Team Automaton



Requirements Specification

Foundations of Software Engineering (EN.605.401.71. FA14)

Authors: Henoke Shiferaw, Shambhavi Shankrit, Ethan Wilansky

Table of Contents

[Introduction 3](#_Toc400444002)

[Information Domain 3](#_Toc400444003)

[Functional Requirements 3](#_Toc400444004)

[Diagrams 5](#_Toc400444005)

[Non Functional Requirements 6](#_Toc400444006)

[Usability 6](#_Toc400444007)

[Performance 6](#_Toc400444008)

[Testability 6](#_Toc400444009)

[Supportability 7](#_Toc400444010)

[Interoperability 7](#_Toc400444011)

[Implementation Constraints 7](#_Toc400444012)

[Communication 7](#_Toc400444013)

[Design constraints 7](#_Toc400444014)

[Data structures 7](#_Toc400444015)

[Databases/Repositories 7](#_Toc400444016)

[Programming language and style guidelines 7](#_Toc400444017)

[Implementation tools 7](#_Toc400444018)

[Software subsystems 7](#_Toc400444019)

# Introduction

The following document outlines the Requirements Specification necessary for the Computer player subsystem of the Clue-Less game. Outlined below is a brief summary of the Information domain, functional requirements written through user stories, non functional requirements, and implementation constraints. The document also include diagrams outlining actions during a Computer player’s turn and classes to be implemented in the Clue-Less game.

## Information Domain

The computer subsystem encapsulates all the capabilities necessary to interoperate autonomously in the game of Clue-Less. Information to be stored in this subsystem includes:

* The selected player (string)
* Current position of all the other players (dictionary)
* Cards that have been dealt (list)
* The cards that have been suggested and revealed (matrix - row: card, column: player)
* Player: location (string)

## Functional Requirements

These requirements follow the Scrum Agile Methodology User Story format: *As a <type of user/entity>, I ant <some goal> so that <some reason>*[[1]](#endnote-1)

As a computer player, I want enter a game so that the user can play against it.

As a client application, I want to be able to add multiple computer players so that the user’s number of requested players can be fulfilled.

As a computer player, I want to select a player so that the computer player has a token to move around with and can interact with other players.

As a computer player, I want to know who’s playing so that the players and possible suspects can be introduced.

As a computer player, I want to know which players turn it is so that the player can tell when it’s their turn and when it isn’t.

As a computer player, I want make a move so that I can move to different rooms, or hallways.

As a computer player, I want to make suggestions so that I can narrow down the list of suspects, weapons, and rooms to make an accusation.

As a computer player, I want to make an accusation to win the game or get eliminated.

As a computer player, I want to reveal a card to someone who made a suggestion so that I can interact with the player who made the suggestion.

As a computer player, I want to be able to maintain player cards to make better suggestions and move forward to making a correct accusation.

As a computer player, I want to hear suggestions made so that I can possibly eliminate options.

As a computer player, I want to hear an accusation so that I can know whether the player who made the accusation won the game or is eliminated.

As a computer player, I want to end my turn so that the game can be facilitated to the next person’s turn.

As a computer player, I want to start my turn so that I can make suggestions, accusations and move.

As a computer player, I want to store whatever move so that I can make a different move in case the initial move is invalid.

As a computer player, I want to respond to suggestions even after I make a false accusation so I can allow the game to continue and participate.

Information to be stored in memory include, player locations, player cards, current moves.

## Diagrams

The following UML diagrams show various aspects of the Automaton subsystem. The first diagram, also included in the Interface Specification Document, is a high-level view of the interaction between the Automaton subsystem and the other subsystems in this solution. The second diagram shows one of the more complex sequences that must be coded, making a move. The third diagram shows data flow from the perspective of an autonomous player, from the start of a new game to losing or winning the game.

