUTION
Written by:
Hang Kang, Simon Krafft, Qianyu Liu, Alistair Russell, Sean Ryan, Wayne Tam

Table of Contents

1. System Overview

1.1 Purpose

This Software Architecture Document (SAD) is the second part of the construction of the Bomberman project. The document will describe the system, focusing on aspects and requirements that are relevant the system architecture. The purpose of the document is to present an overview of the system's software architecture and design. The document includes detailed description of all classes, and their associated attributes and operations.

The SAD is a general guide for the design team during the implementation phase. The document is intended for current developers working on the project, as well as for future developers who maintain the system and for client who requests the project. It should be noted that this document is subject to change in the later phases, and may be modified when necessary.

1.2 Architectural Design

This software architecture design emphasizes on different individual modules, components, and classes. Such model allow each subsystems to be developed separately, which, at the end, will be linked together. Doing so, we reduce interdependencies between classes. Most importantly, the design is subject to change when necessary. Therefore, by dividing the system into subsystems, we improve and make the design flexible to any change in the future.

1.3 Overview

The following sections will address the detailed architecture design of the game system.

The View (Section 2) consists of several unified modelling language (UML) diagrams. The UML diagrams draw the skeleton of the implementation architecture of the software. It will map out all the required classes and interfaces, with links that depicts the relationships between classes. Doing so will help to ease the link of all subsystems and requirements of the project. The class diagrams also provide a high-level overview of all the attributes and operations included in each classes of the architecture.

The Software Subsystem (Section 3) describes the purpose of subsystems, addressing the concern of the service and/or responsibility of each subsystems. Examples of subsystems that will be mentioned are Enemy, Bomberman, Bomb classes. The section is also made up of all attributes and operations mentioned in the View section, organizes them into different subsystems, and describes in detail of the functions of each attributes and operations in correspondence to the Software Requirements Specification (SRS) document from Phase 1. Lastly, this sections describes the interface offered by each subsystem, defining the signatures of methods if possible, as well as any constraints on the input and/or output

The Analysis (Section 4) explains how the proposed architecture will address functional requirements described in the SRS document using traceability matrices. Analysis summarizes any quality requirements identified in the SRS. This section also identifies any features and/or requirements are likely to change, if any, and addresses how the system design will need to be modified.

The last section, Design Rationale (Section 5) provides explanations and justification to any non-obvious design decisions made while producing the architecture.

