Software Development: Improvements & Reflection

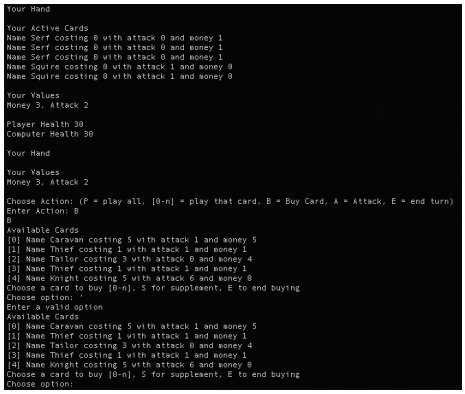
Alexander McFarlane

March 23, 2016

Abstract

The initial project plan was re-evaluated after several significant delays. A control flow was created to facilitate a coherent development plan. The project aimed to build on a Spiral project approach by managing miniature iterative development projects through Github's "Issue" and "Milestone" tracking system. Risk was continually documented controlled and evaluated and testing was performed on each change. Third parties were used to assess the UI and enhance the quality of the final product.

(a) The new "Shop" interface.



(b) The original interface.

Figure 1: Improvements to the card purchasing interface, rebranded as "The Shop". The final product aims to exemplify a commitment to quality in all areas of the development process. The simple, intuitive nature layout is a reflection of this commitment.

1 Summary of Changes

In this section we discuss the summary of changes made to the code. All issues discussed can be found at the Github Repository [5] in full detail. Details of the github repository are found in

Appendix D.

1.1 Aims

The following aims based on notes from the lecture and workshop series, established a basis for setting project goals and prioritising issues.

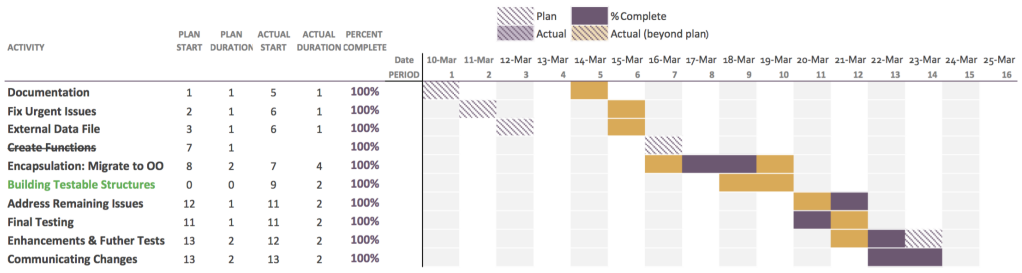
Working Code The final submission must be functioning code with a functioning game

Documentation Risks and issues should be documented as should the approach to fixing them. Code should be documented to allow for future work.

Risk Management Risky sections of code must be isolated from critical points and removed where feasible. A spiral approach will be taken to minimise development risk.

Encapsulation Code should be factorised (object orientated) to encapsulate routines

Testing There should be significant testing at each state. Unit tests should be provided on crucial base components.



ACTIVITY	PLAN START	PLAN DURATION	ACTUAL START	ACTUAL DURATION	PERCENT COMPLETE
Documentation	1	1	5	1	100%
Fix Urgent Issues	2	1	6	1	100%
External Data File	3	1	6	1	100%
~~Create Functions~~	7	1			100%
Encapsulation: Migrate to OO	8	2	7	4	100%
Building Testable Structures	0	0	9	2	100%
Address Remaining Issues	12	1	11	2	100%
Final Testing	11	1	11	2	100%
Enhancements & Futher Tests	13	2	12	2	100%
Communicating Changes	13	2	13	2	100%

Figure 2: The project progression according to the predefined Milestones. This Gantt chart serves to illustrate the progression of the project after completion. During development, progress according to predefined deadlines was was tracked using Github's Milestones feature. The removal of "Create Functions" and addition of "Building Testable Structures" will be discussed in what follows.

1.2 Milestones

The Milestones can be seen to reflect the improvement aims and were documented in detail on [5] throughout the course of the project. We will discuss the cancellation of "Create Functions" later in subsection 2.2.

1. Documentation
2. External Data File
3. Fix Urgent Errors
4. ~~Create Functions~~

5. Migrate to Object Orientated (OO) Classes
6. Building Testable Structures
7. Address Remaining Issues
8. Final Testing
9. Enhancements and Further Tests
10. Communicating Changes

In what follows, we engage in a detailed discussion and assessment for each phase of development[1].

1.2.1 Documentation

Expected Time 2 hours

Actual Time 10 hours

Delays None

Change Summary The issues and Milestones were moved to Github [5], where they were extensively documented. The code was heavily commented and a control flow of the game was created. The control flow can be viewed in Appendix A and provides an intuitive formal documentation of the initial code. Practically, this helps the developer identify key areas of duplication and encapsulate groups of similar procedures and was highly useful later on in migrating to OO code, subsubsection 1.2.4. We can instantly identify three classes from this analysis: the Game Engine; the Computer and the User.

1.2.2 External Data File

Expected Time 3 hours

Actual Time 8 hours

Delays Severe delays due to unexpected external pressures. Further delays as this was a fundamentally bad idea implementing bad code to conform to a bad Milestone ordering.

Change Summary Implemented pseudo-class structure to be immediately absorbed into an OO solution.

> ***Remark*** *This Milestone implemented bad code to conform to a pre-defined plan. This caused significant delays in the later Milestone: "Encapsulation: Migration to Object Orientated Code" and the addition of a further Milestone: "Building Testable Structures". This is discussed later, in detail through, subsubsection 2.2.3.*

[1] In terms of the time estimates and actual time taken, we note that two different unit systems of "days" and "hours" are used. This is deliberate as certain Milestones were completed in one linear session, and hence, given a timing in "hours". Those with "days", were split across several blocks, however, we can estimate that "1 day" is equivalent to "8 hours" of work.

1.2.3 Critical Issues

Expected Time 6 hours

Actual Time 6 hours

- Reassess urgency of critical issues
- Fix all critical issues
- Test the solution

Delays Severe delays in starting due to unexpected external pressures.

Change Summary The critical issues were handled swiftly with extensive testing at each implementation. Several other issues were uncovered and documented during this process: Where a quick fix was available it was also provided. Some non-critical issues were moved to later Milestones that may rely on more involved fixes such as Object Orientated programming.

