Jungle Game
API Design Document

Group 11

Jiaming HAN, Suizhi MA, Yunfei LIU and Yuxing PEI

A COMP3211 Homework Assignment

THE HONG KONG
POLYTECHNIC UNIVERSITY
香港理工大學

November 4, 2022

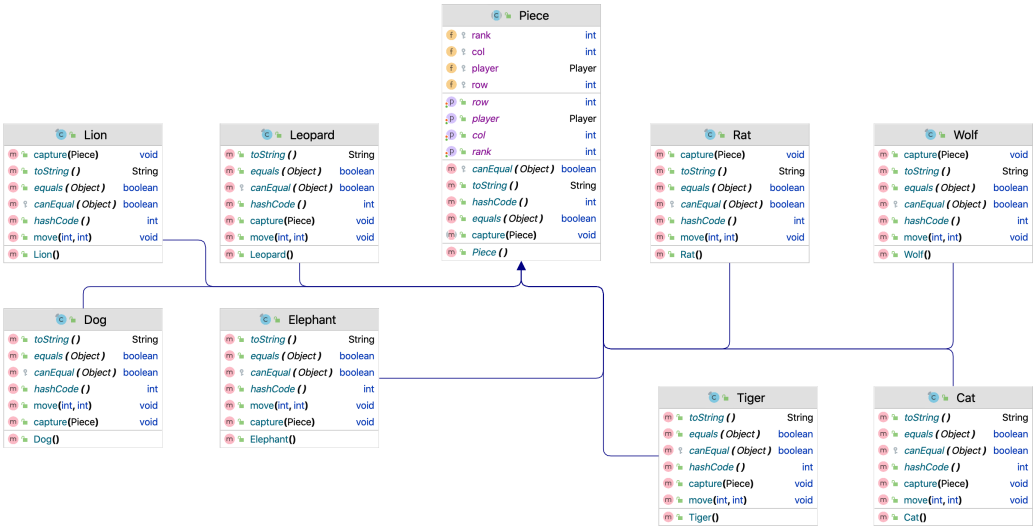

Figure 1: Different animal pieces for the game

1 Code Components

UML diagrams are drawn according to the project structure, and it follows the Model-View-Controller structure [1, 2, 3], which are shown as follows:

1.1 Model

Different animals are programmed as extended classes *piece* in the program, and each of them has their own attributes and public attributes, which is shown in Figure 1.

Landscapes are the blocks on board, which may carry special features including water, a den, and a trap. Those unique features can affect the result of the game significantly. The landscapes are extracted as classes in the project, and the structures of the projects are displayed in Figure 2.

Apart from other objects, the game manager is the core part of our design because it manages the overall activities of the game, and players are also considered important parts of the game. An overview of the project is displayed as Figure 3.

1.2 View

The view, or display of the game are designed as the structure in Figure 4.

1.3 Controller

The controller is usually the input of user, and similar to most of the games, it is defined like Figure 5.

