Game Software Lifecycle for Mobile Devices
A Case Study on iOS

Dimitrios Bendilas

Master Thesis

Supervisor: Dr. Alexander Chatzigeorgiou
Examiners: Dr. Christos Georgiadis
Dr. Nikolaos Samaras

Department of Applied Informatics

University of Macedonia
Thessaloniki

November 2012

Abstract

Mobile applications have become part of everyday life for hundreds of millions of people, who use their portable devices for dozens of tasks, including playing games. This thesis studies the smartphone and tablet market, and presents a variety of statistical data about most major mobile platforms. Further details are given on game applications, including the currently popular free-to-play model. An approach to what makes games different from other types of software is also presented, along with a typical lifecycle for a game application. The thesis provides a detailed case study of A Clockwork Brain, a free-to-play mobile game for iOS, describing many aspects of the development process, the product lifecycle, numerous design and development problems that occurred, as well as key decisions made during the production.

Keywords: game development, mobile applications, case study, game design, prototyping, iOS, balancing, usability, beta test, freemium, free-to-play, analytics

Acknowledgments

It gives me great pleasure in acknowledging the help of my thesis Supervisor, Dr. Alexander Chatzigeorgiou. His confidence and support throughout the preparation of this thesis means a lot to me. I also thank him for being one the best, most generous and most open-minded professors I have ever had.

This thesis would not have been possible without the passionate work of each and every one on the Total Eclipse team, who went the extra mile to create a great game, while having so limited resources. A special appreciation is reserved for my brother and business partner Argiris, who always keeps us true to our cause.

I also wish to thank separately one of the team members and dearest friend, Dr. Maria Sifnioti, for her valuable insights that helped me a lot with my research.

My deepest gratitude goes to my better half, family and close friends for their love and support.

Table of Contents

Index of Tables

Index of Charts

Index of Figures

Glossary

DRM	Digital Rights Management
FPS	Frames-Per-Second
GDD	Game Design Document
IAP	In-App Purchases
QA	Quality Assurance
URL	Uniform Resource Identifier
XML	eXtended Markup Language

Introduction

In the few years of its existence, the smartphone market has displayed a rapid growth; smart device adoption has been 10 times faster compared to that of the PC during the 80's and 3 times faster than the recent social network phenomenon [1]. In the first quarter of 2012 alone, more than 150 million smartphone and tablet devices have shipped, accounting for a 50% year-to-year change [2]. At the same time, the total number of applications available across all major mobile application stores has grown to more than 1.5 million [3], [4], [5], [6]. What is typically required in order to sell an application through one of the app markets is to register to the distributor's program, create an application that adheres to the design and technical requirements the distributor has defined, and submit it for review [7], [8], thus making the entry barrier for a developer low. This is perhaps one of the reasons that there are hundreds of thousands of companies and individuals developing mobile applications [9].

Games are the most popular app category, as people consume most of their time playing games, on both smartphones and tablets [10]. This is probably the reason many developers create mobile games, which account for almost 18% of the total applications on the App Store [3], [11]. However, mobile game development is not trivial and the success of a mobile game is not guaranteed. The purpose and nature of games, as well as the current state of the mobile application ecosystem make it difficult for any game to succeed financially.

This thesis consists of two parts. The first part examines the current state of the mobile market and provides data regarding device and platform use, as well as the app ecosystem. It discusses the current developer trends in mobile game development, in relation to platforms and devices. It also analyzes what makes players enjoy a game and describes the difference between gaming software and other types of software, explaining its idiosyncrasies and difficulties. Finally, it describes the creation process of games and a typical software lifecycle in game development.

The second part describes in detail the creation process of A Clockwork Brain[1], a puzzle & brain training game made for iOS by Total Eclipse[2], a Greek game development studio. The case study focuses on the design and development side of the project, analyzing some of its most important aspects, from the pre-production phase to the completion of the project and subsequent updates. It provides information regarding the scope, the team and its dynamics and describes many of the steps taken to create the final product. A large section is devoted to game design, where many of the game

[1] www.aclockworkbrain.com

[2] www.totaleclipsegames.com

design elements are described. Key decisions made to improve the game play and other aspects of the application, are analyzed, from both a design and programming angle.

A number of engineering problems are listed as well, together with the implemented solutions, related to issues such as network connectivity, caching, building for multiple devices and more. Whenever possible, class diagrams, flow charts and source code fragments are included, in order to better illustrate the adopted solution in an important engineering or design problem. Finally, some business-related aspects are presented, such as using analytics software to track, evaluate and optimize the monetization of the application.