2. Views

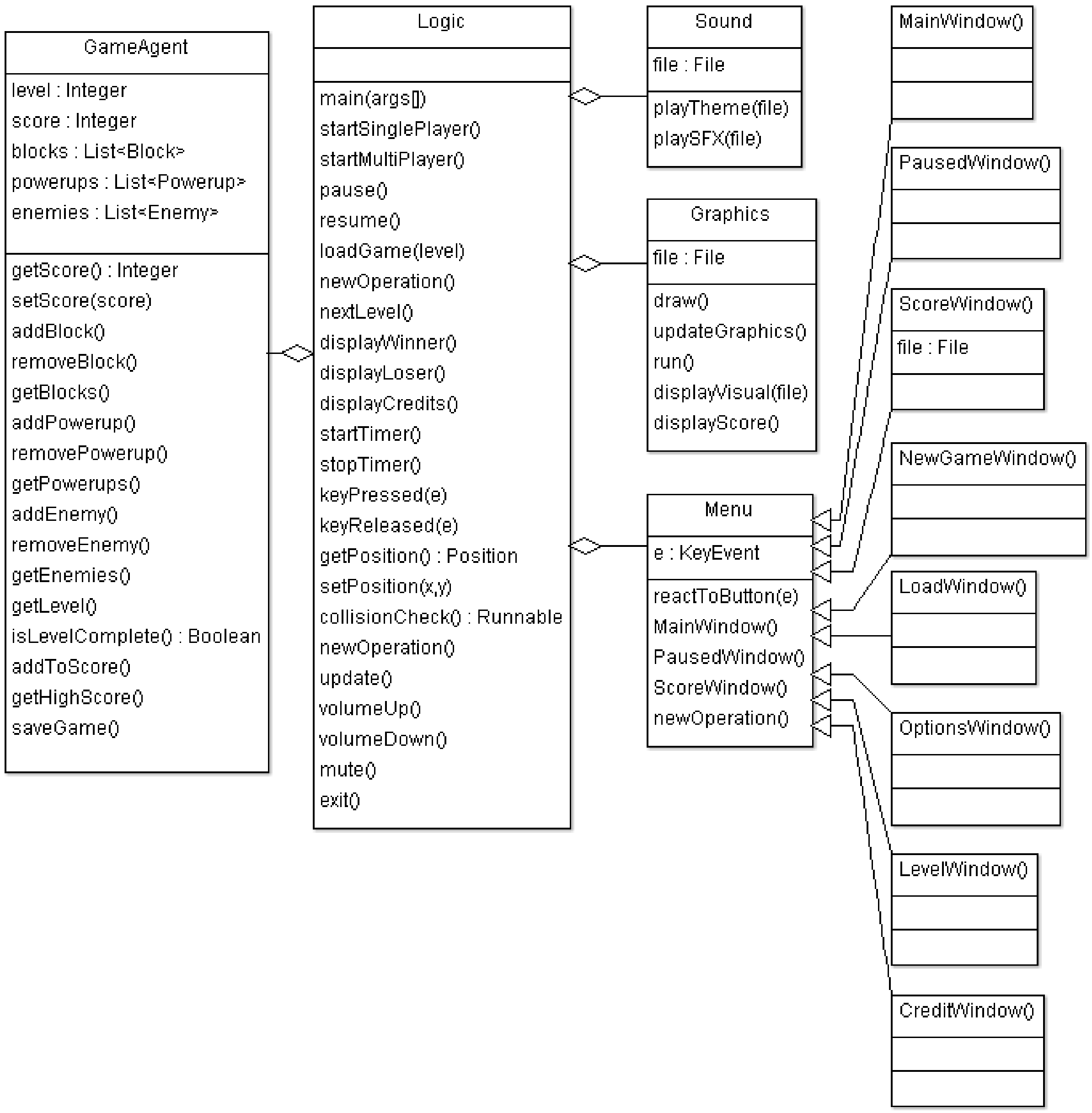
2.1 General Overview

A general diagram describing interaction between packages:

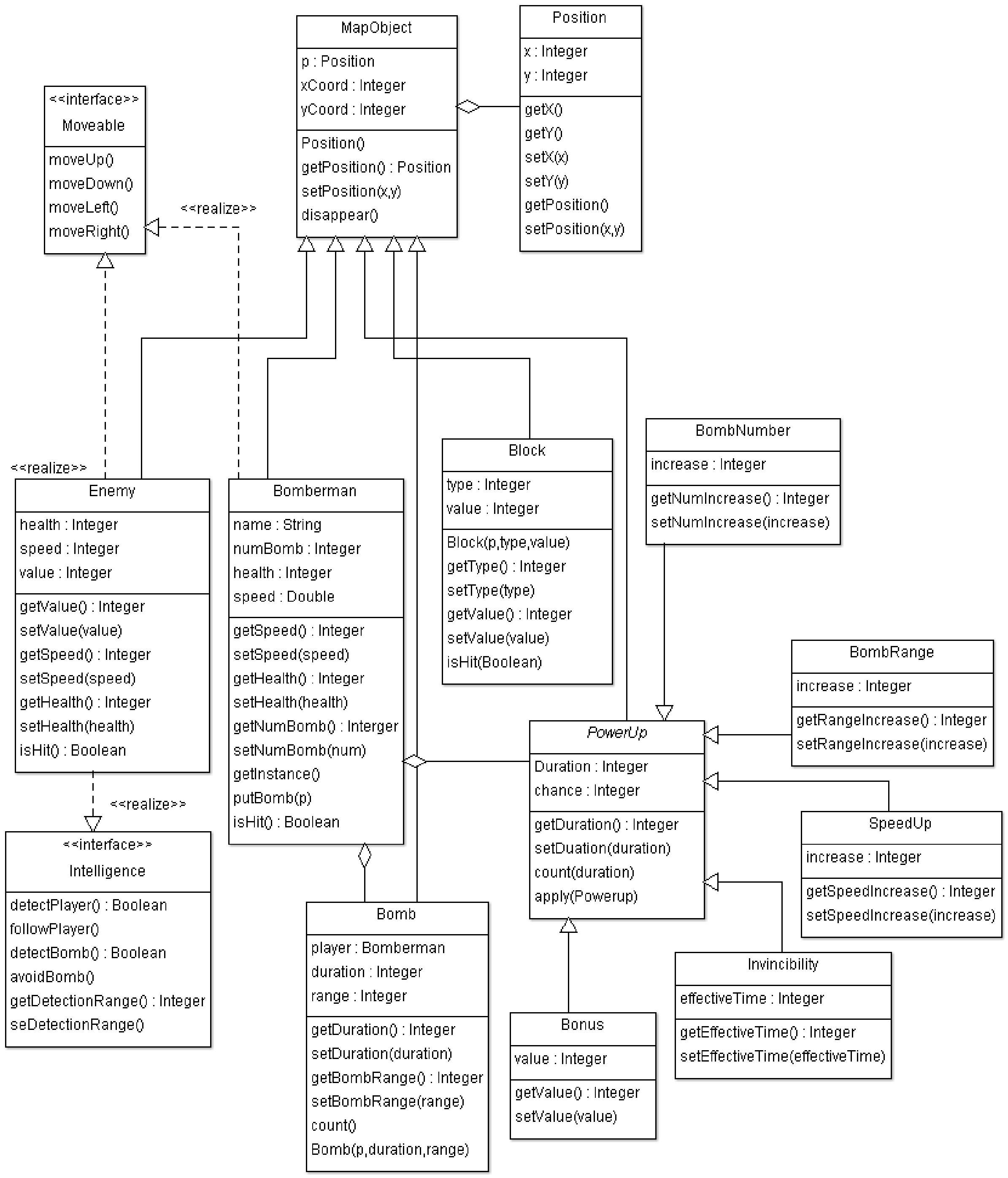
2.2 Detailed Unified Modeling Language (UML) Diagrams

