Designing and Implementation of a Dig Dug-like Game in C++

Deepam Yasvant Ambelal #719369 **Nabeel Seedat #719484**

School of Electrical & Information Engineering, University of the Witwatersrand, Private Bag 3, 2050, Johannesburg, South Africa

ELEN3009 – Software Development 2

Abstract: This report presents the design and implementation of a Dig Dug-like game called Monster Hunter: Dig Dug Edition. The game is developed using C++ with the SFML graphics library. The game provides three minor features: extra lives for the player, red monsters having a disembodied/ "ghost" form when chasing out of a tunnel, good graphics and one major feature of the lightning harpoon which will inflate/explode the enemy after a few seconds in this final submission. The domain is modelled in a 2D plane with four major domain entities: Sorceress, Genie, Lightning and Tunnel. The design made use of three independent layers (data, logic and presentation) with a game class serving as an intermediary class. A critical evaluation of the dynamic behaviours of collisions, the monster chasing algorithm and user input handling is presented. The game is a success in terms of the achieved functionality/game play. Furthermore, the usage of good programming practices and object-oriented techniques such as functionality decoupling, good usage of modern, idiomatic C++ and separation of concerns through good layering techniques results in an efficient game. There are shortfalls in terms of game speed issues, the chase algorithm, the monolithic game class, poor data encapsulation, as well as, partial violation of the DRY principle. Unit testing is successfully implemented on all functionality aside from graphics and user input which can be rectified through re-factoring of the code or the usage a mocking framework. It is proposed that the issues in design and functionality must be addressed, whilst also adding new features to ensure long term viability and an improved game.

Keywords: C++, Domain, DRY principle, Monolithic, SFML

1. Introduction

The aim of this project is to create a game that can emulate the Namco's classic Dig Dug game using C++ with an SFML library. The key concept is to construct the game using object-oriented programming.

This report serves as a high-level documentation and evaluation of the game produced. It provides insight into the design decisions, game modelling used and structure of the design. Issues around the game layering, Object Oriented (OO) design, as well as, domain object modelling decisions are presented. A critical evaluation of the design is then presented to assist developers to improve upon the current structure and design.

2. Project Specifications

The task is to produce a Dig Dug-like game that mimics the mechanics of the original game. However, the design can follow any game theme and storylines.

2.1. Requirements

The requirements for the project are:

- To implement using object oriented design style.
- Create an executable file that can run on a Windows system with a splash screen and game instructions.
- Create a comprehensive unit test executable file using Google Test framework.
- Provide release notes on each GitHub submission.
- Produce a technical reference manual for the final submission.

2.2. Constraints

The constraints of the game's design are:

- ANSI/ISO C++ language is to be used.
- Restricted SFML and Google Test libraries provided. by the ELEN3009 course are to be used.
- OpenGL libraries cannot be used.
- The screen resolution of the game cannot exceed 1920×1080 pixels.

2.3. Assumptions

A screen size of 1280×800 is chosen as the game's window would fit on most laptop screens without being cut-off.

Only 64-bit executable copies of the game is developed as users would most likely use a 64-bit operating system to execute the game. There is no possibility of making a separate 32-bit copy or backward compatible executable as it is not viable.

2.4. Success Criteria

The project's success criteria is evaluated based on the game's functionality which is divided into three categories:

- Basic functionality
- Minor feature enhancements
- Major feature enhancements

Basic functionality is the minimum criteria that the game needs to have to for an acceptable submission. Major and minor feature enhancements are criteria the game needs to meet to achieve a higher than acceptable submission

The lists below provide details on the specifications the implemented game will most likely achieve, whilst excluding the criteria not achievable.

Basic Functionality proposed for implementation:

- Dig Dug, multiple red monsters, and the earth game objects exist.
- Dig Dug moves correctly through the earth based on player input and digs tunnels wherever he moves.

- The monsters move autonomously and chase the player. At the start of the game at least one monster is present in the same tunnel as Dig Dug.
- The game ends if Dig Dug collides with a monster or Dig Dug shoots all the monsters.

Minor features proposed for implementation:

- Good graphics.
- The red monsters can also change into their disembodied/" ghost" forms and drift from tunnel to tunnel in order to chase Dig Dug.
- Dig Dug has more than one life and his remaining lives are depicted on the screen.

Major features proposed for implementation:

- The behaviour of Dig Dug's harpoon mirrors the original game more closely. Monsters are not destroyed instantly instead, it takes a few seconds for them to inflate and pop, and Dig Dug is incapable of moving while this is happening (and he is vulnerable to attack by other monsters). Dig Dug may prematurely stop inflating a monster, that is, stop before it pops. In these cases, the monster deflates and continues to move around as before.

With the evaluation of the game submitted for Submission 2 rated as "Basic functionality achieved" and the criteria aimed to meet in the above list for this Final Submission, the game is aimed to achieve a "Good" rating.

Additional success criteria are related to the design. The solution required the use of Objected Oriented principles, have good layering (presentation, data and logic) and separation of concerns. It must make good use of modern, idiomatic C++14. The Don't Repeat Yourself (DRY) principle must be adhered to and aspects associated with poor software design avoided [1].

3. GAME'S THEME

The graphic art used in the original Dig Dug game can be considered low resolution and very pixelated art compared to graphic art developed in current generation games. To make the game have better aesthetics, sprites that animate are sourced from Nintendo DS sprite sheets and unanimated sprites are fan-art.

The Dig Dug character is chosen first. It is decided that Dig Dug is a red hair girl with a black cloak. It is assumed that she is a Sorceress and thus, the game is fantasy themed. Effectively, Sorceress plays a role as a monster-hunter in the game. The red monsters in the Dig Dug game are chosen as Jafaar genie from Aladdin. The harpoon weapon is chosen to be lightning magic as the art used the splash screen depicted Sorceress to have lightning magic. Since lightning would not result logically result in a monster inflating, creating an explosion that gradually envelops the monster is used. When the monster is completely enveloped, it will be killed like a fully inflated monster.

