|  |
| --- |
| CPSC 2720 Spring 2018 |
| BBG Adventure |
| Team Big Boys  Letter: G |
|  |
| **Gideon Richter – Justin Creig – Jesse Huss** |
| 16/02/2018 |

Table of Contents

[Revision History 3](#_Toc506502246)

[Introduction 4](#_Toc506502247)

[Project Management 5](#_Toc506502248)

[Team Organization 5](#_Toc506502249)

[Risk Management 5](#_Toc506502250)

[First step – Identify 6](#_Toc506502251)

[Second step – Evaluate 6](#_Toc506502252)

[Third step – Eliminate 6](#_Toc506502253)

[Development Process 7](#_Toc506502254)

[Coding Conventions 7](#_Toc506502255)

[Code review 8](#_Toc506502256)

[First stage – Personal 8](#_Toc506502257)

[Second stage – Peer 8](#_Toc506502258)

[Third stage – Project 8](#_Toc506502259)

[Team Communication 9](#_Toc506502260)

[Change Management 9](#_Toc506502261)

[Software Design 10](#_Toc506502262)

[Design Diagrams 10](#_Toc506502263)

[Design Rationale 10](#_Toc506502264)

[Appendices 12](#_Toc506502265)

[UML Class Daigram 12](#_Toc506502266)

[UML Sequence Diagrams 13](#_Toc506502267)

[References 14](#_Toc506502268)

# Revision History

|  |  |  |
| --- | --- | --- |
| **Revision Date** | **Changes Made** | **Name** |
| 2/10/2018 | Added title page and logo | Gideon Richter |
| 2/11/2018 | Added introduction, team organization, code review guidelines, team communication | Gideon Richter |
| 2/12/2018 | Added risk management section | Jesse Huss |
| 2/12/2018 | Added change management section and risk management table | Gideon Richter |
| 2/14/2018 | Added code conventions and UML class diagram | Gideon Richter |
| 2/15/2018 | Added sequence diagrams | Gideon Richter |
| 2/15/2018 | Added design rational section  Added references section | Jesse Huss |

# Introduction

BBG Adventure is a text-based adventure game developed as part of a group project for the course Practical Software Development & Design at the University of Lethbridge. The project has three major sections – Design, Implementation, and Testing. The members of the Big Boy team are: Gideon Richter, Jesse Huss, and Justin Creig.

The adventure plot revolves around a lost father, who, approaches an abandoned manor house in search of directions – only to never return. It is the Hero’s quest to enter the house and find out what happened.

Upon entering the house, the Hero discovers a mad scientist who has transformed the father into a duck with his Electro-object-polymorphizer. Unfortunately, after duck-ifying the father, the machine blew up and parts were sent flying throughout the grounds. Too busy with his other mad-scientist obligations, the Hero is enlisted to search for the three missing parts and return them.

In their search of the house, the Hero encounters rats, ghouls, and school of piranhas while traversing a dark, scary, dark-scary cellar, a spooky gravesite, and manor grounds.

Of course, not every adventurer will return from their adventure – be warned, there are many ways to die in this adventure, or end up locked in the house forever.

But let’s not dwell on failure, there are many ways to succeed! The Hero is provided with a save/load system that allows them pick up where they left off, or travel through time after coming to an untimely end. Furthermore, an in-game action manual provides the hero with all necessary (or unnecessary) text actions that the game can parse.

This document serves as an overview of the design process of the game thus far. The topics include:

* Project Management: Team Organization, productivity, and risk management
* Development Process: Code review and conventions, issue tracking, and team communication
* Software Design: UML class/sequence diagrams, use cases, design principles and patterns

# Project Management

Provide a description of and address any foreseeable problems. TODO: Intro: Summarize contents of sections subsections.

## Team Organization

A close up of a logo

Description generated with very high confidenceThe Big Boys team will be organized as a democratic or open structured team, where all members will have the same opportunity to dictate and participate in team activities.

