Software Development - Planning and Risks

B087965

February 5, 2016

1 Introduction

This assignment represents part 1 of 3 related assignments which form the assessment for the Software Development module. The aim of the assignment is to follow through the software development process starting from a supplied prototype written in Python. The initial assignment is to identify defects in the prototype code, suggest a plan for improving the code and to generate a risk assessment to accompany that plan.

2 Perceived Defects in the Code

At one level, there are no defects in the code. It runs and allows a user to play a game. It serves as a prototype and can be seen as a proof of concept. Thus, although the code does not necessarily conform to software engineering "best practice", it does serve a useful purpose in this regard[1]. However, if we are to assume that the aim of the current phase of the software project is to convert the prototype code into a published (publishable) package[2] then this code is clearly not fit for purpose.

The prototype supplied consists of a single Python script file (`dbc.py`) containing just over 400 lines of code. There is a small class defined to represent a `Card` object at the top of the file, but the remainder of the code is in the form of a single block which is executed when the script file is passed directly to the Python interpreter.

A list of some of the code "defects" follows.

- Limited use of objects

  Apart from the single `Card` object, the remainder of the code is serial, procedural code. Model objects within the game are represented using associative arrays rather than objects. This makes the code less easy to understand and gives rise to other code defect symptoms (see also Magic Strings and Code duplication). The use of objects would help to reduce code redundancy, increase compartmentalisation/encapsulation and will thus make the code more maintainable overall.

- Poor design/use of objects

  The game play is heavily dependent on different types of card. Each card type has a set of characteristics distinctive to that card. However, the code enforces this "by coincidence" - multiple instances of the same card type are created throughout the code with the parameters supplied as constructor arguments each time. If the rules of the game play were changed such that a particular card type was given a different strength, cost or wealth contribution value, refactoring the code would be far harder than should be the case as changes would need to be made at multiple points throughout the code wherever the affected card type was being created.

- Magic Strings

  Because the majority of the objects within the game code are represented using associative arrays and given that such arrays are indexed using arbitrary values, there is no checking of pseudo-object attributes: this renders the code susceptible to subtle "mis-naming" bugs. Reference to a players 'hand', for example, requires that precise string to be typed in multiple places with no way for the interpreter to check the correctness. Whilst Python - as an interpreted language - doesn't provide compile-time checking, defining objects with clearly-named attributes would represent a safer implementation.

- Obscure variable names

[1] Additionally it serves a useful function for the purposes of this course

[2] A. Grant - personal communication

The use of expressive variable names can improve code readability. The code as supplied contains a mixture of expressive names and shorter, more obscure names. For example, the variable `p0` conveys little meaning to the casual observer. Further variables are potentially misleading and should be refactored e.g. a variable called `max` is used on line but is not storing a maximum value in the conventional sense. The `Card` object has a property called `values` which gives no clue as to the original programmer's intent.

- Unnecessary use of an array

  The `values` property referenced above is not only poorly named but is also a 2 element array variable in which the two members of the array do not seem to be related. This property should be refactored into two separate, well-named properties.

- No use of functions

  The entire code is contained within a single `main()` block. Unit testing is therefore next-to-impossible to perform. At the very least, the code should be split into smaller, discrete functions.

- Repeated code

  Large sections of logic are repeated in multiple places within the code block. Again, breaking the code up into separate functions would allow us to reduce the repetition. We can also remove some degree of repetition via a simple re-ordering of the existing code in relation to the main game logic loop.

- No Unit Tests

  As mentioned above, the lack of functions makes testing impossible. The lack of tests is itself a defect. Having a set of unit tests provides confidence that changes made to the code have not had unexpected side-effects and that each tested function is continuing to return the same results for the same given input as it was prior to the change.

- Unused variables

  There are variables defined within the code that are not used. Clutter like this makes the code less clear and are considered to be "code smells". For example, the `clan` property referenced in the `Card` constructor is never used. Nor is the variable `notending` ever read (Lines 143 and 163). Both of these should be removed from the code along with any other instances that come to light during the refactoring process.

