CMP-6013Y

Final report marking sheet

Student's name: **Christopher Alastair Irvine**	Student's Reg: **100036248**
Marker's name:	Supervisor ☐ 2nd marker ☐
Title: **CODEX: A Progressive Web App in React for table-top role-playing games**	

	Marks
Introduction, related work and context	/20
Design / Methodology / Experimental plan	/20
Outcome / Analysis / Results	/30
Discussion, evaluation and conclusion	/20
Quality and accuracy of writing	/10
Overall mark	/100

Comments

Signed:	Date:

CMP-6013Y

Presentation marking sheet

Student's name: **Christopher Alastair Irvine**	Student's Reg: **100036248**
Marker's name:	Supervisor ☐ 2nd marker ☐
Title: **CODEX: A Progressive Web App in React for table-top role-playing games**	

	Marks
Explanation of problem and solution	/10
Software design / analysis of investigation / results of experiments	/30
Clarity of slides	/10
Structure of presentation	/10
Oral performance, including timeliness	/10
Evidence of understanding and reflection	/20
Quality of supporting flyer	/10
Overall mark	/100

Comments

Signed:	Date:

Student's name: **Christopher Alastair Irvine**	Student's Reg: **100036248**	
Title:		
Supervisor: **Dr Katharina Huber**	Signed:	Date:
2nd marker:	Signed:	Date:

Marks	Supervisor	Second marker	**Agreed mark**
Report			
Presentation:			

Comments (made available to the student)

Christopher Alastair Irvine

Registration number 100036248

2018

CODEX: A Progressive Web App in React for table-top role-playing games

Supervised by Dr Katharina Huber

University of East Anglia
Faculty of Science
School of Computing Sciences

Abstract

This project is the tits and here's why...

Acknowledgements

I would like to thank Dr Katharina Huber for taking on the supervision of this project, and guiding me towards success. Additionally I would like to thank Wizards of the Coast for their generosity and kindness in allowing the use of their Intellectual Property for this project.

Contents

List of Figures

List of Tables

1 Introduction

hello there

2 What is CODEX?

CODEX was a project that has two components. The first, produce a progressive web app built in ReactJS that was developed using a single developer Agile software engineering methodology, the second component was to evaluate the quality of that methodology. By the end of this section we will have an appreciation of the CODEX app, in terms of both context and purpose (see Sections 2.1).

2.1 Context

The CODEX an app was based around the popular tabletop role-playing game known as Dungeons and Dragons (D&D), created and published by Wizards of the Coast. This game, and the principles that bring the game to life, drove the requirements which formed the basic structure of the CODEX app. Therefore, in order to understand the functionality, principle and importance of the CODEX app, we must first gain an understanding of D&D.

The author of this paper has multiple years of experience with D&D, both as a *Player* and *Dungeon Master* (DM). The CODEX app began as a personal project for the author of this paper, which grew into a tool to evaluate a single developer Agile software engineering methodology.

2.1.1 What is Dungeons and Dragons?

The game of D&D, at time of writing, was a popular tabletop role-playing game where a group of friends engage in a grand fantasy adventure powered by a narrative (Gygax and Arneson (1974), Peterson (2017)). The story would be divided into manageable blocks of time known as *sessions*. Each session may take between two and four hours to run, however this is not strictly enforced and is decided by the group of players. The events of each session, and the larger campaign, was created and controlled by a player

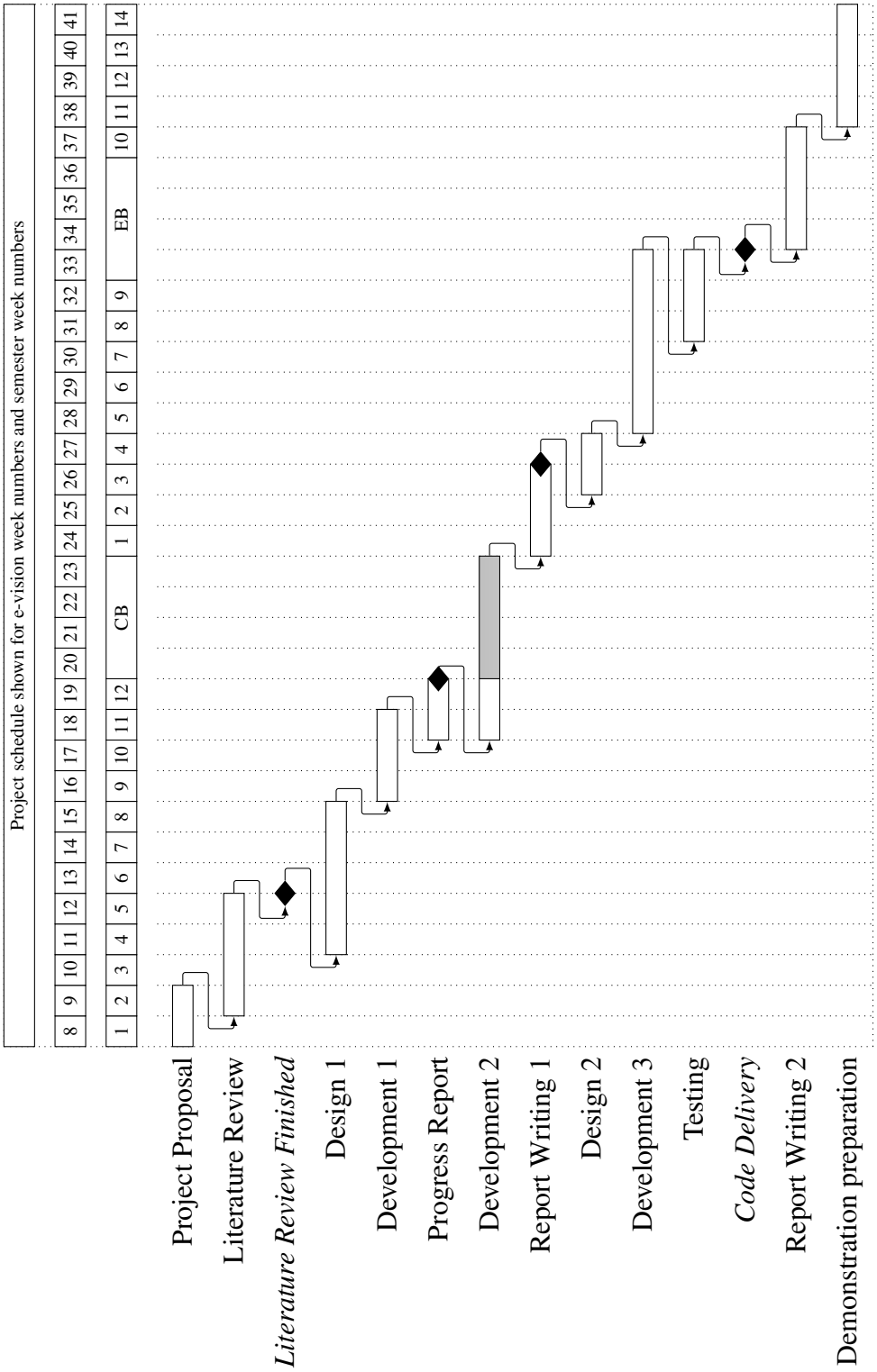


Figure 1: CODEX Gantt Chart, outlining the major tasks and deliverables

known as the *Dungeon Master* (DM) (for more details on DMs please see Section 2.1.2). However, the campaign would not always occur as written. The players, who were clueless as to the next plot in the campaign, might take the story in an entirely different direction several times a session. The DM would have to react to these unexpected inputs and redraw the narrative during each session, or attempt to guide the players subtly back on track (Mearls and Crawford, 2014b).

Whilst it was possible for DMs to purchase pre-written campaigns, published either by experienced DMs or Wizards of the Coast, many DMs would take it upon themselves to create their own worlds in which the campaign would occur or alter the pre-written campaigns to suit the needs of the group. For every game of D&D was inherently different from each other, including those following the same campaign book. Because of this difference it was hard to estimate how many sessions a campaign might take.

The players would form an entity known as the *party*, that would consist of the characters that the players embodied for the duration of the campaign. Each player controls one character, known as the *player character*, within the party. The party would serve as the groups device in how they interact with the world, this is true for the DM who would not embody one character, but every other character within the world. These characters would be referred to ask *non-player characters* and would include both allies and enemies of the party. For that reason, the external perception of the D&D might be that the players were against the DM. Nothing could be further from the truth, as it was the relationship that existed between the players and the DM that would shape the outcome of the campaign (Ewalt, 2014).