Figure 1: Example democratic team structure

The team members will have the following roles:

* Team Lead: Gideon Richter; organize and keep everything on track
* Design Lead: Justin Creig; ensure adherence to good object-oriented design
* Quality Assurance Lead: Jesse Huss; ensures implementation of design specifications

Furthermore, all team members will fill the role of Software Developer and Software Tester, while contributing to project documentation.

## Risk Management

As with any project, there are risks involved that can hinder the completion of the project. Foreseeable risks will be dealt with in three steps; each ensuring that problems are dealt with before they occur.

Identifying problems allows team members to be on the lookout for developing issues. You can not fix something unless you know that it exists.

Evaluating risks will be done as a team during scheduled meetings. Team members are encouraged to speak their minds regarding the project; as part of a democratic team structure this is crucial to our success.

Eliminating risks is a continuous effort. Initially to reverse any risks upon exposure, and secondly to make sure they are not repeated. An elaboration of these steps follows:

### First step – Identify

1. Identify the risk, likelihood of occurrence, and severity
2. Identify aspects of the project that will be affected
3. Categorize risk as:
   1. Time: team scheduling, lack of or misuse of time
   2. Technical: major design changes, project scope, learning new technologies
   3. People: loss of a team member, family emergencies/illness, not contributing

### Second step – Evaluate

1. Evaluate potential risks based on likelihood and severity of the impact. Risks are ranked as:
   1. Likely, severe impact
   2. Likely, minimal impact
   3. Unlikely, severe impact
   4. Unlikely, minimal impact

### Third step – Eliminate

1. If possible, remove the risk. Removing as many risks as possible is key for a productive project. Easy risks to remove include:
   1. Examples:
      1. Design flaws caught early
      2. Lacking knowledge or skill, assign tasks based on capability
2. If the risk can not be removed, reduce it. This makes the risk more manageable.
   1. Examples:
      1. Deal with scheduling in the initial stages of the project. i.e. Using scheduling software to see when team members are available.
      2. Discuss project complexity to determine attainability.
3. Plan to deal with any risks that may still occur. This may include:
   1. Examples:
      1. Internal deadlines that provide sufficient time before official deadlines.
      2. Break project up into smaller parts. i.e. Key functionality/requirements, features that can be added once core of project is complete if time allows.

|  |  |  |  |
| --- | --- | --- | --- |
| Risk | Category | Severity | Action Taken |
| Team availability | Time | High | Scheduled meetings must be no more than 20 mins. Default mode of communication is Discord. |
| Difficulty working on project away at home | Technical | Moderate | Team members must have *at least* a functioningVPN and a Git client on their home computer. |
| Lack of team communication | People | High | Team members are assigned small, modular tasks with clear deadlines. This means tasks can be established with less communication while maintaining certainty. |
| Lack of direction resulting from unknown project definition | People | Moderate | Game plot has been established quickly as to de-abstract discussions and ideas. |
| Project deadlines and procrastination | People | Moderate | Recognize that we can finish the project while maintaining that it is sensible to not underestimate time requirements. |

# Development Process

## Coding Conventions

|  |  |
| --- | --- |
| ID | Convention Description |
| 1 | Every .cpp file should have an associated .h file |
| 2 | Header files should be self-contained |
| 3 | Definitions for template and inline functions should be in the same file as their declarations |
| 4 | All header files should have #define guards to prevent circular inclusion |
| 5 | Define inline functions when they are smaller than 10 lines |
| 6 | The use of `using namespace \*` should be avoided to prevent namespace pollution |
| 7 | When definitions don’t need to be used outside that file, declare them static |
| 8 | Avoid virtual method calls within constructors |
| 9 | Data members should be private |
| 10 | For functions, parameter order is inputs then outputs |
| 11 | Write short functions |
| 12 | Document function overloading |
| 13 | All names should be written in English |
| 14 | The prefix ‘is\_’ should be used for Boolean variables and functions |
| 15 | Abbreviated names should be avoided |
| 16 | Use explicit type conversions |
| 17 | Infinite loops should use while(true) |
| 18 | Use underscore case for names |
| 19 | Curly braces shall be on the same line |