- Mismatch with the description of the game play

  The code appears to contain items not listed in the assignment's description of the game play. For example, there is a clear reference in the code to a `supplement` deck which is not mentioned in the description of the game at all. Either the code is defective, or the requirements are incomplete (or both).

- Logic, Model and Presentation code interspersed. No separation of concerns.

  The monolithic nature of the code as supplied means that the code that represents the objects within the game, the code that controls the logic flow and gameplay and the code that interfaces with the player/user is all intermeshed within the same "function". Again, this makes the code very difficult to test and even harder to re-purpose. If we were to attempt to introduce a different interface to this game, for example, it would require significant effort. A more modular code, with Model, View and Logic/Controller code compartmentalised into separate well-defined components would require only an alternate View module to be implemented.

- There is a complete lack of documentation

Good code can/should be self-documenting and there is a strong argument for not over-commenting code as the non-functional comments often become out-of-date and inaccurate/misleading. However, the complete lack of comments makes this code difficult to understand. Once again, refactoring the code into more discrete functions/objects would make this documentation process simpler as the code can then be documented at the function/class level.

3 Pre-coding considerations

3.1 Evolution vs "clean slate" development

The code as supplied is described in the coursework assignment as "a prototype version". There are differing opinions and styles of development that can be brought into play when using prototypes: these range from systems where prototypes are little more than limited sketches and are designed to be thrown away right up to methodologies that incrementally improve and extend the prototype until it becomes the finished product. It is not clear what the original intention of the prototype code is within this project, but given the number of defects and difficulties within the code, it might be a sensible choice to use it solely as a guide to a completely new code rewrite. The new code could be designed and built from the ground up to use object-oriented techniques.

However, on the other hand, the code as supplied *is* functional and provides a concrete base from which a finished product *might* emerge.

3.2 Choice of programming language

Although the prototype code was delivered using the Python interpreted language, no constraints were detailed in the assignment documentation. Since we are considering the adoption of a "green field" approach to the code development, now is an ideal time to consider alternative choices.

Certainly, we are not fully familiar with Python as a language. Our major experience in recent years has been with Java code. In addition, the tooling for developing code alongside extensive unit tests is very mature in Java which will help to ensure rigour in the code development. Furthermore, there is a suggestion that an alternate user interface may be required for the game. If that were the case, then Python code might not be as convenient a start point from which to implement that additional requirement as would be code written in Java which can readily be integrated into a graphical framework.

3.3 Delivery model

The primary choices to be made with regard to the delivery of the will be whether to deliver regular versions of the code as it develops or to simply deliver a single working version at the end of the project. Our personal preference is to deliver successively more complete working versions, each containing more functionality than the previous one. However, this requires regular contact with – and commitment of resources by – the client in order for them to provide sufficiently detailed feedback on the working of the intermediate versions.

3.4 Risk considerations

A complete rewrite of the code would provide an opportunity to switch to a different language. That provides a bonus in terms of developer familiarity and ability to incorporate rigorous unit testing which is a major positive. There is a risk that lack of developer knowledge of Python could slow the project sufficiently badly as to threaten its viability. However, electing to start from scratch also carries with it risks. Certainly developing completely new code requires a greater effort than incremental refactoring of an existing code base. Also the requirement specification is less than optimal, with many ambiguities and apparent missing details (where code functionality exists which is not referenced in the design documentation). Developing anew risks one or more of those implicit requirements being missed. It is clearly a safer option to incrementally improve the current prototype, refactoring and reworking the code until it reaches a functional and maintainable state.

In addition, the machines on which the code must ultimately be deployed do not have the Java maven tooling installed by default which imposes a further development burden creating scripts to download, install and setup the client's environment so that they will be able to build and run the code.

With regard to interactions with the client to assess partially completed versions of the code, we consider this likely to be possible but we would not wish to *rely* on that as part of the critical path in the project.

