Software Architecture Document

Adventure Game Engine
Build 3

(Features and Constraints)

Prepared by: *2393336 Chung Mak*

6176526 Pei Yu

9092994 Hongxian Gao

6137687 Yan Wang

Team: *Team 6*

Date: *December 8, 2009*

Table of Contents

Revision History

Name	**Date**	**Reason For Changes**	**Version**
Team6	*2009-11-26*	*Architectural Baseline*	*1.0*
Team6	*2009-11-30*	*Design pattern*	*1.1*
Team6	2009-12-08	Final Document	2.0

1. Introduction

1.1 Purpose

This Software Architecture Document serves the purpose of describing the design and architecture of the Adventure Game Engine. It is aim to help the game designer understand how to create a new Adventure Game using this game engine and it is aim to help game player to run the adventure game.

1.2 Project Scope and Product Features

As a software system, the Game Engine is designed to create and develop games. In the project, our objective is to design and implement a Text Adventure Game Engine.

In Build Three, game engine is extended to allow game player to specify puzzles for the player to solve. This included adding constrains to movement, as well as constraints for whether a given object may be picked up. And some additional commands will need to be provided for game designer's use.

1.3 Definitions, Acronyms and Abbreviations

Adventure games: Adventure games (or *adventures*), part of the *Interactive Fiction* (IF) genre, are games in which a story is told as part of the game. The player's actions have direct impact on the game and its outcome. Actions often include puzzles, seeking treasure and interacting with in-game characters, usually referred to as *Non-Player-Characters* (NPCs).

Text-based Adventure games: A text-based Adventure game is designed for the player by typing text input to control the game, and the game state being relayed to the player via text output.

Game Engine Designer: The people who design and implement the game engine.

Game Designer: The people who design and implement specific games, using the game engine which you provide.

Game player: The people who actually play the games.

Actions/Command: A text-based Adventure game is designed for the player by typing text input to control the game, each command contains a verb**,** and many also contain one or more nouns. For example, north, take key, drop cage and turn lamp on.

M.V.C.: [4] Model–View–Controller (MVC) is an architectural pattern used in software engineering. The pattern isolates business logic from input and presentation, permitting independent development, testing and maintenance of each.

Strategy Pattern: Indentify the aspects of the application which are likely to vary, and separate them from those which will stay the same.

Observer pattern: The observer pattern (a subset of the asynchronous publish/subscribe pattern) is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. It is mainly used to implement distributed event handling systems.

1.4 References

[1] The Eclipse Development API document.

[2] http://www.rickadams.org/adventure/

[3] http://en.wikipedia.org/wiki/Interactive_fiction

[4] http://en.wikipedia.org/wiki/Model-view-controller

[5] http://en.wikipedia.org/wiki/Observer_pattern

2. Analysis Model

2.1 Assumptions

A-01. Related software:

The document assume that the project will be executed on the Eclipse IDE of version above 3.2.2

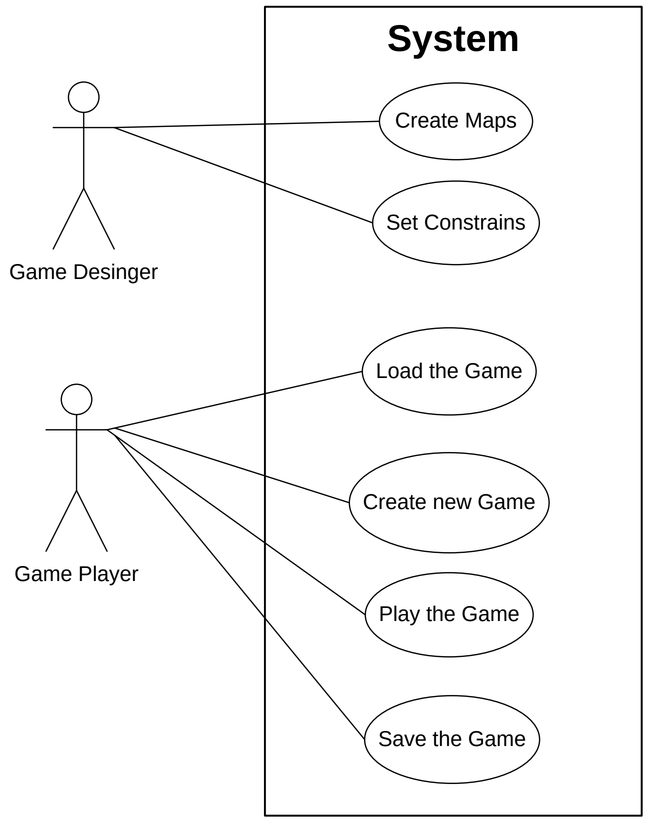
2.2 Analysis Model

2.2.1 User Case

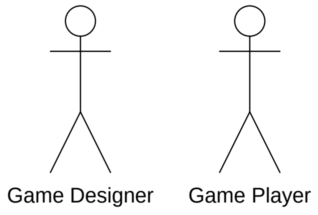
The following sections illustrate how the Game Engine actually works by describing the scenarios and using Use Case Diagram to explain how each element contribute in the system.

2.2.2 User Case Diagram

The following Use Case diagram provides functional Use Cases of the overall Game Engine. It shows us a macro view of the system.



2.2.3 Actor



2.2.4 User Cases Scenarios

(1) Use-case UC1: Create Maps

Primary Actor: Game Designer

Stakeholder: Game Player

Precondition: Map Editor Tool

Post Condition: A map created

Main Success Scenario:

- Start the Map Editor Tool
- Click the menu item “Create Map”

- Click the menu item “Add Room” to add Rooms
- Click the menu item “Add Connection” to add Connections
- Click the menu item “Add item” to add Items for the room
- Click the menu item “Add Character” to add Items for the room
- Click the menu item “Remove" to remove selected Rooms and selected Connections

(2) Use-case UC1: Set Constrains

Primary Actor: Game Designer

Stakeholder: Game Player

Precondition: Map Editor Tool

Post Condition: A map created

Main Success Scenario:

- Start the Map Editor Tool
- Click the menu item “set probability” to set probability for selected connection
- Click the menu item “set door” to set door for selected connection
- Click the menu item “set specific item” to set specific item for selected connection
- Click the menu item “set specific room” to set specific room for selected connection
- Click the menu item “set magic word” to set magic word for selected connection
- Click the menu item “set probability” to set probability for selected room
- Click the menu item “set specific item” to set specific item for selected item
- Click the menu item “set specific room” to set specific room for selected item
- Click the menu item “set magic word” to set magic word for selected item

(3) Use-case UC2: Load the Game

Primary Actor: Game Designer

Stakeholder: Game Player

Precondition: a Command Line Interface and a file stored a Game data

Post Condition: a Game data read from a file and show the Game information on Command Line Interface