The game is named "Monster Hunter: Dig Dug Edition" based on a popular Nintendo DS game "Monster Hunter" The sprites were edited using a combination of Microsoft PowerPoint's image editing tools and sprite editing Piskel applet.

4. DOMAIN MODELLING

All objects displayed in the design are modelled as domain entities and co-ordinates exist in a two-dimensional plane, each given an initialized (x, y) co-ordinate. The following game objects that exist in the two dimensional plane are: Sorceress, Genie, Lightning and Tunnel. The game entities are stored in vectors of objects (the design rationale is discussed in section 7.1). These game entities are able to collide with each other, as well as, being drawable upon rendering of the game onto the screen.

Sorceress can move in an up, down, left or right direction, depending on user input. The directional keys are used to move Sorceress on the screen. The Genie can also move up, down, left and right but only in tunnels. When the Genie decides to "ghost", it can move in any direction and through the walls of the tunnels. Sorceress has lightning magic that can kill monsters. This attack occurs in the direction of Sorceress and is triggered by pressing the spacebar key. The screen is a closed area for Sorceress and the Genies as they cannot leave it. However, the lightning magic attack can partially exist outside the bounds if the user decides to shoot outside the boundary.

5. DESIGN STRUCTURE

The game is separated into three separate layers where each class belongs to an individual layer [2] [3]. These layers are:

- Presentation
- Logic
- Data

It is important that layers are decoupled from each other [4], such that interactions are tightly controlled through an intermediary. This allows for each class and layer to have designated responsibilities, with information not freely known between classes [2] [3]. Layering is based on these principles and is discussed below. The complete class interaction structure of the game is presented as a UML diagram in Figure 1 of the Appendix. A diagram showing the class separation of layers is shown in Figure 2

5.1. Presentation Layers

This layer is responsible for rendering the game and processing game state updates to the screen using the SFML Library based on interactions between the game environment and the characters.

5.1.1. Graphics Class

The `graphics` class is responsible for all interactions with the SFML Library, including but not limited to the graphic sprites displayed, movement functions of graphics on the screen, setting of sprite positions and text usage.

The aim of this class is to separate other game objects and their behaviours from their implementation using the graphics library (i.e. SFML) [2]. This is done based on the objective of separation of concerns, by incorporating a class whose responsibility deals with SFML alone. A key benefit of this design decision is that should there be changes to SFML that only `graphics` is impacted. It also provides the ability to change the graphics library from SFML to another graphics library without impacting the entire codebase. Only the `graphics` class would be impacted which is due to the reduced coupling between classes. This allows for superior code re-usability.

The respective logic objects which utilize the `graphics` class have a private data member which store the graphic associated with the object. The implementation is done through the use of `std::shared_ptr` to `graphics` (See Figure 3 in the Appendix). The implementation of smart pointers is beneficial in terms of memory management and prevents memory leakage due to automatic pointer destruction. The implementation of the `graphic` class with functions primarily virtual in nature is due to the significant inheritance of the `graphics` class throughout the logic layer. While polymorphism is not utilized significantly in the current implementation, it is a design decision for future code usage, should it require polymorphic inheritance to be implemented when using a larger number of similar objects. The virtual functions would then allow for base level function implementation.

5.2. *Logic Layer*

The logic layer consists of object classes and the manipulation of the following: Sorceress, Tunnel, Lightning and Genie. Other logic classes presented facilitate the interactions between the four main domain entities and assist in updating the game state based on these interactions.

5.2.1. Sorceress

The `sorceress` class is responsible for the initialization and movement of the Sorceress sprite and a bounding rectangle. The bounding rectangle is an invisible rectangle which is drawn around the sprite to be used for the collision testing discussed in 5.2.5. The sprite and rectangle will move and animate based on the direction the user inputs. The handling of the input is dealt with by the `controls` class in Section 5.2.10.

5.2.2. Tunnel

The `tunnel` class is responsible for generating the tunnel images on the screen. The initialization function reads in co-ordinates from “tunnel-coords.txt” to produce a tunnel map when the game starts. This class has the responsibility of creating tunnels behind the Sorceress sprite where she moves. The presentation layer then polls these created tunnels and draws them.

5.2.3. Lightning

The `lightning` class is responsible for generating the lightning sprites which is the magic weapon Sorceress uses to kill the monsters. The Lightning is set to move in the direction Sorceress faces. An invisible rectangle is drawn behind the image so that the collision between the Lightning and the monster can be polled. The lightning image is set to load gradually so that it gives a propagating effect. The invisible rectangle’s size loads in the same manner. See section 6.3 for Collision testing.

5.2.4. Genie

The `genie` class is responsible for the loading and the movement of the Genie sprite. The movement of the Genie sprite is split into two phases. The first is where the Genie patrol the tunnels it is currently in. The patrol mode exists when Sorceress is outside a certain radius. Once the radius condition is met, it triggers the second mode, where the Genies chase after Sorceress.

The algorithm of monsters following Sorceress is based on the shortest path algorithm, which will be discussed in Section 6.4. The ghosting mechanism is controlled in the `monsterMove` class which controls when the Genie can “ghost” and pass through the tunnel walls.

5.2.5. MonsterMove

The `monsterMove` class is responsible for controlling whether the monsters have the shortest path to Sorceress through the tunnel or not. If the path is through the Earth, then the monster’s Boolean logic is set to true, which means Genie will change sprite changes to a ghost form and vice versa if the path is through the tunnel. Effectively, the purpose of the class is to control the behavioural logic of the monster (i.e. Genie).

5.2.6. Collision

The `collision` class has the responsibility of checking whether a collision has occurred between Lightning and Genie. Once a collision has been detected, the state of the respective object is updated. The collision algorithm is discussed in Section 6.3.