1.2.4 Encapsulation: Migration to OO Classes

Expected Time 3 days

Actual Time 4 days

Aims

- Migrate code to OO class structure
- Testing
- **Mid-Project Progress Review** Finish with a review to assess appropriateness of solutions provided

Disjoint and non-interacting classes of variables are: The User; The Computer; The Central Cards. Suggest, therefore, three classes containing the functionality for each. Will contain initial parameters in a constructor. The Milestone will conduct an informal ***"Mid-Project Progress Review"***: An assessment to ensure that the direction and solutions are appropriate.

Delays Deciding how best to implement tests Evaluating / reassessing the class structure for appropriateness Decided to revisit initial data creation and remove the bad constructors

Change Summary Migrated code to OO class structure: User; Computer; Game Engine. The classes as built presented a challenge when thinking about testing. Building unit tests on these structures would mean the test would be discarded and rewritten later.

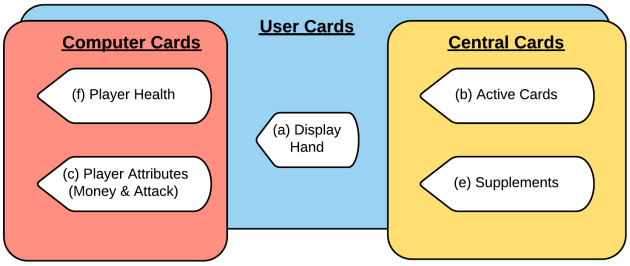


Figure 3: The display output is a clear example of an opportunity for encapsulation. We demonstrate the intersection of several key game constituents and their respective display outputs.

Mid-Project Progress Review Reassessed appropriateness of keeping dictionary format in the classes `pC` and `pO`. After revisiting Lecture notes, [3], it became clear that the beginnings of "bad code" were creeping into the development cycle in order to test against the initially bad structure. This Milestone was consequently closed off with a series of game play and inspection tests. A new Milestone: "Building Testable Structures" was created as a result of this mid-project review to address the previously discussed issues.

1.2.5 Building Testable Structures

Expected Time 8 hours

Actual Time 14 hours

Aims

- Revisit the constructors in the classes and assess appropriateness for testing
- Reassess the creation of cards
- Remove interdependency
- Assess redundancies in initial parameterisation
- Care *must* be taken to ensure improvements are small and regular to minimise risk

The aim of this Milestone is essentially redoing the "External Data File" Milestone as it should have originally been done. These changes can then be iterated through to a better implementation of the classes.

Delays Project was slightly delayed due to come issues in handling Python objects

Change Summary The structure of the base classes is fundamental to the functionality of the game. Learning the lesson from earlier: Bad code was avoided and especially not intentionally implemented. All settings were moved to an external file. A method was created to pass the parameters of the settings directly and create card structures. Some challenges were incurred with Python's handling of certain copied objects which was eventually resolved. Fundamentally, lots of care taken to ensure that each addition was a small addition and was tested thoroughly. Particularly as the commits were done across a short period of time.

1.2.6 Address Remaining Issues

Expected Time 1 days

Actual Time 20 hours

Aims

- Finish tidying up stylistic / elegance issues in code
- Further encapsulate code
- Testing
- Try not get carried away with implementing overly extravagant solutions.

Delays Implementation of colours to mitigate eye-strain. The large number of interdependent areas of code that were fixed led to a number of issues requiring resolutions.

Change Summary Several issues were identified in the earlier development and moved into this Milestone. It became necessary to exert some self control to focus on the task at hand. The priority flags greatly helped with this and shows the effectiveness of a good plan. The code was encapsulated into manageable routines and as the functions were broken down it became more obvious how to subdivide the messy code.

1.2.7 Final Testing

Expected Time 4 hours

Actual Time 10 hours

Aims

- Test on "cplab" computers
- Ensure packages are all working as they should be
- Run tests created and on "cplab"
- Use external parties to test - record results and use for UI improvement

Delays None

Change Summary Tested on the `cplab` machines. The packages all imported as hoped and so there was no need for workarounds. This was perhaps a risky move not checking the compatibility at each stage. However, given experience with the machines that would ultimately operate the code[2]; the action was sensible, as non-standard packages were avoided (see logging routines). Therefore, the only issues that may have arose would be related to terminal colour encodings. Several "willing" volunteers were engaged for testing the software for issues. The test subjects were not developers and this was deliberate as the aim was to look for potential issues and mistakes during general use. The results were highly valuable and identified several issues. A summary of the testing feedback can be viewed in Appendix B.

1.2.8 Enhancements and Further Tests

Expected Time 6 hours

Actual Time Taken 20 hours

Aims

- Assess the state of the UI
- Create a Control Flow
- Draw up plans for a new UI
- Implement new UI
- Testing

This stage is the beginning of the enhancements.

Delays Lack of a coherent development plan for the UI. Significant time testing and debugging as a result.

Change Summary The results from the user tests were used to motivate a new UI. A control flow was created for the gameplay generated by the `game()` loop as shown in Figure 4 but this was not extended to the entire code base[3]. There was no clear route of development since there was never a real coherent plan of implementation for the UI, this led to a significant increase of defects as new code was introduced ad-hoc. This was possibly one of the most challenging parts of the development cycle and also the worst managed. Significant effort had been put into the rest of the project and so this phase was often at risk of over engineering the final product. It was consequently difficult to determine when to do a feature freeze and this is a failure of the aims for this Milestone. A final iterative series of testing and coding aimed to iron out the last of the bugs and caught a large number of defects after spending time away from the code.

[2]The `_CPLAB_` computers are notoriously slow and so it would have probably been more tedious to `_ssh_` in every time something significant was added than to implement a workaround.

[3]The functionality of the game itself it fundamentally the same, the flow would have had to require significant detail to demonstrate the differences and the value could not be justified for the effort.