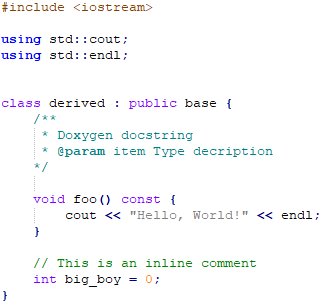


Figure 2: Example Code Conventions

## Code review

Code review will occur in three stages. These stages help ensure that code entering the production codebase of the game is up to standard and is bug-free.

### First stage – Personal

Prerequisites:

* Code should compile
* Code should be free of warnings
* Code should follow project coding conventions
* New functionality should be accompanied by (non-exhaustive) unit tests
* Existing unit tests broken by new code should be fixed, replaced, or deleted

If prerequisites are met:

1. Test local merge with up-to-date develop branch and ensure no conflicts
2. A GitLab pull request should be made develop branch
   1. with sufficient description
   2. with no merge conflicts
   3. with no unnecessary/temporary files
   4. and assigned to at least one other group member

### Second stage – Peer

1. Visually review changes
2. Locally checkout branch
3. Locally merge reviewed branch with up-to-date develop branch
4. Run unit tests
5. Merge pull request into develop branch

### Third stage – Project

* Periodically, all new changes are reviewed again and merged into master branch
* This ensures that the master branch is always stable not directly modifiable (unless in extraordinary circumstance)

## Team Communication

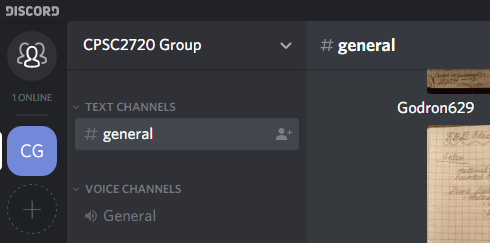
Team communication will occur mainly through a discord channel. Other methods such as texts and emails are also acceptable. Communication regarding code review should take place on GitLab pull requests and issue tracking.

Figure 3: discord channel

## Change Management

GitLab’s Issue tracking is not only useful for filing bug reports, but also for acting as a job board. Issues can be created, organized, and assigned to a team member. It is through this issue tracking system that we will handle change management as well as programming tasks.

All members of the team are free to assign themselves to unassigned issues, although high priority or resource specific issues may be assigned to a team member.