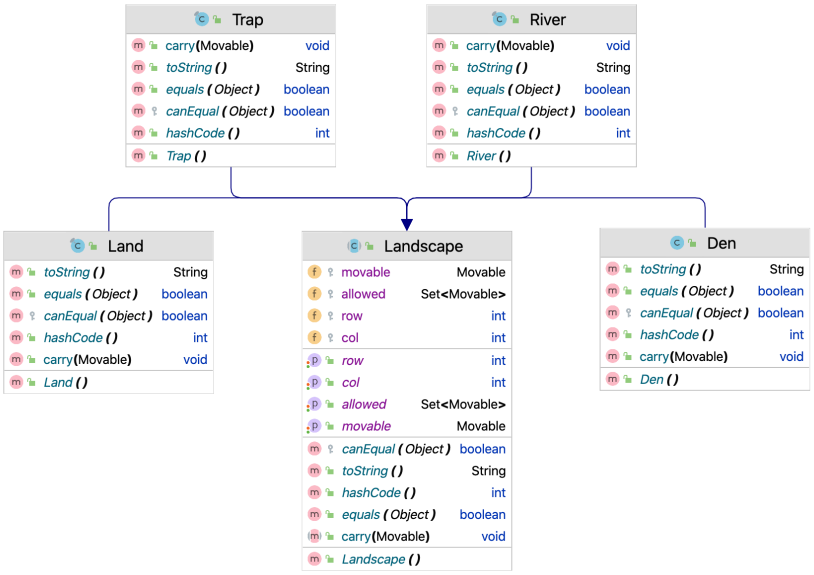

Figure 2: Landscapes in game board

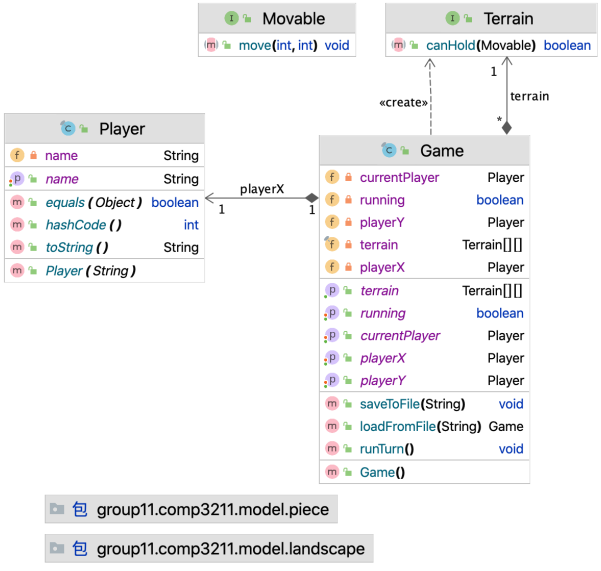

Figure 3: An overview of the project structure

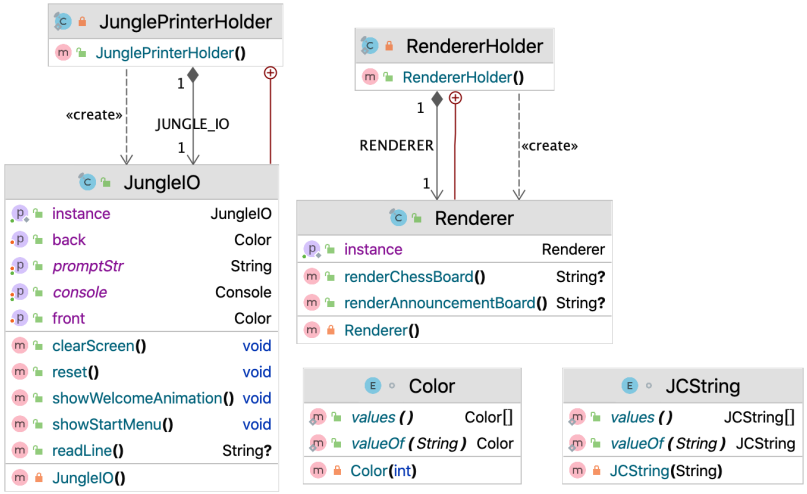

Figure 4: *View* part of the MVC structure

1.4 View

The view, or display of the game are designed as the structure in Figure 4.

2 Diagrams

2.1 Sequence Diagram

The sequence diagram [4, 5] of our project is displayed in Figure 6.

2.2 State Model

The state model [6] of our project is displayed in Figure 7.

2.3 Activity

The activities proposed [7] in the program are displayed in Figure 8.

3 Important Design Decisions

3.1 Language Choosing

Two languages can be chosen to finish this project, and it's important to choose the coding language because it is the base of the project, we discussed and analyzed the differences and advantages of those two languages, and below are the pros and cons of using those languages. [8]