3.4.1 Decision

Having reflected on the points raised above, and balancing the benefits against the risks we consider the option to rewrite in Java to be too great a risk to the project both in technical terms and in the amount of resource required to successfully complete. We therefore have elected to adopt an evolutionary development model, continuing to work in Python. We will deliver a single working version of the code at the end of the project so as to eliminate the risk associated with lack of client engagement during the development timetable.

4 Work Plan

A Gantt chart detailing the proposed work plan is provided in Figure 1. In calculating the lengths of tasks, we have assumed a single developer resource contributing 10 hours per working week of effort. Tasks have been ordered such that only one unit of developer effort is active at any time. The Gantt chart was generated using the Smartsheet system[3].

4.1 Remove Duplicated Initialisation Section

The code as supplied initialises the human and computer players' hands and then loops round the game play. Once the game is complete, the code then repeats the initialisation code before starting a new game. By modifying the location of the initialisation code we can remove the second (duplicated) version of this block of code.

Time estimate:	1 hour including manual checking
Resource required:	Developer time

4.1.1 Risk considerations

The risks associated with this action relate to mistakes being made in the logic of the refactored code or the unanticipated discovery of side-effects or subtle differences between the two separate copies of the initialisation block. We consider the likelihood of this occurring to be low, although the impact of any such eventuality would be moderate. We propose to mitigate against this risk by careful inspection of the deleted code. The time to do this has been factored in to the time estimate.

4.2 Card object design and usage

4.2.1 Document assumptions about Card characteristics

There is no formal reference list of agreed card types and their characteristics. There are a number of card types defined in the code that are not mentioned in the summary of the card game that was supplied as the assignment. We will extract the full set of cards as used in the code supplied and document them to provide a single definitive reference against which we can develop

[3]http://www.smartsheet.com/

Time estimate:	2 hours
Resource required:	Developer time

4.2.2 Solicit client confirmation of documented assumptions

Having documented the assumed characteristics, we will seek client confirmation and obtain “sign off” against that document.

Time estimate:	Up to 5 days
Resource required:	Client response

4.2.3 Implement Card object design

Using the documented characteristics described above, we will firstly extract the card object into a separate module and then define specific subclasses or instances of cards to represent each card type.

Time estimate:	2 hours
Resource required:	Developer time

4.2.4 Implement Unit Tests for Card objects

Using the documented characteristics described above, we will create test classes to check the values and characteristics of the various card types for correctness and to prevent their accidental modification by subsequent refactorings.

Time estimate:	1.5 hours
Resource required:	Developer time

4.2.5 Refactor current Card object usage

We will replace the current parameterised construction of card objects within the main code with references to the specific sub-classes/instances of card object that we have implemented

Time estimate:	1 hour
Resource required:	Developer time

4.2.6 Risk considerations

Although the card characteristics are not all described in the game play document, they are all defined in plain sight in the code. Thus the risk of failing to identify all the card types is very low. Brief manual inspection of the code suggests that there are no cards defined in different ways in different parts of the code. Therefore there are no ambiguities that require clarification. The sign off from the client is an item that is out of our control and we must allow a reasonable amount of time to achieve this. However, the likelihood of radical changes being required as a result of client input are believed to be minimal.

We intend to develop the card definitions and their associated tests using scripting from the value patterns within the current code. Again the risk posed by this activity is low, even though incorrect card type definitions would have a major impact on the success of the project.

4.3 ‘Player’ object design and implementation

4.3.1 Document assumptions about Player characteristics

Player and computer opponent objects within the code are currently implemented using associative arrays. There is much setup and handling code duplicated between the player and computer objects. This could be reduced by the implementation of a common object. A large part of the player object behaviour is listed in the assignment documentation. However, we would benefit from explicitly stating those requirements in a single formal document. This will also allow us to fill in any gaps in behaviour not covered in the original document.

