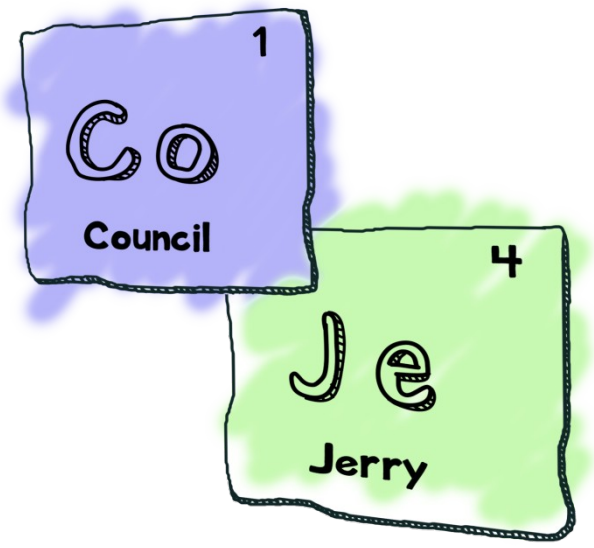


The Council of Jerry's Presents

Jerry's Adventure

By

Nathan Tipper
(aka Boss Jerry)
Vincent Cote
(aka Handsome Jerry)
Jace Riehl
(aka Philosopher Jerry)
Michel Martel
(aka Artsy Jerry)

Table of Contents

Revision History

Revision Date	Change(s) made	Name
2017-01-26	.hgignore added	Nathan Tipper
2017-01-27	.hgignore modified	Nathan Tipper
2017-01-27	Added a directory structure and editted the .hgignore file to use the glob syntax.	Nathan Tipper
2017-02-02	Uploaded .rtf file	Nathan Tipper
2017-02-02	Added DesignLeadDocs directory	Vincent Cote
2017-02-02	Added title page	Michel Martel
2017-02-02	Added the Master Document with the title page	Michel Martel
2017-02-07	Created version 1 of CodingConvensions document	Nathan Tipper
2017-02-09	Finished and added the remaining parts of project management	Vincent Cote
2017-02-10	Added code review process documentation	Michel Martel
2017-02-10	Added introduction, change management, and procedures for configuration management	Jace Riehl
2017-02-10	Revised change management	Jace Riehl
2017-02-12	Added UML diagrams	Nathan Tipper
2017-02-12	Edited UML diagrams formatting	Nathan Tipper

2017-02-12	Edited Character UML to include class inventory	Nathan Tipper
2017-02-12	Communication tool completed and added to the master doucment	Michel Martel
2017-02-12	Added Team report	Vincent Cote
2017-02-13	Design Rationale added	Nathan Tipper
2017-02-13	Revision History added	Jace Riehl
2017-02-13	Changes to CharacterUML01 to reflect Design Rationale	Nathan Tipper

Introduction

Jerry's Adventure is a text based adventure game that follows the story of an average Jerry as he tries to find his way back home. Through a series of unfortunate events, Jerry finds himself launched into another dimension where he has to use his below average intelligence in conjunction with his debonair confidence to stumble his way through the alternate worlds in order to find someone who can help him get back to his garage. On his adventure, Jerry will have to battle distractions, unknown creatures, and his own mental limitations to make it back alive.

This document will help us achieve the amazing game described above. We will be covering subjects such as: how our team is going to work together; how we will stay organized; the different roles each one of us will undertake; address the with the risks of working in a team; how we will effectively communicate as a team; and a detailed outline of our design process using UML diagrams as well as how those diagrams use appropriate principles of OOD.

Jerry would be proud!

Project Management

This section is split into two sections: **Team Organization** and **Risk Management.** The team organization section describes how the team is organized as well as each specific roles from all team members. The risk management section describes foreseeable problems that could occur during the development and testing phase of this project. The topics covered range from illness to slacking team members.