Class diagrams for each subsystem

2.2.1 Logic, Menu, Sound, Graphics and GameAgent



2.2.2 MapObject, Position and Intelligence



3. Software Subsystems/Modules

3.1 Animations

Purpose

This class uses the buffered images to create the game animations.

Interface

Animations(int speed, BufferedImage...args)
This constructor creates a new animation.

void runAnimation()
This methods run the animation.

nextFrame()
This creates the next frame of the animation.

void drawAnimation(Graphics g, int x, int y,int scaleX, int scaleY)
This methods draws the animation at the specified coordinates.

3.2 Sound

Purpose

The purpose of Sound class is to load and play the audio files to the game. Each level has a corresponding music theme. Actions (death, explosion, destruction) and event announcements (advancement, game over) have a sound effects. All audio files will be located in a specific folder.

Interface

Sound(String filename)
This constructor creates a new sound object.

void play()
This method plays the current sound.

3.3 Menu

Purpose

The purpose of a Menu class is to provide the frame of the *Bomberman* game. This frame will output the main menu screen.

Interface

public void render(Graphics g)
This method creates the main menu.

3.4 Position

Purpose

The purpose of Position class is to pinpoint the location of each object on the map.

Interface

Position(int x, y)
This constructor initializes a Position point.

int getX()
This returns the x coordinate of the object.

void setX(int x)
This sets the x coordinate of the object.

int getY()
This returns the y coordinate of the object.

void setY(int y)
This sets the y coordinate of the object.

3.5 Moveable

Purpose

Moveable adds movements to all classes that implement it. Moveable is an interface implemented by Bomberman and Enemy class.

Interface

void moveUp(int velocity)
This method moves the object upward at a specific speed.

void moveDown(int velocity)
This method moves the object downward at a specific speed.

void moveLeft(int velocity)
This method moves the object leftward at a specific speed.

void moveRight(int velocity)
This method moves the object rightward at a specific speed.

3.6Intelligence

Purpose

The purpose of Intelligence is to serve as a controller for the advanced enemy. Intelligence is how the enemy moves and reacts with accordance to other object on the map. Enemy implements Intelligence.