Time estimate:	2 hours
Resource required:	Developer time

4.3.2 Solicit client confirmation of documented assumptions

Having documented the assumed characteristics and behaviours for the player object, we will seek client confirmation and obtain “sign off” against that document.

Time estimate:	Up to 5 days
Resource required:	Client response

4.3.3 Refactor current object usage

We will code up the newly designed player object using the documented characteristics. We will replace the current associative array implementation with the newly-coded player object.

Time estimate:	5 hours
Resource required:	Developer time

4.3.4 Implement Unit Tests for Player object

Using the documented characteristics and behaviours for the player objects, we will implement a set of test classes to check for the correct operation of the player object against those criteria.

Time estimate:	5 hours
Resource required:	Developer time

4.3.5 Risk considerations

The behaviour of the player object will encapsulate a large portion of the game play and as such, is critical to the success of the project. Misinterpretations or inaccurate assumptions and specifications will have a serious or catastrophic effect on the project outcomes. Failure to obtain “sign off” from the client in a timely fashion may result either in delays to the project or in wasted effort developing code that subsequently turns out to be not required or is incorrect. In addition, the code as written will not lend itself readily to refactoring and this is therefore a complex and difficult task. To mitigate against the risks within this task, we should move the tasks relating to player object requirements documentation and subsequent confirmation by the client to the start of the project. In longer projects, this removes the task from the critical path and allows for slippage in scheduling within this work package.

4.4 'Central' object design and implementation

4.4.1 Document assumptions about 'Central' object characteristics

The 'central' object forms another core part of the game functionality and is – like the player object – currently implemented using associative arrays. In keeping with our move towards object-oriented code, we intend to recode this behaviour in object form. A large part of the 'central' object behaviour is listed in the assignment documentation but the code has been noted to contain extra details not mentioned elsewhere e.g. the 'supplement' deck. Explicitly stating the requirements in a single formal document gives us a reference against which to code and will allow us to fill in any gaps in behaviour not covered in the original document.

Time estimate:	1 hours
Resource required:	Developer time

4.4.2 Solicit client confirmation of documented assumptions

Having documented the assumed characteristics and behaviours for the 'central' object, we will seek client confirmation and obtain "sign off" against that document.

Time estimate:	Up to 5 days
Resource required:	Client response

4.4.3 Refactor current object usage

We will code up the newly designed 'central' object using the documented characteristics. We will replace the current associative array implementation within the main game loop with the newly-coded 'central' object.

Time estimate:	3 hours
Resource required:	Developer time

4.4.4 Implement Unit Tests for Central object

Using the documented characteristics and behaviours for the Central object, we will implement a set of test classes to check for the correct operation of the object against those criteria.

Time estimate:	5 hours
Resource required:	Developer time

4.4.5 Risk considerations

Like the player object, the correct functioning of the 'central' object is critical to the success of the project. Misinterpretations or inaccurate assumptions and specifications will have a serious or catastrophic effect on the project outcomes. Failure to obtain "sign off" from the client in a timely fashion may also result either in delays to the project or in wasted effort developing code that subsequently turns out to be not required or is incorrect. Although not apparently as complex as the player object, the refactoring of this functionality remains a non-trivial task. To mitigate against the risks within this task, we should move the tasks relating to 'central' object requirements documentation and subsequent confirmation by the client to the start of the project to run in parallel with the definition of the player object characteristics.

4.5 User interaction object

4.5.1 Document requirements

The current code contains multiple print statements and accepts input directly from the user assuming a keyboard and terminal interface. If the code is to be extended to offer a different interface then the interactions with the user need to be abstracted out so that the core logic of the game does not need to be modified to permit the new interface to work. In order to implement this properly, we need to define the characteristics required of a generic interface and then implement them in the context of a command-line system. Such considerations are a part of System architecture design and so do not require direct client input.

Time estimate:	3 hours
Resource required:	Developer time

4.5.2 Write code for User Interaction object

Based on the defined requirements, we will implement object-oriented code to fit those requirements

Time estimate:	4 hours
Resource required:	Developer time

4.5.3 Implement Unit Tests against the User Interaction object

We will implement a series of unit test classes to check the behaviour of the newly-developed code against the defined requirements set.