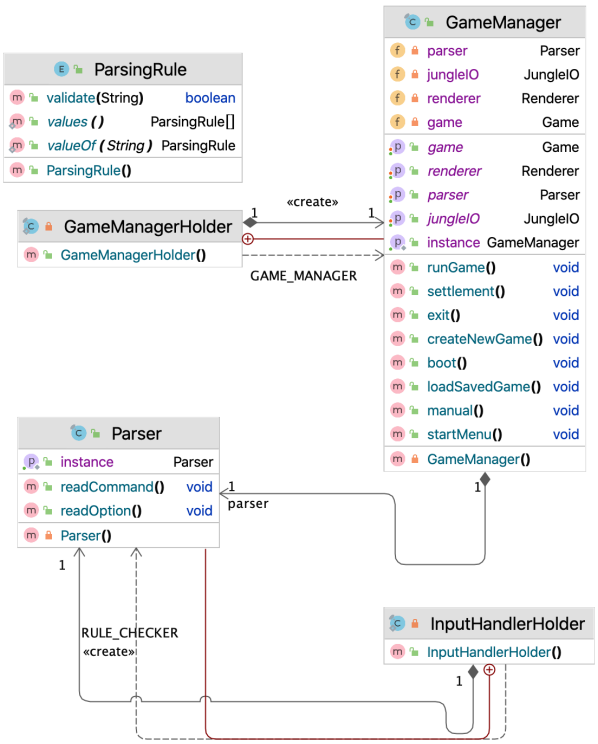

Figure 5: *Controller* part of the MVC structure

3.1.1 Python

Pros

- Interpreted programming language
- Object-oriented and procedural paradigms
- The code is very concise and clean
- No need to declare variables

Cons

- Slow running speed
- Difficult to configure environment for clients
- Complex multi-threading
- High memory consumption

