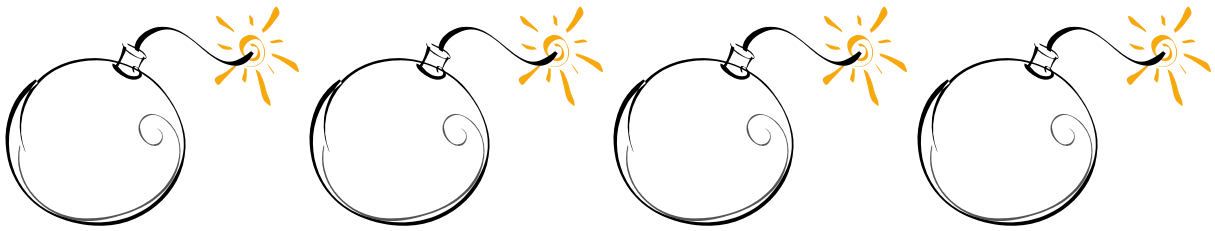
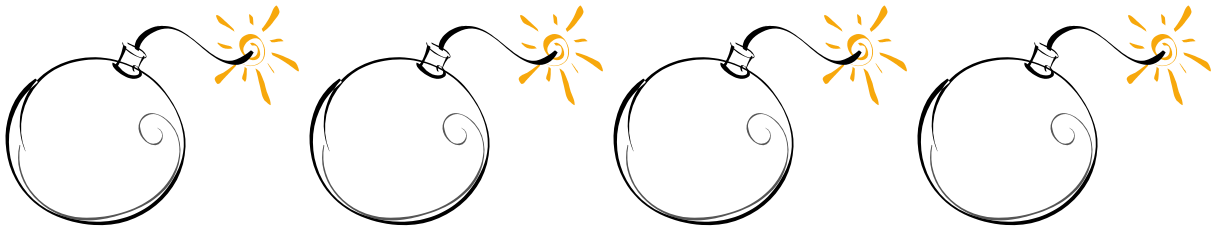


SOFTWARE ARCHITECTURE DOCUMENT

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

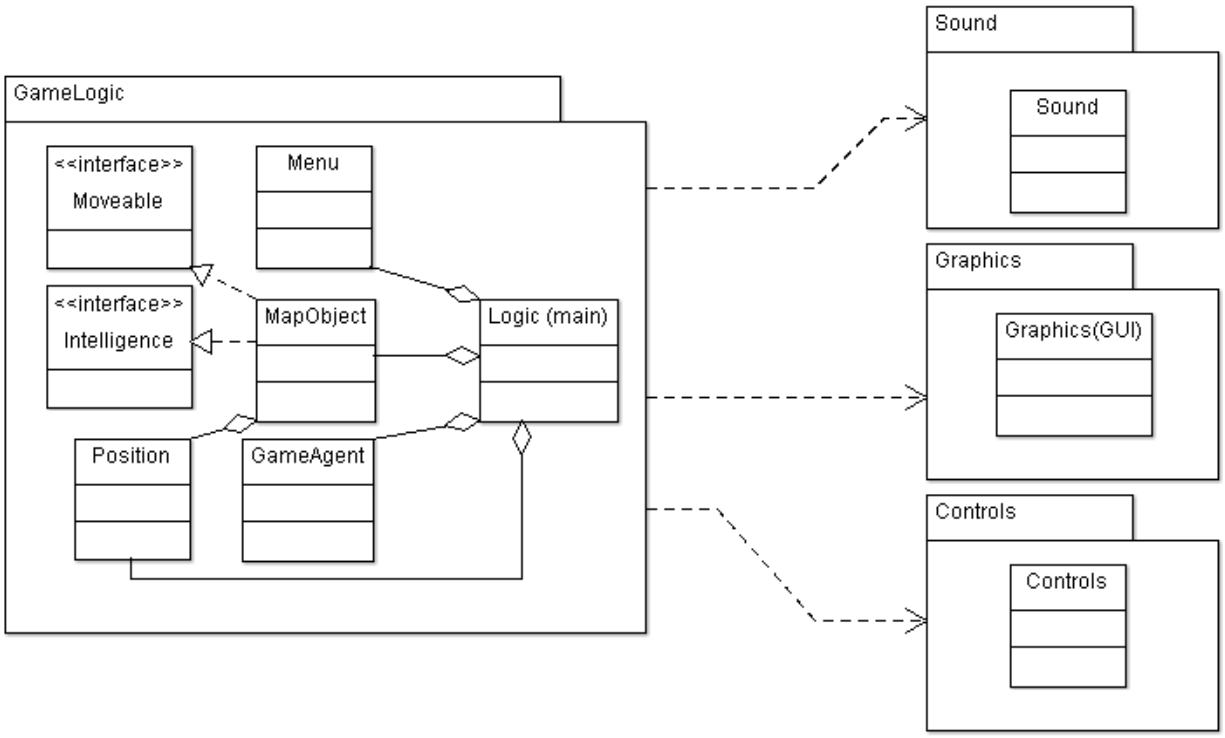
The Analysis (Section 4) explains how the proposed architecture will address functional requirements described in the SRS document using traceability matrices. Analysis summarizes any quality requirements identified in the SRS. This section also identifies any features and/or requirements are likely 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