Team Organization

All none code related documentation is to be done using LibreOffice Writer in order to keep a set standard for all team members. As this is a school project and is used as a learning tool for all team member we understand that we are all designers, programmers, leaders and documentation gurus. With that being said the group is still structured with each member having a specific role. These roles are a guideline and will help the team know who to approach with specific issues concerning the different aspects of the project. The specific roles are as follows:

- **Team Lead:** Nathan Tipper
- **Design Lead:** Vincent Cote
- **Quality Assurance Lead:** Jace Riehl
- **Documentation Lead:** Michel Martel

Risk Management

As with any project, uncertainty is always present and can create chaos if the team is not prepared. The team has come up with a few different problems that could occur on a this project as well as some solution on how to deal with said problems if they occur. Being aware that any of these scenarios could occur at any phase during the project is a good method to help prevent and/or handle these issues. The most important issues discussed are as follows:

- **Unforeseen Major Life Event.** This includes, but not limited to, death in the family, major illness rendering the team member unable to work for extended period of time, unforeseen financial setback (lose of house, claiming bankruptcy). As this is the most extreme case and (hopefully) the least likely, we have agreed as a group on what to do in these events. If a team member undergoes one of these unfortunate situations, the remaining team members will take on the responsibilities and divide them equally between each other until the team member is ready to come back.
- **Loss of Team Member.** The loss of a team member could be due to the events listed above or simply by having a team member drop the class. In any case, we feel we have a strong enough team to divide the work into three and still meet the deliverables and deadlines. If two are more team members leave the group a meeting with Dr. Anvik will be arranged to discuss options.
- **Unproductive Team Member.** An unproductive team member does not always mean the individual is lazy (although sometimes it is the case). We value a productive, healthy and open team management style, this means the team members support each other in all aspects inside and outside of the project. We encourage each other at all milestones no matter how small or large, this can lead to higher productivity through positive encouragement. If these prevention methods do not help and a team member is still being unproductive a meeting between all team members is to be called to discuss the issue.
- **Inexperienced Team Member.** It is understandable that sometimes egos can get in the way and lead to a team member taking on more than he/she can handle. Based on the teams values listed above, the group will gain a good understanding of each team members strength and weaknesses. The idea is to help each other guide us into tasks at which we are best at. If a team member falls behind due to a lack of experience, the other team members will intervene in

order to help the one who is falling behind. If needed tasks may be reassigned to better suite the team members strength.

Development Process

Coding Conventions

Throughout the lifetime of this project, the Council Of Jerry's has decided to follow the following coding conventions:

- **Starting blocks on the next line of any scope.**
  - Ie :

```
Class Class1
{
   ...
};
```

- **Using camelCase for both variables (attributes) and functions (members)**
  - Ie:

```
int thisIsAVariable = 1;
void thisIsAFunction()
{
     ...
}
```

- **Class names are to be capatilized and then use camelCase**
  - Ie:

```
Class ThisIsAClassThatDoesNotFollowExistingConventions { ... } ;
```

- **Variables, functions, and class names should be clear and consise and be named with respect to their function.**

- Example:

```
int numberOfJerrys = 4;
```

- **Comments should be short and give a broad overview, leaving the variable, function, and class names to speak to the functionality of the code.**

- **All nested scopes should be indented a level in for a readibility.**

- **Each header file should start with a ifndef, def statement to ensure header files are not duplicated throughout the code. The defined statement should be read as the name of the class in all capital letter followed by "_H."**
  - Example:

```
#ifndef SOMECLASS_H
#define SOMECLASS_H

class SomeClass { … };

#endif // SOMECLASS_H
```

