Phase 2- System Architecture Document

Brett Leighton, Martin Zhang, Leotard Niyonkuru, Mete Kemertas, Marc-Andre Cataford

March 3rd 2014

Table of Contents

1 System Overview

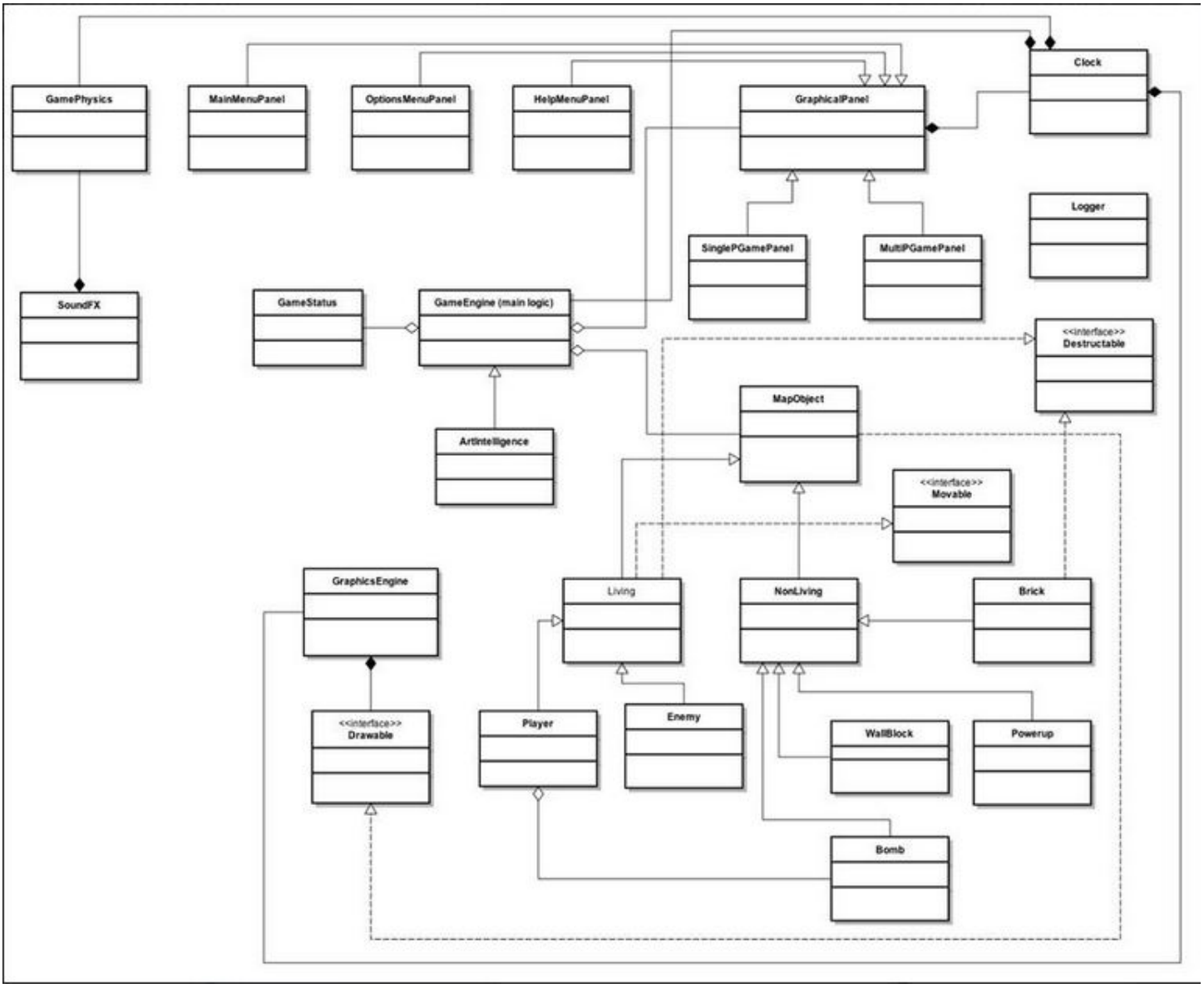
1.1 Purpose

The purpose of this document is to present a detailed description of all classes, objects, interfaces, and associated methods used in the system. The document also describes the relationship between these components in terms of inheritance, composition and aggregation, making use of UML diagrams. This document is intended for current developers working on the project, future developers wishing to make improvements to the project, and finally the client. It is to be used as a point of reference when writing code for the game itself. It should be noted that this document is subject to change, and may be modified by any intended party at any time if need be.

1.2 Scope

The piece of software to be written will be a game called Project Bomberman, modeled after the strategic, maze-based video game Bomberman, originally developed by Hudson Soft. The game can be played by one or two people, and will feature a 2D graphical user interface in which the player will be able to control a character and destroy enemy units by detonating bombs. As a game, the software will have an entertainment value to those who play it, with the development teams goal being to maximize this entertainment value while also meeting requirements defined in the Software Requirements Specification document. This design document presents an overview of the games intended architecture, without delving too deep into details regarding implementation.

2 Views


GamePhysics
MainMenuPanel
OptionsMenuPanel
HelpMenuPanel
GraphicalPanel
Clock
SoundFX
GameStatus
GameEngine (main logic)
SinglePGamePanel
MultiPGamePanel
Logger
<<interface>>
Destructable
MapObject
ArtIntelligence
<<interface>>
Movable
GraphicsEngine
Living
NonLiving
Brick
<<interface>>
Drawable
Player
Enemy
WallBlock
Powerup
Bomb