Main Success Scenario:

- Start the command line interface
- Input “load” command on the command line interface to load the Game file

(4) Use-case UC3: Create a New Game

Primary Actor: Game Player

Stakeholder: Game Player

Precondition: Command Line Interface

Post Condition: a new Game is created

Main Success Scenario:

- Start the Command Line Interface
- Input “new” command on the command line interface to create new game

(5) Use-case UC4: Play the Game

Primary Actor: Game Player

Stakeholder: Game Player

Precondition: Command Line Interface and a game file is loaded

Post Condition:

Main Success Scenario:

- Start the Command Line Interface
- Input “north” command on the command line interface to make the player go north
- Input “south” command on the command line interface to make the player go south
- Input “east” command on the command line interface to make the player go east
- Input “west” command on the command line interface to make the player go west
- Input “northeast” command on the command line interface to make the player go northeast
- Input “northwest” command on the command line interface to make the player go northwest
- Input “southeast” command on the command line interface to make the player go southeast
- Input “southwest” command on the command line interface to make the player go southwest
- Input “up” command on the command line interface to make the player go up
- Input “take item” command on the command line interface to pick up the named item
- Input “pick up item” command on the command line interface to pick up the named item
- Input “drop item” command on the command line interface to drop item
- Input “put down item” command on the command line interface to put down item
- Input “inventory” command on the command line interface to display the list of items which the player is currently carrying
- Input “score” command on the command line interface to display the player’s game score
- Input “fire item” command on the command line interface to fire item to selected character
- Input “throw item” command on the command line interface to throw item to selected character
- Input “attack character” command on the command line interface to attack selected character
- Input “eat item” command on the command line interface to eat selected character
- Input “drink item” command on the command line interface to drink selected character
- Input “lock item” command on the command line interface to fire selected item
- Input “unlock item” command on the command line interface to fire selected item

(6) Use-case UC5: Save the Game

Primary Actor: Game Player

Stakeholder: Game Player

Precondition: Command Line Interface and a running Game file

Post Condition: the Game file is saved

Main Success Scenario:

- Start the Command Line Interface
- Input "save" command on the command line interface to save the game

3. Architectural Design

3.1 System Architecture

The Model View Controller pattern isolates data logic from input and rendering. This "Separation of Concerns" allows each layer to be developed, tested and maintained independently. The Model view Controller pattern is also suitable for our Build Two.

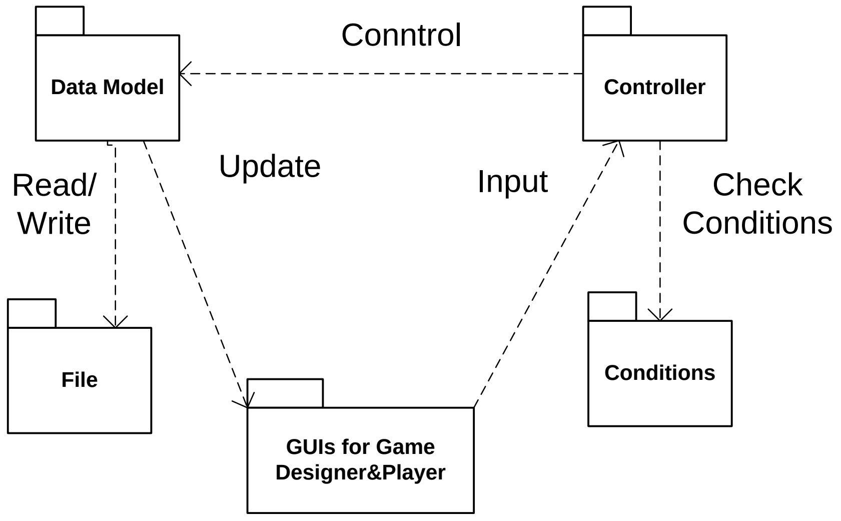
In Build Three, certain constrains are added. Consequently, game designer may add constrains to create complete world map by using graphic user interface.

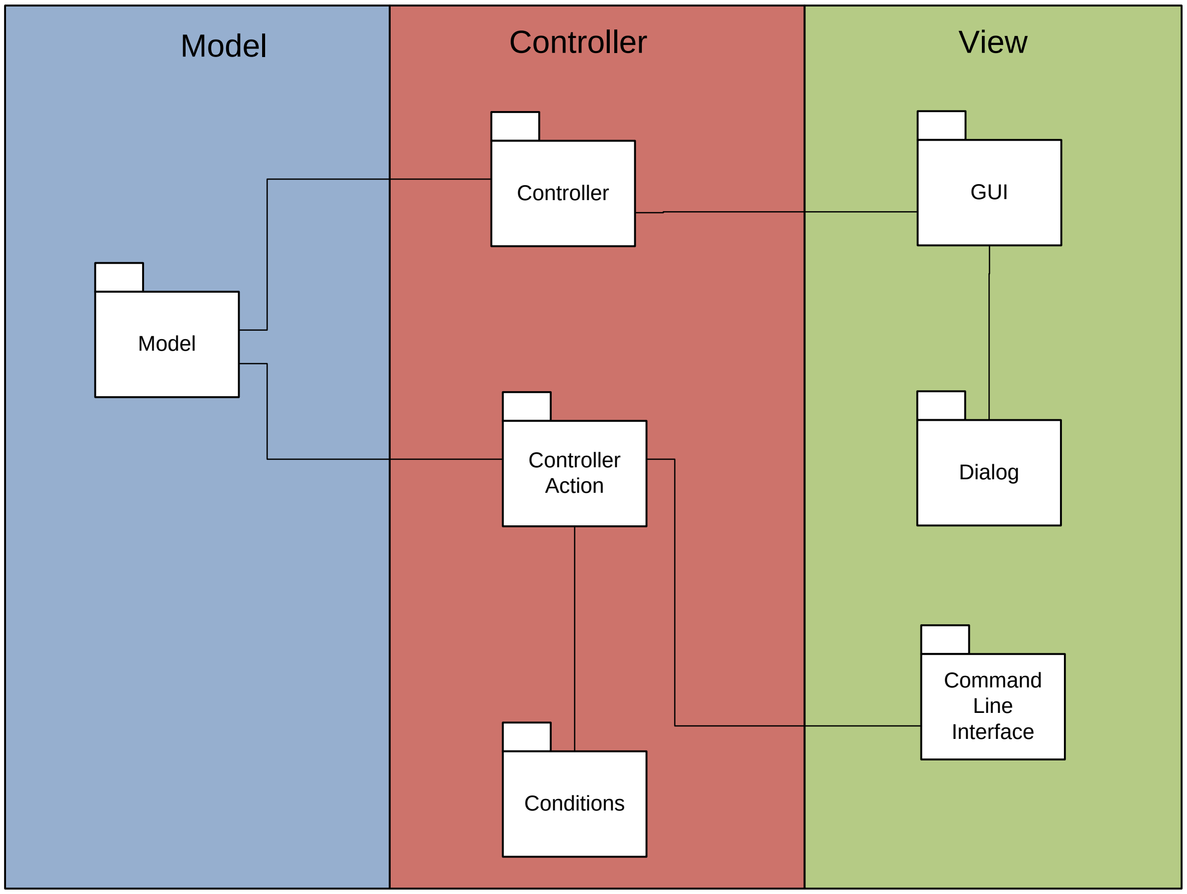
In Build Three, main focus is on playing games with some constrians. The command line interface works solely on getting game player's command and parsing the command syntax in order to notify controller.
Controller invokes functions according to the command syntax and changes the model. Then the controller passes the model to the command line interface to be rendered.