- **Use accessor methods for users to use instead of public attributes.**

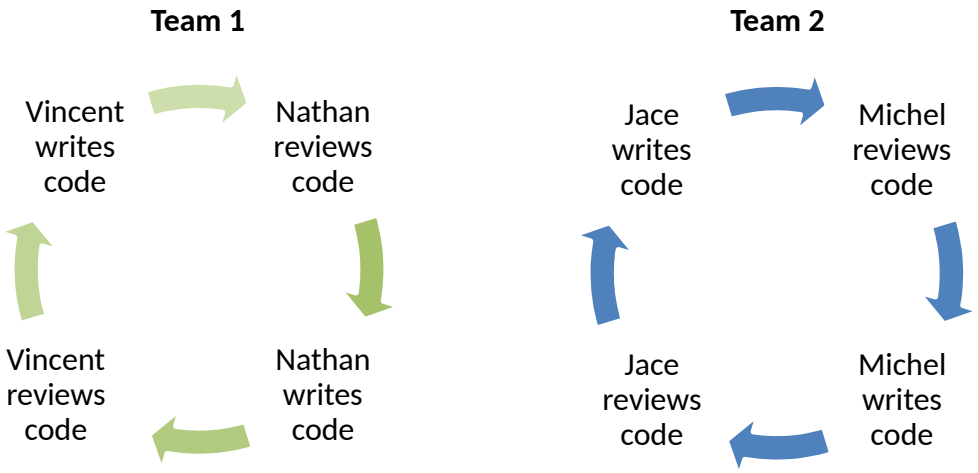
- **Documentation of all members in header files is recorded and is required to have all of the following:**
  - ❑ What the function does.
  - ❑ Description of parameters and what the return value is.
  - ❑ How the function modifies the object.
  - ❑ The pre-condition and post-conditions of the function.
  - ❑ Any restrictions the function may have.