3 Software subsystems & modules

3.1 Classes

3.1.1 GameEngine (main logic) *3.1.1.1 Purpose* The GameEngine class contains the main logic of the system. It calls upon GameStatus to add all map objects to its lists of game objects as well as set up the GraphicalPanel object to display gameplay on the players screen. *3.1.1.2 Interface* void main(String[] args)
Main method. Creates a GraphicalPanel and presents the MainMenuPanel, then waits for user input.

GameEngine()
Constructor for the GameEngine method.

void newSingleP()
Starts a new single player game by setting up an appropriate GameStatus and GraphicalPanel.

void newMultiP()
Starts a new multiplayer game by setting up an appropriate GameStatus and GraphicalPanel adapted to a 2P multiplayer game.

void loadGame()
Prompts user to select a save file, and then loads GameStatus information from that file.

void showLeaderBoard()
Tells the GraphicsEngine to display the Highscore panel.

void showInstructions()
Tells the GraphicsEngine to display the Instructions panel.

void showOptions()
Tells the GraphicsEngine to display the Options panel.

boolean checkPaused()
Checks whether the game is paused, and outputs a boolean.

void displayWinner()
Triggers the You won game event.

void displayLoss()
Triggers the Gameover game event.

void keyPressed(KeyEvent e)
Detects a keyboard event (user has pressed a key)

void keyRelease(KeyEvent e)
Detects a keyboard event (user has released a pressed key)

void actionPerformed(ActionEvent e)
Detects a general event triggered by the user.

3.1.1.3 GraphicalPanel
3.1.2.1.1 Purpose
The GraphicalPanel method is a Swing component that is the superclass to any GUI element later used by the game software. It represents what the user will see on-screen.
3.1.2.1.2 Interface
GraphicalPanel()
Constructor for the GraphicalPanel class.
void redraw()
Generates a new panel and replaces the current one with it. Used for refreshing panels back to their initial state.
3.1.1.4 MainMenuPanel
3.1.1.4.1 Purpose