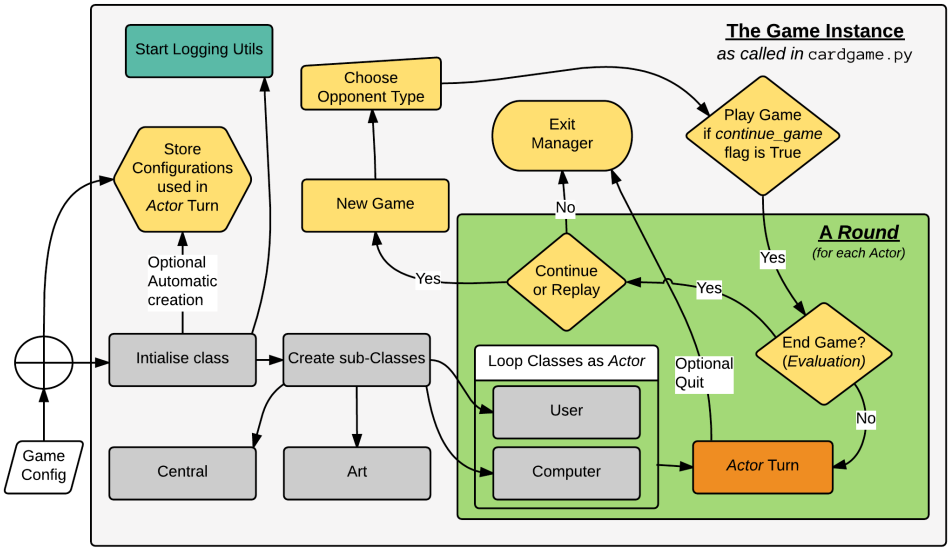


Figure 4: An overview of the new `game()` class being used to create a game loop. Here, class objects are shown in grey; the `game()` functions are shown in yellow. The green box shows an encapsulated routine that constitutes a game "round". Each round contains a loop to promote each player to "Actor" wherein, they act out a"Turn". The process of a "Turn", is another similarly encapsulated series of routines.

1.2.9 Communicating Changes

Expected Time 3-4 days

Actual Time 3 days

Aims

- Document Issues for Future Work
- Document Game
- Document Aims
- Document Issues Faced
- Document Lessons Learned

Delays None

Change Summary Created an image documenting the key relationships in the code which can be seen in Figure 5. Demonstrated high level changes in the game structure with the game instance flow as seen in Figure 4. The game has been fully documented and this documentation can be found in the file `README.md`, provided on the repository, as well as a copy in Appendix C. This also documents how the code is called in relation to the game play. Secondly, the repository details have been documented and can also be found in `README.md` or in Appendix D.

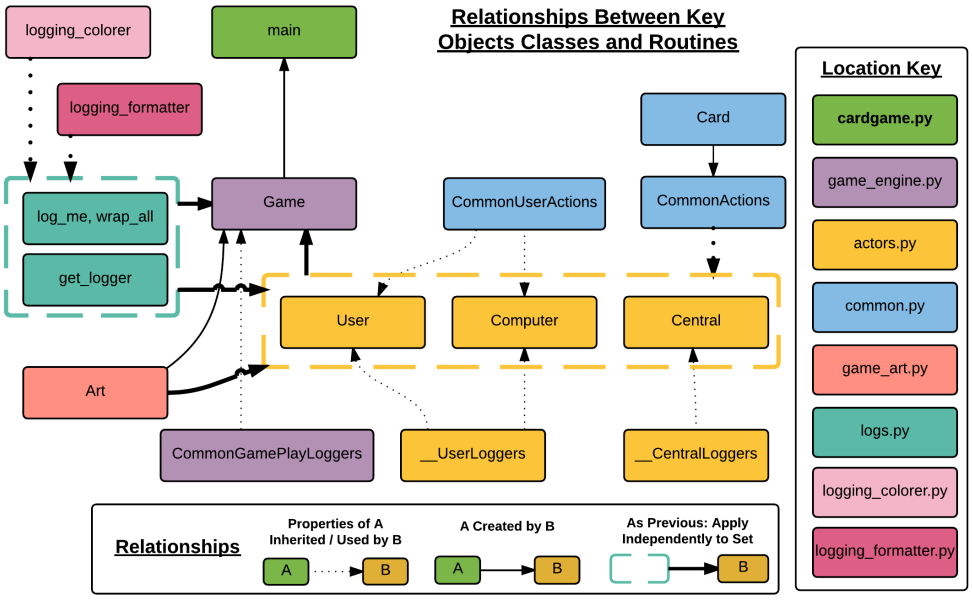


Figure 5: Here we demonstrate key relationships between new code structures and their respective external file names.

2 Reflection

2.1 Risk and Project Management

The project posed a variety of risks to the developer. Key areas being, feature creep / gold-plating, silver bullet syndrome (attempting to fix everything with one solution) and inadequate design / short changed quality. The management approach was one taken to mitigate the negative effects of these factors, should they occur.

2.1.1 Risk Management

It is clear from the development cycle that when Milestones were not obvious or properly thought out, development ran into problems. Key examples in this project were the initial issue log and then

during the project; moving the settings to an external data file and the enhancement of the UI. Risk management was taken very seriously through the development of the product. The development hours were very long and spanned several days, which can be seen from the Github commits. As a consequence, several tactics taken from the course [2] [4] [3] were used to combat trivial mistakes:

- Defect creep minimised by treating each bug as a mini-project
- Testing was carried out extensively alongside development
- Third party testing was used to ensure quality
- Functionality was encapsulated in an iterative manner
- A mid-project review was undertaken to ensure

These tactics can be seen throughout the issue tracking. There are cases in which trivial issues naturally became fixed by other issues but fundamentally all were tested before committing a build to Github. The testing cycle can be seen in [2] and using this approach the first focus was on providing a strong base structure with unit tests across the initial base classes and settings and then moving into more comprehensive tests. One shortcoming was perhaps a lack of unit tests further down the development cycle due to time constraints which can be seen in one of the defects left as further development.

2.1.2 Project Management

The attempt was to follow a Spiral Model approach as seen in [cite] by continuously identifying issues and setting them up as mini projects. The risk was that the project would become a "Code like Hell" approach due to the time constraints and every effort was taken to avoid this. An example defect which required significant discussion and documentation can be seen in Issue #82. In each defect there are key aims for development and suggested fixes for a miniature project cycle. At a higher level, each Milestone is step up with an estimated time-scale, aims and results. The overall time taken was also logged for a comparison to the estimate to help in future Milestone time estimations. In each defect a summary or at least an indication of the fix has been put in the comments and then at a higher level, each Milestone was summarised with results and what was actually done. This approach allowed a clear documentation of risks and identification of shortfalls to address in further iterations.

2.1.3 Unexpected Risks Encountered

The overall work schedule vastly differed from that of the original plan. There were several reasons for this: A lack of practical direction in the initial report; Pressures of external projects and human error. There was a significant increase across various courses as some deadlines and other projects became highly involved. The lack of direction in the initial analysis did not help to give enough motivation to start Milestones on vague directions.

Facilitation of Human Error The initial report in fact had a significant knock on effect in terms of facilitating human error which in turn led to a large delay of several weeks. We move to discuss the weakness of the code review and the issue documentation and how the lessons learned during the process.