3.1.2 Java

Pros

- Complied programming language

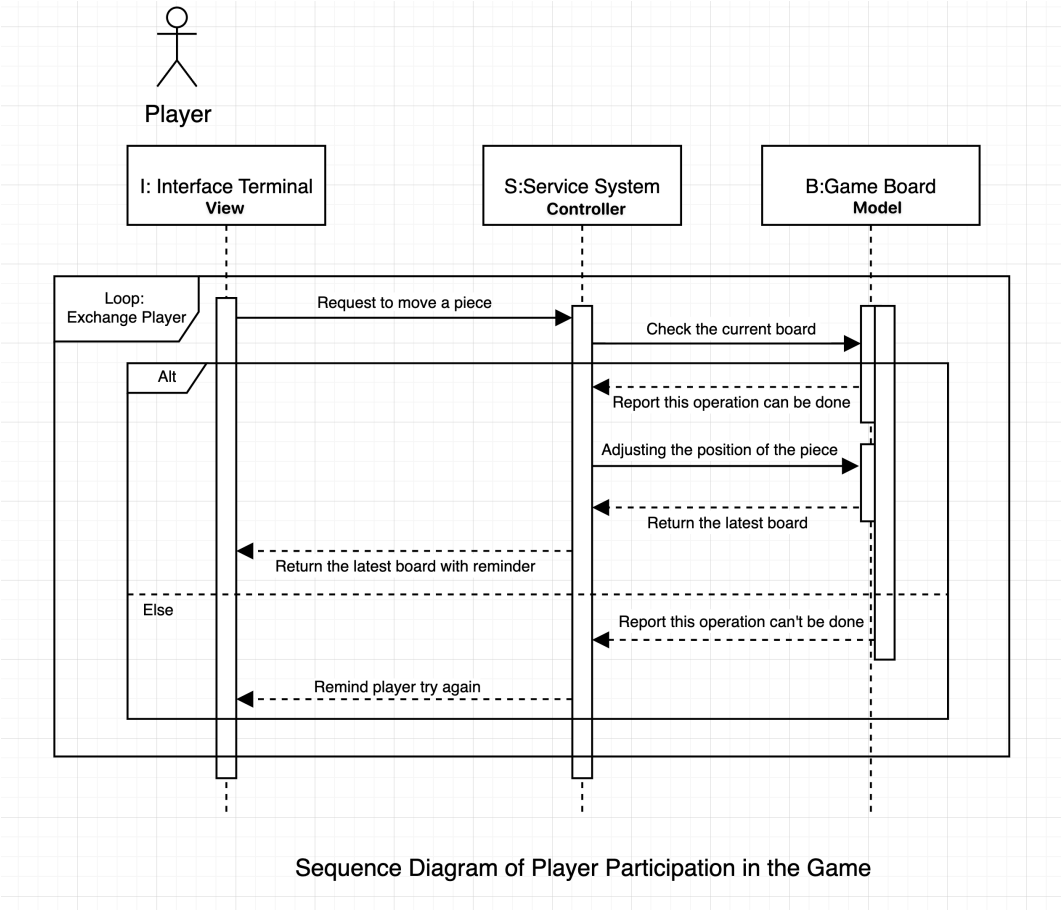

Figure 6: Sequence Diagram for Player Participation in the Game

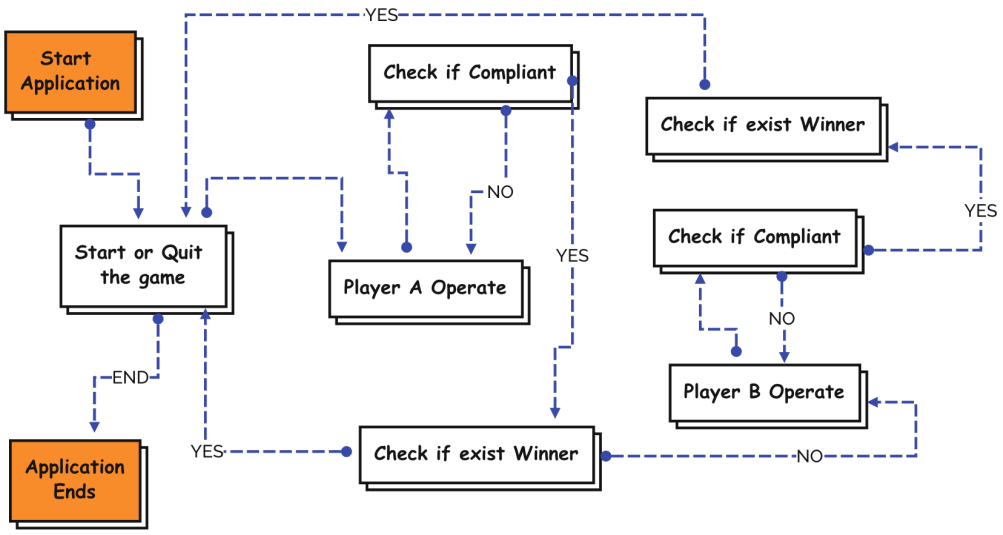

Figure 7: State Model

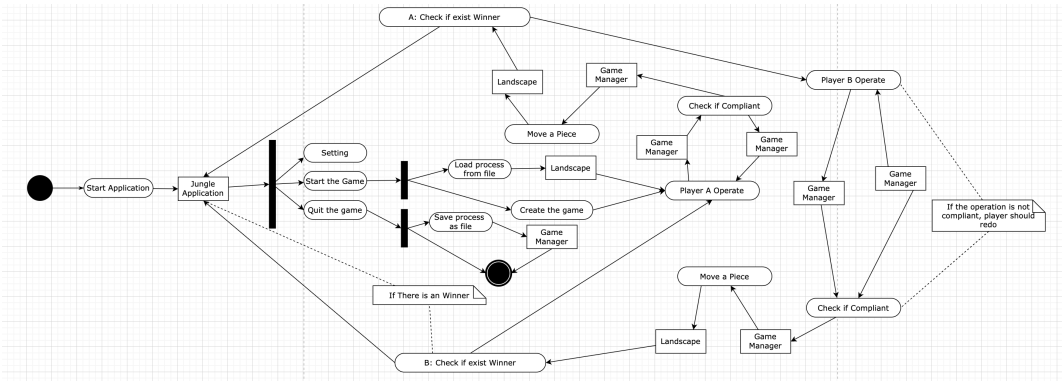

Figure 8: Activity

- Widely used in industry
- Automatic garbage collection
- Faster speed than Python

Cons

- Difficult program structure
- Lower code readability
- No backup facility

3.1.3 Decision

After detailed discussion and comparison with these two languages, we finally decided to choose **Java** as our programming language with the following reasons:

a) Java provides a more detailed and clearer project structure, which is considered essential for this software engineering project. [9]

b) Java has a better performance, which can reduce the latency of the program and use fewer system resources.

c) Java runs in JVM which is widely installed in most computers, making the program easy to run.

d) Java is easier to maintain because of it's the object-oriented feature, and it's easier to collaborate among group members.

3.1.4 Game Class Design

To achieve the MVC principle and object-oriented programming, all data of the game is packed under the Game class including the following components:

- The status of each position on the playing board, which is saved in a 2-D matrix that is straightforward to be understood. The reason why it is saved in a 2-D matrix is that the space occupied by the matrix is negligible for a modern computer, and the data type is very easy to save and use.
- The signal of the current status of the game, for example, running or stopped, is also saved in the game class, which shows whether the game has been started or not. As some features should be enabled only when the game is running, the program needs to keep tracking of the game status.
- Player information is saved in the class as well. As we tend to develop a player system to better simulate real-life game, players' names and other related information are saved in the class for the system to distinguish different users. More importantly, the program needs to know which user is current playing to grant permission of moving the pieces for his/her side.