Contains the basic panel for the main menu.
3.1.1.4.2 Interface
MainMenuPanel()
The constructor draws the main menu panel and adds Swing buttons to it.
3.1.1.5 OptionsMenuPanel
3.1.1.5.1 Purpose
Contains the basic panel for the options menu with the appropriate Swing buttons..
3.1.1.5.2 Interface
OptionsMenuPanel()
The constructor draws the options menu panel and adds Swing buttons to it.
3.1.1.6 HelpMenuPanel
3.1.1.6.1 Purpose
Contains the basic panel for the help menu.
3.1.1.6.2 Interface
HelpMenuPanel()
The constructor draws the help menu panel and adds Swing buttons to it.
3.1.1.7 MultiPGamePanel
3.1.1.7.1 Purpose
Contains the basic panel for a multiplayer game.
3.1.1.7.2 Interface
MultiPGamePanel()
The constructor draws the multiplayer game panel with sections for each players score, the game timer and the game field.
3.1.1.8 SinglePGamePanel
3.1.1.8.1 Purpose
Contains the basic panel for a single player game.
3.1.1.8.2 Interface
SinglePGamePanel()
The constructor draws the single player game panel with separate sections for the players current score, for the game timer and for the game field.
3.1.1.9 GameStatus
3.1.1.9.1 Purpose
The GameStatus class is responsible for everything related to the current status of the game, containing references to all map objects currently active, as well as the capability to save the game.
3.1.1.9.2 Interface
GameStatus()
The constructor for the GameStatus class, initializes empty lists for each MapObject tye that will be placed on the game map by GameStatus methods.

void addPlayer(Player p)
Adds a Player to the list of Players and to the game field.

void removePlayer(Player p)
Removes a Player from the list of Players. This is used when a player dies, with the added convenience of effectively clearing any powerups that a player has.

void addWall(Wall w)
Adds a Wall to the list of walls.

void removeWall(Wall w)
Removes a Wall from the list of walls.

void addBrick(Brick b)
Adds a Brick to the list of bricks.

void removeBrick(Brick b)
Removes a Brick from the list of bricks.

void addEnemy(Enemy e)
Adds an Enemy to the list of enemies.

void removeEnemy(Enemy e)
Removes an Enemy from the list of enemies.

void addPowerUp(Powerup p)
Adds a PowerUp to the list of powerups.

void removePowerUp(Powerup p)
Removes a PowerUp from the list of powerups.

void addExplosion(MapObject exp)
Adds an explosion to the list of explosions.

void removeExplosion(MapObject exp)
Removes an explosion from the list of explosions.

void saveGame(File destination)
Saves the game to a file with specific destination.
3.1.2 MapObject *3.1.2.1 Purpose*
The purpose of the class MapObject is to provide a general representation of the game objects that will appear on the map (such as Enemies, Bricks, Wall Blocks). It includes common functionalities of all objects.
3.1.2.3 Living
3.1.2.3.1 Purpose
The purpose of the Living class is to represent living units in the game: those that implement Movable. It is a subclass of the MapObject class.
3.1.2.4 Player
3.1.2.4.1 Purpose
The purpose of the Player class is to represent the players character in the game. It serves as an anchor point for the graphics/sound data supplied by the SoundFX and GraphicsEngine components.
3.1.2.5 Enemy
3.1.2.5.1 Purpose
The purpose of the Enemy class is to represent enemy unit in the game. It is a subclass of the Living class.
3.1.2.6 NonLiving
3.1.2.6.1 Purpose
The purpose of the NonLiving class is to represent non-living units in the game. It is a subclass of the MapObject class.
3.1.2.7 Bomb
3.1.2.7.1 Purpose
The purpose of the Bomb class is to represent bombs in the game. Inside the bomb class is an attribute called timer, which will detonate the bomb after a certain amount of loops. It is a subclass of the NonLiving class. The Bomb class is also an aggregation of the Player class.
3.1.2.7.2 Interface