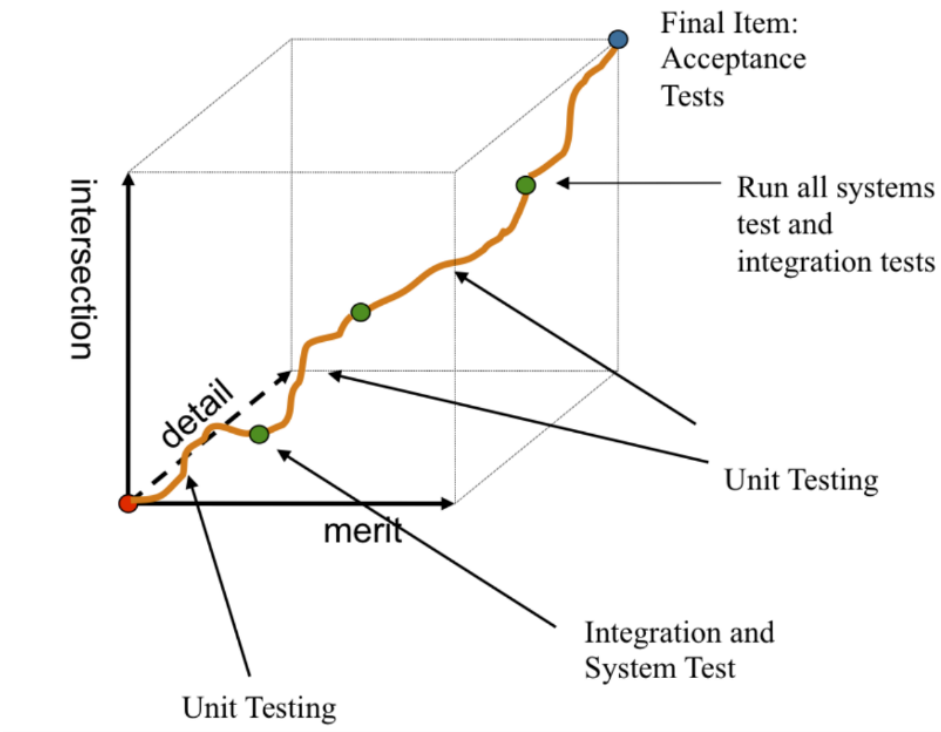


Figure 6: testing

The issues were documented in an Excel spreadsheet and the code review was kept in a hard paper copy. The paper code review contained a highly detailed breakdown of all code functionality and modular links in a control-flow style. This was accidentally binned as it was stored with other loose paper and with it the developer lost all knowledge of the non-trivial areas of the code functionality. This set development back several days when it came to look at the code.

The second human error and possibly what led to the largest development delay was the corruption of the issues in the Excel spreadsheet. There is a direct correlation with this error occurring and the lack of detail in the initial documentation of issues. To make sense, the issues relied heavily on the line number and the correspondence to a short description. Alone and without familiarity with the code, neither bits of information provided any real insight into the actual issue. The error arose in the Excel spreadsheet being saved with a filter that didn't extend to the issue line-numbering. Consequently, the data became largely meaningless. This was an important lesson for the developer in providing detailed issue documentation.

These two factors in combination with a lack of detail in the initial work schedule and the high fluctuation of external pressures meant that development became delayed for several weeks. Other projects were able to take precedence due to the inability to work on a small well defined set of corrective actions in turn.

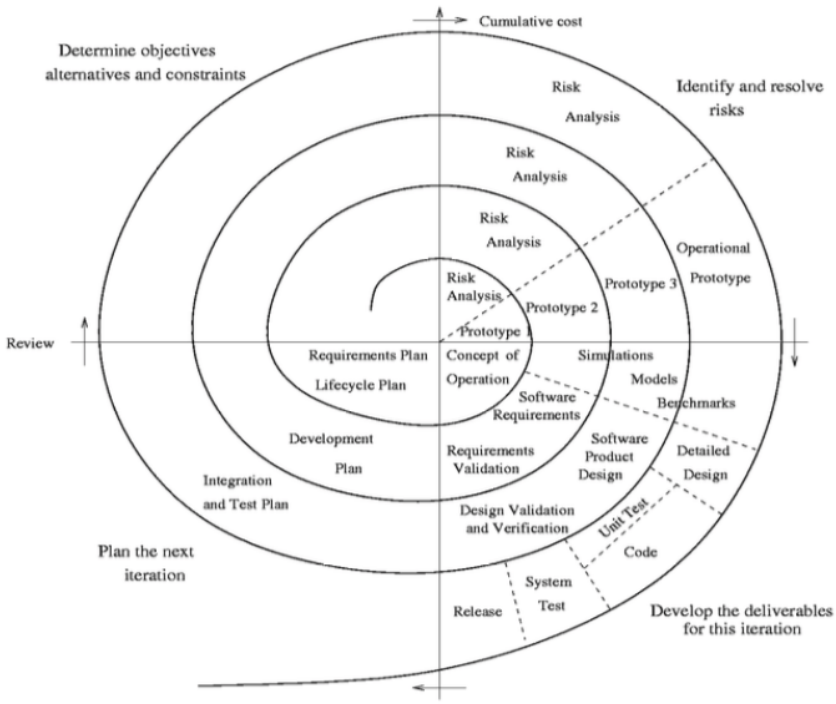


Figure 7: spiral

2.2 Lessons Learned

The approach to the final project varied from the initial proposal. The initial proposal had several flaws in that it was too theoretical. This became even more obvious when it came to implementation and consequently required a rework to avoid the previous failures.

2.2.1 Coherent Issue Tracking

Despite discussing the risks of disjointed tools in project management; there were two different tools used to track and record bugs as well as tracking progress. Moreover, the final code must be presented in a repository format. Naturally, it made sense to utilise a repository that could also track issues and Milestone progression. Thus, the project was moved to Github where issues could be visibly logged and tracked whilst being linked to snapshot code versions.

The documentation of errors was also redone. A criticism of the previous report was a lack of detail and visibility in the discussion of code issues. Incidentally, a serious developer error highlighted the need for a better issue documentation, which is discussed in the following section subsubsection 2.2.2, as it largely rendered all previous work unusable. The net result was a natural move of all issue tracking to Github.

Figure 8: An example Issue logged in Github provides a documentation of all problems faced during development and serves as a basis for future work.

The issues and Milestones were used to group issues into different categories. This allowed the developer to provide a great level of detail to avoid a similar issue with clarity. We see in Figure 8 an example linking the code version, the code snippet with the issue and the suggested corrective action.