Despite the complicated nature of D&D the equipment that was needed to play the game was minimal. A group of players would only have needed a single set of dice, a copy of the core books (*Dungeon Master's Guide (Mearls and Crawford, 2014a), Players Handbook and Monster Manual (Perkins, 2014)*), a set of character sheets (1 per player), pens and paper. Everything else, such as more dice, a *Dungeon Master Screen*, Campaign Books, Battle Mats and Tokens were supplementary. The dice that was used for D&Dare as follows; twenty-sided, twelve-sided, two ten-sided (one in increments of ten, the other in increments of one), eight-sided, six-sided and four-sided. Figure 2a is an example of what a D&D table might have looked like, whereas Figure

(a) A typical D&D set up. From left to right: DM note pad on top of a couple of official Wizards of the Coast core books (*Monster Manual* shown), set of dice, battle mat and tokens, a player character sheet, player's set of dice and the Wizards of the Coast *Players Handbook*

(b) An example set of dice that was needed to play D&D. From left to right: 4-sided, 6-sided, 8-sided, 10-sided, 12-sided, 20-sided and a 10-sided dice in increments of 10s

Figure 2: Examples of the typical equipment used in a game of D&D.

2b is an example of the different dice used for D&D[1].

Throughout the campaign the party would have to face many challenges designed by the DM. These challenges might be full scale battles involving multiple powerful enemies and allies, to small scale skirmishes between only a handful of characters. Alternatively challenges could be a battle of the wills where the party have to negotiate themselves out of a tight situation with the law to passing through a magical trap set by an ancient being in order to acquire an item of treasure. The effectiveness in which the party would handle these situations are decided through a simple mathematical system.

Every character within a D&D world had a set of *attributes* that would dictate how that character interacted with the environment around them. These attributes are; *strength*, *dexterity*, *constitution*, *intelligence*, *wisdom* and *charisma* (more details of these at-

[1]When discussing these dice further in this paper, they will be referred to as dX where X is the number of sides the dice has (for example the 20 sided dice is referred to as *d20*). The exception to this is the secondary d10 (which has increments of 10) which is referred to as the *d100*.

tributes are available in Appendix **??**). These attributes were rated between 1 and 20, with 20 being the highest a character could achieve without magical means. The score would be divided by 10, rounding up to the nearest whole number, creating an *ability modifier* which would be applied to the role a dice (this number could be negative) If a character was highly skilled in a particular task, then a *proficiency bonus* would also be applied. This calculation is summarised in the below equation 1.

$$skill\ check = d20 + ability\ modifier + proficiency\ bonus \tag{1}$$

These skill checks allowed the players to interact with the world, via the player character, in a quantitative manner. The higher a skill check was the better that character performed in that challenge. Challenges would have a *difficulty class*, an arbitrary number which indicates the level of skill (or luck in some cases) needed to perform that task. If the challenge was to strike an enemy, it would be easier to wound someone wearing no armour than it was to wound a skilled warrior in plate armour.

Winning a game of D&D was no simple manner either, as there were no set win conditions. It depended entirely on the campaign that was being ran by the DM. A typical example might be that the party were a group of loyalists who were tasked with overthrowing a usurper and restoring the rightful monarch to the throne. That party would *win* where they to do so, however the objectives might change where the party to learn of some new, unsavoury, information about their beloved monarch. A campaign might end without achieving the original goal, or shorter campaigns may end with the start of a new story for the party. It all depended on the attitude of the players towards the game.

For a short, worked example of D&D please refer to Appendix **??**, where a scenario is explored and explained.

2.1.2 How does a Dungeon Master differ from a Player?

In Section 2.1.1, we explored some of the principles that gave D&D life, however the most important principle to the CODEX app was the difference between a DMs and Players. As briefly explained above, a DM was a member of the group who would control the narrative of the campaign and dictate what was occurring during each ses-

sion. Whilst the DM may write the campaign narrative themselves and would of had a great deal of control over it, they would not own the story. That belong to the group as a whole. Player input during the sessions shaped the world and story as the party progressed through the campaign, forming relationships with characters that the DM portrayed.

It was the responsibility of a DM that the party was enjoying the campaign, narrative and challenges. As a result the DM would had to have known every intricacy of the world at any given time, or be able to improvise when they did not know the answer. Only a DM could bring a world to life. Therefore the amount of information that a DM would have to deal with would have been immense. Every DM would have a folder (either physical or digital) full of notes on places, people and events that resided within the DMs world. This folder would not only contain the events of the past, but plans for the future. Skilled DMs would rely on their folder more than any other resource during a session to ensure the correct reaction to party input. The maintenance of a DM folder was costly in both time and effort (Mercer, 2016). Whilst there was no definitive time spent in order to prepare for a session, DMs could easily spend over ten hours a week planning their D&D campaign (Holmes, 1980).

However, DMs were part of the game just like the players were. D&D was never the party against the DM, instead it was a symbiotic relationship where they would explore the world together and share the experience. Whilst a DM knew what was planned to happen next, it was decided by the players if what actually occurred.

2.1.3 Why was the CODEX app developed?

As mentioned in Section 2.1, CODEX began as personal project for the author of this paper. The original functionality of the app was to be a platform for tracking combat during games of D&D. Since then the functionality has grown to include a note taking system and several other minor features that will be discussed in Section 4. Whilst there are several other systems (explored Section 3.1) that assist a DM, they often only perform one task. The CODEX app aimed to be the assistance app for DMs by possessing a wide range of functionality.

DMs spend a lot of time and effort in running their games of D&D(as discussed in

Section 2.1.2). The CODEX app attempted to remove some of the more laborious tasks from DMs and put more enjoyment into the game by removing some of the pressure. As of the time of writing, the principle functionality the CODEX app was to track combat and provide an interactive database of information about the world for quick reference.

3 Related Work

In this Section we will explore the work that was pertinent to the CODEX project. We will have a greater appreciation of the CODEX app, by evaluating the similar systems available at the time of writing, and and gain an understanding of the Software Engineering discipline in addition to the Web-App architecture through the review of academic works.

3.1 Dungeon and Dragons Apps

At the time writing, there were several applications that existed to assist in the running of D&D. These apps had varying levels of functionality, however they differed from the CODEX app. For example, the official D&D Beyond app (Wizards of the Coast and Curse, Inc., 2018) (discussed further in Section 3.1.1), had a huge range of functionality including digital character sheets and a quick reference database. This service is aimed towards the players of the game, not towards the DMs, therefore D&D Beyond does not support session planning and encounter generation to the same level as the CODEX app.

3.1.1 D&D Beyond

D&D Beyond was a web app that was developed by *Curse, Inc.* and published by Wizards of the Coast (the creators and owners of D&D). The purpose of D&D Beyond was to digitise the experience of D&D without removing the table-top, social aspect (see Section 3.1.2). However, D&D Beyond was not just a web-app that held digital records of a User's character. There was an additional emphasis on community and sharing between the Users through the use of Forums and the ability to publish individual content through the site.