void detonate()
This method detonates the bomb, and runs the four clearObjects methods.
void clearObjectsUp()
This method clears all MapObjects above the bomb upon detonation, with distance according to the Powerups the player has.
void clearObjectsDown()
This method clears all MapObjects below the bomb upon detonation, with distance according to the Powerups the player has.
void clearObjectsLeft()
This method clears all MapObjects to the left of the bomb upon detonation, with distance according to the Powerups the player has.
void clearObjectsRight()
This method clears all MapObjects to the right of the bomb upon detonation, with distance according to the Powerups the player has.
3.1.2.8 Wall
3.1.2.8.1 Purpose
The Wall class represents the immovable, indestructible lattice that creates boundaries of the map. It is a subclass of the NonLiving class.
3.1.2.9 Brick

3.1.2.9.1 Purpose
The Brick class represents the destructible obstacles that are randomly generated at the beginning of each game. It is a subclass of the NonLiving class.
3.1.2.9.2 Interface
Brick(int Xpos, int Ypos)
This is the constructor for the Brick class. It requires the x and y coordinates as inputs.
3.1.2.10 Powerup
3.1.2.10.1 Purpose
The Powerup class represents the powerups randomly generated when a Brick or Enemy is destroyed. It is a subclass of the NonLiving class.
3.1.2.10.2 Interface
Powerup()
The constructor for the PowerUp class. Creates a Powerup of a random type, sharing the X and Y coordinates of the brick or enemy destroyed.
void applyPower(Player p)
Applies the powerup to the player.
3.1.3 GamePhysics
3.1.3.1 Purpose
The GamePhysics class monitors collision detection between game objects. It keeps track of collision conditions and uses the GameStatus to access the positions of active MapObjects.
3.1.3.2 Interface
GamePhysics(GameStatus gs)
The constructor for the GamePhysics class. Takes GameStatus as input.
checkCollisions()
This method checks for collisions between two MapObjects.
3.1.4 Clock
3.1.4.1 Purpose
The Clock class keeps track of in-game time and acts as a synchronization device for the softwares threads. It supplies timing information to the game softwares modules to preserve gameplay coherence.
3.1.4.2 Interface
startTime()
Starts the clock, which will synchronize the GamePhysics and the GraphicsEngine

stopTime()
Stops the clock when the game is over.

3.1.5 Logger
3.1.5.1 Purpose
The Logger class collects data about the games runtime and records it to a runtime log. Implementation is built in to Java 7.
3.1.6 GraphicsEngine
3.1.6.1 Purpose
The GraphicsEngine class generates on-screen graphics based on GameStatus data.
3.1.6.2 Interface
GraphicsEngine(GameState state)
The constructor for the graphics engine. Takes GameState as input. Links the graphics engine to the Clock class for thread synchronization.

void updateGraphics()
Graphically updates the position of all objects in GameState based on their X and Y coordinates.
3.1.7 ArtIntelligence
3.1.7.1 Purpose
The purpose of the ArtIntelligence class is to build the strategy employed by the enemy units. It will handle the decisions made by the enemy such as whether to go up, down, left or right.
3.1.7.2 Interface
findPath()
Checks the units surrounding and finds an available path for further movement depending on GamePhysics collision data.
targetPlayer()
Checks Player proximity and if available, initiates an attack.

3.1.8 SoundFX

3.1.8.1 Purpose

The SoundFX class will contain all methods relating to playing sound effects.

3.1.8.2 Interface

SoundFX(String inputFile)

The constructor for the SoundFX class.

playSound()

Plays the sound once.

stopSound()

Stops the current sound from playing.

playLoop()

Plays the current sound in a loop.

3.1.9 Movable ¡¡Interface¿¿

3.1.9.1 Purpose

The movable interface provides prototypes for each Movable implementation, i.e. extended classes of the Living superclass.