2.2.2 Code Documentation

We can clearly see that a lack of a practical plan has a direct correlation to facilitating delays and error. To correct these issues, a rigorous plan was put into practice using the tools provided in Github to manage a highly detailed set of issues and Milestones and great lengths were taken to provide robust documentation of both issues and the code base. The code, in parts, is highly difficult to understand and so documentation in the code base avoided the risk of managing multiple processes as discussed in the earlier risk report. Moreover, this also meant that a sleep deprived developer would not bin it in haste! The initial commit shows code that was heavily documented inline as a result.

Usability of Documentation Building on ideas learned during the process. It was felt that a control flow would provide great benefit not only for development but also for presenting final results. A simple control flow is an intuitive way to demonstrate to an end user the changes made to the code as seen in Figure 4 and Figure 5. A futher step was also taken to document the code routines in relation to actual game-play as seen in Appendix C.

2.2.3 Writing Bad Code to Conform to a Plan

One of the key lessons learned, is directly from a Lecture in Software Development:"Identify and tackle bad code before it cripples the project". The scope of a plan is to plan out how to complete

a task before attempting it. Consequently, plans can have specific goals and Milestones that are not sensible once the task is being undertaken and conversations with peers, suggest that this was a prevalent issue. The risk is that the developer may attempt to rigidly stick to a given plan by implementing 'bad' code to facilitate the completion of a Milestone that required a rework. A clear example in this project was the movement of data to an external file. The self-imposed constraints were: To preserve the original structure of the input data to simplify tests; To avoid the risk of gold-plating with an OO solution until a later Milestone. The simple task of moving the game settings to an external file ended up becoming a highly involved task that fundamentally revolved around writing bad code that in the end used a class anyway to create global variables in a complicated manor and significant effort had to then be put in to ensure that no errors were made as a result. The lessons learned are significant and can be seen in the Milestone to create functions. It soon became apparent that many functions would be the same and that the developer would be duplicating efforts to write bad code. The process was therefore never started and instead of naïvely spending time writing lots of disparate functions, the move was seamless to a class implementation.

3 Future Work and Conclusions

During the later part of the project, the remaining risks were documented and listed under the title of "Future Development". This provides a good documentation for further work and provides a clear route of development should the project be picked up by another developer.

The project provided a comprehensive process into looking developing high quality software. For a physicist attempting implement abstract theoretical models, solid development and testing is crucial. This project has taught me several key lessons in project management and testing processes which will be highly effective when applied to more complex projects.

Appendices

A Original Code: Encapsulation

In this section I present work that was done in the

A.1 Game Overview

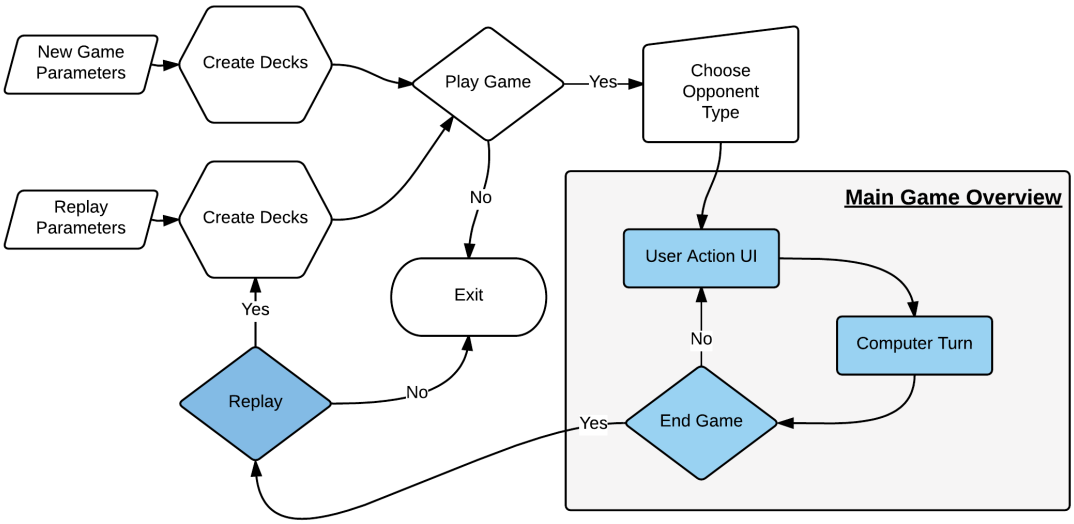
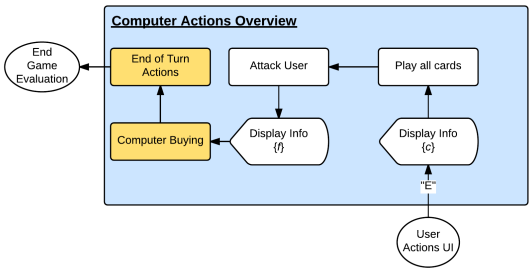
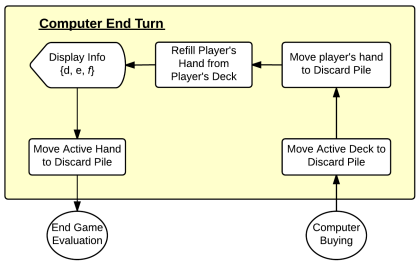