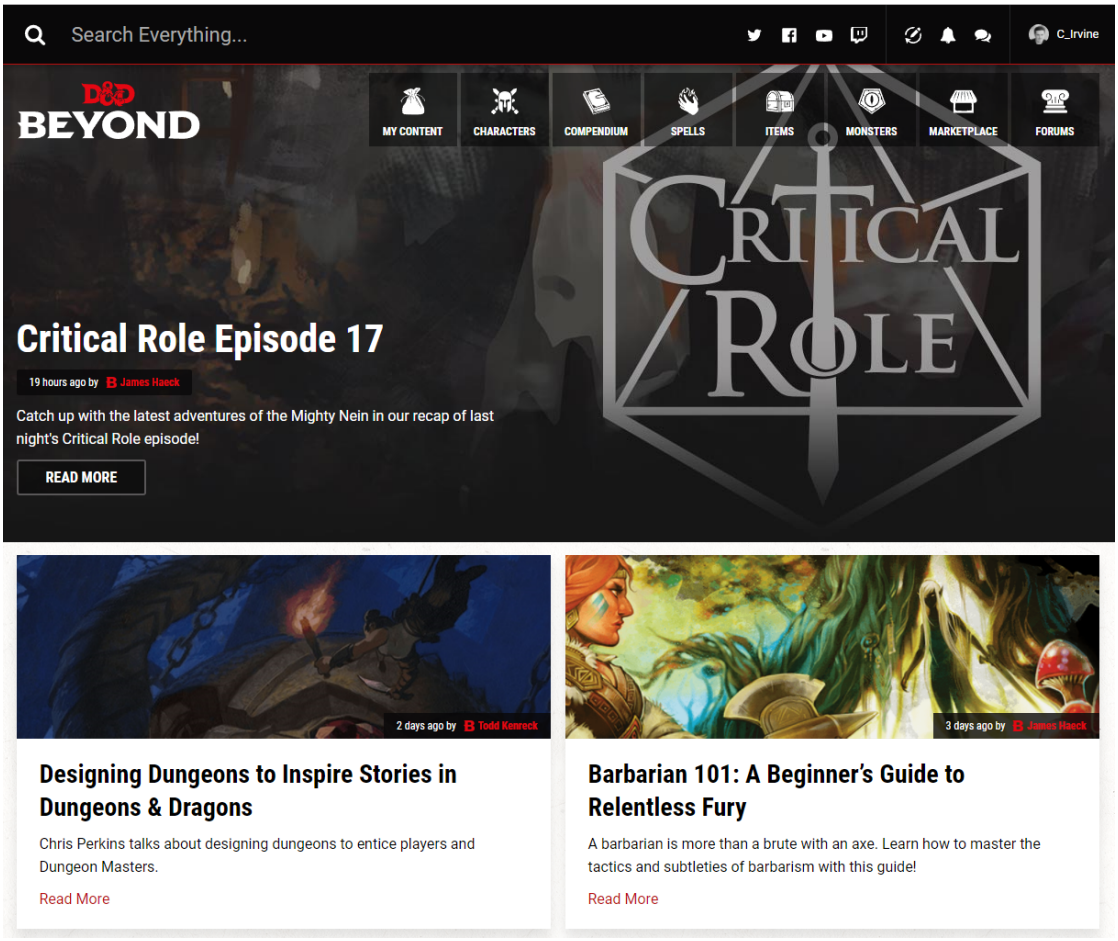
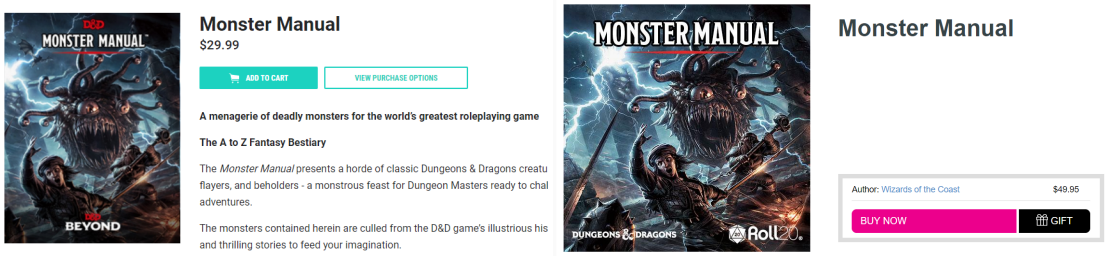


Figure 3: A screen shot of the D&D Beyond web-app homepage (05/05/2018), showing the various functionality of the service. We can see several articles (with more off-screen); separate sections for personal content, characters, compendium (campaign section), spells (see the Extended D&D Explanation in portfolio for more details on spells), items, monsters, marketplace (where users may purchase electronic books to unlock content) and forums in the navigation bar. Additionally there is search, social media, chat and account functionality in the header of the screenshot.

Figure 3 is a screenshot taken from the D&D Beyond homepage. Whilst a full description of the functionality of D&D Beyond is available in the Design Document held within the project portfolio, we should understand some of the key features of the ser-

vice. Firstly, the ability of digitally store a character, with a full description of abilities and spells attached to that character was critical to many players, who were frustrated at the constant searching through the core rule books of the game frustrating. Secondly, D&D Beyond gave the community the ability to directly share and use content created not by Wizards of the Coast but by other players of the game. Whilst this was happening on alternative sites such as the Reddit D&D Forum (Various and reddit inc, 2018) (a popular forum website), D&D Beyond brought the community back to Wizards of the Coast and centralised it. However, one of the largest downfalls of D&D Beyond was the compulsion to purchase the Wizards of the Coast books through the web-app, even if the user might own a physical copy already. Only by purchasing the book (potentially for a second time) could the user access the majority of the content on the web-app (see Figure 4a).



(a) D&D Beyond *Monster Manual*

(b) Roll20 *Monster Manual*

Figure 4: Screenshot taken from the D&D Beyond and Roll20 web-apps showing the option to purchase a digital copy of the *Monster Manual*, unlocking the relevant content

3.1.2 Roll20

Roll20 (The Orr Group, LLC, 2018) was a popular web-app that was developed to enable games of D&D to function without the need of a group of people coming together at a table. Instead, the group would use Voice over IP (VoIP) technologies (such as Skype or Roll20's internal VoIP system) to communicate and the Roll20 app would serve as the gaming table (see Figure 2a).

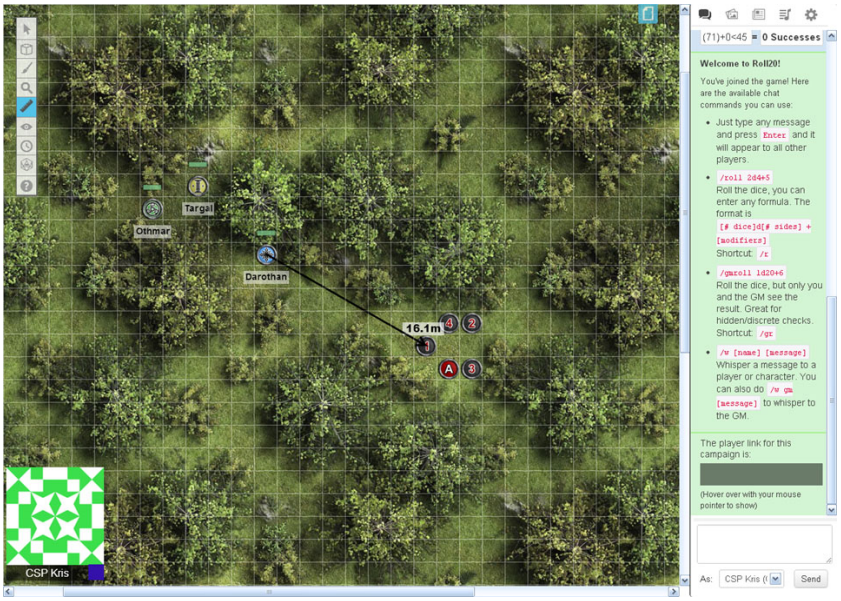


Figure 5: A screenshot taken from the Roll20 web-app showing the table-top, text and voice chat functionality (Richards and The Orr Group, LLC, 2012).

Figure 5 shows a typical Roll20 virtual table-top in use. In comparison, we can see that the grid in the screenshot is very similar to the battle mat shown in Figure 2a. The DM of a group can use Roll20 to plan encounters and scenes for the party, and illustrate certain points to greater effect than at a physical table. All dice rolling can be performed from the Roll20 chat system, with graphical dice rolling across the virtual table-top. Without the Roll20 web-app some groups would not be able to function. However, for some groups the removal of the personal touch of meeting in person was not acceptable.

Roll20 was more than just a virtual table-top, as a quick reference database could be accessed through the purchase of Wizards of the Coast books in a similar manner to D&D Beyond (see Section 3.1.1), as demonstrated in Figure 4.

3.1.3 Kobold Fight Club

Kobold Fight Club (KFC) (Toltz and Barzilai, 2016), named after one of most popular monsters in D&D, was a website that assisted many DMs in creating and balancing encounters with enemies during a campaign. KFC only had a single function at the time of writing, which was the encounter planning. KFC was an example of a website

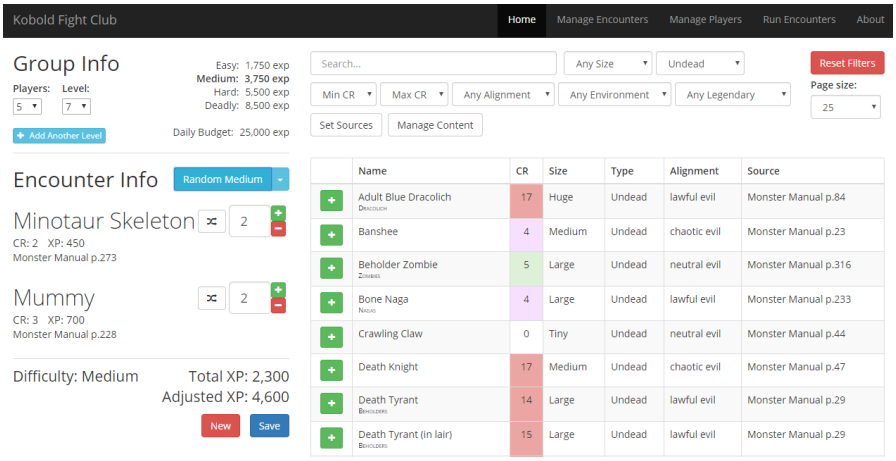


Figure 6: A screenshot showing an encounter designed using the Kobold Fight Club website, a popular D&D encounter building tool

that provided a single service to an excellent standard with a clean UI that was easily understood, as shown in Figure 6. KFC does have some weaknesses, in order to make the service free and open to all the developers could not include the attributes of each monster. Instead they can only provide a page reference for that monster.