5.2.7. Explosion

The `explosion` class is responsible for cycling through the explosion sprites to give the explosion an expanding effect while it is enveloping the monsters. This represents the inflating monster in Dig Dug. A timer function is utilized for this purpose, as the amount of time after Genie is struck by lightning will represent a different cycle in the sprite.

5.2.8. KillGenie

The `killGenie` class is responsible for polling the state of Genie. Based on the result of the poll, it has the responsibility for removing the killed Genie from the Genie vector. This is done so that when all the Genies are killed, the “You Win” screen is triggered.

5.2.9. KillSrs

The `killSrs` class is responsible for deducting life points from Sorceress’s health based on same collision mechanism as will be expounded in section 6.3. When the health reaches zero it provides a trigger for the screens to change to “You Lose”

5.2.10. Controls

The `controls` class manages all user input. It does not store any data, but rather it serves as a link between the logic of game objects and input behaviour.

5.3. *Data Layer*

This layer loads in information from outside the executable file. This is simply in the form of initialization co-ordinates at the start of the game, textures utilized to render the SFML sprites, as well as, fonts for text for in-game text display.

It is decided to load all data resources at the start of the game, as loading resources from memory is an expensive operation [5]. Therefore, instead of constantly loading resources when needed, which would negatively impact game performance by introducing lag at import time, a design decision is taken to load all resources at start-up/run time to mitigate this potential issue. This assists in separation of concerns by creating a data layer of resources which the logic layer and presentation layer is able to access.

5.3.1. *Background*

This `background` class has the purpose of loading full screen graphics [3] (1280×800 pixels) which are the following:

- Start game splash screen
- Instructional splash screen
- Game background
- “You Win” screen
- “You Lose” screen

All the above graphics excluding the game background are displayed until the user inputs the correct key to progress the game to the next screen or to terminate the game.

5.3.2. *tunnel-coords.txt*

This text file contains co-ordinates of initial tunnel positions where the `tunnel` class places tunnel sprites to create a tunnel chain.

5.3.3. *sansation.ttf*

This is a font style used to display text on the screen.

5.3.4. *Scoring*

The `scoring` class is responsible for storing the data about the in-game Sorceress’s number of lives remaining. It also initializes the font and sets the correct string variable of lives remaining which is then drawn by `game`.

5.4. *Layer Interaction*

5.4.1. *Game*

The `game` class serves the role of the “Game-engine”. The `game` class firstly renders the game window and objects themselves onto the screen based on initial conditions. Thereafter, it checks user input, as well as, logic level updates, which then changes the game state and this is re-drawn on the screen. The `game` class has a highly defined responsibility of drawing and updating game objects, hence it inherits from each game object in order to monitor the state changes in the logic layer and to correspondingly render the correct game screen. The `game` class has knowledge of all the states of all the objects, in order to process the requests and updates. This is compared to the game objects themselves that only have knowledge of themselves, as well as the `graphics` class through a `std::shared_ptr`. It can be said that the `game` class does have the responsibility to serve as the intermediary between the presentation and logic layers. Lastly, it is responsible for ending the game by drawing the correct end game background: player wins (all monsters killed) or player loses (Sorceress loses all her lives)

6. LAYER INTERACTION

Various key features of the game need to be expounded upon in terms of achieving the specific game functionality. This involves the polling of user input, the updating of game objects with regard to the gameplay of chasing and collision detection. Lastly, how these changing game states are dealt with using the `game` class which bridges the presentation and logic layers.

6.1. *Game Engine*

The `game` class is the core of the game as it is responsible for linking the three data layers together. The `game` class serves as the game engine, whose responsibilities are defined as rendering the game, checking for game state updates and appropriately enacting these changes upon the continuous rendering of the game [2].

As a consequence of these responsibilities, the `game` class needs to interact with both the presentation and logic layer in order to render the game, whilst also checking for updates in the logic layer. As was discussed, the `game` class, thus has knowledge of all game objects in order to process these changes.

The game engine polls for user input in order to end the game mid-game, as well as, when a player wins or loses in order to exist. As a consequence, the `game` class has interactions with the `controls` class.

The `game` class then polls changes in the position of Sorceress and thus, polls the `sorceress::update()` function.

The `game` class then checks for two types of collisions, discussed in Section 6.3. Firstly, it interacts with the `collision` class for updating collisions between Lightning and Genie. The update effects are subsequently detected when polling the game object seen by `game`.

As a result, the `killGenie::kill_genie` function is then polled, in order that if the state of the `genie.alive` object has changed to false, that the Game engine will erase the Genie from the vector of Genies, such that it is no longer rendered.

The second collision detection is between Sorceress and Genie, such that if a collision is detected, the `scoring` class object is called by Game to update the number of lives Sorceress has remaining.

The last poll done is for the `explosion` class. If the collision between Genie and Lightning is greater than 2 seconds, then the explosion object will then be rendered by the `game` class.

Once all updates have taken place the game objects are rendered such that the presentation layer accurately reflects the changed states in the data and logic layers. The rendering is done using the `graphics` class through the `std::shared_ptr` to the `graphics` class.

This logic can be seen in Figure 4 in the Appendix

6.2. *User Input*

User Input is linked to the `controls` class. The class checks for different user inputs linked to movement, shooting of lightning and escaping the game. The aim is to decouple user input/controls from both the game engine [3], as well as, the respective game objects that utilize user input (for example Sorceress moves based on user input).

Since user input can move Sorceress (in 4 directions), fire Lightning and end the game – each of these inputs is associated with a member of an enumeration class. These are then polled in a switch statement flow of control; such that multiple events cannot be triggered simultaneously. This is done whilst, still separating concerns as the `controls` class has the responsibility of checking a key and returning a Boolean to the object calling it.

6.3. *Collision*

Each game object (Sorceress, Lightning and Genie) has a rectangle bounding box around the sprite in order to define the object.