Figure 9: The Original Game: A High Level Overview

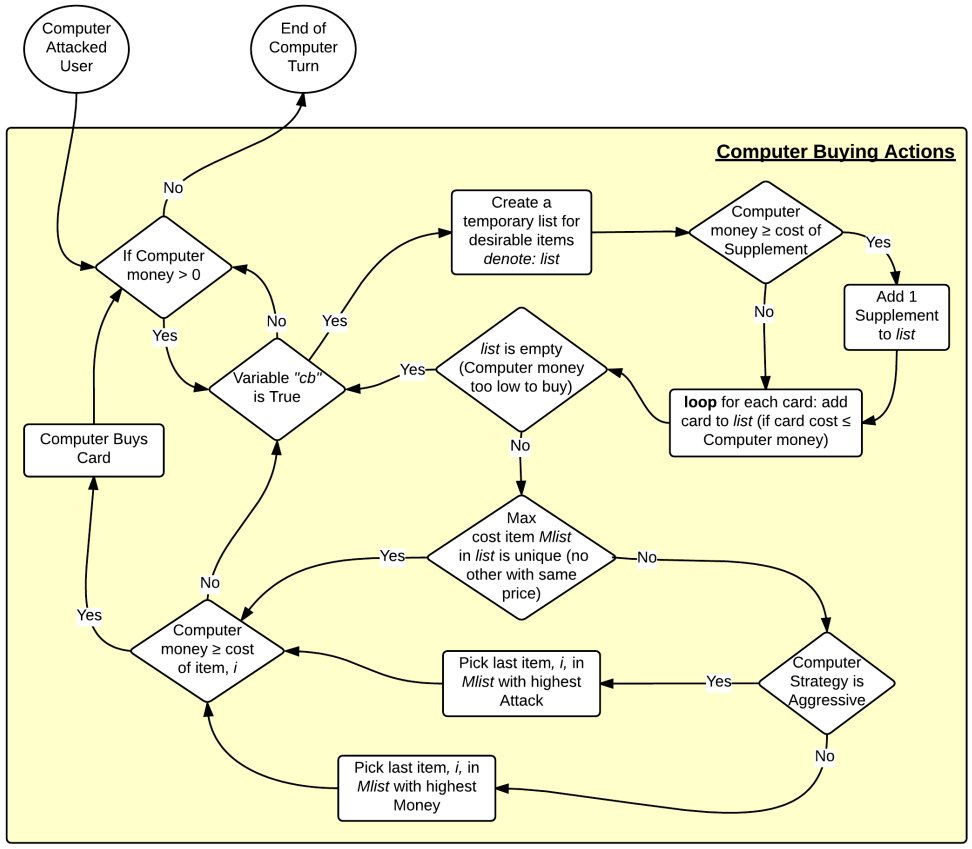
A.2 Computer Actions



(a) The Computer's Actions: An Overview



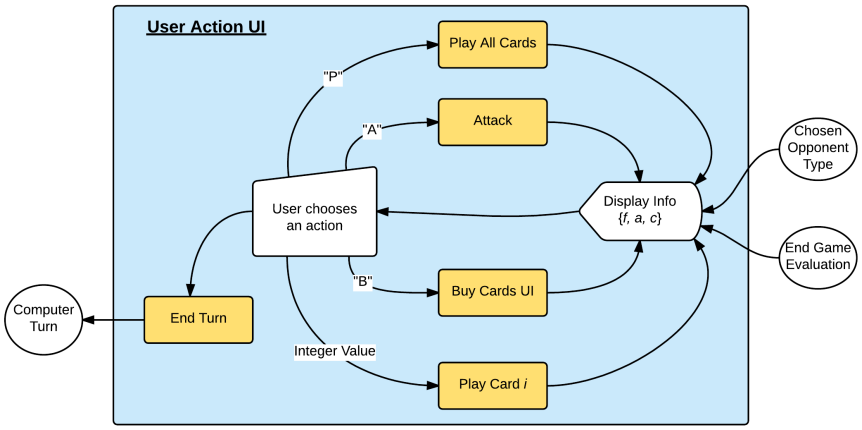
(b) The Computer's End Turn Actions



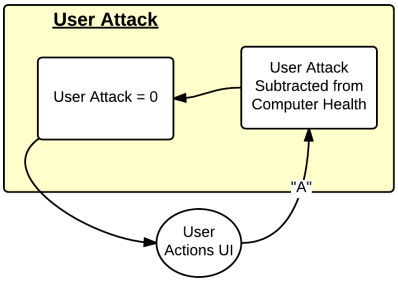
(c) The Computer's Card Purchasing Actions

Figure 10: The Computer's Actions in detail

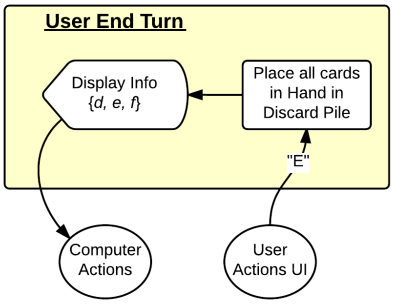
A.3 User Actions



(a) The User's Actions: An Overview

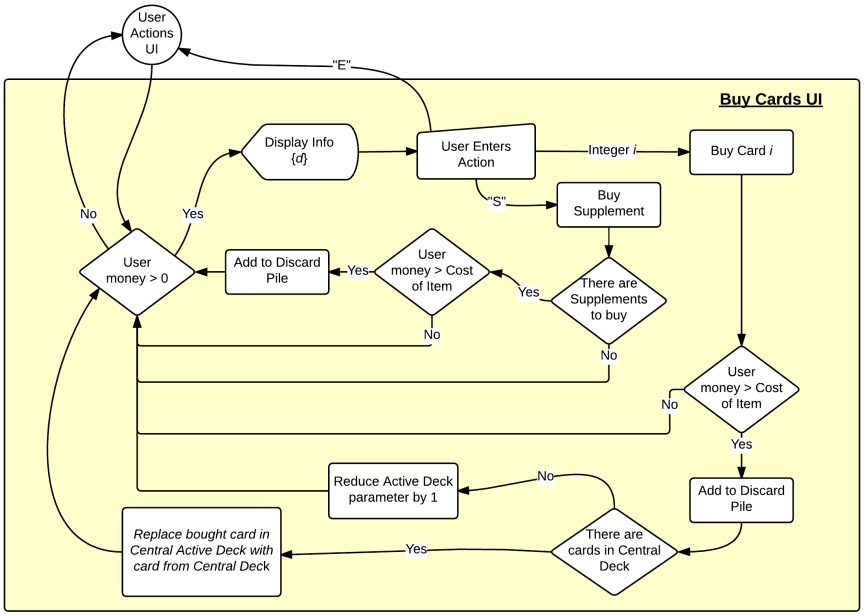


(b) The User's Attack Actions

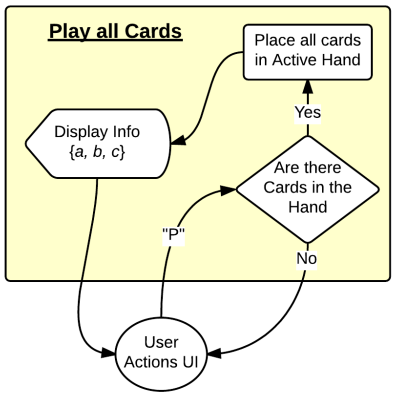


(c) The routine for ending the User's turn

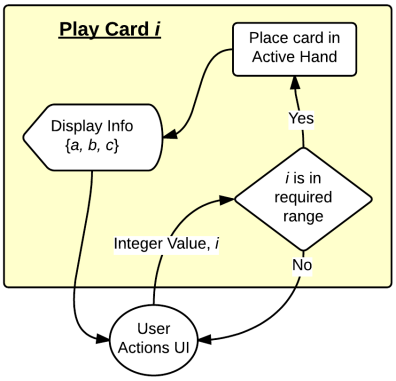
Figure 11: The User's Actions in detail, continued.