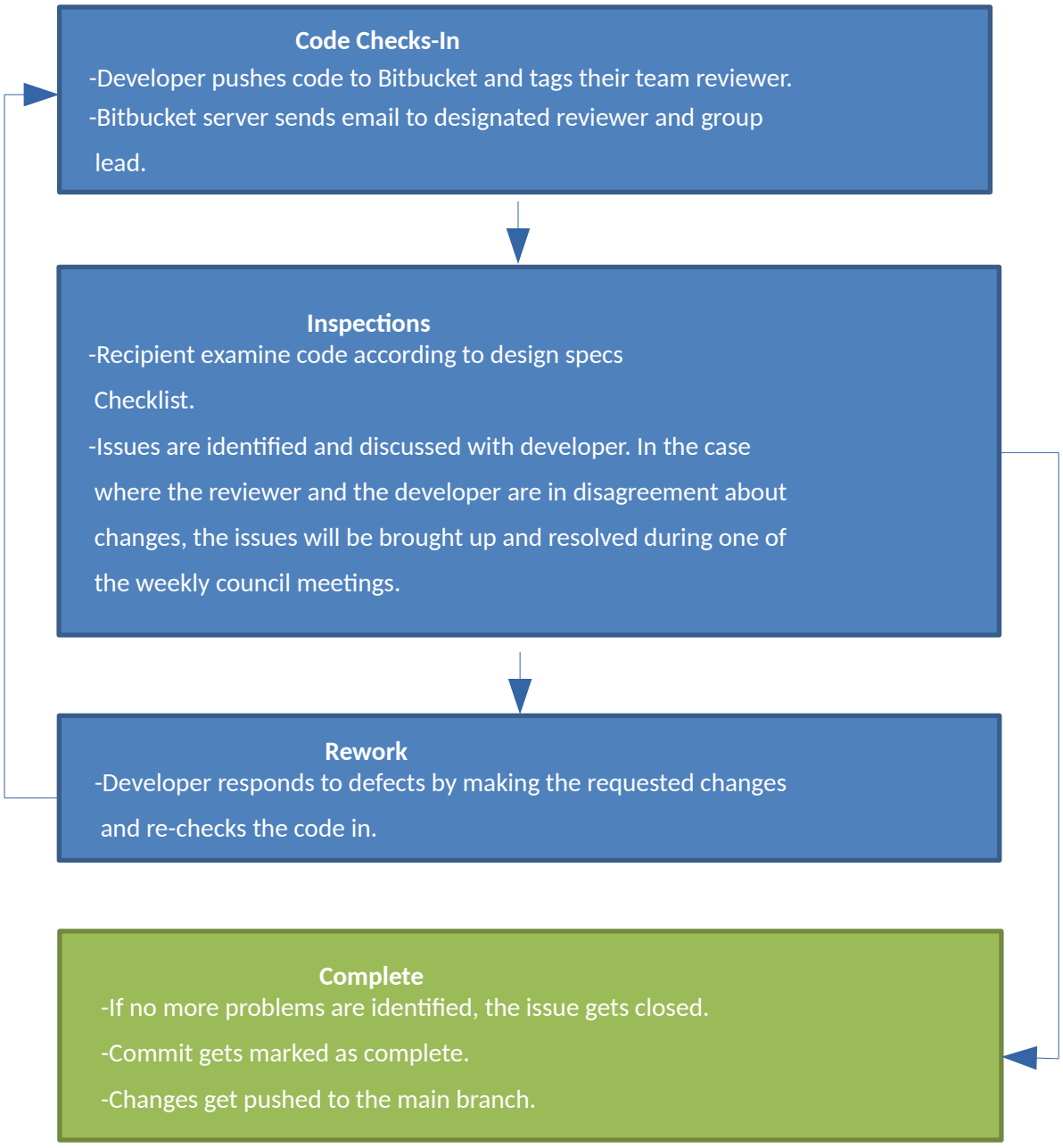
Procedures for Configuration Management

As agreed upon by the council of Jerry's, Pull requests are to be handled in a 'pull requests' fashion, where if there is a branch off it will require a pull request to merge back into the trunk which must then be approved by the design lead. For more information on merging back into the main trunk, refer to the code review process section.

Code Review Process

The code review process will revolve around a buddy system where completed code gets reviewed by one of the paired team member. The teams and processes are as follow:





In order to maintain a high quality review standard, any code written and reviewed should adhere to the following checklists:

Checklist for Developers

- ❑ My code compiles
- ❑ My code has been developer-tested and includes unit tests
- ❑ My code includes doxygen formatted documentation where appropriate
- ❑ My code is tidy (indentation, line length, no commented-out code, no spelling mistakes, etc)
- ❑ I have considered proper use of exceptions
- ❑ I have eliminated unused imports
- ❑ The code follows the Coding Standards
- ❑ Are there any leftover test routines in the code?
- ❑ Are there any hardcoded, development only things still in the code?
- ❑ Was performance considered?
- ❑ Was security considered?

Checklist for Reviewers

- ❑ Comments are comprehensible and add something to the maintainability of the code
- ❑ Comments are neither superfluous nor scarce
- ❑ Types have been generalized where possible
- ❑ Exceptions have been used appropriately
- ❑ Repetitive code has been factored out
- ❑ Functions have all been defined appropriately
- ❑ Classes have been designed to undertake one task only
- ❑ Unit tests are present and correct
- ❑ Common errors have been checked for
- ❑ Any security concerns have been addressed
- ❑ Performance was considered
- ❑ The functionality fits the current design for the game
- ❑ The code is unit testable
- ❑ The code complies to coding standards

Communication Tools/Channels

The primary methods of communication for the duration of the project will be face to face meeting as well as using slack for text based communication regarding general information, random information, design documentations, game design, game structure and repository documentation; all of which are subdivided into their own channels. If face to face meetings are not feasible due to scheduling conflicts the team is to arrange a video conference call on Google Hangout at the earliest opportunity.