Two collision events are checked: (1) collision between Sorceress and Genie and (2) Genie and Lightning. For the specific collision, each object within the vector is iterated through using STL commands and a check is carried out if the global bounds of one object rectangle intersects the other objects rectangle global bounds. If an intersection occurs, that indicates a collision and specific actions as outlined in Section 5 are then carried out.

Figure 5 of the Appendix illustrates the mechanism behind collision detection.

6.4. *Monster Chase Algorithm*

The monster chasing Sorceress algorithm relies on monster having the knowledge of Sorceress's current position in (x, y) co-ordinates

The Genie function therefore, has access to these co-ordinates. Initially, the shortest path algorithm used was Dijkstra's algorithm, which aimed to find the shortest path within the tunnel. However, the implementation of the disembodied movement or "ghosting" feature required the algorithm to be modified.

A shortest path algorithm based on the Pythagoras theorem is implemented. The algorithm calculates the horizontal (x) and vertical (y) distance between the monster and Sorceress.

The shortest path distance is calculated based on Pythagoras Theorem. If Sorceress is within a certain radius/proximity of the monster, the monster then enters chase mode along the shortest path.

Finally, if the Genie moves out the tunnel, then the Genie moves into ghost mode to continue chasing the player out of the tunnel.

The algorithm is a simple chase algorithm, which highlights the design trade-off made. It is decided to prioritize functionality of the simple algorithm, over a more complex algorithm which would compromise on basic functionality.

7. Design Analysis

In order to critically analyse the strengths and weaknesses of the proposed solution, there must be an analysis of the code's design, as well as, the functionality of the game.

7.1. *Code Design*

Software development has certain common practices which are considered as poor design decisions (i.e. "code smells") [1]. The assessment of code design will be based on these common factors.

7.1.1. *Strengths*

The design of the 3 layers and the separation of concerns thereof, is achieved successfully as is shown in Figure 2. Firstly, as is discussed in further detail below, the graphics library (i.e. SFML) is decoupled and this is beneficial with regard to separation of concerns. The layering is good as classes belong to only a single layer. Therefore classes do not have intimate knowledge of classes in the same layer or different layers. This allows for hiding of functionality from other classes which is beneficial. The communication between layers is tightly controlled with only specific intermediary functions (e.g. a getter function or update caller function). In the case of presentation to logic, `game` class serves as the intermediary. All of this means that there are highly distinct layers which are beneficial to the design.

Pure data classes are avoided in the design of the code. The `background` class fulfils the role of loading sprite resources from memory and also stores the background sprites. It also has a shared pointer to the `graphics` class. The `tunnel` class is responsible for reading in and storing the positions of tunnels and drawing of tunnels. Lastly, `game` class can access the data. These design decisions conform to good practice.

A design decision is taken to make use of smart pointers, in particular shared pointers to the `graphics` class. This is a good use of idiomatic C++, as well as, beneficial in terms of memory management. However, this will be covered further under analysis of Functionality.

A good usage of modern C++ 14 was implemented in terms of the `graphics` class. The usage of virtual functions in the class is important as game objects derive from the `graphics` class. The virtual function would allow the derived class implementation to be executed. Whilst, pure polymorphism is not implemented, a design decision is taken that since virtual functions were implemented that polymorphism would be able to be easily implemented should it be required. Thus, it is a design decision with the future in mind. Further analysis concerning the usage of modern C++ 14 is discussed in the functionality section.

Most classes in the design are relatively short (approximately 40 lines). This is beneficial as it allows for easier debugging of code, as well as, having tight, singular defined responsibilities of each class which prevents the problems that occur with monolithic classes. This means that classes have well defined responsibilities linking well to the notion of separation of concerns and non-overlapping responsibilities.

Class intimacy is avoided as each class only knows about the inner workings of its own class. This is encapsulation is achieved through the usage of specific functions which control access to the class. For example: access to Genies movement is only possible through the `updateMovement` function accessed only through the `monsterMove` class which controls access. Alternatively, the knowledge about Sorceress's position is tightly controlled through the getter functions. Another example is that the intermediary `game` class, manages the `controls` class and Sorceress has knowledge of the changes in `controls` based on user input.

The `graphics` class (representing SFML) has been decoupled from the rest of the code. A specialized class to deal with the graphics library (i.e. SFML) is beneficial to control access to the library itself. Also, if the SFML library functionality is changed to an alternate graphics library (such as SIGEL, SDL or Allegro), only the graphics class needs adjustment. This is good practice with regard to separation of concerns, as well as, allowing code re-usability.

A vector of objects of Sorceress, Genie, Lightning and Tunnel is created instead of a vector of object pointers. The comparison can be seen in Figure 6 of the Appendix [6]. A vector of objects becomes contiguous in memory, whilst the pointers are not contiguous in memory as can be seen in Figure 6 [6]. The trade-off is that a vector of objects requires memory to be manually managed but allows for superior performance when using iteration [6]. The vector of pointers will have better memory management as the vector of objects is allocated on the heap and thus, only destroyed when the object itself is destroyed. The design decision of a vector of objects is beneficial due to the specific implementation in the game. Iteration through the vector is crucial to functionality as this speed/performance has been prioritized. There are only a few objects in the solution thus the memory drain isn't significant. The manual deleting or destruction of memory takes place in the code when Sorceress or Genie is killed or after Lightning is fired. Therefore, this mitigates the memory issue in favour of performance.

7.1.2. Weaknesses:

The majority of the classes are short however, the `game` class can be seen as a monolithic class, as it has 2 distinct responsibilities which is drawing the game and updating the game state. Section 8 will discuss how this can be rectified, especially since `game` has knowledge of all classes being the intermediary function. The monolithic nature of the class also meant that the class functions (in particular the main game loop) are quite long which is not good practice.

There is partially duplicated code in terms of reading in and storing sprites from memory. The necessity of functionality is prioritized over the DRY principle. Section 9 will deal with overcoming this weakness.