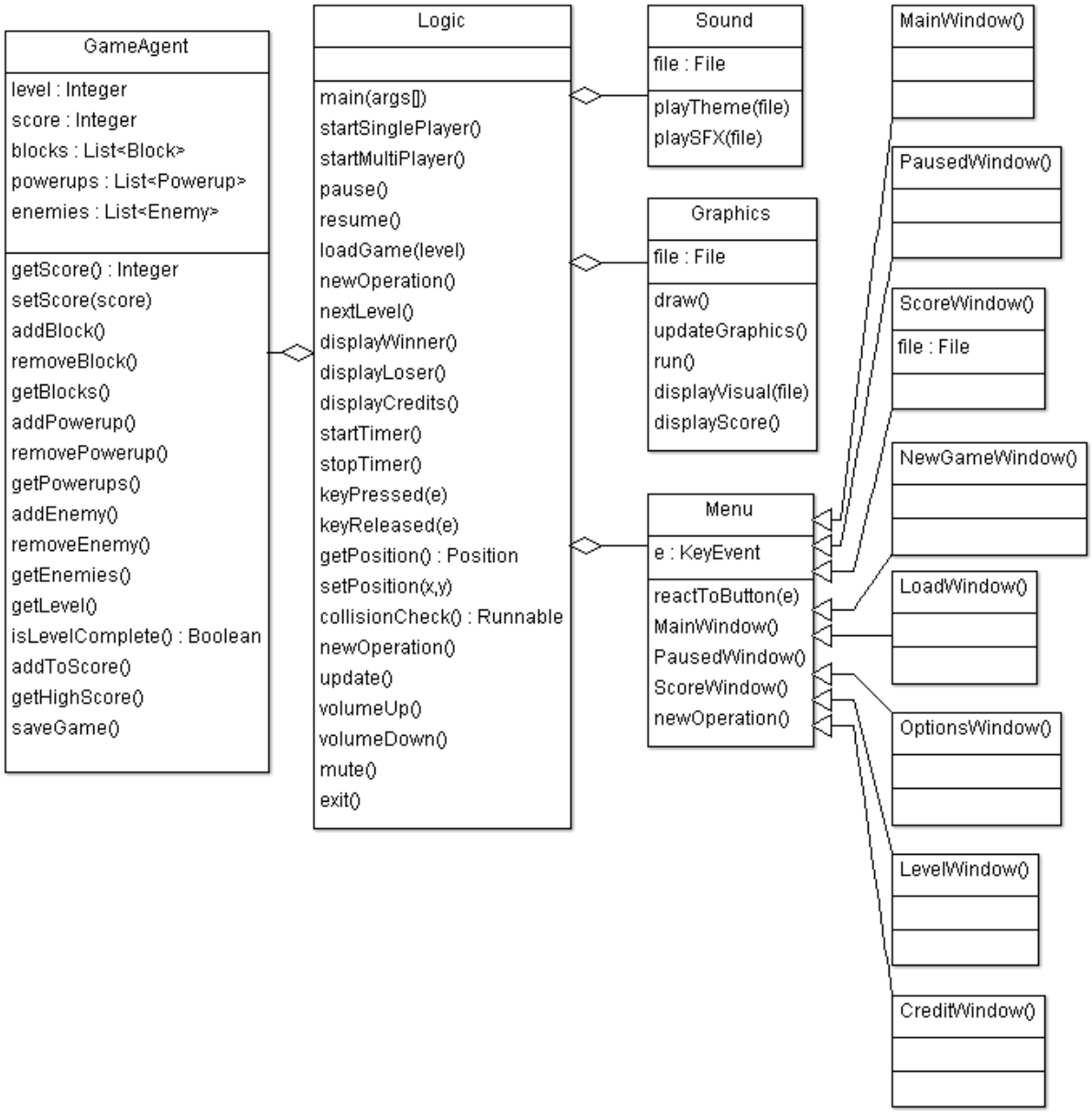
A general diagram describing interaction between packages:



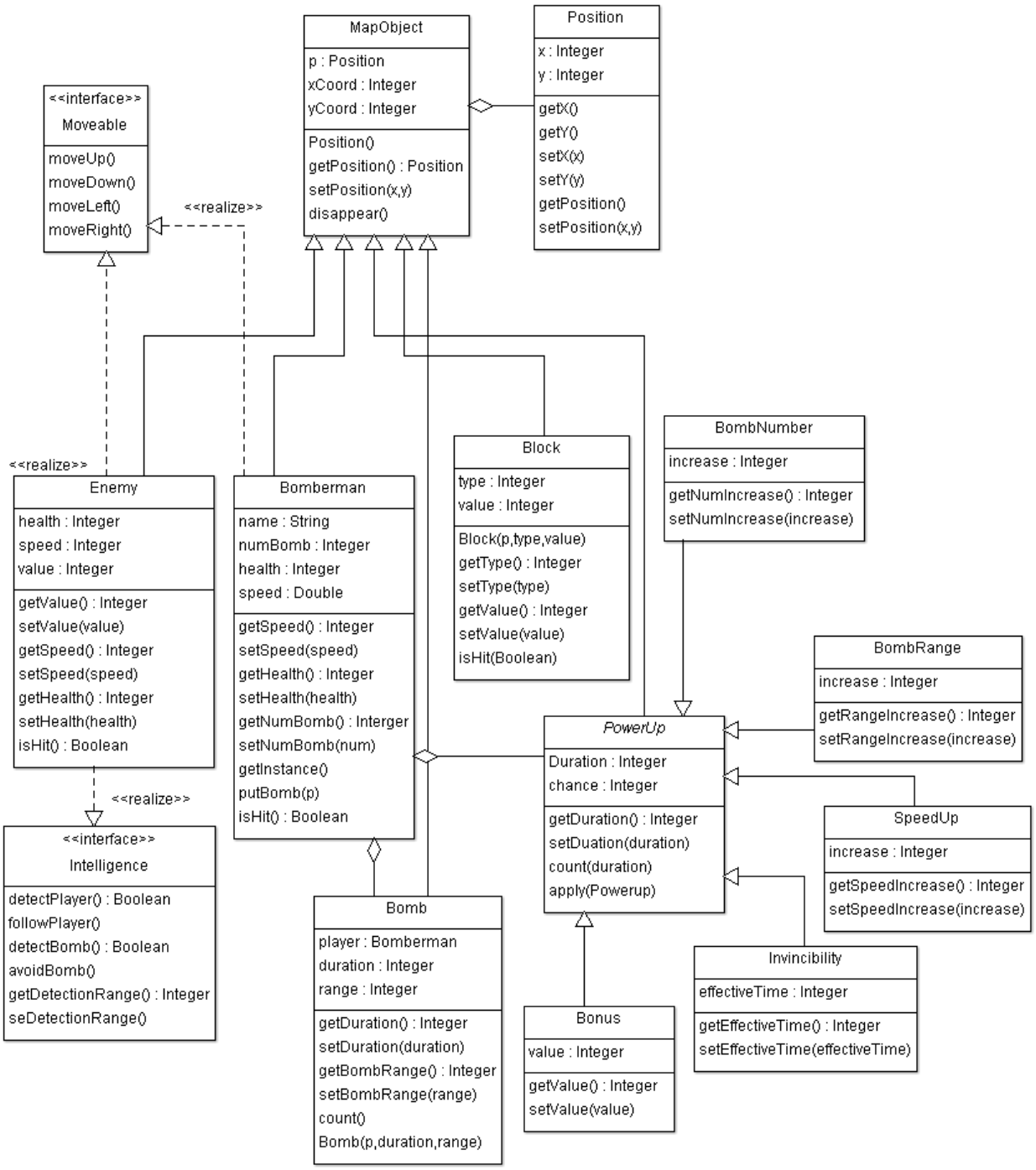
2.2 Detailed Unified Modeling Language (UML) Diagrams