(a) The User's 'Shop' Interface Actions



(b) The User's actions when playing all cards



(c) The User's actions when playing a single card

Figure 12: The User's Actions in detail, continued in Figure 11

A.4 End Game Actions

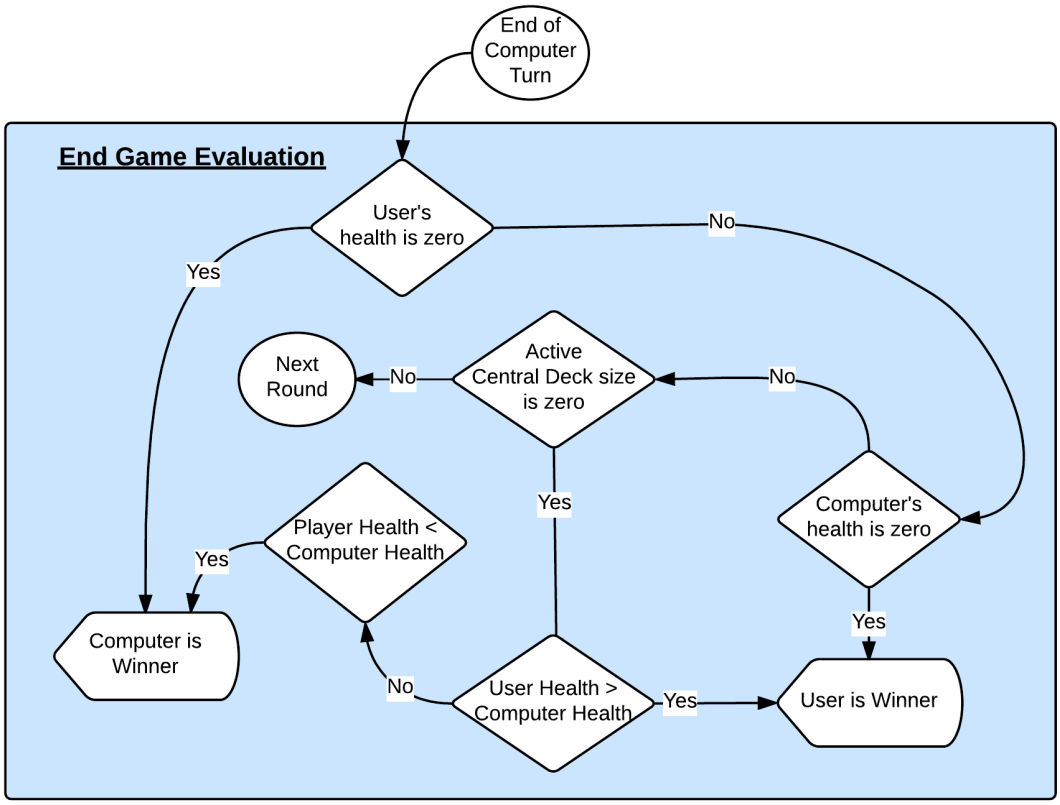


Figure 13: The routine for ending the game

B Third Party Testing

A summary of feedback from Third Party Tests. Test subjects are denoted as *[Marios]*, *[Aaron]* and *[Theo]* who were all highly willing, yet confused volunteers. The raw feedback sheets can be seen in the Issue: #90 within the repository [5].

Confusion in the Gameplay *How confusing is the game?*

- How do I play? *[Marios][Theo][Aaron]*
- What are supplements? *[Marios]*
- How do I buy a card? *[Marios]*
- Why did my hand vanish after I bought cards? *[Marios][Theo]*
- What are available cards? *[Theo]*
- Attempting to attack without playing cards *[Theo][Aaron]*
- What does playing a card do? *[Theo]*

UI / Layout *Points relating to the UI / layout*

- Who am I? *[Marios]*
- What is "Your Hand"? *[Marios]*
- "[0]" was taken literally as user attempted to buy cards. *[Marios]*
- How much money do I have after the shop closes? *[Marios][Aaron]*
- Difficult to read. *[Marios][Theo]*
- Layout changes - where do the indices go? *[Marios]*
- What is my Attack? *[Theo]*
- Not inputting "Y" ends the game. *[Aaron]*
- What is money - why do I need it? *[Aaron]*
- Confusion that "E" doesn't end turn as well as exiting the shop. *[Aaron]*
- Suggestion for titles to indicate the game state e.g. "*You are in the shop*" *[Aaron]*

Other / Bugs *Are there any spelling issues or unclear sentences / little mistakes*

- "or *an* Greedy Opponent" - bad grammar. *[Marios][Theo]*
- Loose spaces in front of certain lines. *[Marios]*
- When the User kills the computer, the game interface persists. *[Theo]*
- Possible bug in money summation in the shop interface *[Aaron]*
- If player loses with negative health, their health is lower in a replayed game. *[Aaron]*

C Game Overview and Implementation

This section contains the full documentation of the game. We also include relevant functions and classes that are used in the implementation of the game actions.
To introduce the game, and for simplicity, we will assume certain settings and will denote in *italics* that a parameter can be adjusted by the file `_config.py_`.

C.1 Setting up the game

The game is played by two **Players**: the real-life user, **The Player**, competing against the computer, **The AI**. Each Player is assigned a **Player Deck** that is individual to each player consisting of*10* Cards,*2 Squires and 8 Serfs*. Each Player is assigned a **Discard Pile (Discard Deck)**, which starts empty. Each Player is assigned an **Active Hand**, which starts empty. Each Player is assigned a **Player Hand**, which starts empty. A **Central Deck** of*36* Cards is also created that consists of*a variety* of different Cards. A **Central Active Deck** will begin empty. Players are both assigned*30* **Health** at the beginning of a **Game** instance. A **Supplement** is identical to a normal Card in this game and has no special functionality at this stage. However, they are treated as separate entities in the code and UI to allow for future development. They are played and bought as normal Cards.
The game is set up through the function `_new()_` within the class `_Game()_` in `_game_engine.py_`. Players are managed by the `_User()_` and `_Computer()_` classes found in `_Actors.py_`. The Central Deck and Central Active Deck are managed by the class, `_Central()_` also found in `_Actors.py_`

C.1.1 Settings

The settings available to configure for gameplay are:

- The properties of Cards (Attack, Name, Cost, Money and define custom variants)
- The variety of Cards in the Player Deck for each Player
- The variety of Cards Cards in the Central Deck
- Supplements in the Central Deck
- The number of Cards that constitute a Player Hand for each Player
- The starting Health for each Player
- The Name of each Player

These can be configured in the file `_config.py_`.