Change Management

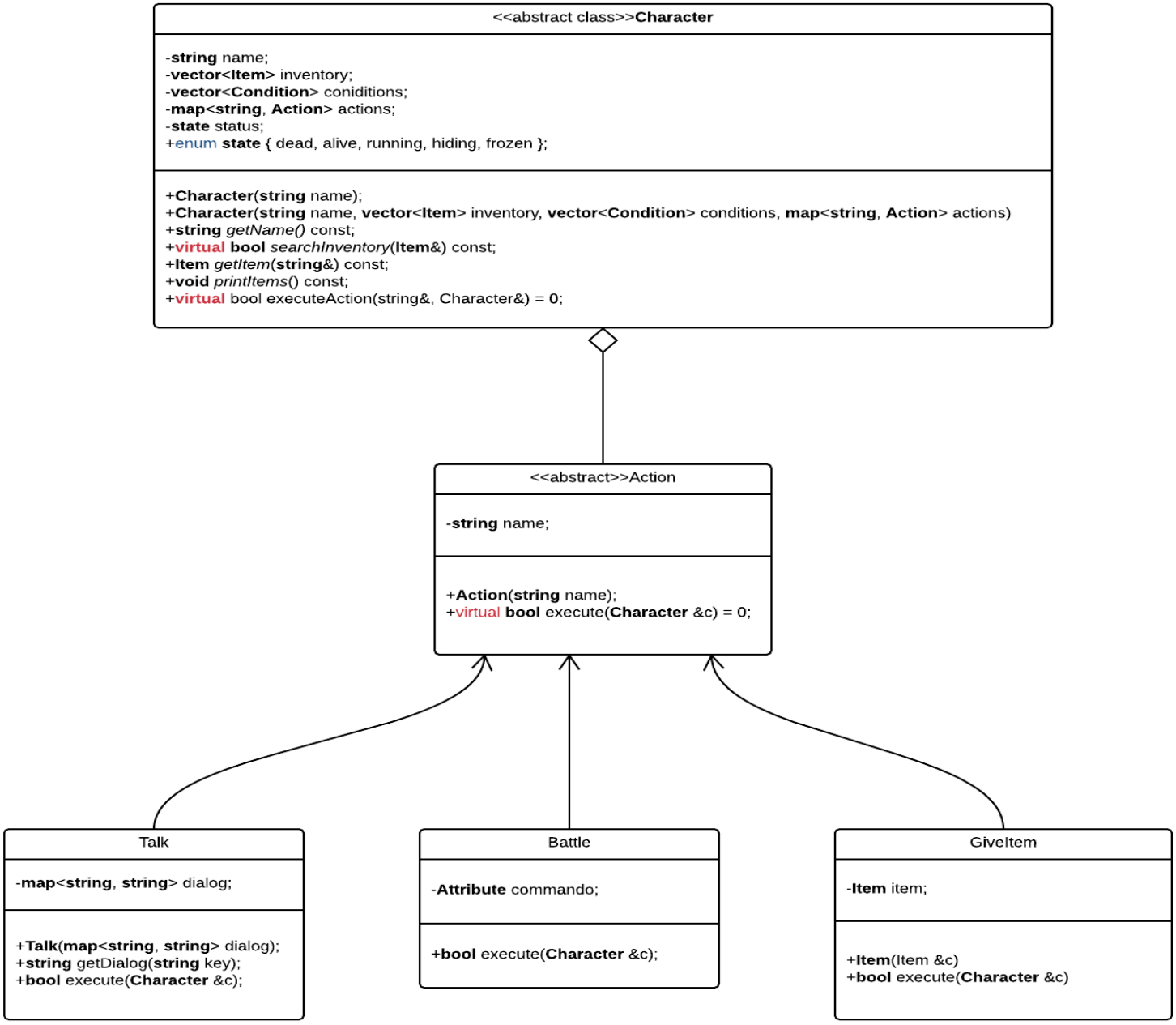
Any group member will be able to issue bug reports that they encounter from the program through bitbucket. This ensures that bugs are not forgotten about and ensures that the team lead is involved in the whole process. When a bug is found, the reviewer who identified the bug(s) will assign them to the developer who wrote the function/class. Only the reviewer who identified the specific bug is allowed to close it. Once the bug has been resolved, the developer is to report back to the reviewer who will test it again until it is agreed that it is fixed or that it is more of a feature than a bug; at which point the issue will be closed through bitbucket.