Model includes data such as room, item, connection, map, character and player.

Conditions package include all these constrain of the map.

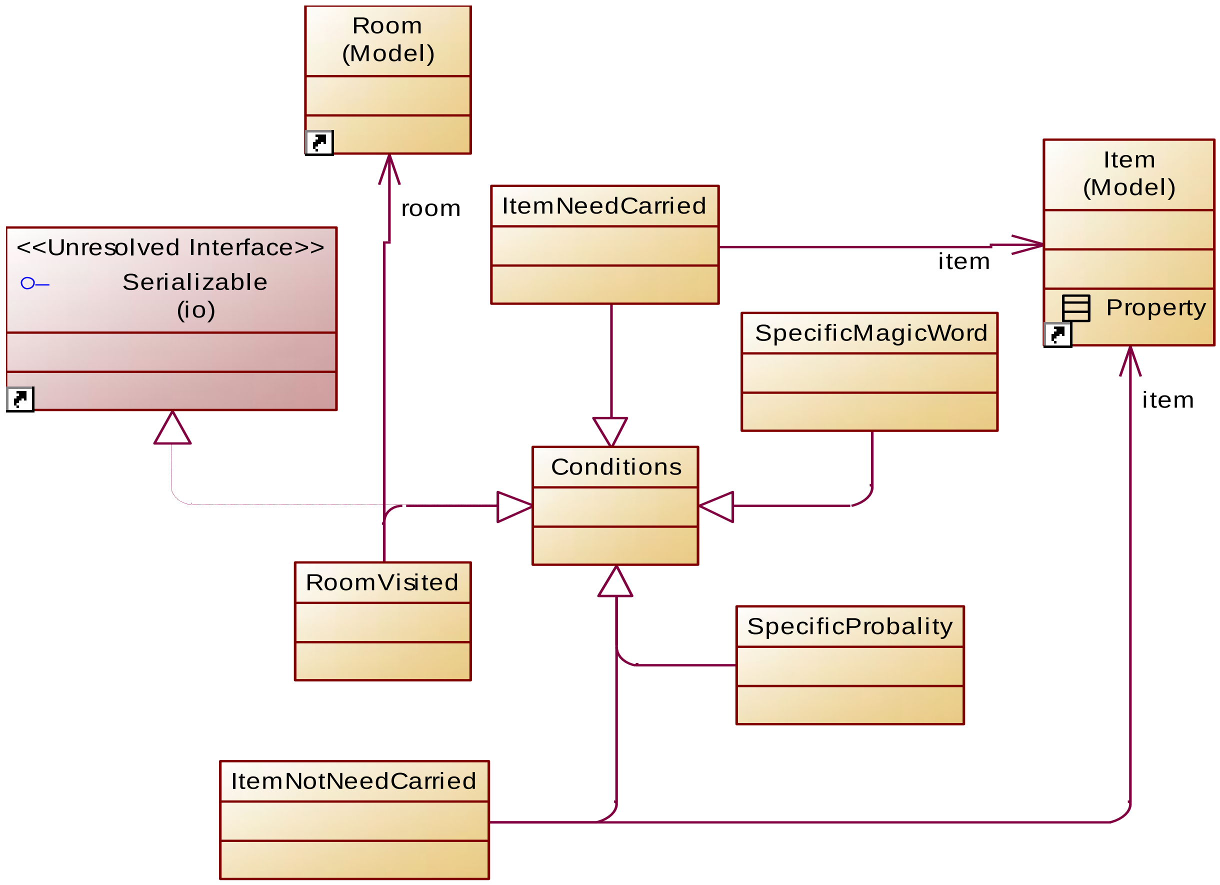
In the java implementation, each of the sub-system will be wrapped by a package. The following diagrams show the relationship between each sub-system.


Conntrol
Data Model
Controller
Update
Input
Read/ Write
Check Conditions
File
GUIs for Game Designer&Player
Conditions