Interface

void setDetectionRange(int range)
This method sets the range in which the enemy can detect.

int getDetectionRange()
This method returns the range in which the enemy can detect.

boolean detectPlayer()
This method checks whether Bomberman is in the proximity

void followPlayer()
This method adds the object into the list of blocks

boolean detectBomb()
This method adds the object into the list of blocks

void avoidBomb()
This method adds the object into the list of blocks

3.7 GameObject

Purpose

The purpose of the GameObject class is to provide a general representation of the game objects, such as all the characters, bombs, power ups. The class includes functionality present in all the objects.

Interface

GameObject(Position p, ObjectId id)
This constructor initializes a new game object.

float getVelX()
This method returns the velocity of the object in the x direction.

float setVelX()
This method sets the velocity of the object in the x direction.

float getVelY()
This method returns the velocity of the object in the y direction.

float setVelY()
This method sets the velocity of the object in the x direction.

void moveUp(int velocity)
This method moves the object upward with a specific velocity.

void moveDown(int velocity)
This method moves the object downward with a specific velocity.

void moveLeft(int velocity)
This method moves the object to the left with a specific velocity.

void moveRight(int velocity)
This method moves the object to the right with a specific velocity.

ObjectId get Id()
This method returns the Id of the object.

Position getPosition()
This returns the position of the MapObject.

void setPosition(Position p)
This sets the position of the MapObject.

3.8 Player

Purpose

The Player class defines all the attributes and operations appropriate to the Bomberman that are not included in GameObject. Bomberman extends GameObject.

Interface

Player(Position p, Handler handler, ObjectId id)
This constructor initializes a Bomberman character.

void tick(LinkedList<GameObject> object)
This method is used to constantly update important variables such as the moving speed and colision related variables.

void Collision()
This method handles collision between the player and other objects on the map.

void render(Graphics g)
This method renders the player object on the map.

Rectangle getBounds()
This method returns the boundaries of the player object. This method is for collision purposes

boolean isBomded(Bomb bomb)
This method checks if the player was hit by a bomb.

Boolean isExited()
This method checks if the game exited.

3.9 Enemy

Purpose

The Enemy class defines all the attributes and operations appropriate to the enemy that are not included in MapObject. Enemy inherits GameObject and implements Intelligence.

Interface

Enemy(Position p, Handler handler, int type, ObjectId id)
This constructor initializes an enemy character.

Void tick(linkedList<GameObject> object)
This method is used to constantly update enemy variables such as position.

void Collision(LinkedList<GameObject> object)
This method handles collision between enemies and objects on the map.

Rectangle getBounds()
This method returns the bounds of the rectangle that contains the enemy. Used for collision purposes.

Int getValue()
This method gets the value of an enemy. Used to calculate the score.

void setValue(int pointsWorth)
This method sets the value of an enemy.

int getSpeed()
This method returns enemy’s current speed

void setSpeed(int newSpeed)
This method sets enemy’s speed.

int getHealth()
This method returns enemy’s current health

void setHealth(int health)
This method sets enemy’s health.

boolean isHit()
This method checks if the enemy is hit.

void moveUpAI()
This method is used to move the AI upwards.

void moveDownAI()
This method is used to move the AI downwards.

void moveRightpAI()
This method is used to move the AI to the right.

void moveLeftAI()
This method is used to move the AI to the left.

void setDetectionRange(int range)
This method sets the detection range of the enemy.

int getDetectionRange()
This method returns the detection range.

boolean detectPlayer()
This method checks if the player is within range of the enemy.

void followPlayer()
This method makes the enemy follow the player when detectPlayer() returns true.

void moveRandomly()
This method makes the enemy move randomly on the map.

boolean isBombed(Bomb bomb)
This method checks if the enemy has been hit by a bomb.

3.10 Bomb

Purpose

The Bomb class defines all the attributes and operations appropriate to the bomb that are not included in GameObject. Bomb inherits properties and methods from GameObject class.

Interface

Bomb(Position p, ObjectId id)
This constructor initializes a bomb.

void tick()
This method constantly updates variables related to the bomb object.

void render(Graphics g)
This method renders a bomb on the map.

Rectangle getBounds()
This method returns the bounds of the rectangle that contains the bomb.

boolean exploded(Bomb bomb, LinkedList<GameObject> object)
This method handles the explosion of the bomb.