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 Logic

Purpose

The purpose of the Logic class is to manage all the other classes and interfaces to produce a playable game. Methods in this class includes, but is not limited to, starting or resuming a single player or multiplayer game, exit the game.

Interface

This class implements the KeyBoardListener interface and the Runnable java interface.

void main(String args[])
Main method; creates the main menu and acts as appropriate depending on player input.

void startSinglePlayer()
This method starts a new game in single player mode.

void startMultiPlayer()
This method starts a new game in multiplayer mode.

void setPlayerName(String name)
This method allows set player's name to one the user inputs. This method will be implemented in such a way that it will throw an error message if user's input has more than 10 characters.

void pause()
This method pauses the game

void resume()
This method resumes the game from paused state.

void loadGame(int level)
This method will prompt the player to select a saved game file. When selected, the file will be read and the game will start from that level indicated. Method should throws an exception when no game is saved.

void nextLevel()
This method proceeds to the next level.

void displayWinner()
This method displays a text stating the player(s) has won the game.

void displayLoser()
This method displays a text stating "Game Over"

void displayCredits()
This method displays a text stating “Game Over”

void startTimer()
This method starts the internal game timer. The timer is responsible for keeping all game elements up to date. If possible, the timer will use some form of multithreading to execute update tasks concurrently.

void stopTimer()
This method pauses the internal game timer

void keyPressed(KeyEvent e)
This method acts accordingly when user presses a key.

void keyReleased(KeyEvent e)
This method acts accordingly when user releases a key.

void setPosition(double x, y)
This method sets the x and y coordinates of the object.

doube[] getPosition()
This method returns the x and y coordinates of the object.

Runnable collisionCheck()
This method provides a Runnable class that can be used to check for collisions between game objects.

void update()
This method updates objects’ position and state on the map.

void volumeDown(Sound s)
This method decreases the music volume.

void volumeUp(Sound s)
This method increases the music volume.

void mute(Sound s)
This method sets the music volume to none.

void exit()
This method returns player to the main menu.

3.2 Graphics

Purpose

The purpose of Graphics class is to define the hape of all MapObjects, and placing the object at the position intended based on the position of the object. The class implements Runnable.

Interface

void draw()
This method adds the shapes of objects to the map.

void updateGraphics()
This method updates the shapes of objects by placing them at the new position of the object.

void run()
This method is used to call the update() method, since Graphics implements Runnable.

void playVisual(File f)
This method plays corresponding visual effect from action and event library.

void displayScore()
Returns the high score.

3.3 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

void playTheme(File f)
This method plays corresponding level music in a loop

void playSFX(File f)
This method plays corresponding sound effect from action and event library

3.4 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, in-game menu screen and their sub-menus. Menu implements Graphics.

Interface

void reactToButton(KeyEvent e)
This method is responsible for all button presses accordingly.

MainWindow()
This constructor initializes the basic GUI and draws the main menu panel.

PausedWindow()
This constructor draws the in-game menu panel.

ScoreWindow(File f)
This constructor draws the High Scores menu panel, displaying the score information contained in the file.

NewGameWindow()
This constructor draws the new game menu panel.

LoadWindow()
This constructor draws the load game menu panel.

OptionsWindow()
This constructor draws the options menu panel.

LevelWindow()
This constructor draws the select level menu panel.

CreditWindow()
This constructor draws the credit menu panel.

3.5 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.

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

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

3.6 Moveable

Purpose

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

Interface

void moveUp()
This method moves the object upward.

void moveDown()
This method moves the object downward.

void moveLeft()
This method moves the object leftward.

void moveRight()
This method moves the object rightward.