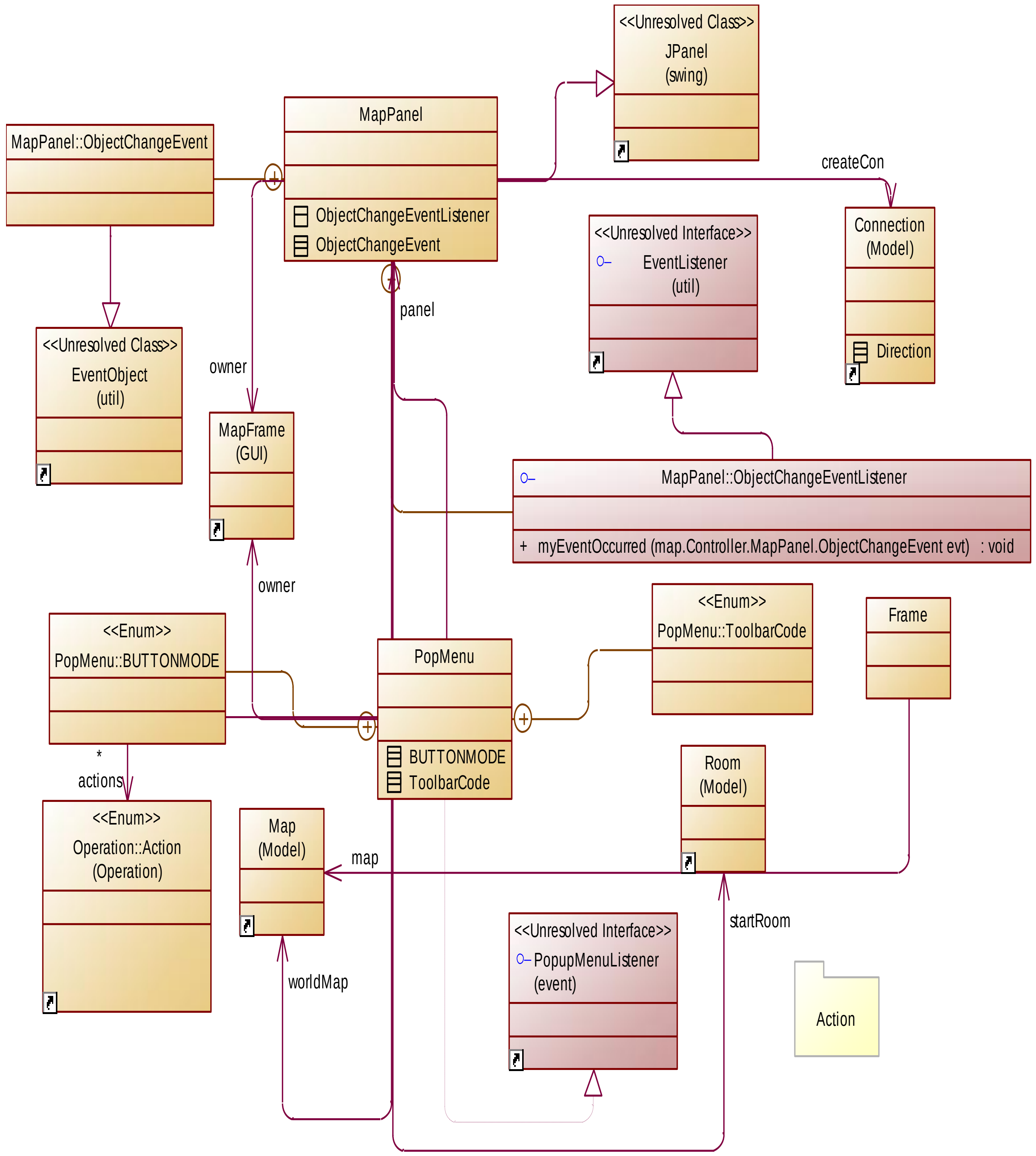

Model
Controller
View
Controller
GUI
Model
Controller Action
Dialog
Command Line Interface
Conditions