Time estimate:	4 hours
Resource required:	Developer time

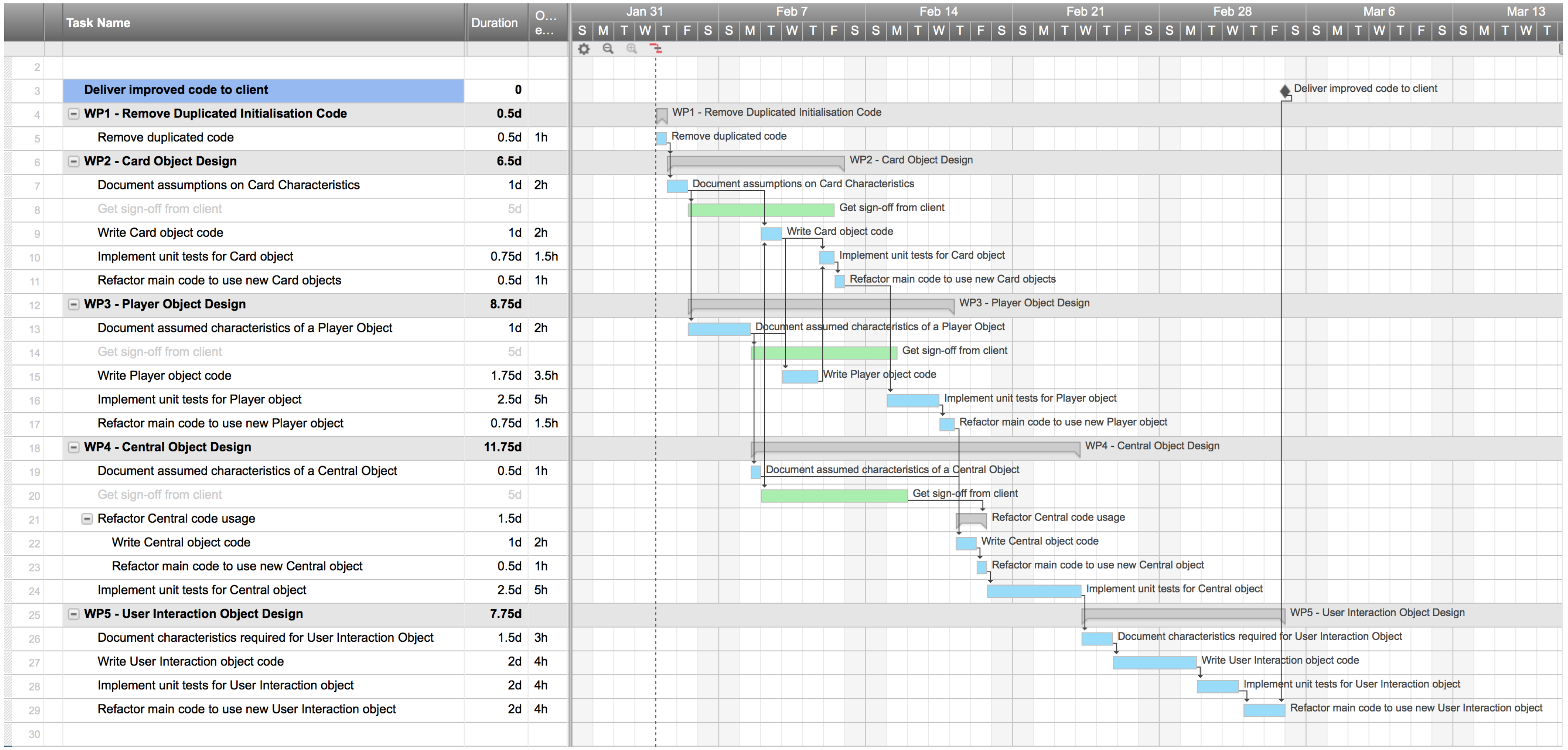
4.5.4 Refactor the main loop to use the User Interaction object

We will modify the main loop code to use the user interaction object rather than the hard-coded command-line/terminal embedded in the supplied code.