3.1.5 Menu Design

All program events are running by an instance *GameManager*. It manages the life cycle of each game from starting the game to determining the winner of the game. After the user started the game, *GameManager* will start running and it provides four different options for users to choose from: Start game, load game, help, and exit, which corresponds to different use cases when using our program.
For starting a game, it is most often used to start a new game directly. Considering sometimes users may not be able to finish a game, and they are not willing to give up, we offer a function of saving the game and users are available to save the game. Load games can be used to load saved games before and let users continue playing. Here, one of the advantages of using a game class mentioned before is that we can save and load games easily.
The rest two functions are supplementary functions. For help, it is designed for users who are not familiar with the rules, and we can help them with how to play the game. Note that the help information will also be displayed in the latter stage after the users execute the wrong operations or feel confused. The exit function is easy to understand: it is just designed for users to exit the game.

3.1.6 Game Manager

The instance of class *GameManager* will manage the whole game, and it falls on **the middle layer between user interface and model layer**. The main duty for class *GameManager* is performing like a bridge and managing processed user commands. For example, change the status of *Game* instance.
Another highlight of this kind of design is that it can **decouple** model and user input effectively. In this case, the model will not solely depend on user inputs, which is more convenient for the assignment of game tasks. Another advantage of this design is that because of the decoupling operation, user input is separated from the model, which makes

the model easier to debug. Also, the way of a game manager is suitable for API design and makes module development becoming possible.

3.1.7 Jungle I/O Decision

To provide a better user experience, we designed and implemented a new class different from the default class of I/O. The newly designed class is intended for handling user inputs from users with different computer backgrounds. The current user interface of Java is designed for programmers which may be difficult to get along with for users who are not majoring in Computing. By extracting all the input and output as a separate class in our project, there are more advantages compared to integrating implementation.
One of the typical advantages is portability. As the core codes are running under processed data from the new I/O class, the data type of core codes is easy to be defined and it is portable once the code is written to consider most of the operating systems. The advantage of this design is that the final project can be saved in a flash drive and played on any computer without installing it at first.
Another advantage is multiple platform capability. For example, if clients want to immigrate to another platform such as iOS, we only need to redesign the I/O part and we can reuse the code for the core part, which is very time-saving and reduce additional cost.

3.1.8 Parser Input Class

Parser input is designed for considering different possible inputs of the user. According to the designing principle of the user interface, we need to consider the worst situation that the user will input, and the error cases should be handled properly to avoid confusion among users. The user input class can also define an input grammar uniformly which makes it easier to add extensions. In the future, when the game extends to a larger scale, such as many people playing remotely at the same time, or when the game rules need to be revised, the input can be revised without affecting other modules.

3.1.9 Instance of Game

The reason why we decided to make each game an instance is that the instance helps save and load the game, or return to the previous stage of the game. For example, when a user needs to save the game, the program can save the instance of the game very conveniently. Similarly, when the user is trying to load a game, the program can load the game very fast. The object-oriented intention of the project can be further respected based on the operation of making each game an instance.

3.1.10 Adjustment for Different Landscapes

According to the game rule, several different landscapes require separate considerations. For example, the river only allows a mouse to go into it so there needs verification for whether the move is valid, especially for river blocks. We designed **different classes** for all those special blocks. The reason why we define them in this way is that in future updates, we can edit the additional features directly in those blocks, which makes it easier to maintain and

decrease redundancy for the codes.
Each different type of block is implemented independently, and they retain their features within their classes. While a piece moves into the special blocks, their feature will be enabled to the piece, which follows the game rule.

3.1.11 Output Render

A render is designed and implemented in our project, render aims to receive the original instance from *GameManager* and transform the data to a display-friendly form: For example, transforming data representations to characters and adding an edge of the board. The reason why we are doing this is that we need to display the board to the users to let them know the current situation of the game, otherwise they may not know what is happening in the game and it is not feasible to let them realize all details of the board. A refreshing screen to keep them updated is of vital importance. The render will pass rendered data to the I/O module and display it on the screen to have a direct display of the chess pieces, which can be understood by the general public.
The major advantage of adding this output render is the layout is very intuitive. Just like users are playing games on the field, the display will show the electronic board for both sides of the player, and they can move the pieces and see how the pieces are moving. Which is essential information for them to think of the next step and how to win.
Another advantage is that the output is easy to be revised. For example, if a graphic user interface is asked to be implemented in this program in the future, we only need to rewrite the rendering part and change it from command-line form to graphic form. The data (i.e. position of the pieces) form received from *GameManager* is not necessarily to be modified because the difference of outputs devices can be resolved in the render, and the render can output corresponding encoded results depending on different displays.