3.11 Power

Purpose

The Power class defines all the attributes and operations appropriate to power-ups that are not included in GameObject. Power inherits from GameObject.

Interface

Power(Position p, int type, ObjectId id)
This is the constructor for the Power class.

void tick(LinkedList<GameObject> object)
This method updates variables related to powers.

void render(Graphics g)
This method renders a power on the map.

void speedUp(int type)
This method defines the power-up speedUp. It increases the speed of the player.

void doubleUp(int type)
This method defines the power-up speedUp. It doubles the current score.

void healthUp(int type)
This method defines the power-up healthUp. It gives the player one extra life.

void rangeUp(int type)
This method defines the power-up rangeUp. It increases the range of the bomb explosion.

boolean consumed(Power power, LinkedList<GameObject> object)
This method checks if the power has been consumed by the player.

void selectPower(int type)
This method decide the chance of making a power-up appear when destroying a block.

3.12 Block

Purpose

The Block class defines all the attributes and operations appropriate to blocks that are not included in GameObject. Bomb inherits properties and methods from GameObject class

Interface

Block(Position p, int type, ObjectId id)
This constructor creates blocks at the indicated coordinate positions, type, and value.

void tick(LinkedList<GameObject> object)
This method constantly updates all variables related to blocks.

void render(Graphics g)
This method renders the block on the map.

Rectangle getBounds()
This method returns the bounds of the rectangle that contains the block.

Boolean isBombed(Bomb bomb)
This method checks if the block has been hit by a bomb.

3.13 Game

Purpose

The game class contains the main method and all the necessary methods to run the game and create all the objects. It extends canvas and implements Runnable.

Interface

void init()
This method initializes all the necessary components to run the game.

void start()
This method defines the start of the game.

void run()
This method is main game loop.

void tick()
This method constantly calls all the tick methods from objects on the map.

void render()
This method renders the running game.

void LoadImageLevel(BufferedImage image)
This method is used to load a specific level.

Texture getInstance()
This method returns the texture.

void main(String args[])
This is the main method of the program. It creates the game window.

3.14 KeyInput

Purpose

The KeyInput class creates the controls of the game. It extends KeyAdapter.

Interface

void keyPressed(KeyEvent e)
This method handles all the events that happen when a key is pressed down.

void keyReleased(KeyEvent e)
This method detects when the key is released.

3.15 MouseInput

Purpose

The MouseInput class creates the areas that can be clicked and what happens when you do. It implements MouseListener.

Interface

void mousePressed(MouseEvent e)
This method handles all the mouse click events.

3.16 ObjectId

Purpose

Interface

3.17 SpriteSheet

Purpose

Interface

SpriteSheet(BufferedImage image)
Constructor for the SpriteSheet class.

BufferedImage grabImage(int col, int row, int width, int height)

3.18 STATE

Purpose

This enum class hold the variables for the different states of the game.

3.19 Texture

Purpose

The Texture class creates the appearance of every object in the game using an image.

Interface

Texture()
Constructor for the Texture class.

3.20 Credits

Purpose

This class is used to create the sub-menu: Credits.

Interface

void render(Graphics g)
This method renders the items in the credits menu.

3.21 GameCover

Purpose

This class is used to create the cover of the game. The first image that appears when launching the game.

Interface

void render(Graphics g)
This method renders the cover image of the game.

3.22 GameOver

Purpose

This class is used to create the game over screen when the player loses.

Interface

void render(Graphics g)
This method renders the Game Over screen.

3.23 Highscores

Purpose

This class is used to create the sub-menu: Highscores.

Interface

void render(Graphics g)
This method renders the high score menu.

void scoreList()
This method displays the list of high scores.

void addList()
This method adds a high score to the list.

3.24 HowToPlay

Purpose

This class is used to create the sub-menu: how to play.

Interface

void render(Graphics g)
This method renders the instruction menu.

3.25 InGameMenu

Purpose

This class is used to create the in game menus.

Interface

void render(Graphics g)
This method renders the in-game menu when pressing escape.

3.26 LevelSelect

Purpose

This class is used to create the sub-menu: Level select.

Interface

void render(Graphics g)
This method renders the level select menu.