Time estimate:	4 hours
Resource required:	Developer time

4.5.5 Risk considerations

User interface code is notoriously difficult to test thoroughly so there is a clear risk that bugs may be introduced during this process. We will attempt to mitigate the risk by designing the code in as modular a fashion as possible so as to maximise the testing possibilities. Ultimately, the option to roll back the code to a version prior to the introduction of the new interface will remain a possibility.



	Task Name	Duration	O… e…
2			
3	**Deliver improved code to client**	**0**	
4	**WP1 - Remove Duplicated Initialisation Code**	**0.5d**	
5	Remove duplicated code	0.5d	1h
6	**WP2 - Card Object Design**	**6.5d**	
7	Document assumptions on Card Characteristics	1d	2h
8	Get sign-off from client	5d	
9	Write Card object code	1d	2h
10	Implement unit tests for Card object	0.75d	1.5h
11	Refactor main code to use new Card objects	0.5d	1h
12	**WP3 - Player Object Design**	**8.75d**	
13	Document assumed characteristics of a Player Object	1d	2h
14	Get sign-off from client	5d	
15	Write Player object code	1.75d	3.5h
16	Implement unit tests for Player object	2.5d	5h
17	Refactor main code to use new Player object	0.75d	1.5h
18	**WP4 - Central Object Design**	**11.75d**	
19	Document assumed characteristics of a Central Object	0.5d	1h
20	Get sign-off from client	5d	
21	Refactor Central code usage	1.5d	
22	Write Central object code	1d	2h
23	Refactor main code to use new Central object	0.5d	1h
24	Implement unit tests for Central object	2.5d	5h
25	**WP5 - User Interaction Object Design**	**7.75d**	
26	Document characteristics required for User Interaction Object	1.5d	3h
27	Write User Interaction object code	2d	4h
28	Implement unit tests for User Interaction object	2d	4h
29	Refactor main code to use new User Interaction object	2d	4h
30			

Jan 31
Feb 7
Feb 14
Feb 21
Feb 28
Mar 6
Mar 13
S M T W T F S S M T W T F S S M T W T F S S M T W T F S S M T W T F S S M T W T F S S M T W T F S S M T W T

Figure 1: Gantt chart of proposed Work Plan

5 Risk Analysis

No.	Risk	Likelihood	Impact	Category	Mitigating Action
R1	Lack of Developer Time	Possible	Severe	High	Conduct regular reviews of progress against schedule to ensure slippage is caught as early as possible. Work nights and weekends to catch up.
R2	Lack of Python Knowledge	Possible	Severe	High	Google provides the answer to 90% of coding problems. Again, review progress regularly to ensure that the schedule is not slipping as a result of lack of previous Python exposure
R3	Failure to obtain critical client sign-off	Possible	Moderate	Low	The majority of the requirements that we will document are based on the actuality of the current code prototype. We have designed the Work Plan schedule to push the client sign-off as early in the process as possible to allow time for slippage. We have also scheduled "pre-emptive" development work to be under way whilst waiting for client feedback. In the absence of client engagement and sign-off, we will fall back to implementing code that functions in the same manner as the current prototype.
R4	Refactoring introduces bugs	Possible	Severe	High	Define clear code properties and behaviour in a requirements document. Build unit tests to enforce the assertions defined in the requirements.
R5	Poor time estimates (schedule slippage)	Possible	Severe	High	As per R1, extra developer time may possibly be found at nights and weekends should it become necessary. In addition, the splitting of the project workload into discrete work packages will allow delivery of a working code block without all packages being complete. Although the code will not be in a complete state, it will be functional at the end of Work Package 4. Thus, we will be able deliver functional code.