3.1.1 UML Class Diagram

3.1.1.1 UML Class Diagram for Conditions Package

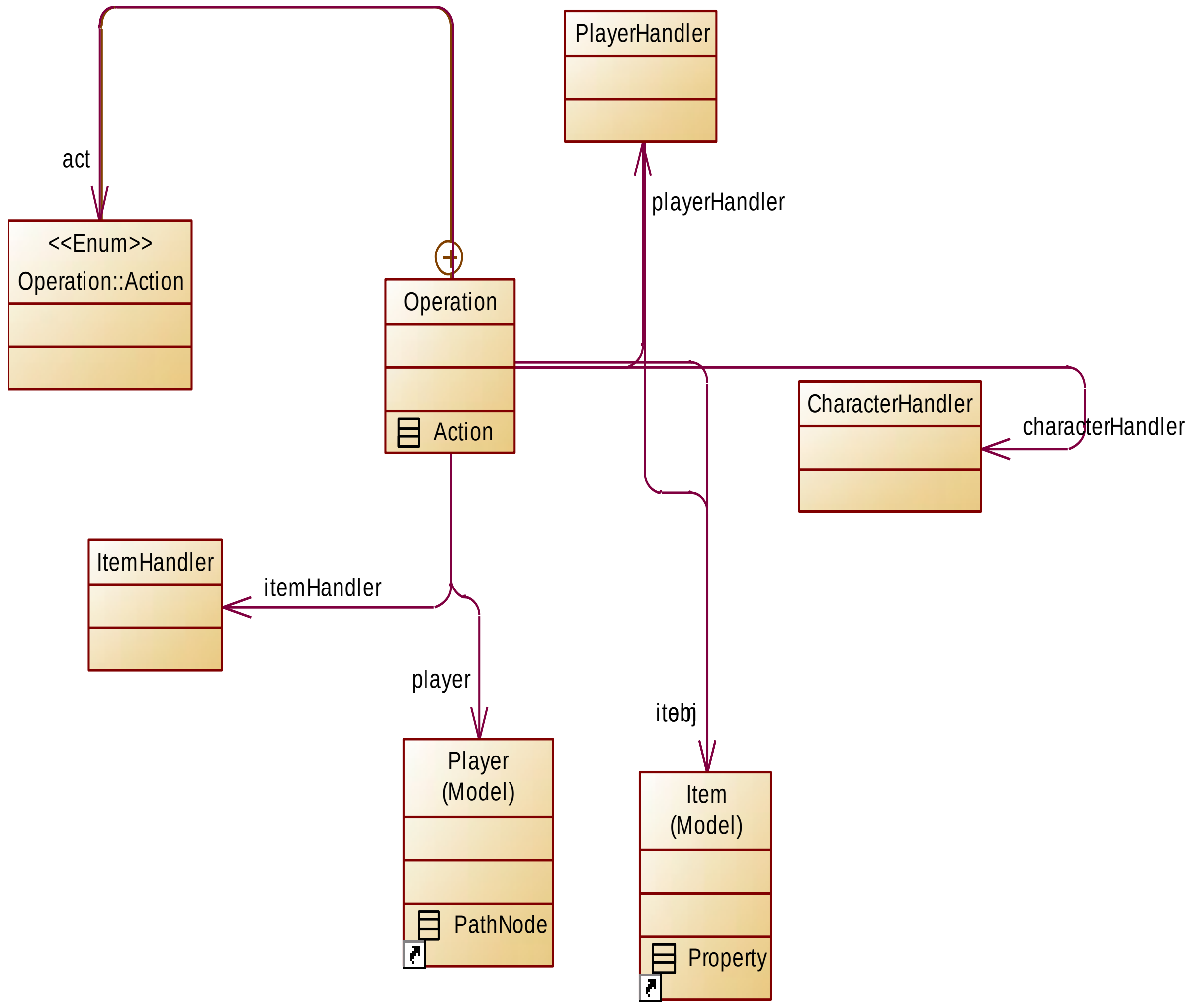


3.1.1.2 UML Class Diagram for Controller Package

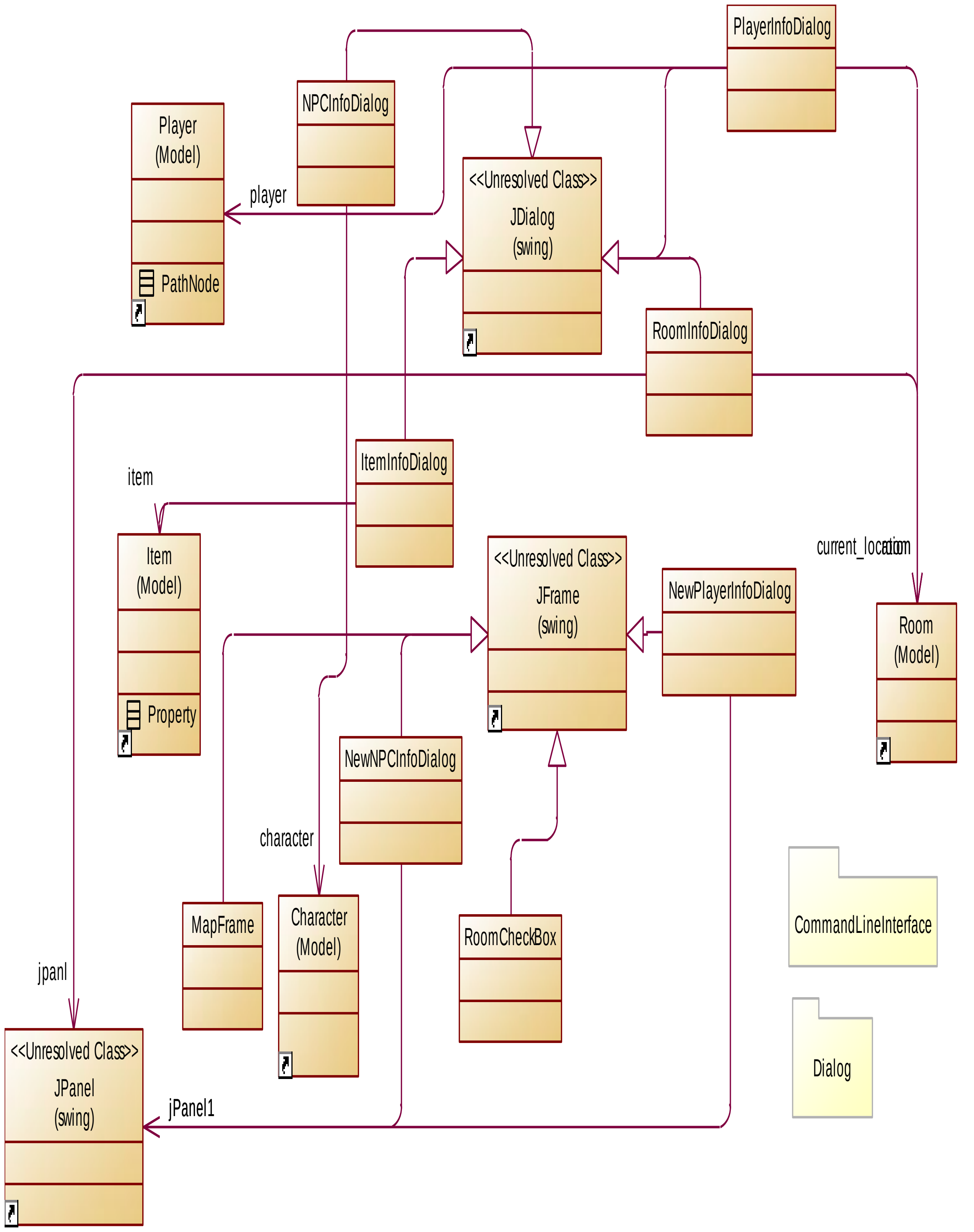


<<Unresolved Class>> JPanel (swing)
MapPanel
ObjectChangeEventListener
ObjectChangeEvent
MapPanel::ObjectChangeEvent
createCon
Connection (Model)
Direction
<<Unresolved Interface>> EventListener (util)
<<Unresolved Class>> EventObject (util)
owner
panel
MapFrame (GUI)
MapPanel::ObjectChangeEventListener
+ myEventOccurred (map.Controller.MapPanel.ObjectChangeEvent evt) : void
owner
<<Enum>> PopMenu::BUTTONMODE
PopMenu
BUTTONMODE
ToolbarCode
<<Enum>> PopMenu::ToolbarCode
Frame
*
actions
<<Enum>> Operation::Action (Operation)
Map (Model)
map
Room (Model)
startRoom
worldMap
<<Unresolved Interface>> PopupMenuListener (event)
Action

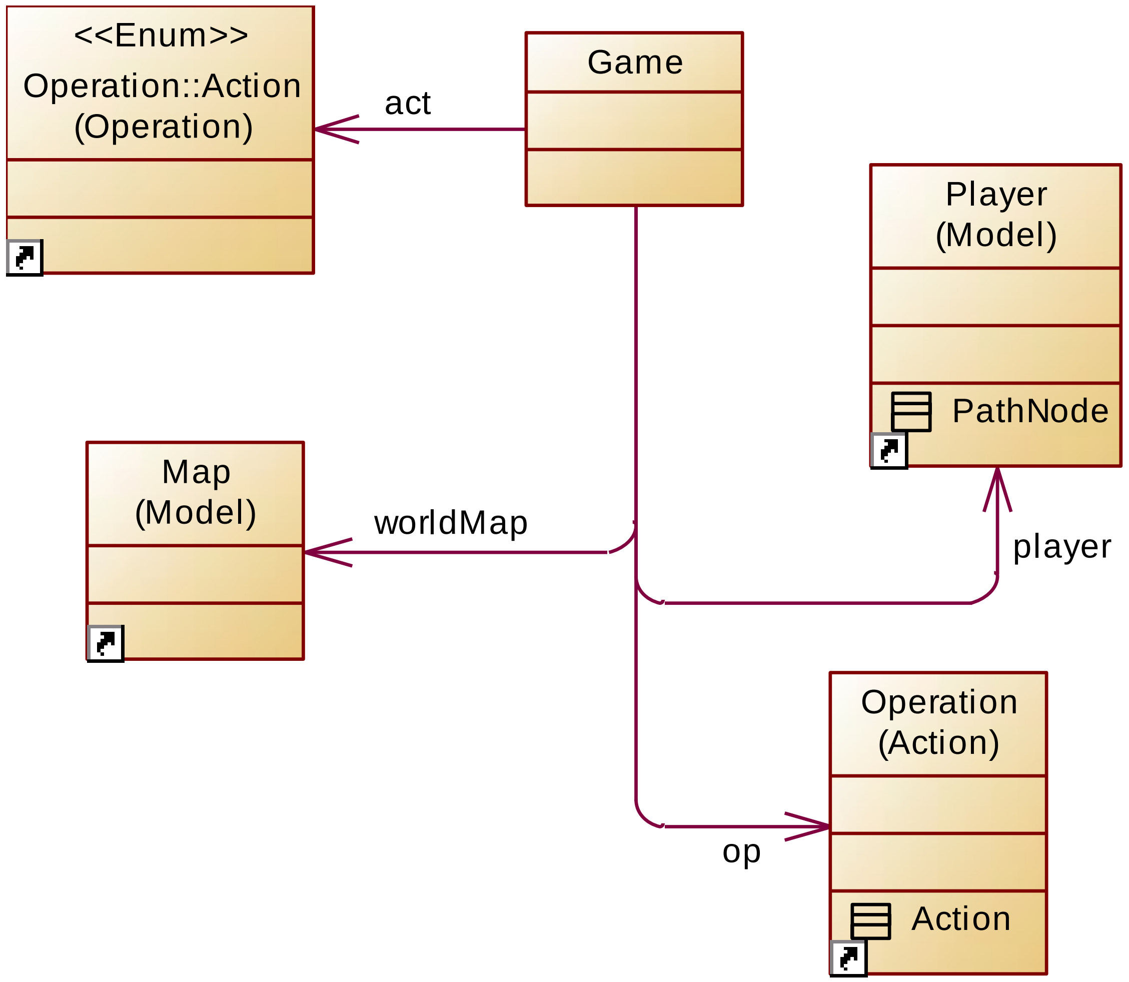
3.1.1.3 UML Class Diagram for Controller Action Package