3.27 PauseOptions

Purpose

This class is used to create the sub-menus in the in-game menu.

Interface

void render(Graphics g)
This method renders the different sub-menus when using the pause function.

3.28 Crate

Purpose

This class is used to create the destructible blocks. It extends GameObject.

Interface

Rectangle getBounds()
This method returns the bounds of the rectangle that contains the crate

boolean isBombed(Bomb bomb)
This method checks if the crate has been hit by a bomb.

void render(Graphics g)
This method renders a crate on the map

Crate(Position p, float y, ObjectId id)
Constructor method for the crate class.

3.29 ExitDoor

Purpose

This class is used to create the exit door that brings the player to the next level. It extends GameObject.

Interface

ExitDoor(Position p, ObjectId id)
Constructor method for the ExitDoor class.

void render(Graphics g)
This method renders an exit door on the map.

Rectangle getBounds()
This method returns the bounds of the rectangle that contains the exit door.

3.30 BufferedImageLoader

Purpose

This class is used to load the images.

Interface

BufferedImage loadImage(String path)
Load the images and returns a buffered image.

3.31 Handler

Purpose

This class mainly handles all the game objects by storing them in a LinkedList of type GameObject.

Interface

void tick()
This method constantly updates variables related to the handler.

void render(Graphics g)
This method renders the objects contained in the handler list of objects.

void addObject(GameObject object)
This method adds an object to the list of objects

void removeObject(GameObject object)
This method removes an object from the list of objects.

3.32 window

Purpose

This class is used to create the windows needed for the game.

Interface

window(int w, int h, String title, game game)
Constructor method for the window class.

4. Analysis

This section demonstrates the coherency of the Software Architecture Document and the Software Requirement Specifications.

4.1 Functional Requirements

Categories

Logic	**Graphics**	**MapObject**	**Intelligence**	**Game**	**Sound**
Logic	Graphics	Position	Intelligence	GameAgent	Sound
	Menu	Moveable			
	GUI	Bomberman			
		Enemy			
		Bomb			
		Powerup			
		Block			

Traceability Matrix

The following traceability matrix demonstrates the coherency of the SAD and SRS

Requirements	**Logic**	**Graphics**	**MapObjects**	**Intelligence**	**Game**	**Sound**
3.1. CT.1.1			moveUp()			
3.1. CT.1.2	keyPressed()		moveDown()			
3.1. CT.1.3	keyReleased()		move Left()			
3.1. CT.1.4			moveRight()			
3.1. CT.2.1	keyPressed()					
3.1. CT.3.1	keyPressed()					
3.1. MN.1.1		reactToButton()				
3.1. MN.1.2	keyPressed()					
3.1. MN.2.1		MainWindow()				
3.1. MN.3.1	pause()	PausedWindow()				
3.1. MN.3.3	resume()					
3.1. MN.4.1		NewGameWindow()				
3.1. MN.4.2	setPlayerName()					
3.1.		ScoreWindow				

MN.5.1		()				
3.1. MN.5.2	update()					
3.1. MN.6.1	loadGame()	LoadWindow( )				
3.1. MN.6.2	loadGame()	LoadWindow( )				
3.1. MN.7.1	loadGame()	LevelWindow ()				
3.1. MN.7.2	loadGame()	LevelWindow ()				
3.1. MN.8.1		OptionsWind ow				
3.1. MN.8.2	update()					
3.1. MN.8.3		draw()				
3.1. MN.9.1		CreditWindo w()				
3.1. MN.9.2	displayCred its()					
3.1. MN.10.1					saveGame()	
3.1. MN.10.2					saveGame()	
3.1. MN.11.1		OptionsWind ow()				
3.1. MN.12.1	keyPressed( )					
3.1. MN.12.2	keyPressed( )					
3.1. MN.12.3	exit()					
3.1. MN.12.4	resume()					
3.1. MN.13.1	startGame()	NewGameWi ndow()				
3.1. MN.13.2	startGame()	LevelWindow ()				
3.1. MN.13.3	startGame()	LoadWindow( )				
3.1. MN.14.1						playSFX () playThe me()
3.1.	resume()	playVisual()				