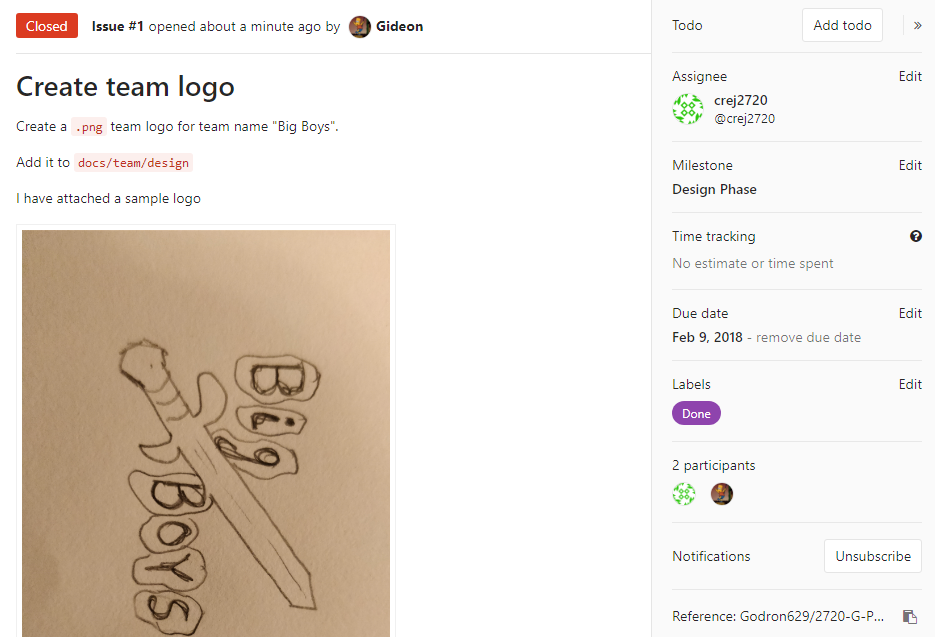


Figure 4: Example GitLab Issue

# Software Design

In this section we will describe classes and example use cases within the game. There are 5 classes, and already there have been discussions about class additions and abstractions that will make the design better.

The game experience is handled by the Game class, which interprets console commands, and appropriately dispatches tasks. The Game task also keeps track of where the Hero is and their inventory.

The Thing class is used to represent items and characters within the game, which are held within the Inventory class. Inventory provides methods to add, remove, and search for Things.

The Room class describes the surroundings may be loaded and saved to/from an external file or database, so in-game actions persist.

## Design Diagrams

* Appendix Figure 5: Class diagram
* Appendix Figure 6: Pick up item (success) sequence diagram
* Appendix Figure 7: Talk with NPC sequence diagram

## Design Rationale

The design for this project evolved around the idea of wanting a simple and easily modifiable system of classes that will allow for the most flexibility in the later stages of the project. This will be helpful being that it is still early in the project and when changes ultimately need to be made, it will be easy to do so.

In this section we will describe how our project follows the SOLID principles of object-oriented design.

* S – Single-responsibility principle.
  + This principle states that every class has only one job. As seen in Appendix Figure 5: Class diagram all of the classes in our project follow this principle.
* O – Open-closed principle.
  + This principle states that objects or entities should be open for extension but closed for modification. Basically, functions and classes should be abstracted as much as possible so that more functions or classes may be overridden or extended rather that having to make many modifications to solve the same problem.
  + Currently our class structure follows this principle, however there has been discussions about potential class abstraction and addition.
* L – Liskov substitution principle.
  + This principle states that every subclass/derived class should be substitutable for their base/parent class. For instance, if a class diagram had a class named “square” that was derived from “shape” then “square” and “shape” should be able to be used interchangeably.
  + The design of our class diagram has the functionality this principle talks about. We have classes for inventory items as well as characters which are both derived from “thing” which implies that “thing” may be used interchangeably with both items and characters. This of course will include some sort of type casting.
* I – Interface segregation principle.
  + This principle states that “A client should never be forced to implement an interface that it doesn’t use or clients shouldn’t be forced to depend on methods they do not use.” (kayandrae07) Basically, if an interface has dozens of methods but only a couple are being used, split the interface up into smaller interfaces and get rid of things that are not being used or are unnecessary.
  + Since our project is relatively small it should be easy to keep this principle present in our design without much thought. Having such a small number of classes makes it very easy to keep track of what methods are present. As well with tools like “gcov” finding dead code will be an easy task.
* D – Dependency inversion principle.
  + This principle stresses the need for abstraction. It states that high level classes should not be concrete or depend on the low-level class, but that they need to depend on abstraction.
  + From our class diagram it is clear that our project follows this principle as well. Our “thing” class which is currently our highest-level class does not depend on any other class, however our low-level classes are abstracted from the “thing” class.