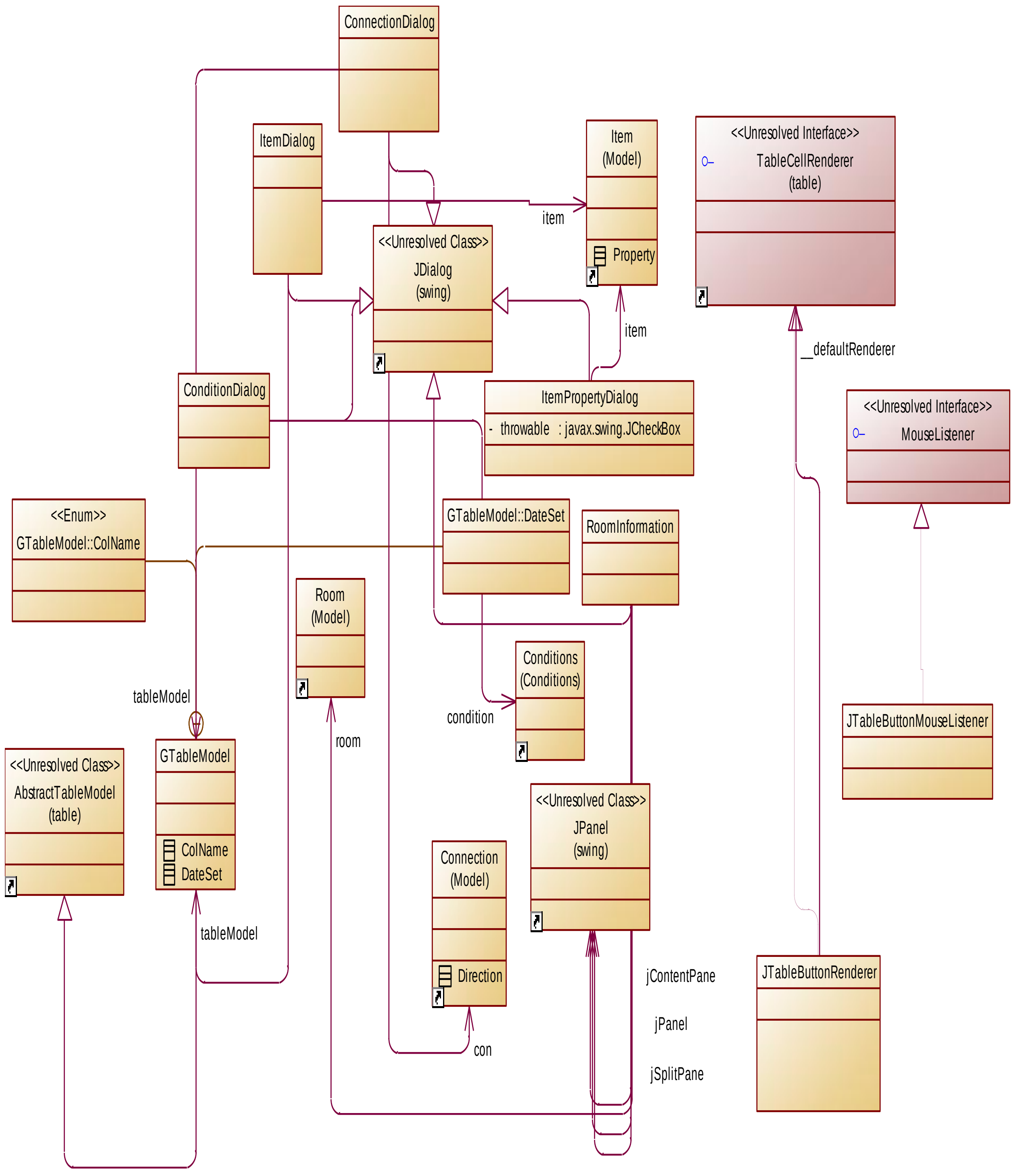
3.1.1.4 UML Class Diagram for GUI Package



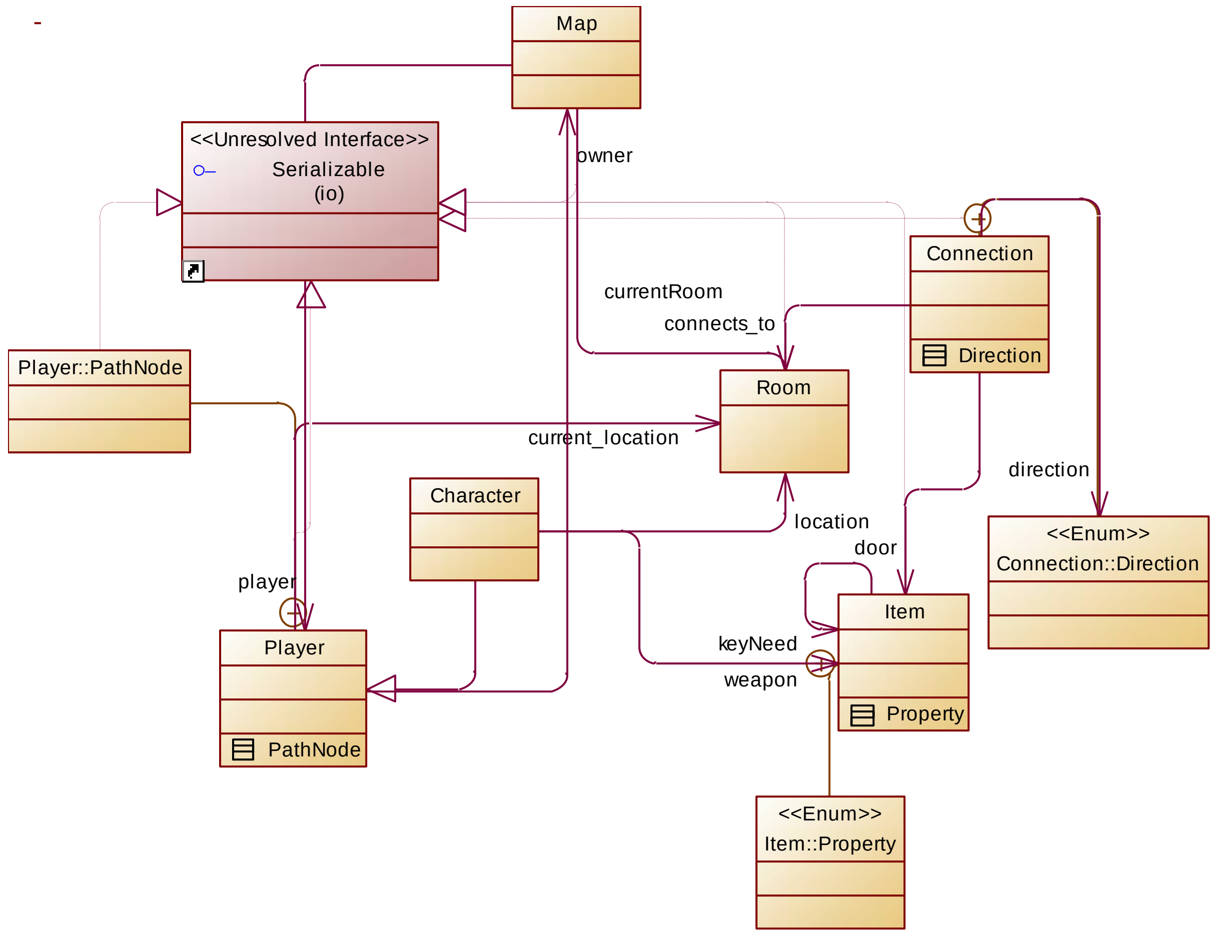
3.1.1.5 UML Class Diagram for GUI CommandLineInterface Package