MN.15.1			
3.1.BM.1.1		draw()	
3.1.BM.1.2		draw()	
3.1.BM.2.1	keyPressed()		
3.1.BM.2.2	keyReleased()		
3.1.BM.2.3			setSpeed()
3.1.BM.2.4			move()
3.1.BM.2.5			move()
3.1.BM.2.6	keyPressed()		putBomb()
3.1.BM.3.1			setHealth()
3.1.BM.3.2		draw()	getHealth()
3.1.BM.3.3			setHealth()
3.1.BM.4.1			setNumBomb()
3.1.BM.4.2	keyPressed()		putBomb
3.1.BM.4.3		draw()	
3.1.BM.4.4			setBombRange()
3.1.BM.4.5			count()
3.1.BM.4.6			disappear()
3.1.BM.4.7			setPosition()
3.1.AI.1.1		draw()	
3.1.AI.1.2		draw()	
3.1.AI.1.3		draw()	
3.1.AI.2.1			move()
3.1.			move()

AI.2.2				
3.1.AI.2.3		move()		
3.1.AI.4.1		setSpeed()		
3.1.AI.4.2				addToScore()
3.1.AI.4.3		setSpeed()		
3.1.AI.4.4				addToScore()
3.1.AI.4.5		setSpeed()		
3.1.AI.4.6				addToScore()
3.1.AI.4.7		setSpeed()		
3.1.AI.4.8				addToScore()
3.1.AI.4.9		setSpeed()		
3.1.AI.4.10				addToScore()
3.1.AI.4.11		setSpeed()		
3.1.AI.4.12				addToScore()
3.1.AI.5.1			detectBomb() avoidBomb()	
3.1.AI.5.2			detectPlayer() followPlayer()	
3.1.BL.1.1	draw()			
3.1.BL.1.2	draw()			
3.1.BL.2.1	draw()			
3.1.BL.3.1		setType()		
3.1.BL.3.2		move()		
3.1.		setPosition		

BL.3.3			()	
3.1.BL.4.1		draw()		
3.1.BL.5.1			isHit() disappear()	
3.1.BL.5.2		draw()		
3.1.BL.5.3			isHit()	
3.1.BL.5.4		draw()		
3.1.LV.1.1		draw()		
3.1.LV.1.2		draw()		
3.1.LV.2.1		draw()		
3.1.LV.2.2		draw()		GameAgent()
3.1.LV.3.1				addEnemy()
3.1.LV.3.2		draw()		
3.1.LV.4.1	nextLevel()	draw()		
3.1.LV.4.2				isLevelComplete()
3.1.LV.5.1		displayScore()		addToScore()
3.1.LV.5.2		displayScore()		
3.1.LV.5.3	update()			gameAgent()
3.1.LV.5.4			-	saveGame()
3.1.LV.5.5		displayScore()		saveGame()
3.1.LV.5.6				saveGame()
3.1.LV.5.7		displayScore()		
3.1.GP.1.1	setPlayerName()			
3.1.GP.1.2	setPlayerName()			
3.1.	setPlayerNa			

GP.1.3	me()					
3.1. GP.1.4	setPlayerName()					
3.1. GP.2.1		draw()	getInstance() setPosition()			
3.1. GP.2.2			move()			
3.1. GP.3.1	pauseGame()					
3.1. GP.3.2	pause () stopTimer()					
3.1. GP.4.1	resume()	playVisual()				
3.1. GP.4.2	resume() startGameTimer()		move ()			
3.1. GP.5.1		playVisual()	isHit()			playSFX()
3.1. GP.5.2			setHealth()			
3.1. GP.5.3			setPosition			
3.1. GP.5.4			apply()			
3.1. GP.6.1			isHit()			
3.1. GP.6.2			disappear()			
3.1. GP.6.3					addToScore()	
3.1. GP.7.1			isHit()			
3.1. GP.7.2			disappear()			
3.1. GP.7.3					addToScore()	
3.1. GP.8.1			consume()			
3.1. GP.8.2			apply()			
3.1. PU.1.1		draw()				
3.1. PU.1.2		draw()				