The experiences described here are first-hand, as the author was part of the development team throughout the project, overseeing the game design and engineering areas and discussing the rest of the production aspects with the other team members on a daily basis. The author co-founded Total Eclipse in 2004 and has since been working in the company as the Lead Engineer and Lead Game Designer.

Part I

Mobile Application Development

The first part of this thesis reviews the current state of mobile game development. It includes a chapter on market analysis and another chapter on game development. The former examines the most important aspects of the mobile industry, the devices and software available, as well as current trends. The later makes a theoretical approach to games and their difference with other types of software. It analyzes the idiosyncrasies of gaming applications and outlines the special challenges game development presents in terms of engineering and management.

Chapter 1

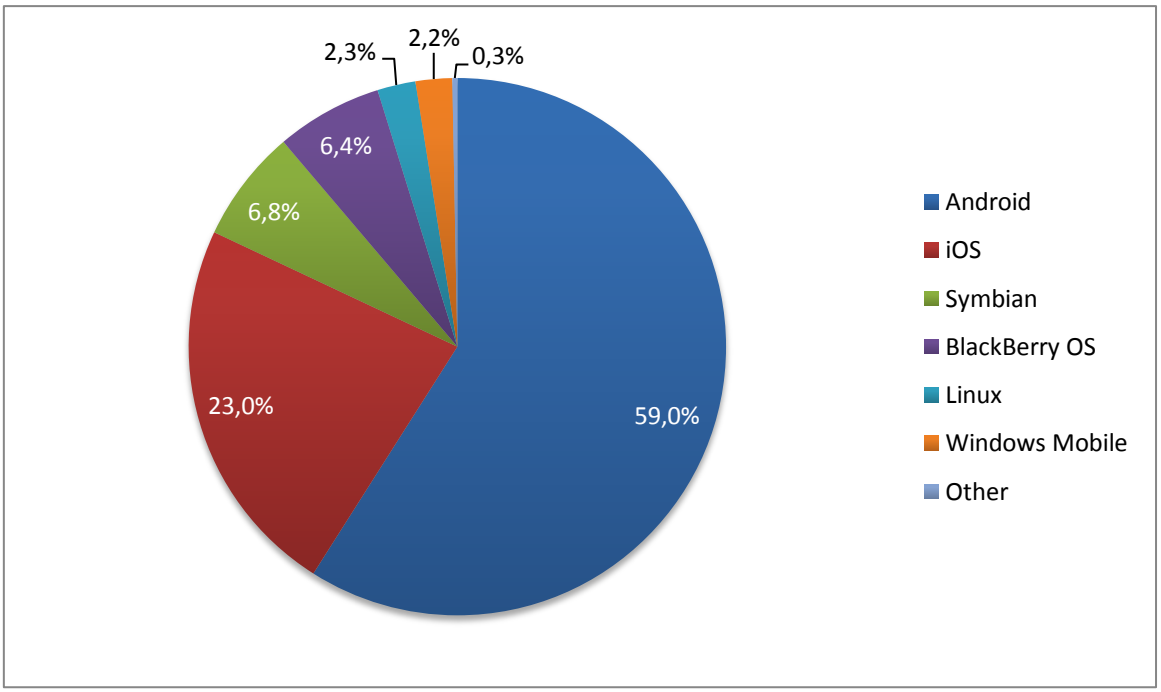
Market Analysis

This chapter describes the current state of the mobile market, focusing on the smartphone devices, the available operating systems and the application ecosystems across the various platforms. The market is analyzed in terms of size and growth of the most popular platforms. Revenue and ranking data for applications and games in particular are also provided. Finally, the opportunity for developers to succeed in the mobile development market is analyzed, using data from surveys that study the revenues of mobile apps and games, as well as studies that provide quantitative information on the top charts on the application stores.

1.1 Overview

The smartphone market has been growing rapidly for the past few years. Farago [1] estimates there were over 640 million iOS and Android devices in use during July 2012. In comparison, 835 million PCs were sold from 1981 to 2000 [12]. In 2011, for the first time ever, global shipments of smartphones and tablets surpassed that of PCs [13] , while at the same time daily use of mobile applications exceeded desktop and mobile web use for the first time [14].