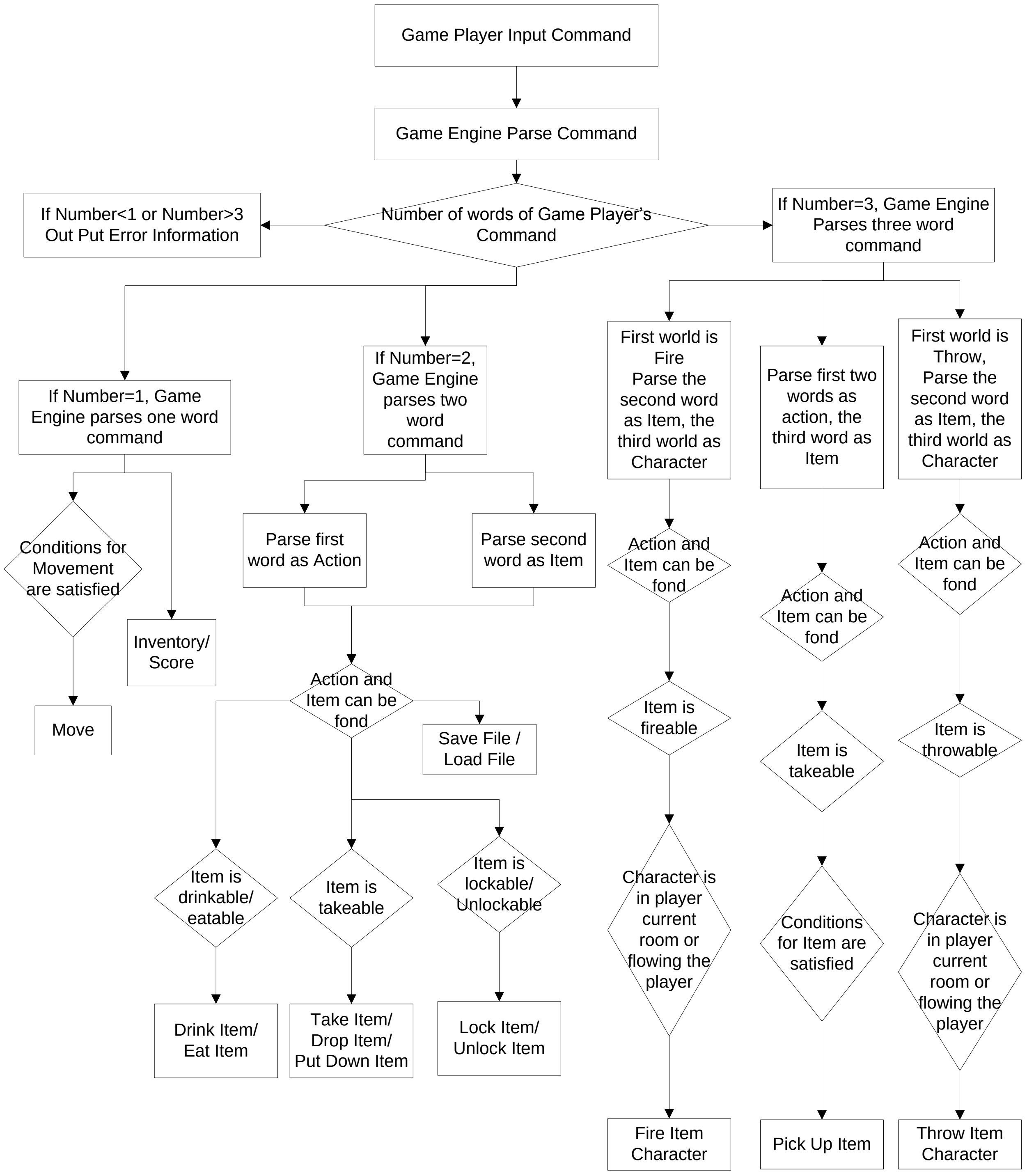
3.1.1.6 UML Class Diagram for GUI Dialog Package



3.1.1.7 UML Class Diagram for Model Package



3.2 Work Flow Diagram for the Game Player


Game Player Input Command
Game Engine Parse Command
Number of words of Game Player's Command
If Number<1 or Number>3 Out Put Error Information
If Number=3, Game Engine Parses three word command
If Number=1, Game Engine parses one word command
If Number=2, Game Engine parses two word command
First world is Fire Parse the second word as Item, the third world as Character
Parse first two words as action, the third word as Item
First world is Throw, Parse the second word as Item, the third world as Character
Conditions for Movement are satisfied
Inventory/ Score
Move
Parse first word as Action
Parse second word as Item
Action and Item can be fond
Save File / Load File
Item is drinkable/ eatable
Item is takeable
Item is lockable/ Unlockable
Drink Item/ Eat Item
Take Item/ Drop Item/ Put Down Item
Lock Item/ Unlock Item
Action and Item can be fond
Item is fireable
Character is in player current room or flowing the player
Fire Item Character
Action and Item can be fond
Item is takeable
Conditions for Item are satisfied
Pick Up Item
Action and Item can be fond
Item is throwable
Character is in player current room or flowing the player
Throw Item Character