Figure . Skeletal Class Diagram Showing Subsystem Interactions

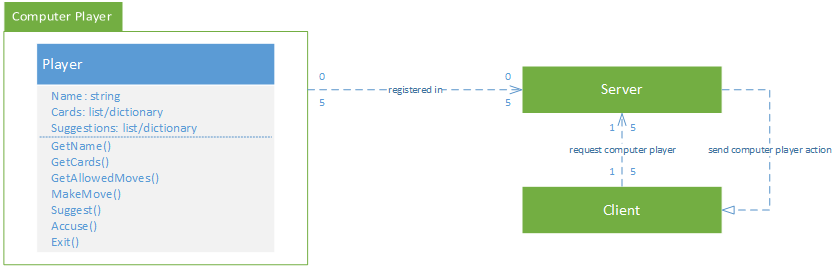


Figure . Autonomous Player Move Sequence Diagram

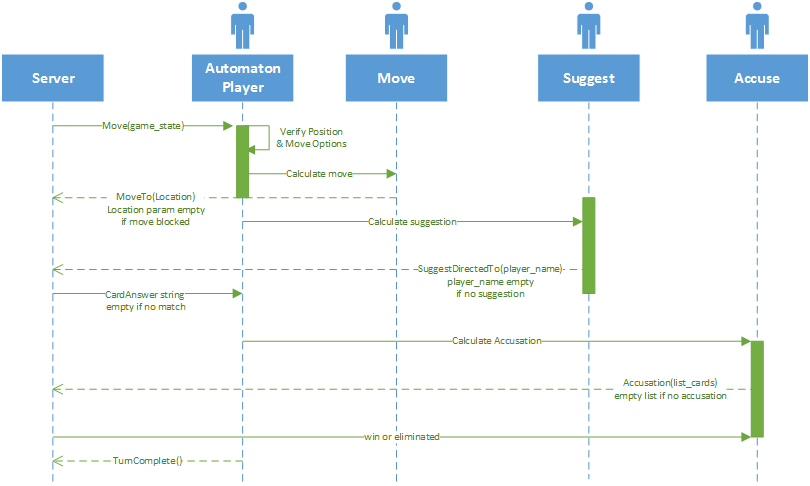
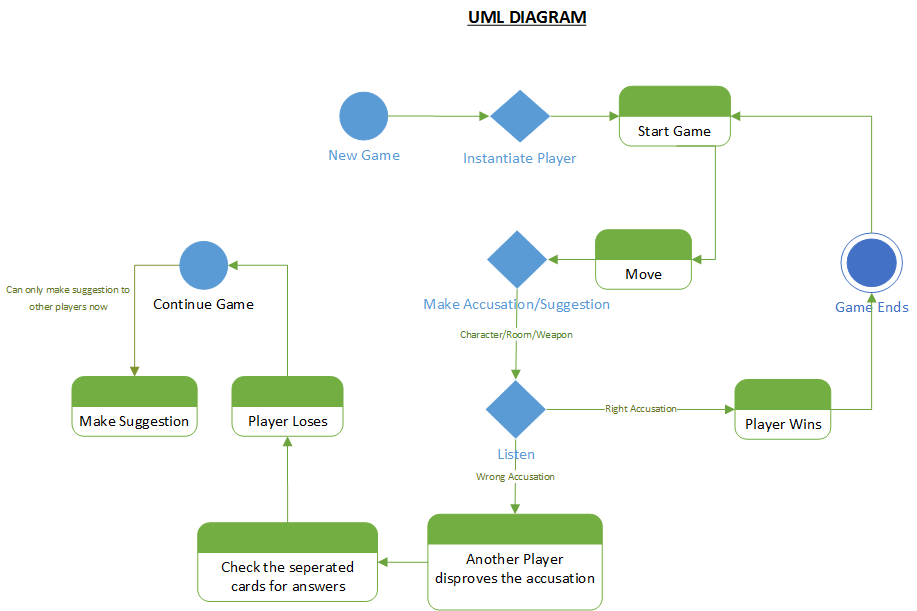


Figure . Autonomous Player Data Flow Start to Win or Lose[[2]](#endnote-2)



## Non Functional Requirements

The following non-functional requirements describe how usable, performant, testable, supportable and interoperable we expect this subsystem to be.