Beside the `sorceress` class, data encapsulation in the `genie` and `lighting` class could be better implemented. This is due to data members being modified directly, due to the absence of getter and setter functions. This is not good coding practice and better use of getters and setters should be used so that data can be hidden from other functions, rather than directly accessing the data members.

Polymorphism isn't well implemented in the code and therefore, leading to the potential repeating of code. For example, an entity class could have been implemented and Genie and Sorceress could have both derived from or inherited from it. Despite the lack of polymorphism there is still a clear hierarchy in the code base.

7.2. Functionality

7.2.1. Strengths:

All the basic functionality features coupled with minor and major functionality has been implemented. The required game objects are implemented. Sorceress is successfully able to move around the map, as well as, dig tunnels whilst Genie monsters chase Sorceress autonomously. Sorceress has multiple lives adding to the game experience. Moreover, gameplay is made more challenging through the implementation of a ghosting/disembodied movement feature for the red Genies. A significant add on is that Sorceress has the ability to fire lightning at the Genies which become enveloped in the explosion and disintegrate if struck for over 2 seconds. Lastly, if the player either wins or loses the game a specific screen is loaded to signify the end of the game.

The collision detection method of rectangle global bound intersection checking used for killing of Sorceress or Genies works well and provides game performance as expected.

Modern, idiomatic C++ 14 is implemented in the functionality both in terms of memory management, as well as, functional usage. Firstly, the usage of smart pointers with regard to the usage of `std::shared_ptr` towards the `graphics` class. This is used opposed to conventional pointers as the pointer is automatically deleted which assists in memory management. Secondly, modern C++ 14 is successfully implemented through functional usage of STL in terms of vector based iterators, as well as, enumerator class types. Additionally, `const` is utilized in `graphics` class.

Lastly, the game possesses good graphics, which are significantly higher resolution than the original game.

7.2.2. Weaknesses:

The chase algorithm has some weaknesses which detract from the game play. The Genie monsters currently chase the monster along the shortest path based on the Pythagoras Theorem. This type of chasing achieves chase functionality (i.e. the basic functionality) however, a different algorithm should be explored (see section 9), in order that it might improve the game experience.

There are game speed issues of variable game speed when tested on different computers. In particular, on systems with higher performance specifications the game plays faster, whilst on lower specification systems the game plays slower. While the frame rate has been limited to try mitigate this issue, it would be wise to address this issue (discussed in Section 9), such that the game plays at the same speed on any computer giving a consistent game experience.

The ghost mode has bugs when the Genie monster moves out of the tunnel and into the Earth to "ghost". The Genie only changes to ghost mode once the entire rectangle bounding box is out of the tunnel and returns to conventional genie mode once the entire rectangle bounding box is within the tunnel bounds. This isn't a huge functional issue, but it does detract from the gameplay experience.

8. TESTING

The Google Test framework is used as the test environment for the project. Unit testing is done to ensure code units work as expected and that functionality is achieved. The aim is to test each functional unit independently, to assist a developer in assessing the code base.

Most functions of each class are tested in the Google test section of code which is implemented in gtest.cpp. The majority of game logic has been covered by the unit testing, however some areas have not been tested and hence will be discussed along with the rationale.

In terms of untested functionality, the areas which are not tested is the graphics (i.e. SFML) modules, as well as, modules requiring user input. Furthermore, whilst most functions which utilized timers are tested by setting the timer condition to true, those functions which purely relied on time rather than a timer condition being met are not tested.

The `controls` class, the Sorceress move function and the game close functions are not tested as they require user input. The Google Test framework does not have functionality to either simulate user input or create an event type of user input. The classes could be tested if they are re-factored into a function containing user input checks and a function containing the resulting functionality. Thus, if the user input is decoupled from functionality in a different way, it might allow for unit tests to be written for these functions.

The `kill_Srs` function in the `killSrs` class and the `mons_lighning` function in the `collision` class is tested, however not all functionality is tested. The functions rely on timers to execute certain conditions and since the googletest framework cannot deal with timers it is not tested. However, the tests involve testing for situations when the timer condition would be met and when it wouldn't be met to still ensure that the unit testing is as thorough as possible.

The `graphics` class in the entirety is not tested. The `graphics` class includes all functionality of the SFML library. The class is not tested on the basis that there is an assumption that the SFML library itself has been tested by the developers and that all units of SFML will function as expected. Hence, SFML and by virtue the graphics class does not need to be re-tested.

A mocking framework (FakeIt or Google Mock mocking framework) was attempted to be implemented as an addition to the Google Test framework. However, the implementation of FakeIt/Google Mock to the code base proved unsuccessful. The merits and rationale of a mocking framework will nevertheless be discussed below as an important consideration for testing.

Ultimately, the tests performed tested most of the functions of the game and indicated that all areas of the game functioned as expected. However, certain areas unfortunately were unable to be tested. This issue could either be rectified by refactoring the code to separate/decouple the game functionality/logic from the user input. Alternatively, a mocking framework such as FakeIt/Google Mock could be implemented. A mocking framework allows the testing of classes which aren't well defined or require user input by creating "mock" versions of the class in order to simulate the classes overall behaviour. This is an important long term consideration as a mocking framework would be beneficial should abstract classes be implemented.

9. FUTURE IMPROVEMENTS

9.1. Design

- The chase algorithm could be modified with another shortest path algorithm which allows for a smarter AI. For example: Johnsons Algorithm or the A* Search Algorithm.
- To mitigate the monolithic `game` class, the class could be split into an interaction class (intermediary between presentation and logic) and an interface class to draw the game [3]. This result in better modelling of responsibilities and separation of concerns. Moreover, it would reduce the interaction that `game` shares with the logic layer.
- Error handling could be better handled, in terms of rather than throwing exceptions, these could be handled such that the game continues.
- Better data encapsulation and data hiding should be implemented in the `genie` and `lightning` classes. This can be rectified with better usage of getter and setter functions.
- Inheritance and polymorphism should be implemented to create a superior domain model. For example: an entity class could be implemented with specific attributes. Sorceress and Genie could then inherit from entity class. Moreover, if the second type of monster is implemented this type of inheritance could prove useful to the design.