Our hope for designing the project around the SOLID principles is to make it easier to extend, modify and test the project. (kayandrae07)

# Appendices

## UML Class Daigram

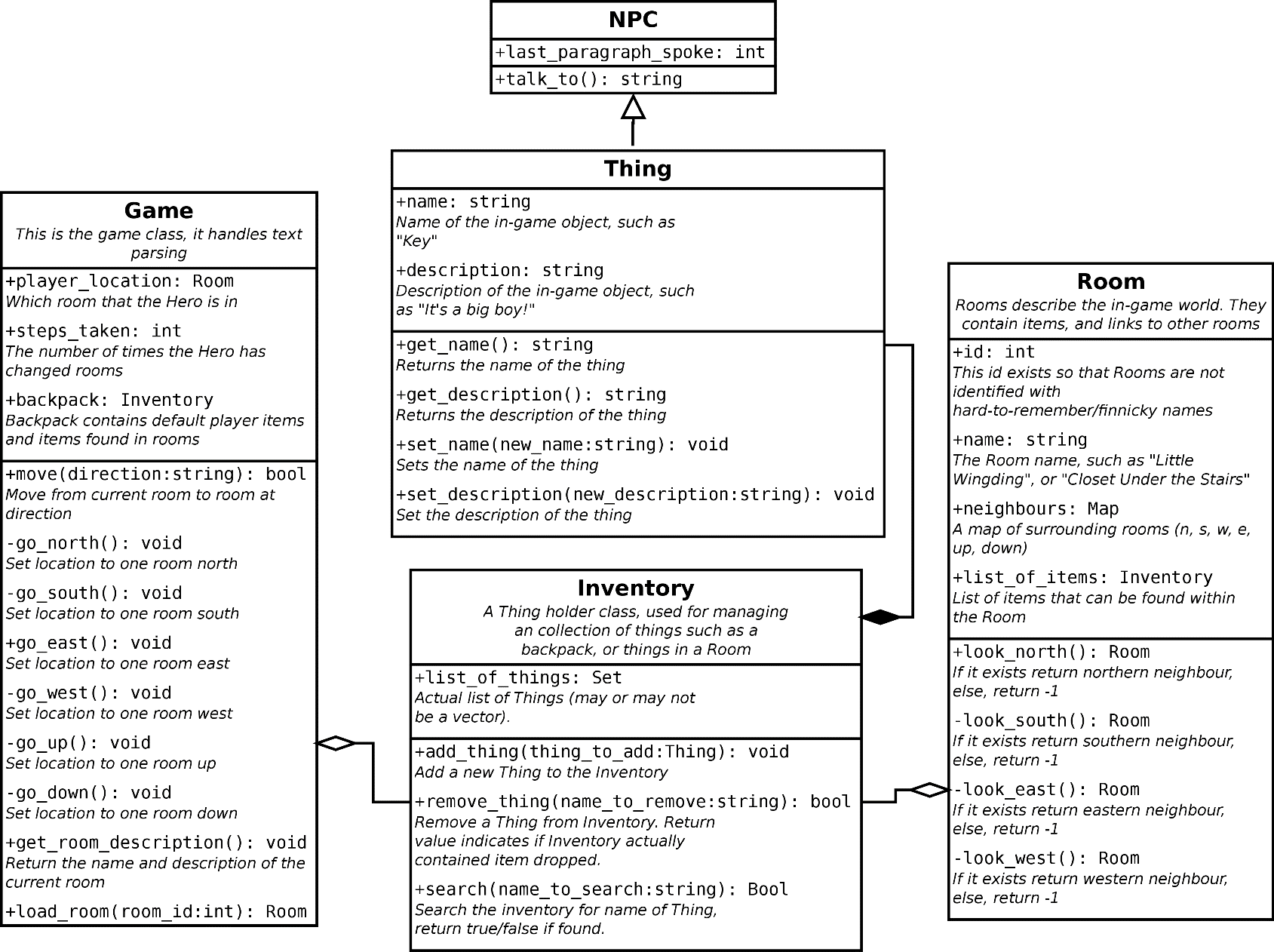


Figure 5:UML Class Diagram

## UML Sequence Diagrams

Figure 6: Talk to NPC Sequence Diagram

Figure 7: Pick Up Item Success Sequence Diagram

# references

kayandrae07. “S.O.L.I.D: The First 5 Principles of Object Oriented Design.” *Scotch*, scotch.io/bar-talk/s-o- l-i-d-the-first-five-principles-of-object-oriented-design.