3.2 Software Engineering

The CODEX project was not only the development of a web-app based around the D&D game. It was the evaluation and examination of an agile single developer software engineering methodology. In order to perform this evaluation, we must first have an understanding for the Software Engineering Discipline (see Section 3.2.1). From there we will learn about the Agile principles and how they changed Software Engineering (see Section 3.2.2). Finally, we will examine two potential single developer agile methodologies which would be compatible with the development of the CODEX app (see Sections 3.2.3 and 3.2.4).

3.2.1 What is Software Engineering?

As stated multiple times throughout this paper, CODEX was not only the development of the CODEX app. CODEX aimed to evaluate an agile software engineering methodology

that was suitable towards the single developer. In this subsection we will learn what Software Engineering was, at the time of writing, before gaining an understanding of the Agile Principles (see Section 3.2.2).

> "The systematic application of scientific and technical knowledge, methods and experience to the design, implementation, testing and documentation of software." – Association et al. (2010)

As explained in the definition above, Software Engineering was application of engineering practices to the development of Software, in order to produce the best Software solution to the proposed problem. These practices were reduced down to a series of stages that each project would be expected to progress through at least once during the project life cycle. Software Engineering is applied to a project is through Software Engineering Methodologies (SEMs). Please note that these stages may have different names depending on the selected SEM, however generally they were known as:

- Requirements analysis
- System Design
- Implementation or Development
- Testing
- Deployment
- Maintenance

Requirements analysis (sometimes called requirements engineering) was when a developer (or team of developers) would discuss the problem with the client in order to create a well defined list of requirements that a potential software system would need to fulfil in order to be a success. Later that list would be prioritised into four lists using a MoSCoW analysis (see Section 4.2 for the CODEX MoSCoW analysis). Those lists being must have, should have, could have, won't have. Several other investigative techniques may be used during the requirements analysis stage of a project, sometimes

referred to as *Systems Analysis*, such as Rich Pictures and Stakeholder wheels. By the end of a requirements analysis stage the project will have a clear set of criteria that it has to meet in order to succeed.

The *System Design* stage was closely related to the requirements analysis stage, they were sometimes merged together by some methodologies. During this stage the developers would design the proposed software system. This does not only mean the graphical look of the system; but the architecture that organises the classes, the methods that populate them and how information is transferred to different parts of the system. This stage would produce a document known as the *Design Document* (see Section 4.2). This document would give future developers the information necessary to contribute towards the development of a system and would be regularly updated throughout each iteration of a project.

During the *implementation (or development)* stage of a project, the developers would build the system as designed from the previous stages. Depending on the SEM being used by the development team, there might be the ability to revisit the requirements and design stages in order to fix unforeseen design errors or adjust to changing client demands.

The *testing* stage of a project may occur after or during the implementation stage, depending on the applied SEM. Developers who practised *Test Driven Development* (TDD) would have a large base of automated tests that would check a large percentage of the code regularly (Astels, 2003). That percentage is known as *code coverage*. Regardless of whether TDD was used during the implementation stage the developers would still have to test the system themselves. This testing would involve the developers attempting to find bugs and/or security issues with the system. Eventually letting individuals (such as prospective users or experts in software security) test the system in order to get feedback and different opinions on the current system. One such method of testing using these individuals might be a *Think-Aloud Evaluation* (Nielsen et al., 2002). Any issues found during this stage would be corrected by the developers (El-Far and Whittaker, 2002).

The *deployment* stage of a system would be a crucial step in any project. Developers had several processes to choose from when deploying their system; direct, parallel,

phased or piloted distribution. Each has their own strengths and weaknesses. Different systems would benefit greater from different deployment processes.

Once a system has been deployed, the maintenance stage may begin. Whilst during this stage additional features may be implemented into the system, they would fall into a life cycle of their own. Maintenance stage activities are limited to the identification and correction of any errors within the system, and it would have no definitive time scale. As long as the system was running, regular maintenance would have to be performed.

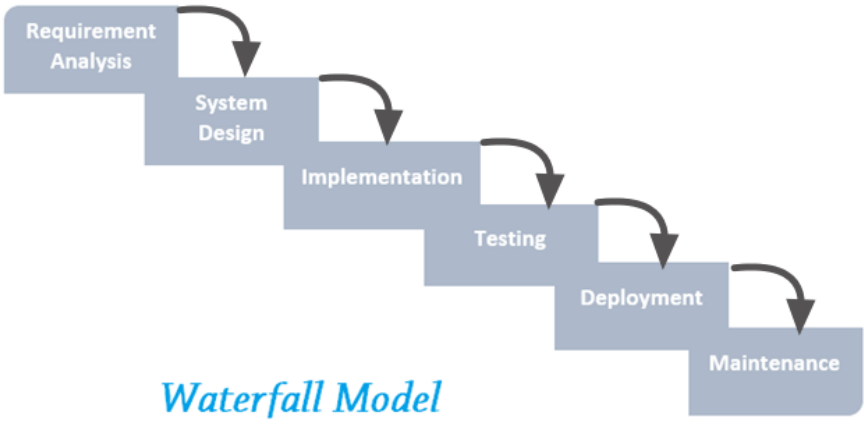


Figure 7: Flow Diagram showing the linear progression of a Waterfall Project.

There are a variety of SEMs that have been tried and tested by several projects, with each possessing strengths and weaknesses. The earliest SEMs followed a *Waterfall model*, where each stage would be completed in a linear sequence (see Figure 7. The popularity of Waterfall models arose from the simplicity they brought from a management view. The lack of iterations allowed a project manager to *"cross off"* each section in sequence and move onto what was next. The industry would eventually realise that the rigid linearity of the Waterfall model prevented the developers adapting to changing client demands.

Iterative SEMs were created, such as *Spiral Model*, that formed the basis for the Agile Principles to be created (see section 3.2.2). The Spiral Model, as seen in Figure 8, follows an iterative design that condenses the system life cycle into four distinct phases: planning, risk analysis, engineering & execution and evaluation. Within the planning

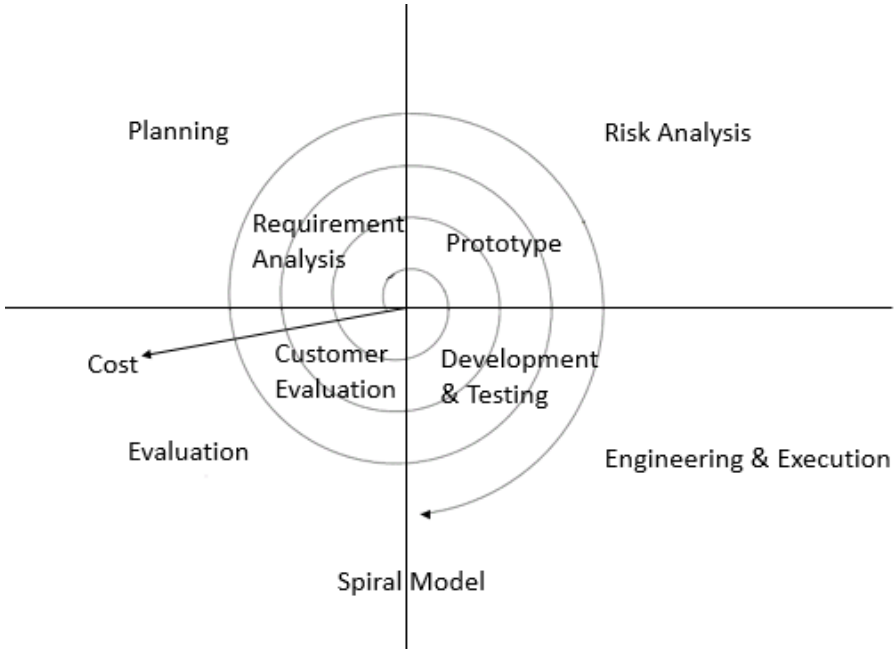


Figure 8: Flow Diagram showing the iterative design of the Spiral Model.

phase the activities that were usually be associated with the requirements analysis and system design stages would be found. Risk analysis, an additional stage added by the Spiral Model, would ask the developer to prototype and examine the designed system to attempt to reduce the number of errors that would need fixing further into the project – reducing costs. The engineering & evaluation stage would see the approved system built and tested, before finally being deployed just prior to the evaluation stage where the users would provide feedback. That feedback was then incorporated into the next iteration of the project, where the spiral would begin anew.

The CODEX app development has followed a methodology, discussed in Section 3.2.3 and was described as a software engineering project. This means that the development followed the systematic application of engineering practices, managed by the Agile Solo methodology.