3.1.9.2 Interface

MoveRight()

Moves the Movable object by one cell to the right (incrementing the x-position attribute by 1)

MoveLeft()

Moves the Movable object by one cell to the left (decrementing the x-position attribute by 1)

MoveUp()

Moves up the Movable object by one cell (incrementing the y-position attribute by 1)

MoveDown()

Moves down the Movable object by one cell (decrementing the y-position attribute by 1)

3.1.10 Destructable ¡¡Interface¿¿

3.1.10.1 Purpose

The movable interface provides prototypes for objects that can be destroyed and erased from the map, such as a bomb or a brick, but not a WallBlock.

3.1.10.2 Interface

isDestroyed()

Checks if an object was destroyed following an explosion. Returns boolean.

3.1.11 Drawable ¡¡Interface¿¿

3.1.11.1 Purpose

The drawable interface provides prototypes for objects that are drawn on the map. More precisely, instances of MapObject fall into this category. Thus, this Interface is implemented by MapObject class.

3.2 Categories

AI	**Physics**	**Audio**	**Graphics**
ArtIntelligence	GamePhysics	SoundFX	GraphicsEngine
GUI	**Game Objects**	**Logic**	
GraphicalPanel	MapObject	GameEngine	
MainManuPanel	Player	GameStatus	
LeaderBoardPanel	Enemy		
SinglePGamePanel	Powerup		
MultiPGamePanel	Bomb		
OptionsMenuPanel	Wallblock		
LeaderboardPanel	Brick		

4 Analysis

4.1 Traceability matrix

Requirement	GUI	GameObject	Logic	Audio
A1.				
A2.	MainMenuPanel			
A.3		Player		
A.4		Enemy		
A.5		PowerUp		
A.6			level()	
A.7	MapObject			
A.8			addPlayer()	
A.9	OptionsMenuPanel			
A.10	GraphicalPanel			
A.11			loadGame()	
A.12			saveGame()	
A.13			showLeadBoard()	
A.14			MultiPGamePanel	
A.15	GraphicalPanel			
A.16			actionPerformed()	
A.17			checkPaused()	
B.1	HelpMenuPanel			
B.4			Clock	
B.2			showInstructions()	
B.5			ArtIntelligence	
B.6			ArtIntelligence	
B.8			saveGame()	
B.9			Clock	
B.10				
B.11			GameEngine	
B.12				
B.13	GameEngine			
B.14			Logger	
C.1				
C.2	GraphicalPanel			
C.4				SoundFX

4.2 Quality requirements

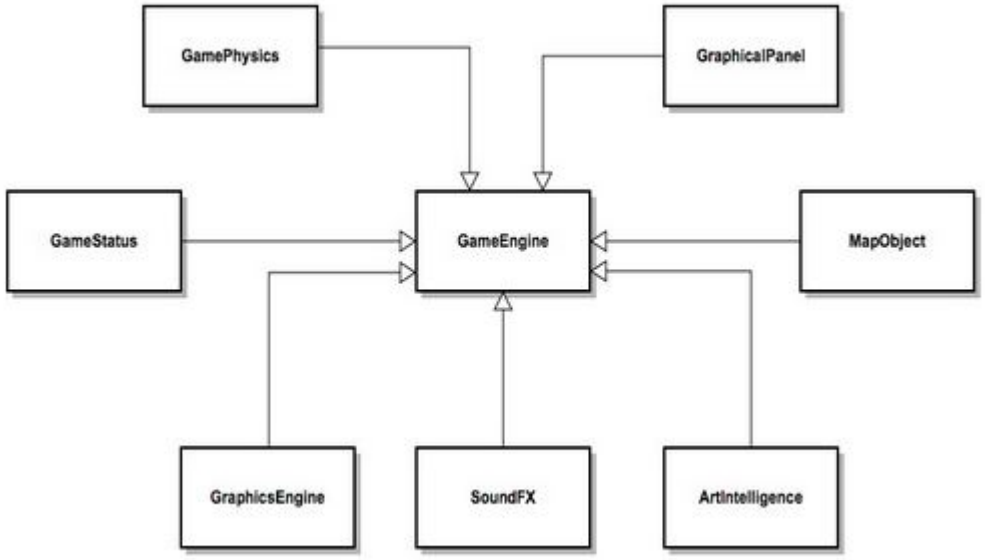
Quality Requirements from SRS:

B1. The game shall have an instructions menu accessible through a button on the main screen.
B2. The instructions shall be of a resolution that is at least equivalent to a 10-point typeface.
These requirements are covered through use of MenuPanel classes.
B3. The game shall not use more than 50% of CPU resources for any given time.
B4. There should be a delay of no longer than .05 seconds between user command and execution of the command.
B7. The game shall run smoothly from startup 99% of the time
B9. After the user finishes a level, the next level should load in less than 5 seconds.
B10. The game shall run at 60 frames per second on a basic 32 mb graphics chip.
B11. The game can initialize within 2 seconds of launching the application.
These requirements are addressed with the Physics and Graphics loops being in two separate threads, minimizing CPU resources. The graphics in the game will be of a low file size and in JPG format, to minimize the size of resources necessary for initial load.
B5. 75 out of 100 average players can fully complete the game
B6. 75 out of 100 average players can finish each level within 2 minutes
The AI and difficulty is easily modified because of the separation of the AI class. In the testing phase, this will make this requirement easily attainable through modification of this class and how it responds to difficulty changes.
B8. The game shall save the level completed, the difficulty selected, the score at the start of the last level started, and the

powerups the player has accumulated. This is solved through the implementation of saveGame() in the GameStatus class. All data needed in the save file is in one place.
B14. The user can enter a username to identify his save file and highscores to the game. This optional quality requirement can be easily implemented in the future through additional user inputs in the saveGame() method in GameStatus.

4.3 Design rationale

The Design Architecture shown in Figure 2 was chosen as a result of a decision process among developers and the consulting teaching assistant.



The structure is broken down into 8 components, one of which is the brain of the game, managing the other 7 and running the main method. The advantage of this model is that none of the 7 classes are related to each other in terms of inheritance, composition or aggregation, which makes it much easier and more simplistic to develop in parallel. This helps coordinate the time budget and team coordination.
A good example of the benefits of this diversification is the drawing of an enemy on the map. The Enemy object will not be required to calculate its own position, nor be affiliated with the drawing. Instead, the calculation of its speed and coordinates is be completed within GamePhysics before being fed to GameEngine, where GraphicsEngine will pull the data and draw the picture. This will happen synchronously thanks to the Clock object shown in the UML diagram in Section 2. The Enemy class merely has to store the information. Another advantage of having separate classes for each of these components is the insurance that if a new version of a component needs to be made later in the software cycle, little to no modification of other components will be required.
As a generalization of game operation, on command of the GameEngine component, the GraphicsEngine, SoundFX and GamePhysics components generate gameplay events that correspond to the current GameStatus component, which represents the tracking of MapObjects and how these change based on user input. A MapObject class is created as a general class to represent each object on the map and their common attributes. This will simplify methods that do not require any specific information contained in subclasses.
The programs visual interactions are provided through GraphicsEngine and GraphicalPanel classes. The GraphicsEngine class is created to display the in-game events and interactions, while the GraphicalPanel forms the outlook for menus of various types (Options Menu, Escape Menu, etc).
Finally, ArtIntelligence class determines the strategy enemy units will employ based on probability calculations and random evaluations, likely depending on the difficulty level of the game. This will ensure the Enemy class remains simple and to the point.

5 Workload distribution

To split the workload evenly and effectively, the project needed to be split into five segments, outlined as follows:
GameStatus: Brett
The GameStatus class.
GameEngine: Mete
The GameEngine class.
MapObject: Martin
The MapObject class, including all subclasses.
GUI: Marc
The entire Graphical User Interface (GUI).
Interfaces/Debugging: Leotard
Interfaces, ensuring coherence between sections, and final debugging.
All members will be responsible for testing their individual sections as well as the final project.