3.1.12 User-defined Exceptions

We know that exceptions are essential parts for programmers to evaluate the bugs of the program [10]. However, for a user without a professional programming background, it is hard to understand the error messages from the computer and they will feel confused and frustrated. To solve this problem, we added several user-defined exceptions when the operation done by the user is not valid or violates some system rules. The exceptions are more understandable and easygoing compared to the professional exceptions presented by Java interpreters. Users can follow the instructions or messages in the exception to fix problems themselves, for example, by reviewing the rules.
Another advantage of using user-defined exceptions is it will be easier for programmers to locate the position of bugs. Programmers can investigate the position of the exception, and quickly find out the cause of the exception because the user-defined exceptions have been designed to anticipate unforeseen bugs.

3.1.13 Instantiate of blocks

For every block in the board, it is initialized as an instance. The reason why it is defined in this way is that the instance can carry a piece or an animal. Similarly, the task of verifying operations can be allocated to landscape blocks, and those tasks are easier to be implemented with different special blocks. An example is water blocks, they can be used to determine whether it is valid for a piece to move to them or not, for capturing process, it can be also checked by the block instance: it can check whether the piece is allowed to move inside whether there is already has an occupation.
One of the advantages of defining in this way is that it is helpful for further extension and writing. Also, because different modules are independent they can be developed independently. And it is also useful for interpersonal collaboration.
Another advantage is that the instance can be put in the matrix we mentioned in the previous introductions, and by doing this we can improve the conciseness and cleanliness of the codes, and increase their readability.

3.1.14 Enumerate Class

As for some of the classes such as constant objects, enumerate classes are used to increase the efficiency of the code. Also, it is very convenient for debugging. If programmers need to modify the value of constants in the future, it will be easy for them to find the meaning and the position of the constants incurred. And code stability of the code can be improved as well because the potential bugs are decreased.

References

[1] J. Deacon, "Model-view-controller (mvc) architecture," *Online][Citado em: 10 de março de 2006.] http://www. jdl. co. uk/briefings/MVC. pdf*, vol. 28, 2009.

[2] A. Leff and J. T. Rayfield, "Web-application development using the model/view/controller design pattern," in *Proceedings fifth ieee international enterprise distributed object computing conference*. IEEE, 2001, pp. 118–127.

[3] M. Aniche, G. Bavota, C. Treude, M. A. Gerosa, and A. van Deursen, "Code smells for model-view-controller architectures," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2121–2157, 2018.

[4] L. Li, "Translating use cases to sequence diagrams," in *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*. IEEE, 2000, pp. 293–296.

[5] X. Li, Z. Liu, and H. Jifeng, "A formal semantics of uml sequence diagram," in *2004 Australian Software Engineering Conference. Proceedings.* IEEE, 2004, pp. 168–177.

[6] R. V. D. Straeten, T. Mens, and S. V. Baelen, "Challenges in model-driven software engineering," in *International conference on model driven engineering languages and systems*. Springer, 2008, pp. 35–47.

[7] M. Brambilla, J. Cabot, and M. Wimmer, "Model-driven software engineering in practice," *Synthesis lectures on software engineering*, vol. 3, no. 1, pp. 1–207, 2017.

[8] S. Khoirom, M. Sonia, B. Laikhuram, J. Laishram, and T. D. Singh, "Comparative analysis of python and java for beginners," *Int. Res. J. Eng. Technol*, vol. 7, no. 8, pp. 4384–4407, 2020.

[9] G. Destefanis, M. Ortu, S. Porru, S. Swift, and M. Marchesi, "A statistical comparison of java and python software metric properties," in *Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics*, 2016, pp. 22–28.

[10] M. P. Robillard and G. C. Murphy, "Designing robust java programs with exceptions," in *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering: twenty-first century applications*, 2000, pp. 2–10.