3.3 Controller Subsystem Architecture

3.3.1 Purpose

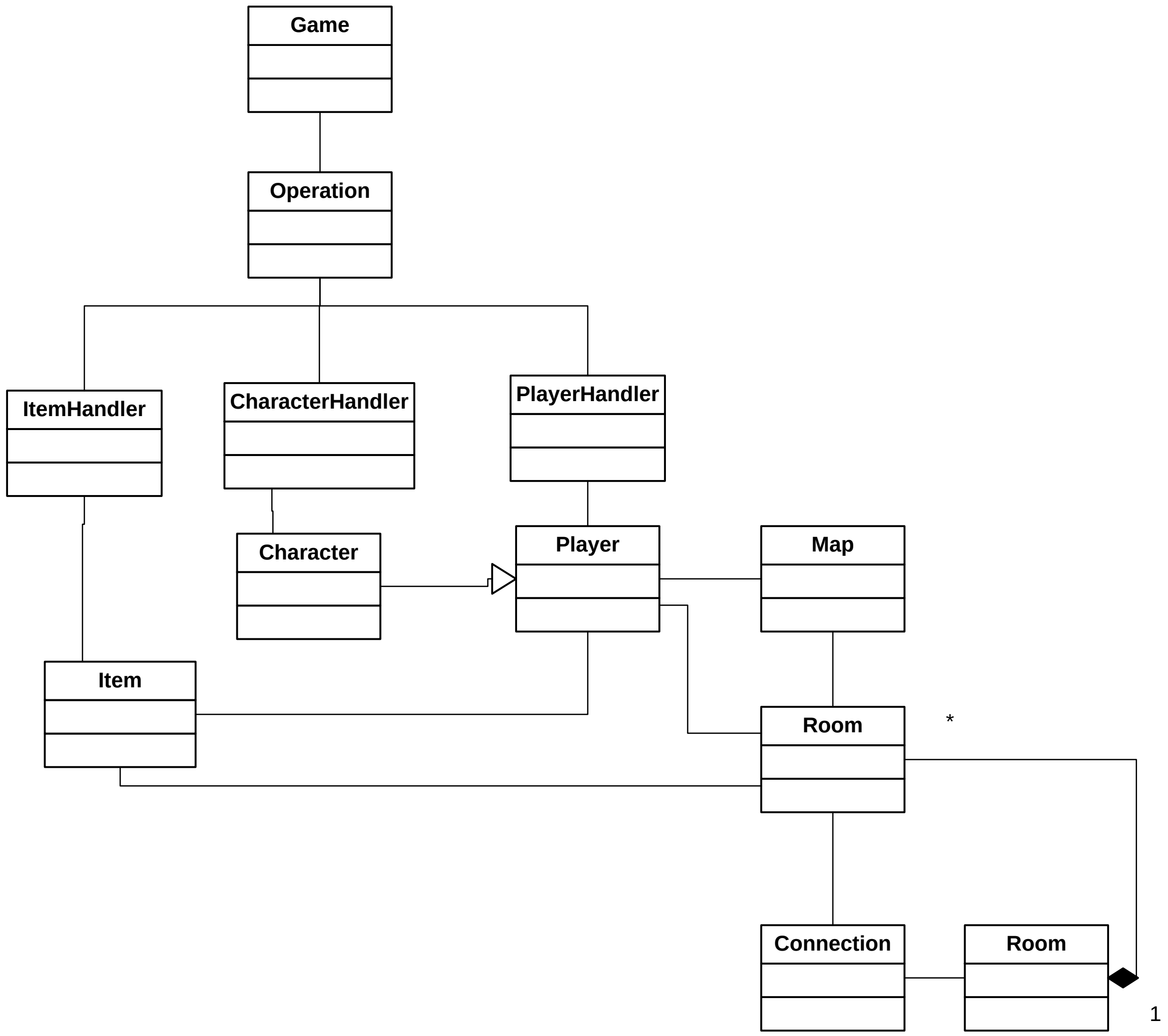
The main function of the sub-system controller is to invoke functions according to the player user's commands getting from command line interface. This is the main subsystem that handles the adventure game playing successfully. Consequently, after each function invoked, the controller passes the update information to command line interface.

3.3.2 Subsystem Class Structure

Considering extends and maintains of the system, the strategy pattern is a suitable. Operation class includes all the operations which are must be implemented in order to make the adventure game work successfully. If new functions are added in the next build, all we need to do is just updating this operation class independently. Indentify the aspects of the application which are likely to vary, and separate them from those which will stay the same.

The second pattern should be considered is observer pattern. Depending on the different subjects of the different operation, two operation handlers are developed as ItemHander and PlayerHander. In playerHander class, it implements these operations such as move function (go north, go south..Etc), inventory function to show the inventory of the player, score function to show the score of the player gotten, loading function of the game, saving function of the game. In ItemHander class, it implements taking item function and dropping item function. These two classes subscribed to the operation class. Once the functions in operation class are invoked by game player, the corresponding concrete function will be implemented by these two classes.

The subsystem class diagram:



3.3.3 Data Persistence

Object serialization is the process of saving an object's state to a sequence of bytes, as well as the process of rebuilding those bytes into a live object at some future time. The Java Serialization API provides a standard mechanism for developers to handle object serialization.

3.4 Data Model Subsystem Architecture

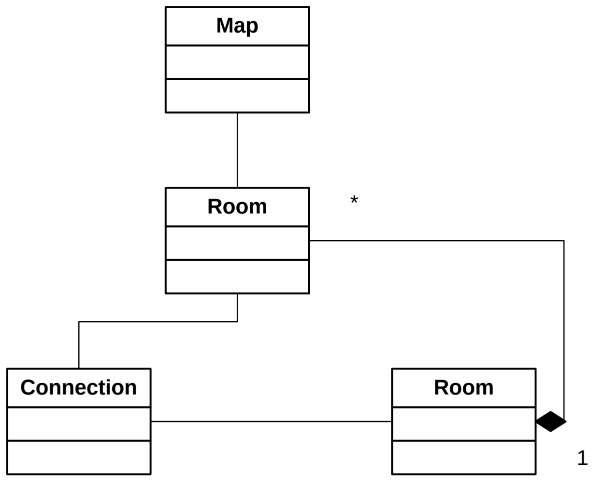
3.4.1 Purpose

The sub-system servers as the Map data structure to handle it for our project. It mainly contains the LoadMap that parses the file and converts it to the suitable data structure for UI displaying, and the SaveMap that accepts our data structure and saves it to the file system.

3.4.2 Subsystem Class Structure

Composite pattern is a good way to represent the map data structure. One map class instance may include one or many room class instances. And one room class instance can has none or many connections and also can connect to other room class instances.

The Class Diagram is as follow:


Map
Room
*
Connection
Room
1