3.7 Intelligence

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(int range)
This method checks whether Bomberman is in the proximity

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

boolean detectBomb(int range)
This method adds the object into the list of blocks
void avoidBomb()
This method adds the object into the list of blocks

3.8 MapObject

Purpose

The purpose of the MapObject 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. MapObject implements Position, Graphics, and Sound.

Interface

Position()
This constructor initializes a new position (x and y coordinates) for a map object.

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

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

void disappear()
This method is used to remove the MapObject from the game, which occurs when a character dies, a bomb explodes, a power up is consumed or a block is destroyed. Necessary animations and sound effects will also occur.

3.9 Bomberman

Purpose

The Bomberman class defines all the attributes and operations appropriate to the Bomberman that are not included in MapObject. Bomberman inherits from the MapObject and implement Moveable. Bomberman also has Bomb and Powerup objects

Interface

Bomberman(int speed, Position p, int health, Bomb b)
This constructor initializes a Bomberman character with properties such as speed, position, health, bomb.

void move()
This method implements from Moveable, which allow the user control the Bomberman.

int getSpeed()
This method returns player's current speed

void setSpeed(int speed)
This method sets player's speed.

int getHealth()
This method returns player's current health

void setHealth(int health)
This method sets player's health.

int getNumBomb()
This method returns the maximum number of bomb the player can place.

void setNumBomb(int num)
This method sets the maximum number of bomb the player can place.

void getInstance()
This method creates first instance of Bomberman that cannot be called again.

void putBomb(Position p)
This method adds bomb to list of bombs and the coordinate.

boolean isHit()
This method checks if Bomberman is hit.

3.10 Enemy

Purpose

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

Interface

Enemy(int speed, Position p, int health, int value)
This constructor initializes an enemy character with properties such as speed, position, health, value.

int getValue()
This method returns enemy's value.

void setValue (int value)
This method sets enemy's value.

void move()
This method implements from Moveable, which allow the user control the Bomberman.

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

void setSpeed(int speed)
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.

3.11 Bomb

Purpose

The Bomb class defines all the attributes and operations appropriate to the bomb that are not included in MapObject. Bomb inherits properties and methods from MapObject class and is a composite of Bomberman.

Interface

Bomb(Position p, int duration, int range)
This constructor initializes a bomb with properties such as position, duration, and range.

int getBombRange()
This method returns the range of the bomb.

void setBombRange(int range)
This method sets the range of the bomb

int getDuration()
This method returns the duration of the bomb.

void setDuration(int duration)
This method sets the duration of the bomb

void count(int duration)
This method count to the duration.

3.12 Powerup

Purpose

The Powerup class defines all the attributes and operations appropriate to power-ups that are not included in MapObject. Powerup class will be implemented as an abstract class since there will be different types of power-ups with different abilities. Powerup inherit from MapObject and is a composite of Bomberman

Interface

int getDuration()
This method returns the duration of the power-up.

void setDuration(int duration)
This method sets the duration of the power-up.

void count(int duration)
This method count to the duration.

int apply(Powerup p)
This method implements the power-up for the Bomberman.

3.12.1 Bonus

Purpose

The Bonus class defines all the attributes and operations appropriate to bonus power-ups that are not included in Powerup.

Interface

Bonus(Position p, int duration, int value, int chance)
This constructor initializes a bonus with a position, duration, value and chance of appearing.

int getValue()
This method returns power-up's value.

void setValue (int value)
This method sets power-up's value.

3.12.2 SpeedUp

Purpose

The SpeedUp class defines all the attributes and operations appropriate to speed increase power-ups that are not included in Powerup.

Interface

SpeedUp(Position p, int duration, double increase, int chance)
This constructor initializes a SpeedUp with a position, duration, increase and chance of appearing.

int getSpeedIncrease()
This method returns power-up's effect on speed.

void setSpeedIncrease(int increase)
This method sets power-up's effect on speed.

3.12.3 RangeUp

Purpose

The RangeUp class defines all the attributes and operations appropriate to bomb range increase power-ups that are not included in Powerup.

Interface

RangeUp(Position p, int duration, int increase, int chance)
This constructor initializes a RangeUp with a position, duration, increase and chance of appearing.

int getRangeIncrease()
This method returns power-up's effect on bomb range.

void setRangeIncrease(int increase)
This method sets power-up's effect on bomb range.