Cards Each **Card** has the properties which will be used in the following discussion:

- **Name** (e.g. Archer)
- **Cost**

- **Attack**
- **Money**

This is managed by the class `_Card()_` found in `_common.py_`.

C.2 Game Instances

A **Game** consists of **Rounds**, described later, which continue until either player has below 1 health or there are no remaining Cards in the Central Deck. A Game instance will begin with the Central Active Deck being filled with*5* Cards at random, selected from the Central Deck. If there are less than*5* Cards remaining in the Central Deck, all remaining Cards in the Central Deck (i.e. a number less than*5*) will be moved to the Central Active Deck. The first **Round** will then be started.

A game instance is managed by `_Game()_` found in `_game_engine.py_`.

C.3 Rounds

A **Round** consists of both Players taking a **Turn** in playing the game, one following the other. In the first Game Instance when the code runs, The Player will go first. When it is either Player's Turn, they are referred to as an **Actor**. A Turn for an **Actor**, begins with The Player drawing*5* *Cards* at random from their Player Deck.

Rounds are managed within `_Game()_` in `_game_engine.py_`.

C.4 Turns

At the beginning of a Turn, the Actor has **Attack** and **Money** both set to zero and*5* Cards are drawn at random from the **Player Deck** and constitute **The Player Hand**. If the case that Player Deck is empty which may happen during a Game instance, the Discard Pile will be moved to the Player Deck and players will select Cards from the Player Deck. The Active Hand is empty at the beginning of each turn.

An Actor then has several options in each turn presented by the **Actor UI**:

- Play a Card (Play All Cards is a variant of this)
- Buy Cards
- Attack
- End Turn
- Quit Game (this is only relevant for The Player)

Turns are managed by `_Turn()_` which is similarly named function in both the `_User()_` and `_Computer()_` classes in the file `_actors.py_`.

C.4.1 Playing Cards

The Actor may only **Play** Cards from their Player Hand. When an Actor **Plays** a Card, the **Attack** and **Money** values are added to the current respective totals. The Card is moved from the Player hand to the Active Hand and the Actor is then presented with the **Actor UI** again. Playing All Cards is the same as playing each*5* Cards one at a time.

Playing Cards is managed by: `_play_all_cards()_`. `_play_a_cards()_` within the class `_CommonUserActions()_` in the file `_common.py_`

C.4.2 Buying Cards

An Actor may **Buy Cards**, and is presented with a UI which we will denote, **The Shop***(as explicitly named in the new code)*. In The Shop, the Actor may **Purchase Cards** or return to the **Actor UI**.

Purchasing Cards is when the Actor successfully selects a Card from the Central Active Hand and it is moved to the Actor's own Discard Pile, subtracting the Cost property of the Card from the Actor's Money in the process (denoted as **Buying** the Card). The Actor may only Purchase the Card if the action of Buying the Card will leave the Actor with positive Money.

(Remark: By Purchasing Cards, the Actor then gains a probability of selecting this Card from the Player Deck at the beginning of their Turn when populating their Active Hand. This will occur only when the Player Deck has been emptied since the Purchase, as described in the section describing the beginning of a Turn.)

The Shop for The Player is defined by `_card_shop()_` within the class `_User()_`. The Shop for The AI is defined by `_purchase_cards()_` within the class `_Computer()_`. Both in the file `_actors.py_`

C.4.3 Attack

An Actor may **Attack** the other Player, now denoted **The Defender** for clarity.

1. The Defender's Health becomes the Actor's current Attack, subtracted from the Defenders current Health

2. The Actor's Attack is set to zero

After an Attack, the Actor returns to the Actor UI.

Attacking is managed by `_attack_player()_` within the class `_CommonUserActions()_` in `_common.py_`

C.4.4 End Turn

The action of **Ending the Turn** consists of appending both the Active Hand and the Player Hand to the Player Deck.

(Note that here we do not randomly then select a new set of Cards for the Player Hand but in fact as we have already defined this process, however, the code will conduct this process here)

Ending the Turn is managed by `_end_turn()_` within the class `_CommonUserActions()_` in `_common.py_`

C.4.5 Quit Game

Only applicable to The Player: this will exit the program.

Quitting the game is managed by `_exit()_` within the class `_Game()_` in `_game_engine.py_` Note that should the player force-close the game `_hostile_exit()_` will manage the process smoothly.

C.5 The Winner

The **Winnner** is then determined as the Player with positive Health if they other Player has met the condition of negative or zero Health. In the condition that the Central Deck is empty, the winner is determined as the Player with the greatest Health, else the outcome is a Draw. The Game will then **End**.

The winner is determined by `_end()_` in the class `_Game()_` which is in the file `_game_engine.py_`.

C.6 Game End

Upon Ending, The Player is asked if they would like to **Replay**. A Replay will start with the Actor that would have come next, should the previous Game instance not have ended.

The End of the Game is also managed by `_end()_` in the class `_Game()_` which is in the file `_game_engine.py_`.

D Repository

The repository is found at `https://github.com/flipdazed/SoftwareDevelopment.git` under the user name `flipdazed`. The repository name is `SoftwareDevelopment`

Running the Code The game is run from:

- `cardgame.py` the debug output level can be adjusted in this file

Configuration The settings are configured in

- `config.py` containing a dictionary of game parameters

Relevant Files Other relevant files for the running of the game

- `test.py` unit tests for the base classes and settings
- `game_engine.py` contains the `game()`class that instantiates players from `actors.py`
- `actors.py` The classes for the players and central deck
- `common.py` Common routines inherited by the player and deck classes
- `logs.py` Logging routines
- `game_art.py` Game art including an alternative welcome message
- `logging_colorer.py` Colouring parameters for the logger
- `logging_formatter.py` Formatting for the logger

Other Folders Other folders in this directory

- `docs/modifiedcode/` Modified code flow diagrams and dependencies
- `docs/originalcode/` Original code flow diagrams
- `docs/usertests/` User feedback summary and response sheets

References

[1] Alistair Grant. *Lecture Notes in Software Development: UI Prototyping*. Feb. 2016.

[2] Nick Johnson. *Lecture Notes in Software Development: Software Testing*. Feb. 2016.

[3] *Lecture Notes in Software Development: Code Refactoring and Defects*. Mar. 2016.

[4] *Lecture Notes in Software Development: Development Models*. Mar. 2016.

[5] Alexander McFarlane. *Software Development*. `https://github.com/flipdazed/SoftwareDevelopment.git`. 2016.