9.2. Functionality

The game was capped at a rating of “Good” from the second submission. Therefore, it is not viable to add features to push the rating up to an “Excellent” rating for the Final Submission. There are missing features from the game to completely emulate the original Dig Dug game.

These features are:

- The addition of a green dragon monster which can breathe fire, chase Sorceress and ghost through tunnel walls.
- Rocks in the earth exist which can be used to kill monsters by making a tunnel underneath the rock. If Sorceress would hold the rock for too long, she would be crushed by a rock.
- Bonus items would appear on the screen which would give the player additional points or health. These items would have a limited lifespan on the screen.
- A score displayed on the screen would provide the player information on the points he/she earn from killing monsters and digging the earth. A high score screen at the end of the game will be provided where players can save their name with their score.
- A pause screen
- Additional levels of progressive difficulty
- Like the original game, the nature of monsters to change after a certain time so that they escape to the surface and leave the screen. This reduces the points Sorceress can achieve.
- Improvement of the aesthetic features of the game where the monsters face the direction which they move in and better quality sprites are used.

10. CONCLUSION

A Dig-Dug like game has been successfully designed and implemented in C++ with SFML. The game achieved basic functionality, three minor feature (Sorceress has extra lives, monsters can chase through tunnels in disembodied/ghost form and good graphics) and one major feature (Lightning which causes Genie to explode). Furthermore, the game has well-defined layers and separation of concerns. The game predominately utilizes good coding practices, as well as, an objected-oriented design approach. It also makes good use of modern, idiomatic C++. That being said there are issues which must be addressed including: code repetition, the monolithic game class, as well as, issues of poor data encapsulation and a lack of polymorphism in the code base. Unit testing proves functionality in most game functions, however classes related to graphics and user input were unable to be tested. Proposed solutions to the issues in the design, functionality and testing are proposed. The game ultimately can be considered a success in meeting the criteria outlined and can be significantly improved by dealing with the future improvements.

References

[1] G. Singh and V. Chopra, “A Study of Bad Smells in Code,” *International Journal for Science and Emerging Technologies with Latest Trends,* vol. 7, no. 1, pp. 16-20, 2013.

[2] D. Conger and R. Little, “Writing C++ Programs,” in *Creating Games in C++: A Step-by-Step Guide,* Berkeley, CA, USA, New Riders, 2006, pp. 27-29.

[3] R. Pupius, “The interface class,” in *SFML Game Development by Example,* Birmingham, UK , Packt Publishing Ltd, 2015, pp. 297-298.

[4] Microsoft, “Resource Management Best Practices,” URL:https://msdn.microsoft.com/en-us/library/windows/desktop/ee418784(v=vs.85).aspx. [Last Accessed 26 September 2016].

[5] R. Nystrom, Game Programming Patterns, Genever Benning, 2014.

[6] B. Filipek, “Vector of Objects vs Vector of Pointers Updated,” 19 May 2014. URL: http://www.bfilipek.com/2014/05/vector-of-objects-vs-vector-of-pointers.html. [Last Accessed 26 September 2016].