3.2.2 What is Agile?

Agile was a set of ideas and principles that would shape the way in which software was developed. The creators of Agile were frustrated with how software was built in the 1980 and 90s. Before Agile, a linear approach was taken (see Figure 7) towards software development. For example, once a set of requirements for a system was decided, it could not be changed because the project management could not allow it. Another frustration was the lengthy and complicated contracts that bound both client and developers to a proposed system. These are but a few frustrations that were felt by the creators of Agile, who wanted to "... uncover better ways of developing software ... " (Fowler and Highsmith, 2001).

What was produced by the desire to produce better software was known as the *Agile Manifesto*, at the core of which were four values:

1. **Individuals and interactions** over processes and tools
2. **Working software** over comprehensive documentation
3. **Customer collaboration** over contract negotiation
4. **Responding to change** over following a plan

The Agile Manifesto additionally states "That is, while there is value in the items on the right, we value the items on the left more.". This distinction is critical to Agile, as the creators recognised that the traditional software engineering methods held a value and a place in the future. However, change must be embraced in order to keep up with the demands of that same future.

With the release of the Agile Manifesto, many iterative SEMs became more popular and replaced the linear waterfall models as the industry standards. Scrum and Kanban are two such SEMs.

The Scrum methodology was governed by a "... simple set of roles, responsibilities and meetings that never change." (James, 2017). The roles within a scrum team are *Product Owner*, *Scrum Master* and *Team*. The Product Owner may be a client or an executive who guides the scrum master and the team towards creating the product that

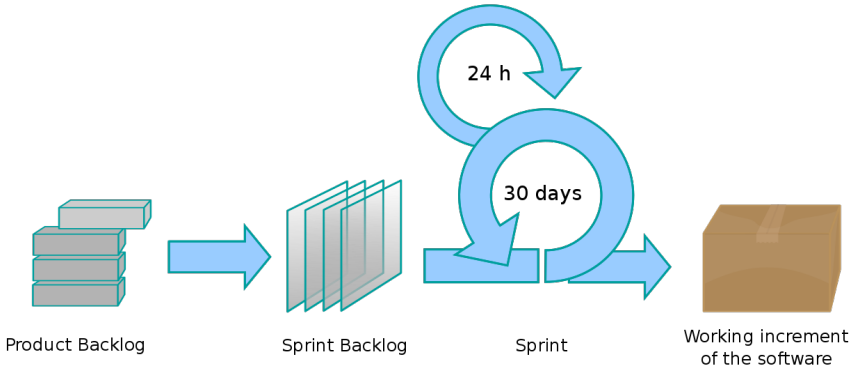


Figure 9: Flow diagram detailing the stages of a Scrum *sprint* cycle. At the beginning of each *sprint* the Team and Scrum Master would select a set of tasks from the *Product Backlog* and move them into the *Sprint Backlog*. This occurs during the *Sprint Planning Meeting*. During the 2-4 week *sprint* the Team would progress through the *Sprint Backlog* tasks, holding a tightly regulated short *Scrum Meeting* daily. At the end of the *sprint* the Team would have produced a potentially shippable increment of working software.

should be delivered. Technical expertise was not required and the Product Owner should not be involved in the management of the development process. The Scrum Master was the facilitator towards both the Team and Product Owner, meaning that this individual would remove any impediments preventing the progress of the project. Scrum Masters do not manage the Team however as the Team is utterly self managing. There were 3 to 6 members in a Scrum Team that would select the work to do every *sprint* in a *sprint planning meeting*. Teams would contain a mixture of professionals including but not limited to software engineers, architects and UI designers (Schwaber, 1997).

Scrum projects were organised into *sprints*. At the beginning of each *sprint* the Team and Scrum Master would select a set of tasks from the *Product Backlog* and move them into the *Sprint Backlog*. This occurs during the *Sprint Planning Meeting*. During the 2-4 week *sprint* the Team would progress through the *Sprint Backlog* tasks, holding a tightly regulated short *Scrum Meeting* daily. At the end of the *sprint* the Team would have produced a potentially shippable increment of working software. This increment would

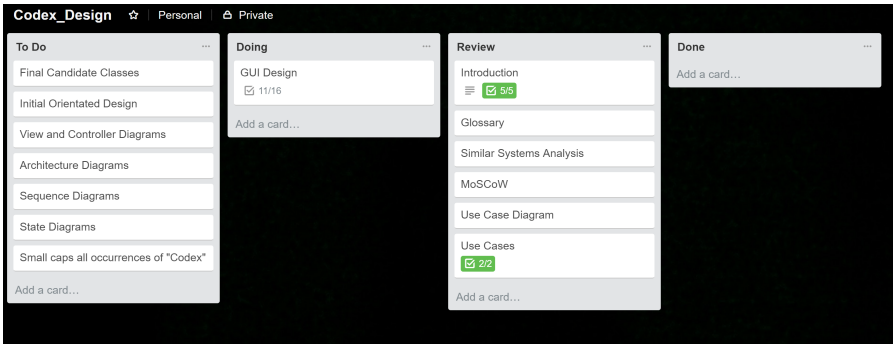


Figure 10: Screenshot taken from the early design stage of the CODEX app. Kanban Board developed using the Trello app (Software, 2011) and includes an optional Review column.

be scrutinised in the *Sprint Retrospective Meeting* which evaluated the work done, and any large scale issues would be solved before beginning the sprint anew with another *Sprint Planning Meeting*.

Whilst some projects would only use Scrum during the implementation stage of a project, perhaps within an iterative Spiral model (see Section 3.2.1 and Figure 8), other project would choose to use Scrum throughout the project life cycle as to prevent confusion through the use of multiple different SEMs.

Kanban was another popular Agile SEM, which whilst closely related to Scrum was a lean methodology rather than iterative. Despite this difference, Kanban was often used in conjunction with iterative SEMs, the most common being Scrum (Anderson, 2010). Kanban was operated by balancing the demands of a project with the available capacity of the development team, with the ultimate goal of reducing the effect of system level bottlenecks. This was achieved through the use of a Kanban Board (see Figure 10), which may have been either physical or digital. Kanban board must include the following columns: *to do* (sometimes called Backlog), *Doing* and *Done* at a minimum. However, additional columns such as; *plan*, *test* (or review) and *deploy* are recommended. The task *tickets* would move from left to right through each of the columns on the board, eventually clearing the Backlog of tasks. Some users of Kanban might have colour coded tickets in accordance with different type of tasks; pink for development tasks and yellow for design tasks for example. Customisation of the tickets

can be taken further when developers apply arbitrary levels of difficulty to the tickets in order to gauge the length of time taken to complete that task.

Kanban projects would not have to go through the regular planning that Scrum projects experiences, because Kanban project were not organised into Sprints. However, Scrum can benefit from the inclusion of Kanban boards to organise and manage the Product and Sprint backlogs.

3.2.3 Agile Solo

Agile Solo was a SEM developed by Anna Nyström in June 2011. The methodology was created using the values stated by the Agile Manifesto (see Section 3.2.2) and aimed to adapt the established working practices of existing Agile SEMs. However the definitive feature of Agile Solo is that the methodology was able to be implemented and used by a single developer.

Nyström selected the suitable components from many different existing Agile SEMs, such as the Sprint Cycle from Scrum (see Figure 9), the Kanban Board (see Figure 10) to name but a few. However, Nyström also realised that single developers rarely work in isolation as typically there would be a supervising individual assigned to the project and a client who would own the project, so adaptations were made to accommodate such individuals.

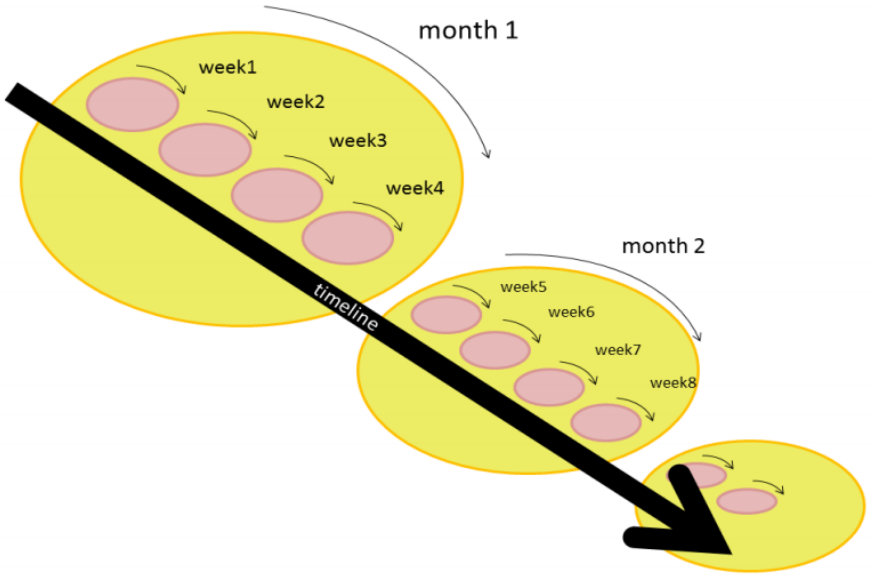


Figure 11: Diagram to represent the Agile Solo Software Engineering Methodology. The project is segmented into *monthly cycles* that would contain *weekly sprints*. The developer would review the work done at the end of each sprint, with the customer and supervisors reviewing the work at the end of each cycle. The monthly cycle would begin anew with feedback from the customer and supervisor to be incorporated into the project.