3.12.4 BombNumber

Purpose

The BombNumber class defines all the attributes and operations appropriate to bomb number increase power-ups that are not included in Powerup.

Interface

BombNumber(Position p, int duration, int increase, int chance)
This constructor initializes a bonus with a position, duration, increase and chance of appearing.

int getNumIncrease()
This method returns power-up's effect on the maximum number of bomb the Bomberman can put.

void setNumIncrease(int value)
This method sets power-up's effect on the maximum number of bomb the Bomberman can put.

3.12.5 Invisibility

Purpose

The Invisibility class defines all the attributes and operations appropriate to invisibility power-ups that are not included in Powerup.

Interface

Invisibility(Position p, int duration, int effectiveTime, int chance)
This constructor initializes a bonus with a position, duration, increase and chance of appearing.

int getEffectiveTime()
This method returns power-up's effect on the maximum number of bomb the Bomberman can put.

void setEffectiveTime(int increase)
This method sets power-up's effect on the maximum number of bomb the Bomberman can put.

3.13 Block

Purpose

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

Interface

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

int getType()
This method returns the integer value of the type property (represent whether the block is destructible or indestructible).

void setType(int type)
This method sets the integer value of the type property (represent whether the block is destructible or indestructible).

int getValue()
This method returns enemy's value.

void setValue (int value)
This method sets enemy's value.

boolean isHit()
This method checks if Bomberman is hit.

3.14 GameAgent

Purpose

The purpose of the GameAgent class is to dedicate a separate class just for storing the state of the game in memory, and when needed, saves the game. This is done by holding references to all the objects currently active in the game. Data such as current level and score is also stored.

Interface

GameAgent(int level, int score)
This constructor initializes a game state with the current level and score. The score and level are set as defined in the arguments.

int getScore()
This returns the current score.

void setScore(int addScore)
This sets the current score.

void addBlock(Block b)
Will add the object b into the list of blocks.

void removeBlock(Block b)
Will remove the object b from the list of blocks.

List<Block> getBlocks()
Returns a list of Blocks.

void addPowerup(Powerup p)
Will add the object p to the list of bonus drops.

void removePowerup(Powerup p)
Will remove the object p from the list of bonus drops.

List<Powerup> getPowerups()
Returns a list of Power-ups.

void addEnemy(Enemy enemy)
Will make the Enemy variable reference object as.

List<Enemy> getEnemies()
Returns the Enemy object in game agent.

void removeEnemy()
Will remove the Enemy from the game agent.

int getLevel()
Returns the level.

boolean isLevelComplete()
Return true if level is completed.

void addToScore(int Score)
This method takes the value of object destroyed and adds to Bomberman's score.

int getHighScore()
Returns the high score.

void saveGame(GameAgent s, File f)
This method saves the current state of the game to the specified file.

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 GameAgent	Graphics Menu GUI	Position Moveable Bomberman Enemy Bomb Powerup Block	Intelligence	GameAgent	Sound

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	keyPressed() keyReleased()		moveUp() moveDown() moveLeft() moveRight()			
3.1. CT.1.2						
3.1. CT.1.3						
3.1. CT.1.4						
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. MN.5.1		ScoreWindow()				
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		OptionsWindow				
3.1. MN.8.2	update()					
3.1. MN.8.3		draw()				
3.1. MN.9.1		CreditWindow()				
3.1. MN.9.2	displayCredits					
3.1. MN.10.1			saveGame()			
3.1. MN.10.2			saveGame()			
3.1. MN.11.1		OptionsWindow()				
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()	NewGameWindow()				
3.1. MN.13.2	startGame()	LevelWindow()				
3.1. MN.13.3	startGame()	LoadWindow()				

3.1. MN.15.1	resume()	playVisual()				
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. AI.2.2			move()			
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. BL.3.3			setPosition()			
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. GP.1.3	setPlayerName()					
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.2.4		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() volumeDown() mute()					
3.1. AV.1.2						
3.1. AV.2.1						playTheme()
3.1. AV.2.2						playTheme()
3.1. AV.2.3						playTheme()
3.1. AV.2.4						playTheme()
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