APPENDIX

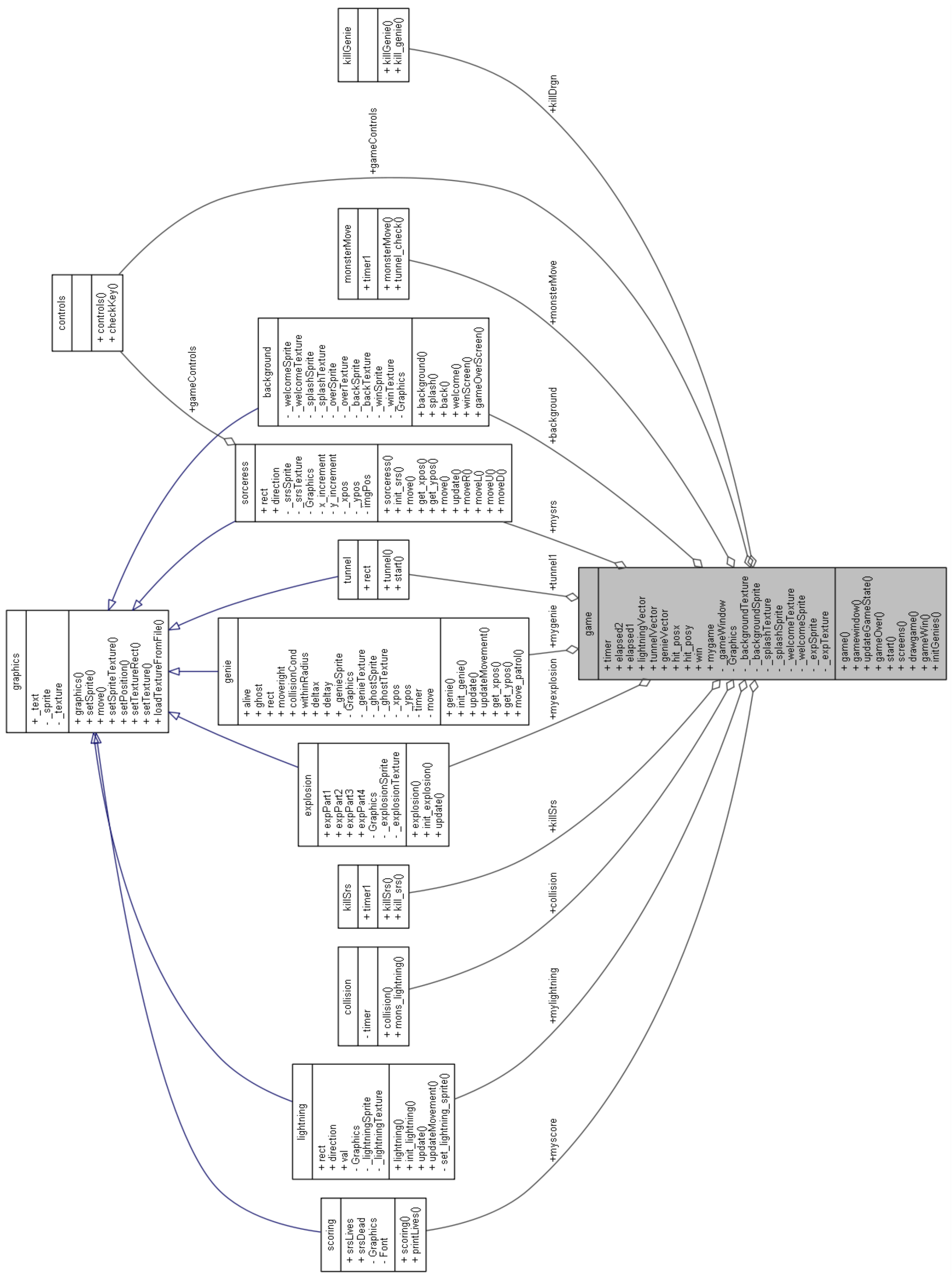

Figure 1: UML Class Interaction Diagram

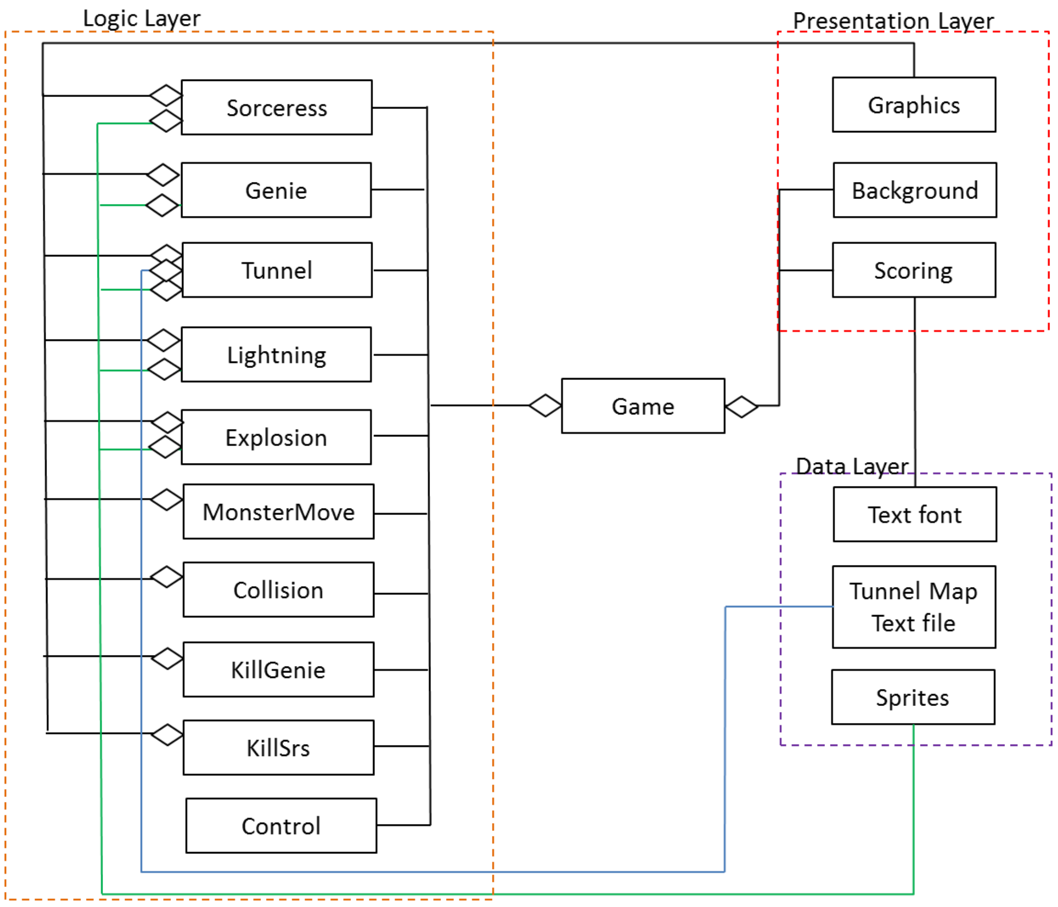

Figure 2: Condensed UML Class Diagram with Layer Separation

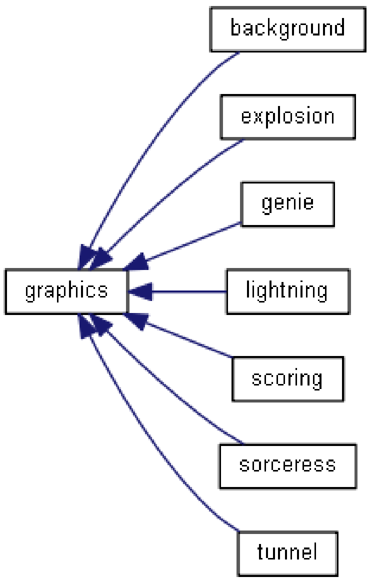

Figure 3: Graphics Class Shared Pointers