The resulting methodology was called Agile Solo and revolved around monthly cycles with weekly sprint iterations. At the end of every sprint the developer would review and test the work done, whilst at the end of each cycle the customer and supervising figures would review and test the system providing feedback to be incorporated in the next cycle. This is demonstrated by Figure 11. Nyström recommends that the product backlog should be managed by a Kanban Board and the software be implemented using Test Driven Development (TDD).

Agile Solo was selected to be the SEM for the development of the CODEX app (see Section 4.1), and shall be evaluated later in this paper (see Sections 5.2 and 6.3)

3.2.4 XP for One

Extreme Programming (XP) was a SEM which would attempt to improve the software quality through an iterative development loop that emphasised responsiveness to changing customer requirements. Typically paired with programming practices such as TDD and pairs programming, practitioners of XP believed that the only important product of development was code (Beck and Gamma, 2000).

Figure 12: Diagram to represent the Extreme Programming iterative life cycle.

As seen in Figure 12, XP was highly iterative, with each task within the SEM having an iteration cycle. XP for One was an untested development methodology which had one critical difference from traditional XP, the removal of Pair Programming (pair negotiation in the diagram) (ExtremeProgrammingChallenge, 2006).

In the event that Agile Solo (see Section 3.2.3) was discovered to be unsuitable towards the development of the CODEX app, XP for One was to be the replacement SEM.

3.3 Web App Technology

The CODEX app was designed to be a Web Application (web app). In this Section we will learn what a web app is and how it differs from the traditional website and application (see Section 3.3.1). ReactJS is a popular library for the JavaScript programming language that was developed specifically to develop web apps. Due to the popularity and availability of supplementary materials of ReactJS, it was selected to be the principle programming language for the CODEXApp (see Section 3.3.2). Semantic UI (User Interface) is a package developed for ReactJS systems, which would allow developers to quickly create professional UI for web apps and was used in the development of the CODEX app (see Section 3.3.3).

3.3.1 What is a Web App?

With the rising popularity of applications in the early 2000s, web developers understood the need to reinvigorate websites with new technologies to bring websites into the modern age. This resulted in the creation web-apps. A system that could operate as an separate application on a computer or mobile device, that could still be accessed through a browser (such as Google Chrome). Today web apps exist in between applications and websites (Fling, 2009).

The CODEX web app will follow the web app three tiered architecture that was established with the technologies emergence. As we can see from Figure 13 the first layer (Presentation Tier) is held locally on the Client's machine. This is what is known as 'Client-Side'. Here HTML, CSS and JavaScript is used to translate data into a format the Client will understand. The second layer (Logic Tier), here languages such as ReactJS perform functions on raw data to generate web content. Traditionally the Logic Tier is hosted on a Server that Clients connect to, this is the 'Server-Side' The third layer (Data Tier) communicates with the Logic Tier through SQL Queries (De Groef, 2016).

3.3.2 ReactJS

ReactJS was a JavaScript library developed by Facebook, Inc. after the acquisition of Instagram (Facebook, Inc., 2017). The library provided a set of components, and the

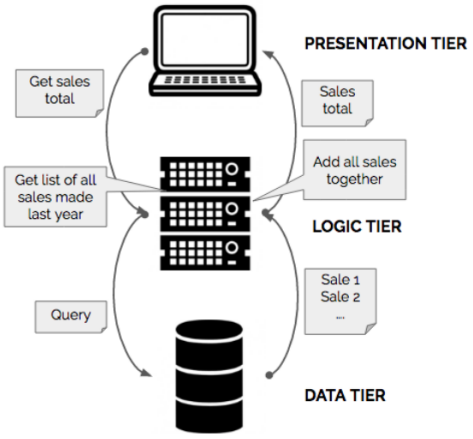


Figure 13: Diagram showing the flow of information between the three tiers of web app architecture. The *presentation layer* allowed the user to interact with the system (and vice versa). The *logic layer* would host the functionality of the system and generated the GUI. Finally the *data tier* would hold the database that supported the web app.

ability to create components, that would be rendered by the ReactDOM. This allowed developers to quickly build user interfaces that were highly customisable without the use of traditional HTML components (Horton and Vice, 2016). As ReactJS was developed around the JavaScript programming language there was pre-existing functionality support. This meant that entire websites and web apps could be developed in JavaScript and ReactJS (with a little CSS for styling).

The CODEXweb app was developed using the ReactJS technologies, as it was one of the most popular web app languages available at the time of writing (Thinkwik, 2017).

3.3.3 Semantic UI

One of the strengths of JavaScript technologies was the incorporation of *open source packages* that would supply a programmer with a set of methods that could be called upon during development. The purpose of including packages instead of developing the methods personally is to save time and reduce costs. Packages allowed developers to focus on the more important tasks (Flanagan, 2006).

Semantic UI was a package whose purpose was to provide a sophisticated set of UI elements and collections that can be easily styled through the use of CSS (Thomason, 2018). Many large organisations (Amazon, Netflix and Microsoft for example) had chosen the Semantic UI package to build their websites. The CODEX web app was developed using Semantic UI elements (see Section 4.3).

4 Development and Implementation

In this Section we will examine how the Agile Solo methodology (discussed in Section 3.2.3) will be implemented into the CODEX project and later evaluated in Sections 5.2 and 6.3. Next we will gain an understanding on the design of the CODEX web app through the extensive Design Document that can be found in the supporting materials. Finally, we will expand upon the knowledge gained in Sections 3.3.2 and 3.3.3 to gain an appreciation of the development of the CODEX web app by dissecting some examples of source code.

4.1 Using Agile Solo

As explained in Section 3.2.3, Agile Solo was centred around the idea of *weekly sprints* encompassed by *monthly cycles* (see Figure 11. Whilst there was no recommendation by Nyström for the management of the project backlog, the CODEX implementation of Agile Solo will utilise the Product Backlog/Scrum Backlog system favoured by Scrum (see Section 3.2.2). At the end of each sprint the developer (the author of this paper) would test the CODEX system as it stood, in order to catch any issues that made it past the automated tests. A fortnightly meeting would be held with the supervisor of the project in order to ensure that the development of CODEX remained on track. At the end of each cycle the current state of the CODEX web app would be presented to a small group of users in order to ascertain the quality of the development done.

The feedback generated from the supervisor and user meetings was stored within the project journal (found in the supporting material) and later incorporated into future developments, where possible. Also found in the journal, would be the initial evaluations

and opinions of Agile Solo from the author of this paper as the development of the CODEX web app progressed.

4.2 Design

The CODEX web app was designed through the creation of a Design Document [2], which was briefly discussed in Section 3.2.1. There we learned that a Design Document should contain a mixture of diagrams and analysis in order to convey to a new developer the design of a software system (McElrath, 2007). These would include the analysis of requirements and similar systems. With diagrams to explain how the system was intended to be used, the GUI design, Class and Architecture Designs with optional additional diagrams that would demonstrate the flow of information throughout select use cases.

The requirements analysis would consist of listing, in detail, the primary functions of the system. The CODEX Design Document elected to use bullet points in order to promote concise descriptions of the functionality. Bullet points additionally allowed the author of the Design Document to label certain functions as sub-functions. For example; the “Account Creation" functionality has detailed under it that there should be “Two ‘levels’ of accounts - Player and Dungeon Master" within the system. From this detailed list a *MoSCoW analysis* can be performed where the developer would designate which functionality was critical to the success of the system and which functionality would not be. MoSCoW contains four categories of functionality; Must Have, Should Have, Could Have and Won’t Have. These categories would be organised into a table, that is shown in Table 1.

The *Similar System* analysis was a vital step for Design Documents, particularly the CODEX app, as it would ensure that the developer was not creating a duplicate for a system that already existed. Therefore, for commercial systems, this analysis may be seen as a form of risk assessment. Typically the findings of the Similar Systems analysis are stored within a table in the early stages of the Design Document. Sections 3.1.1, 3.1.2 and 3.1.3 informed by the findings from the CODEX Similar Systems analysis.

Another key component of a Design Document would be the *Glossary of Terms*,

[2]The CODEX Design Document can be found in the supporting material for this paper.