3.1. PU.2.1		draw()				
3.1. PU.2.2		draw()				
3.1. PU.3.1			apply()			
3.1. PU.3.2			apply()			
3.1. PU.3.3			apply()			
3.1. PU.3.4			apply()			
3.1. PU.3.5			apply()			
3.1. PU.3.6			apply()			
3.1. PU.3.7			apply()			
3.1. PU.3.8			apply()			
3.1. AV.1.1	volumeUp() volumeDow					
3.1. AV.1.2	n() mute()					
3.1. AV.2.1						playThe me()
3.1. AV.2.2						playThe me()
3.1. AV.2.3						playThe me()
3.1. AV.2.4						playThe me()
3.1. AV.3.1						playSFX ()
3.1. AV.3.2						playSFX ()
3.1. AV.3.3						playSFX ()
3.1. AV.3.4						playSFX ()
3.1. AV.3.5						playSFX ()
3.1. AV.3.6						playSFX ()
3.1. AV.3.7						playSFX ()

3.1. AV.4.1	playVisual()
3.1. AV.4.2	playVisual()
3.1. AV.4.3	playVisual()
3.1. AV.4.4	playVisual()
3.1. SL.1.1	draw()
3.1. SL.2.1	playVisual()
3.1. SL.2.2	playVisual()
3.1. SL.3.1	playVisual()
3.1. SL.3.2	playVisual()
3.1. SL.4.1	playVisual()
3.1. SL.4.2	playVisual()
3.1. SL.4.3	playVisual()
3.1. SL.5.1	playVisual()
3.1. SL.6.1	playVisual()
3.1. SL.6.2	playVisual()

4.2 Quality Requirements

Performance

The program should take 3 seconds or less to show each menu screen. Upon selecting play game and exiting the cut scene the character should immediately respond to input from the user.

The interaction with the in-game menu will be minimized as much as possible to ensure smooth transitions into gameplay.

Reliability

The system should not lag or crash as a result of the design. Any associated problems during the running time of the game should be due to each user's respective operating system. We will implement extensive testing requirements to ensure that the software runs smoothly.

Maintainability

Maintainability is not a key focus of this software architecture design, as the game system is only a semester-long project, as specified in the Software Requirements Specification document.

Securities & Privacy

There is no data collected from the user apart from the optional username so this software should not have any security issues. Furthermore the information collected will not be distributed in anyway outside of the game itself. The game will not be connected to the internet, eliminating the need to protect from unwanted access.

Safety

The system will warn the user to not play the game if they are epileptic. A screen will appear upon start-up to display the required warning.

5. Design Rationale

The architecture design is based on a modular approach, break the system down into individual subsystem classes. Such model allow each class to be developed separately, which, at the end, will be linked through inheritance and association. Doing so, we reduce interdependencies between classes and improve design, making the game system flexible to any change in the future.

5.1 Specific Design Decisions:

5.1.1 Main

The Logic class will track and update the position of every game object and must therefore be accessed by almost every other class.

5.1.2 Controls

Since *Bomberman* is a game that is played primarily with keyboard, it makes sense to make the whole software controllable by the keyboard only. Therefore navigating the menus and playing the game will happen through the use of the keyboard.

5.1.3 Storing Game Data

GameAgent class allows to store all game information into a temporary file. While the game is running, any updates should be recorded. After every level completed, the game should automatically transfer the information recorded on the temporary file and save into a saved file. Whenever the user loads the game, the system reads the saved file and begin from the last save.

5.1.4 Intelligence

The difficulty of the enemies will be based on a criterion interface named Intelligence. An enemy with "intelligence" will have the ability to detect the player within a certain range and follow the player. An intelligent enemy will also have the ability to detect a nearby bomb and tries to avoid it. The range of the detection depends on the intelligence level of the enemy.

5.2 Workload Breakdown

The team will be divided into three groups of two, and work will be distributed accordingly. Every team member will be assigned at least one subsystem and will be responsible for coding all the methods described in the SAD for that subsystem. The breakdown of workload is subject to change during the implementation phase, as each member has different level of programming experience. A general breakdown of the workload as followed:

Group 1: Logic, GameAgent, MapObject (Block)

Group 2: Graphics, Menu, Sound, MapObject (Bomb and Powerup)

Group 3: MapObject (Bomberman and Enemy), Position, Moveable, Intelligence