Software Design

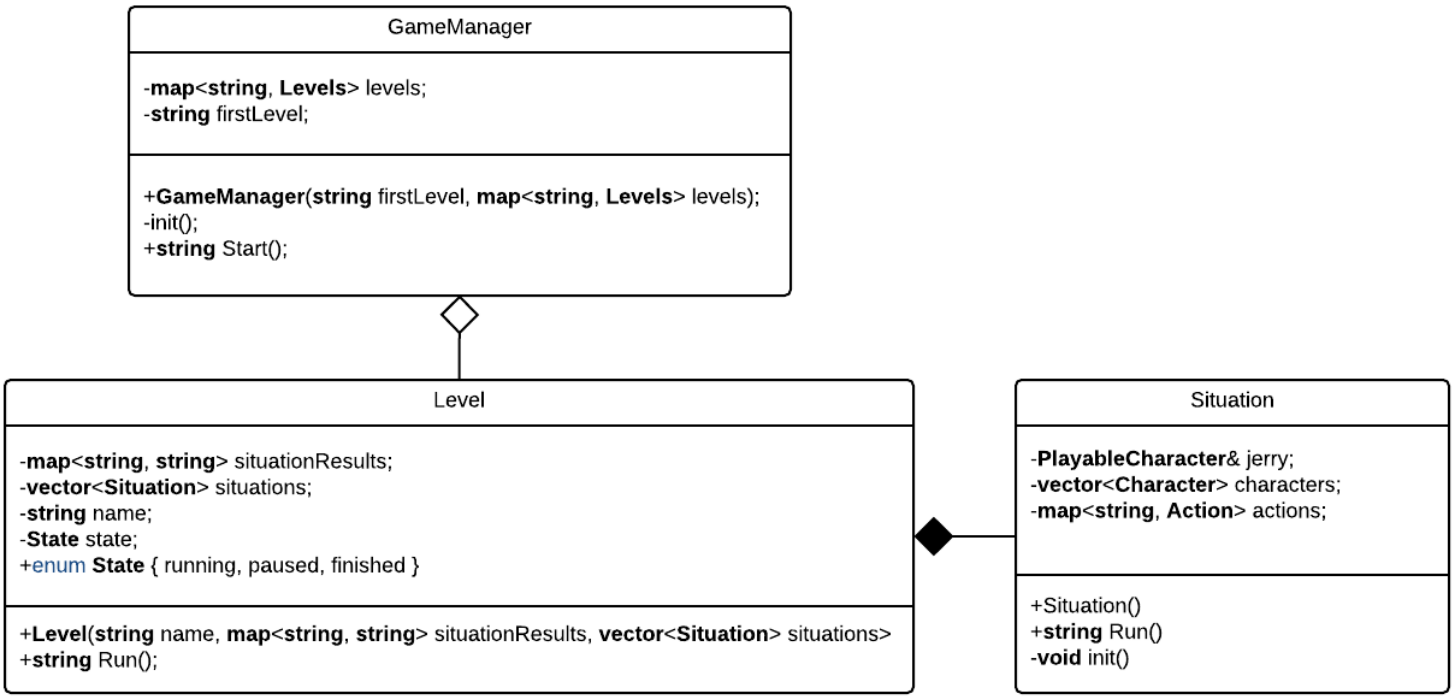
Design

This section includes the details of our game presented as a series of UML class diagrams.