Must have	**Should have**	**Could have**	**Won't have**
Track Combat	Multiple Campaigns per DM Account	Game Scheduling	Full descriptions of D&D
Accounts	Settings	Items	Pre-made Settings & Campaigns
Encounter Planning	Characters		Inter-account Direct Messaging
Random Encounters	WorldWiki		
Populated Database	Run, Save & Delete Games		
Session Planning			

Table 1: MoSCoW analysis for CODEX, showing the differing importance of wanted features to the system as a whole. The *Minimum Viable Product*(MVP) for the project is derived from the *Must have* column. The *Should have* column represents what features CODEX should contain for the system to be complete. The *Could have* is what features might give CODEX an edge over similar systems and what would be "nice to have". Finally the *Won't have* column represents what features will never be part of the CODEX system.

which would inform future developers of the terms required to understand the software system. The terms contained within this glossary would not extend to technical terms, unless they were uncommon at the time of development. Instead the glossary would focus on the contextual terms, in the case of CODEX the majority of the terms were D&D related. The full glossary of terms for the CODEX app can be found in the Design Document.

For every function detailed by the Design Document, there would be an accompanying *Textual Use Case* and *Use Case Table* which were then summarised into a *Use Case Diagram*. The Use Case Diagram would show the authorisation that each type of user would have within the system. In the case of CODEX we have two types of users – players and DMs – and each had two levels of access to the system functions. Also

shown by the Use Case Diagram is which use cases relationships: *include* (meaning they are a prerequisite to) and *extends* (which represents optional behaviours between use cases). See Figure **??** for the Codex Use Case Diagram. The textual use cases and use case tables detail exactly how each function of the app would be used by the user. These details included goals, scope, triggers, end conditions, success and alternative scenarios to name but a few. The developer would study the use cases in order to gain an understanding of what the source code needs to reflect prior to starting development.

The Graphical User Interface (GUI) design of the CODEX web app began with *lo-fi prototyping*, where the developer would roughly sketch the design of the GUI and then quickly receive feedback from prospective users. With the feedback incorporated into the design, the lo-fi prototypes were transformed into *hi-fi prototypes* by creating digital sketches using software. The CODEX web app GUI was designed using MarvelApp (App, 2013), which had a range of tools and allowed the prototyping of colour schemes and icons to be effectively tested. The hi-fi prototypes were shown to prospective users to gather feedback, who reported that the colour scheme was too dark. With the lighter palette integrated into the app, the GUI design was finalised and inserted into the Design Document as images.

Once the use cases and functionality of a system has been decided, the developer could now start allocating the functionality to classes within the source code. This was done through the Class Diagram, which shows the flow of information between the classes that divide the source code. Each class within the diagram would have the class and method names. The method names were designed to be self-explanatory to the developer. This was done so that when the developer was writing the source code, the Class Diagram could be referenced so that development would stay on track and the scope of the project would not creep [3]

The classes generated from the Class Diagram would be inserted into the Architecture Diagram, to create a graphical representation of the system architecture. Each class was allocated into one of the three tiers of the web app architecture (described in Section 3.3 and Figure 13): Presentation, Logic and Data. Similar to the Class Diagram, the

[3] *Scope Creep* is a term used by Software Engineers when the Scope (the functionality) of a system was extending (creeping) past the original parameters of the system in an unplanned fashion.

Architecture Diagram guided the developer during the development process to ensure that each class was performing the tasks allocated during the design process.

4.3 Development of CODEX

The development of the CODEXweb app was scheduled to take 9 weeks (see Figure 1), which allowed for multiple revisions of the design and development in addition to the writing and evaluation necessary for the project.

As mentioned in Section 3.3, the CODEX web app was built using ReactJS (see Section 3.3.2) with assistance from the Semantic UI package (see Section 3.3.3). Before providing examples of the CODEX source code, we must first expand upon our understanding of how ReactJS operated.

ReactJS was operated through the concept of *components*. Developers would create these components which would render the GUI for within the logic layer before sending the image across to the presentation layer. Components could receive inputs from other components and from the database through the *props* list. However components might also be functional, not just presentational. Finally, ReactJS passes information across web pages through an attribute known as *state*, components may be *stateful* or *stateless.*

```
1  import React, { Component } from 'react';
2  import { Header, Grid, Input, Segment, Button, Divider } from '
       semantic-ui-react';
3  import background from '../images/background.jpg';
4  import '../App.css';
5  import Link from 'react-router-dom/Link';
6
7  const Account = () => (
8    <Segment padded>
9      <Header as='h1' className="app-title">Welcome, Dungeon Master
         , to your Codex!</Header>
10     <Input fluid placeholder="Username"/>
11     <Divider hidden/>
12     <Input fluid placeholder="Password"/>
13     <Grid columns={2} relaxed>
14       <Grid.Column>
```

```
15            <Segment basic>
16              <Link to={'/dungeon-master'}><Button primary fluid>
                    Login</Button></Link>
17            </Segment>
18          </Grid.Column>
19          <Grid.Column>
20            <Segment basic>
21              <Link to={'/player'}><Button primary fluid>Reset
                    Password</Button></Link>
22            </Segment>
23          </Grid.Column>
24        </Grid>
25        <Divider horizontal>Or</Divider>
26        <Link to={'/create-account'}><Button secondary fluid>Sign Up
             Now</Button></Link>
27      </Segment>
28    )
29
30    class App extends Component {
31      render() {
32        return (
33          <div className="page-container">
34            <img alt="Background" className="bg-img" src={background
                 }/>
35            <div className="header">
36              <div className="login-container">
37                <Account/>
38              </div>
39            </div>
40          </div>
41        );
42      }
43    }
44
45    export default App;
```

Listing 1: ReactJS source code which generated the GUI for the homepage for the CODEX web app

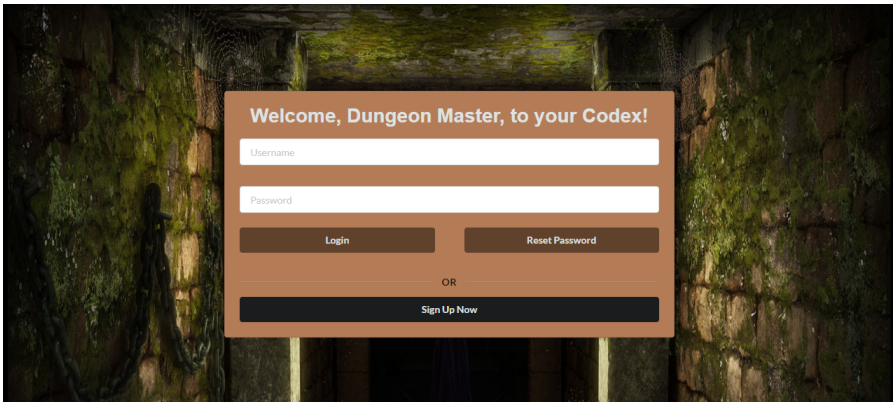


Figure 14: Screenshot taken from the CODEX web app homepage which shows the rendered components detailed in Listing 1

Listing 1 was the source code that generated the homepage of the CODEX web app (shown in Figure 14) once compiled through the *virtual DOM*[4]. The file would compile in the order as we would read it. The necessary components for the subsequent source code would be imported first, in addition to any local assets, such as the `background` image on line 3. Once the imports have finished, the compiler would then render any custom components. In this case the `const Account` would qualify as a custom component. Various Semantic UI elements were used in the construction of the `Account` component. Penultimately the compiler would render the `class App` which itself would be a component (as it extends the React Class `Component`). The final action performed by the compiler to export the `App` class into the webpage, where the virtual DOM would read it and pass the node tier list to the actual DOM to be displayed on the users screen.

The development of CODEX was driven by tests, using the Test Driven Development (TDD) principle, in that each method will produce a testable outcome. Before any development could begin, the developer wrote a test that would ensure no build of the CODEX web app would be deployed when the components where unable to be rendered. This was known as a *smoke test* which has been an industry standard test to include in any

[4]A *virtual DOM* (Document Object Model) would behave in the same manner as the *actual DOM*. Which would construct a node tree that lists all elements and corresponding attributes on a webpage. The difference being that the virtual DOM was significantly faster.

automated testing environment for over a decade (El-Far and Whittaker, 2002). Listing 2 was the source code that was necessary to perform the smoke test on the CODEX web app. When a build of the web app was pushed to the master build on GitHub (Github, Inc., 2018), the code was automatically tested by the web service *CodeShip* (CloudBees, 2018) which would execute the `App.test.js` file before committing the changes to GitHub. As previously mentioned in Section 3.2.3, TDD does not prevent every bug from infecting the master version of a project, TDD would prevent the majority of the developer errors increasing the quality of the code produced.

```
1  import React from 'react';
2  import ReactDOM from 'react-dom';
3  import App from './App';
4
5  it('renders without crashing', () => {
6    const div = document.createElement('div');
7    ReactDOM.render(<App />, div);
8    ReactDOM.unmountComponentAtNode(div);
9  });
```

Listing 2: CODEX web app smoke test

5 Outcome of the CODEX project

In this Section we will examine the development of the CODEX web app (see Section 5.1 and the effectiveness of Agile Solo in managing the development (see Section 5.2). At the end of this section we will have an understanding of the outcomes of the CODEX project and will then be able to evaluate those outcomes in Section 6.