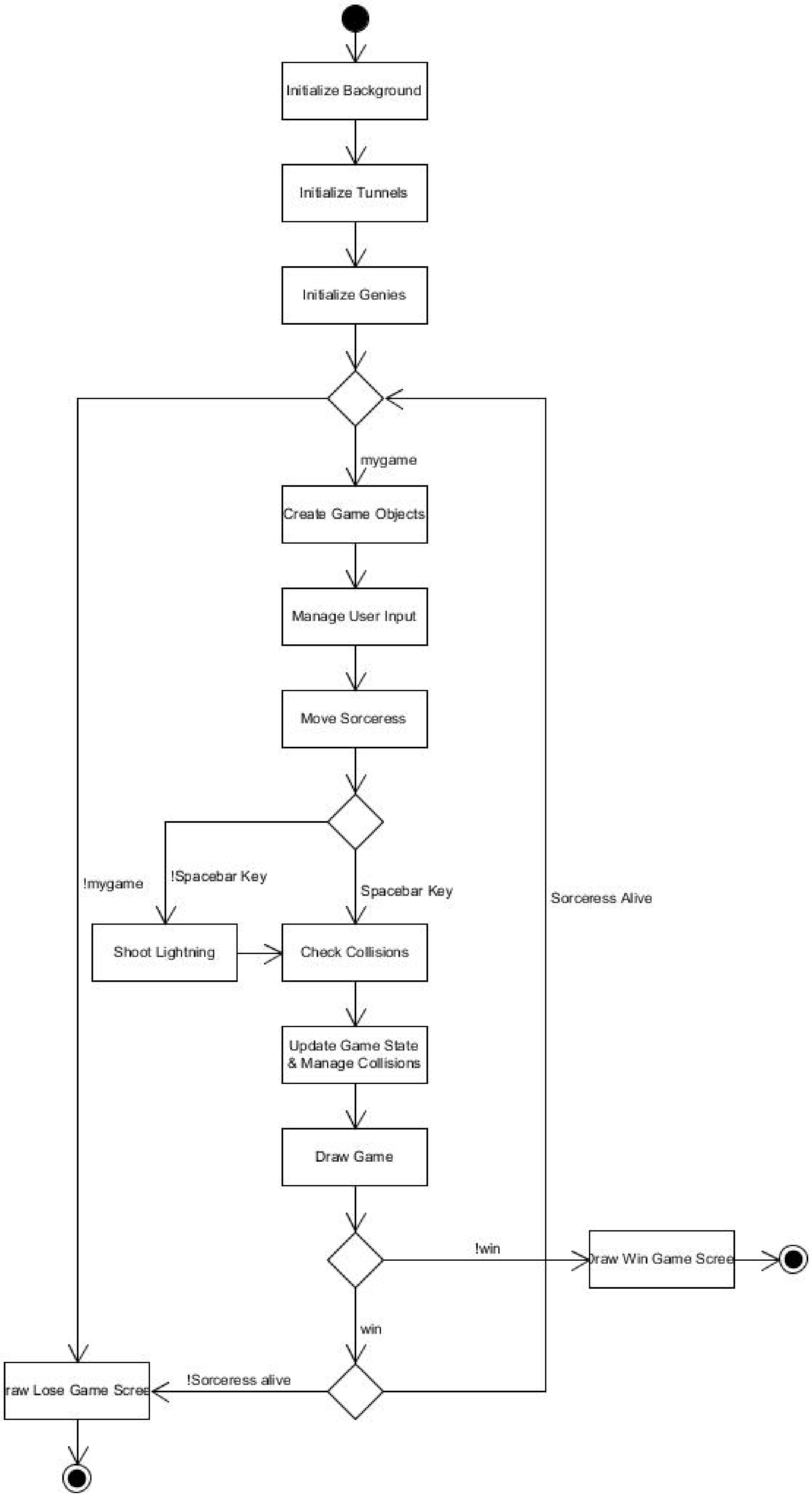

Figure 4: UML Activity Diagram of the Game Class

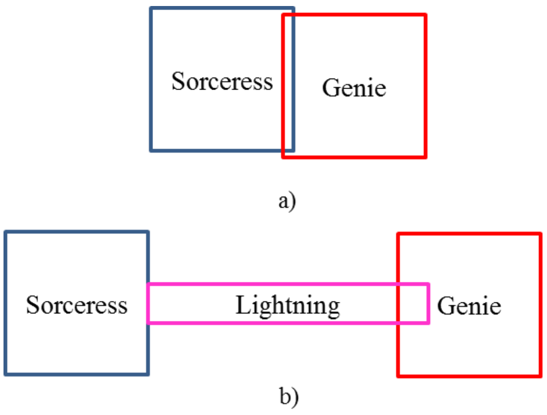

Figure 5: Collision mechanism between objects

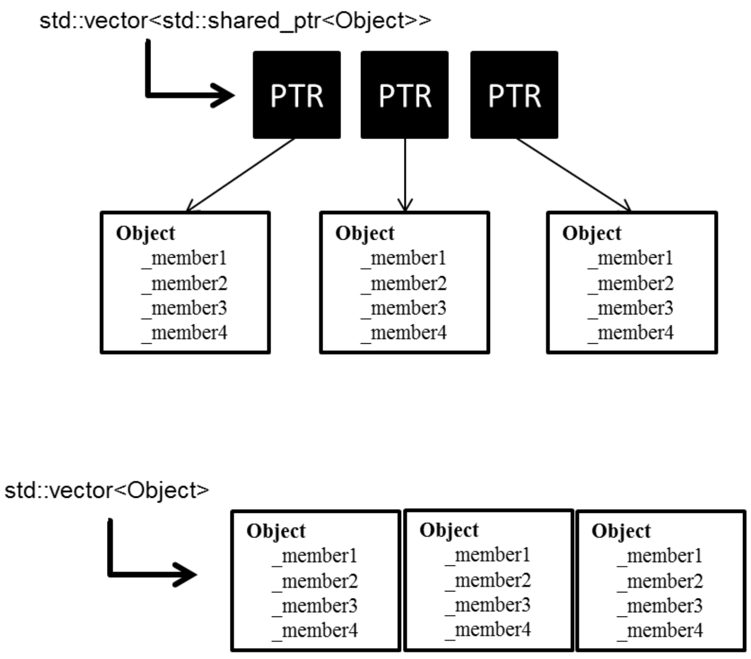

Figure 6: Vector of Objects versus Vector of Pointers (adapted from [6])