### Usability

Ability to interact with Computer player is contingent upon the player knowing the rules of the game and knowing the messages specified under the Interface requirements.

### Performance

Computer player must return with a response in less than a second for a requests from other subsystems.

Memory utilization for the computer player objects will stay below 250kb.

### Testability

Each of the functions must pass specified unit tests before it is allowed to be implemented by the computer player in a game.

Unit tests should provide at least 90% code coverage.

### Supportability

Design changes and bugs reported by any of the teams will get a response within 2 days of the report.

### Interoperability

When the computer player is integrated into the server system, the game operates as defined.

## Implementation Constraints

The implementation constraints defined in these requirements cover the way in which this subsystem will communicate with the other subsystems and the data that other subsystems will send to this subsystem. Design constraints, data structures, databases/repositories dictate how this subsystem will be implemented. Programming language and style guidelines, implementation tools and software subsystems drive the code design.

### Communication

* Interactions amongst computer and user players is strictly defined by the Interface Requirement documentation.
* Computer player assumes user will only use these specified messages.

### Design constraints

After discussions with the Server team, it was decided the Computer player is to be run as a state machine within the server. Each state machine will be instantiated from a class and the subsystem will be a Python module.

### Data structures

* The data between the server and the computer player will be contained in primitive data types such as string or int. And data structures such as lists and dictionaries.
* All methods must contain docstrings for the purpose of documentation generation.

### Databases/Repositories

* No database will be implemented for the computer player.
* Instead, server memory will contain the computer player objects.
* Therefore, it’s important that the computer player occupies a small memory footprint no more than 250kb per computer player object.

### Programming language and style guidelines

* The Computer player system will be programmed in Python 3.4[[3]](#endnote-3).
* The PEP Style Guide[[4]](#endnote-4)

### Implementation tools

* All programmers building the computer player will use PyCharm IDE by JetBrains.
* The code will be contained in source control, specifically GitHub[[5]](#endnote-5).
* Unit testing will be conducted using Python’s unittest framework[[6]](#endnote-6).
* Documentation will be implemented using Sphinx Documentation Generator[[7]](#endnote-7).

### Software subsystems

* High performance numeration data structures: itertools module[[8]](#endnote-8)
* Selection of available player names: random module
* Document generation: Sphinx module

1. Cohn, M. "Scrum." Mike Cohns Blog Succeeding With Agile RSS. Accessed September 16, 2014. www.mountaingoatsoftware.com/agile/scrum. [↑](#endnote-ref-1)
2. Pressman, R. “Software Engineering: A Practitioner’s Approach”. Accessed September 3, 2014. Print. [↑](#endnote-ref-2)
3. Rossum, G. "Python" General-Purpose, High-Level, Open Source Programming Language. Accessed October 2, 2014. https://www.python.org/ [↑](#endnote-ref-3)
4. Rossum, G. "PEP 8" Style Guide for Python Code. Accessed October 1, 2014. http://legacy.python.org/dev/peps/pep-0008/ [↑](#endnote-ref-4)
5. Preston-Werner, T., Wanstrath, C., Hyett, P. "GitHub." Source control implementation on the Git engine. Accessed October 7, 2014. http://github.com [↑](#endnote-ref-5)
6. Rossum, G. "unittest" Unit Testing Framework, Python Standard Library. Accessed October 2, 2014. https://docs.python.org/3/library/unittest.html [↑](#endnote-ref-6)
7. Brandl, G. "Sphinx Documentation Generator." Accessed October 1, 2014. http://sphinx-doc.org/contents.html [↑](#endnote-ref-7)
8. Rossum, G. "itertools" Functions Creating Iterators for Efficient Looping, Python Standard Library. Accessed October 2, 2014. https://docs.python.org/3.1/library/itertools.html [↑](#endnote-ref-8)