5.1 Development Observations

The CODEX web app was developed in ReactJS (see Section 3.3.2) using the Software Engineering Methodology (see Section 3.2.1) known as Agile Solo (see Sections 3.2.3 and 4.1). The effect that Agile Solo had on the development will be discussed in Section 5.2), in this subsection we will discuss the observations the developer recorded during

the development stages of the CODEX web app.

The developer had allocated a total of 9 weeks for the development of the CODEX web app with an additional 7 weeks for the design of the system. This was found to be insufficient for the a web app such as CODEX as the developer found that only the core functionality was able to be achieved within this time frame. The reasons for this were two-fold. The scope of this web app was too great to be tackled within the development time, particularly as the developer was learning ReactJS during the development. Whilst the rate of development increased rapidly as the developer came to understand the nuances of ReactJS (as described in Section 4.3), the deadlines of other projects were obtrusive towards the development of CODEX.

During the implementation the database that supported the app was populated with a small sample of the Wizards of the Coast D&D material. The developer observed that there was no need to transcribe over five hundred pages of statistical data from the D&D core books (Monster Manual, Dungeon Master's Guide and Player's Handbook) and supplementary material (such as Volo's Guide to Monsters (**?**)). The smaller database would allow the development of critical features of CODEX to begin earlier than scheduled and be tested with a more manageable dataset. The database would be fully populated at a later date of development.

Another observation that the developer made during the development of the CODEX web app was that the primary function and goal of the development was to provide a platform for the evaluation of Agile Solo. At the end of the scheduled development time, the developer made the conscious and informed decision to prioritise the evaluation of Agile Solo. Development of the web app continued at a slower rate with the change of focus for the project.

5.2 Effectiveness of Agile Solo

[time management]

[task management]

[TDD]

6 Evaluation of CODEX

boo

6.1 Feedback on the CODEX app

[Chaos report categorisation criteria]

[Testing the web app from a developer point of view]

[Testing the web app from a potential user point of view, limitations from the Wizards of the Coast agreement and gathering feedback]

[Tis a challenged project it is]

6.2 Development Issues

boo

6.3 Agile Solo Evaluation

boo

7 Conclusions

The CODEX project was to software engineer a progressive web app built in ReactJS, using an agile methodology. The principle challenge of CODEX was that, unlike the majority of software engineering project, there was only one developer. Agile methodologies are designed to be used by a group or groups of developers, with designated roles for individuals within the team. As part of the preparation for CODEX, a single developer methodology had to be found, these were *Agile Solo* and *XP for One*. Agile Solo was selected to be the principle methodology for the development of CODEX.

7.1 Agile Solo

Agile Solo, as described in Section 3.2.3, is a methodology that was developed because there was no Agile development methodology designed for solo developer projects. The purpose of a software engineering methodology is to provide a platform to assist with time and task management for a project, in that respect we need to have a certain amount of discipline from the developer when applying a methodology to a project. [It was this discipline that came a critical issue with a solo developer project. When working in a team, each developer is held accountable by the rest of the team. But in solo developer projects no one is holding the developer to account, with the notable exception of a supervisor when present.] However, Agile Solo functioned without flaw when implemented as the original author described. This is evidenced by the project journal, which can be found in the CODEX portfolio, where there are numerous entries that prove the following of Agile Solo. The work that was allocated each cycle was achieved and any issues in time spent of that work was down to an incorrect estimation during the allocation process.

In summary, Agile Solo has proven to be a highly effective and accurate methodology for solo developer projects. The methodology proposed in 2011 by Nystöm does not need any alterations as far as CODEX is concerned.

7.2 CODEX

- The Codex app was a large ambitious project with many complicated features
- Developed in a new language to the developer
- Developer achieved a working system that provided the core functionality of the app that met the requirements
- Many advanced features are easily integrated with more work
- Project was not a success as all features were not completed, however it was not a failure as the requirements were satisfied

- Therefore the CODEX app is a challenged project in accordance with the Standish Group Chaos Report classifications

References

Anderson, D. J. (2010). *Kanban: successful evolutionary change for your technology business*. Blue Hole Press.

App, M. (2013). Marvel app.

Association, I. S. et al. (2010). Systems and software engineeringâĂŤvocabulary iso/iec/ieee 24765: 2010. *Iso/Iec/Ieee*, 24765:1–418.

Astels, D. (2003). *Test Driven Development: A Practical Guide*. Prentice Hall Professional Technical Reference.

Beck, K. and Gamma, E. (2000). *Extreme programming explained: embrace change*. addison-wesley professional.

CloudBees (2018). Codeship. *http://codeship.com/*.

De Groef, W. (2016). *Client- and Server-Side Security Technologies for JavaScript Web Applications*. Phd thesis, Faculty of Engineering Science.

El-Far, I. K. and Whittaker, J. A. (2002). Model-based software testing. *Encyclopedia of software engineering*.

Ewalt, D. M. (2014). *Of dice and men: The story of Dungeons & Dragons and the people who play it*. Simon and Schuster.

ExtremeProgrammingChallenge (2006). Extreme programming for one. *http://xp.c2.com/ExtremeProgrammingForOne.html*.

Facebook, Inc. (2017). Reactjs. *https://reactjs.org*.

Flanagan, D. (2006). *JavaScript: the definitive guide*. " O'Reilly Media, Inc.".

Fling, B. (2009). *Mobile design and development: Practical concepts and techniques for creating mobile sites and Web apps*. " O'Reilly Media, Inc.".

Fowler, M. and Highsmith, J. (2001). The agile manifesto. *Software Development*, 9(8):28–35.

Github, Inc. (2018). Github. *https://github.com/*.

Gygax, G. and Arneson, D. (1974). *Dungeons and dragons*, volume 19. Tactical Studies Rules Lake Geneva, WI.

Holmes, J. E. (1980). Confessions of a dungeon master. *Psychology Today*, 14(6):84–94.

Horton, A. and Vice, R. (2016). *Mastering React: Master the art of building modern web applications using React*. Packt Publishing.

James, M. (2017). Scrum methodology. *http://scrummethodology.com/*.

McElrath, R. (2007). Xml legal document utility software design document. *http://robotics.ee.uwa.edu.au/courses/design/examples/example_design.pdf*.

Mearls, M. and Crawford, J. (2014a). *Dungeon Master's Guide*. Wizards of the Coast.

Mearls, M. and Crawford, J. (2014b). *Player's Handbook*. Wizards of the Coast.

Mercer, M. (2016). Setting up your gamemaster's screen! (gm tips w/ matt mercer). *https://www.youtube.com/watch?v=YRMVTmbe-Is*.

Nielsen, J., Clemmensen, T., and Yssing, C. (2002). Getting access to what goes on in people's heads?: reflections on the think-aloud technique. In *Proceedings of the second Nordic conference on Human-computer interaction*, pages 101–110. ACM.

Perkins, C. (2014). *Monster Manual*. Wizards of the Coast.

Peterson, J. (2017). Fourty years of adventure. *http://dnd.wizards.com/dungeons-and-dragons/what-dd/history/history-forty-years-adventure*.

Richards, K. and The Orr Group, LLC (2012). Crooked staff blog. *http://crookedstaff.blogspot.co.uk/2012/12/experimenting-with-roll-20-virtual.html*.

Schwaber, K. (1997). *SCRUM Development Process*, pages 117–134. Springer London, London.

Software, F. C. (2011). Trello. *https://www.trello.com*.

Standish Group (2014). Chaos report. *https://www.projectsmart.co.uk/white-papers/chaos-report.pdf*.

The Orr Group, LLC (2018). Roll20. *https://roll20.net/*.

Thinkwik (2017). Why reactjs is gaining so much popularity these days. *https://medium.com/@thinkwik/why-reactjs-is-gaining-so-much-popularity-these-days-c3aa686ec0b3*.

Thomason, L. (2018). Semantic ui react. *https://react.semantic-ui.com/*.

Toltz, I. and Barzilai, J. (2016). Kobold fight club. *https://kobold.club/*.

Various and reddit inc (2018). Reddit. *https://www.reddit.com/r/DnD/*.

Wizards of the Coast and Curse, Inc. (2018). D&d beyond. *https://www.dndbeyond.com*.