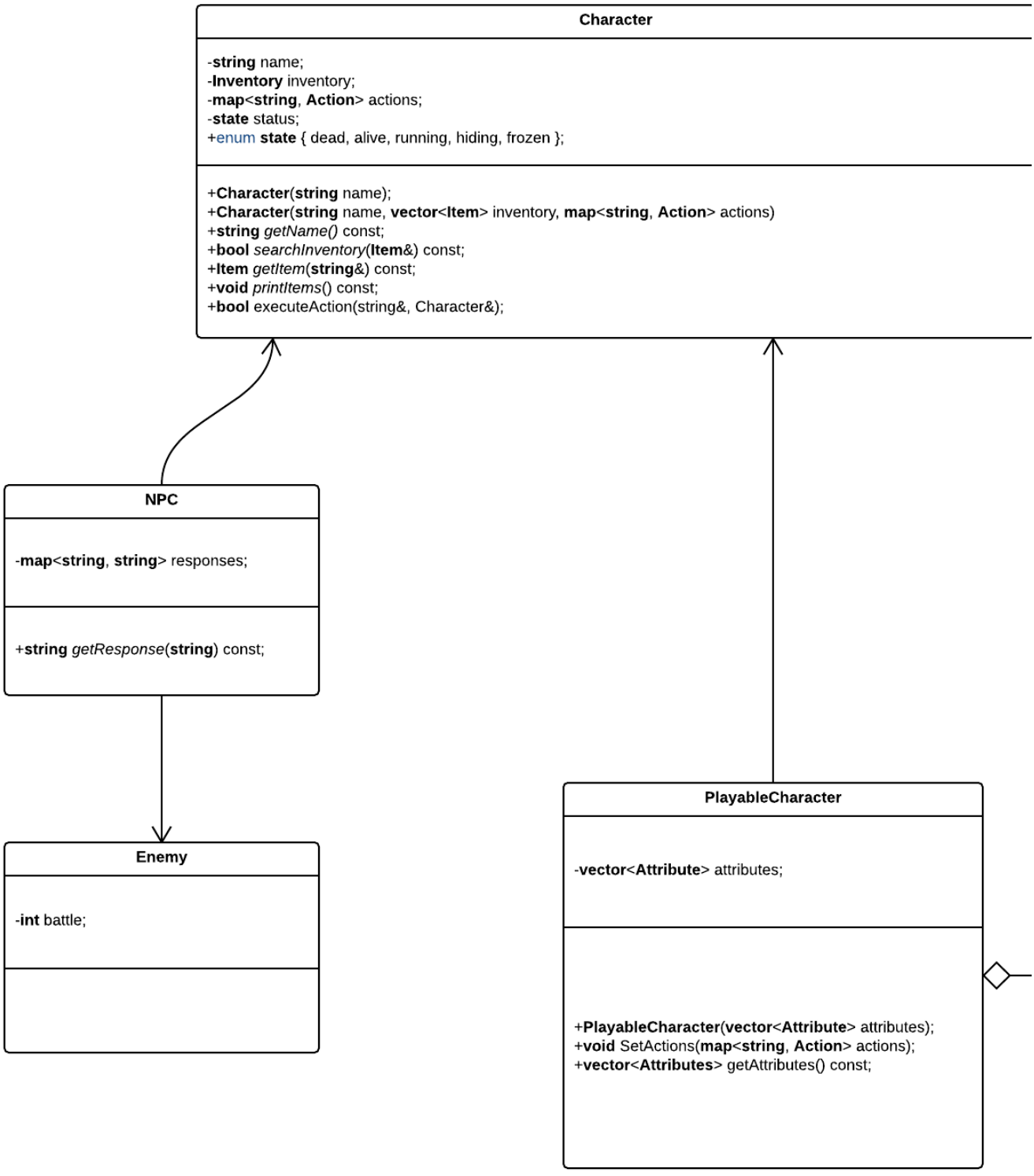
Action UML



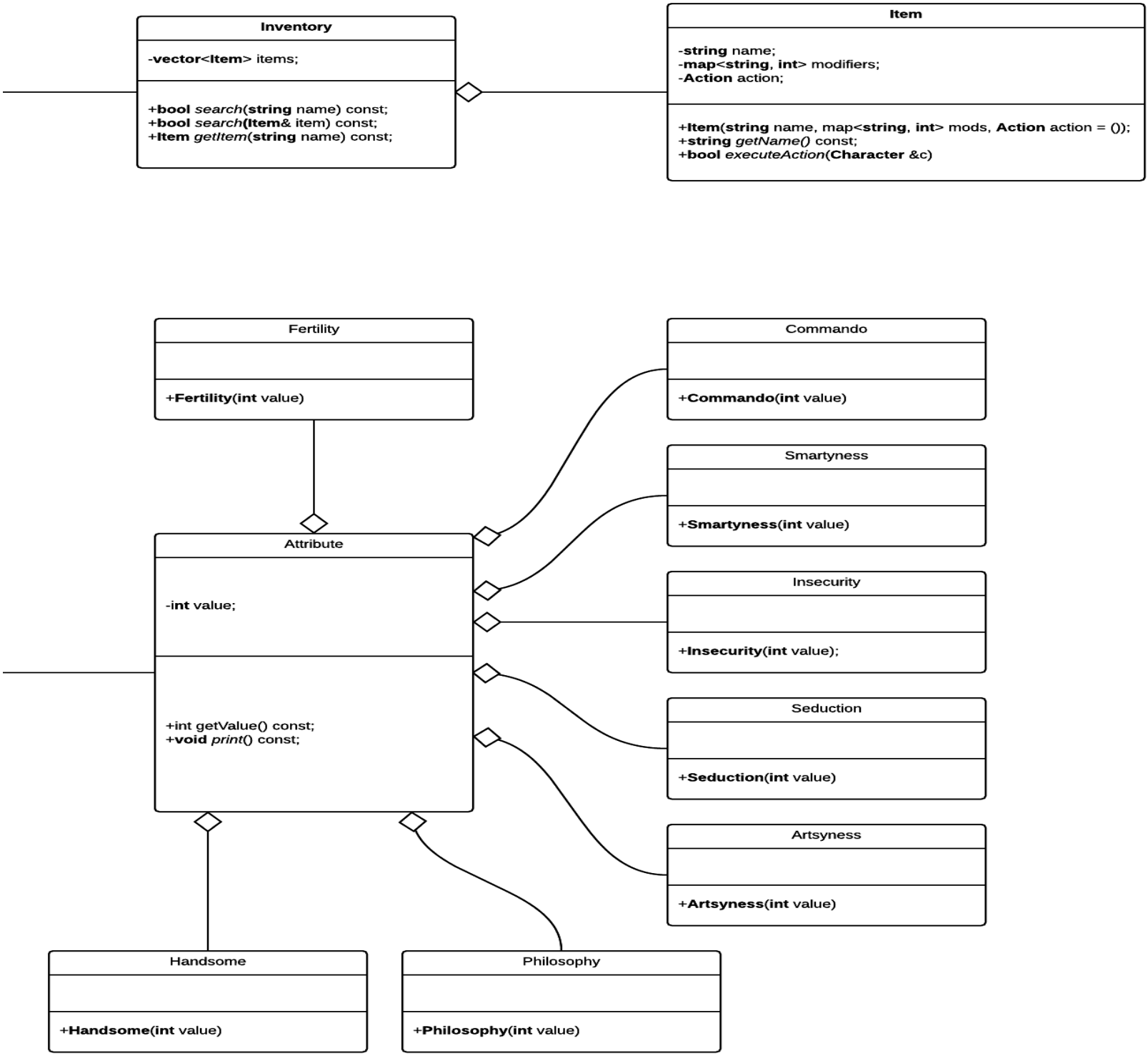
Level UML


GameManager
-map<string, Levels> levels;
-string firstLevel;
+GameManager(string firstLevel, map<string, Levels> levels);
-init();
+string Start();
Level
-map<string, string> situationResults;
-vector<Situation> situations;
-string name;
-State state;
+enum State { running, paused, finished }
+Level(string name, map<string, string> situationResults, vector<Situation> situations>
+string Run();
Situation
-PlayableCharacter& jerry;
-vector<Character> characters;
-map<string, Action> actions;
+Situation()
+string Run()
-void init()

Character UML (LHS)



Character UML (RHS)



Design Rationale

The Council Of Jerry's has designed a game in such a way that we have implemented some sick S.O.L.I.D. principles. If we look at the Character class, we see that it acts as a container for actions and an inventory. Every character has a name and a current status, so those are implemented as well. Looking at the behaviours, the Character simply has accessor methods to return it's current status, as well as functions to access the classes within Character. That is what the Council Of Jerry's has defined as a Character. The Character class has a single responsibility as it just holds the information that a character has with very minimal functionality. It is also open for modification in the sense that it can be derived from and still work as a Character, as well as closed for extension in that the interface is well defined, and easily understood. The class is then further subdivided into two subtypes of Character which have specific functionality, NPC and PlayableCharacter. This is an example of interface segregation, as we have defined two different interfaces which now extend Character's implementation in two very different ways, but still require the information held within Character. The design of Character has also accounted for Liskov substitution, as there is no functionality or restrictions within the sub-types that are not in the base class, meaning we could replace NPC or PlayableCharacter anywhere we have Character, and the functionality would remain the same. The Character class then defines a Inventory class which handles a Character's items within the game. The Inventory class has a singular purpose; to store the items that a Character has. All the functionality for items is stored within the Item class.

Another good example of interface segregation is the way the Jerry's have implemented Attributes. A PlayableCharacter has attributes, but this is represented with its own class, Attribute. From there, we can extend from Attribute and add any singular Attribute that is needed and give any functionality we desire to that Attribute without breaking any code elsewhere in the program.

Lastly, we have Action. Action defines a simplistic interface that includes a pure virtual function, execute. This makes the Action class extensible, while being open for modification, allowing specific actions within the code, such as Talk, to extend from Action and define their own functionality for what the method execute means.