Chart 1.1 Market share of top smartphone operating systems (1Q 2012)

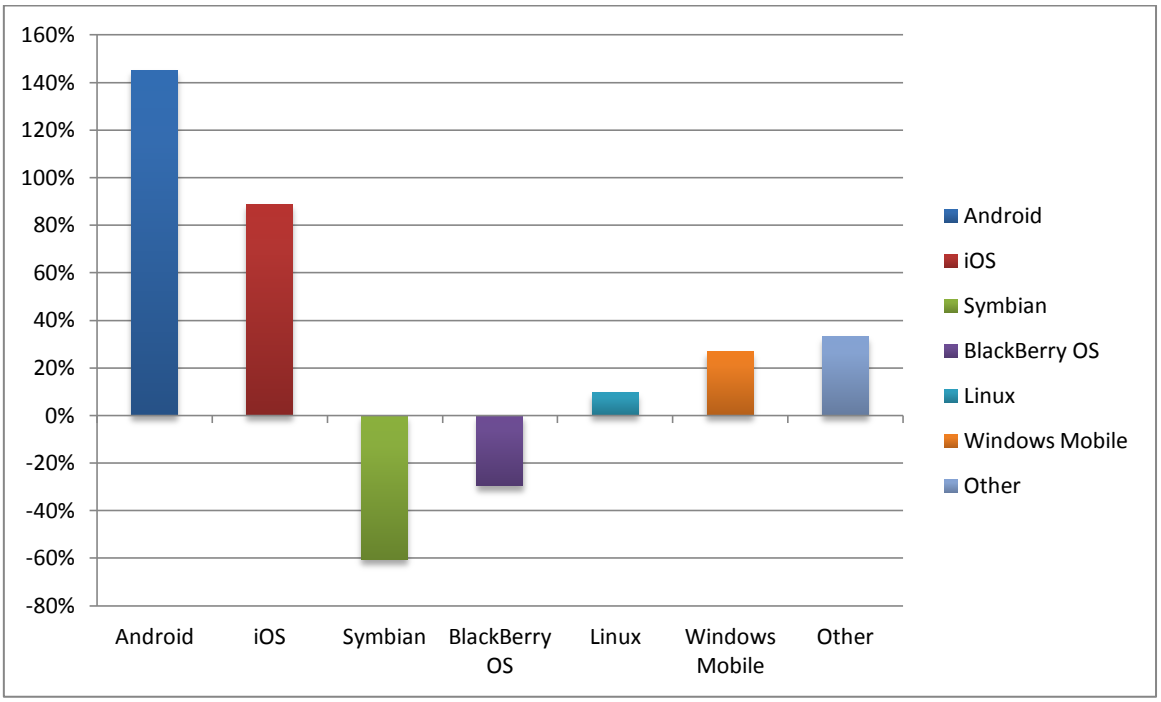


Source: IDC [2]

Android and iOS are the most popular mobile operating systems, having a combined share over 80% of the current market, both achieving a high growth rate compared to 2011, as seen in Chart 1.1 and Chart 1.2.

Of all the platforms, Android, iOS and Windows Mobile are growing fast, while Symbian and BlackBerry are shrinking. Android is displaying the fastest growth, at a 145% rate compared to a year before.

Chart 1.2 Year-to-year change of top smartphone operating systems (1Q 2012)



Source: IDC [2]

1.2 Devices

The number of available devices varies a lot among platforms. Apple's iOS runs only on devices created by Apple, namely the iPhone, iPod Touch, iPad (portable devices), as well as Apple TV. In total there are 14 portable iOS devices [15]. Android has created a much different ecosystem, with 1306 different devices currently running its operating system and supporting Google Play [16], most of which are created by companies other than Google. Windows Phone runs on 35 different devices, manufactured by third-party companies [17] as well as Microsoft. It is obvious that Android is much more fragmented, something that has been causing mobile developers troubles. For example, as Natalia Luckanova [18], one of the team members of mobile hit game Temple Run writes for the Android version of Temple Run:

> *"99.9% of support emails are complaining their device isn't supported. We currently support 707 devices."*

Perhaps the fragmentation of Android devices may affect the developers' intent to invest on the platform. A survey conducted by Appcelerator and IDC in 2012 suggests that in Q1 2012 the interest in Android phones and tablets dropped 4.7% and 2.2% respectively [19]. As the report states, these drops are consistent with a trend of previous quarters, which showed an erosion in Android interest, even though the platform has increased its market share substantially. As the authors say:

> *We believe this is mostly due to the fragmentation Android continues to experience and that Google seems unable to curtail, and the continued success of Apple's iPhone and iPad. This fragmentation, coupled with iPads continuing to outsell all Android tablets combined, has swayed developers increasingly towards iOS and away from Android.*

1.3 Applications

The number of mobile applications built for modern devices is also impressive. In September 2012 the total number of apps on the App Store reached 700.000, with 90% of them being downloaded every month [3], while at the same time the Google Play store had 675.000 apps [5]. In the summer of 2012 Windows Phone Marketplace had 100.000 apps [4], about the same as BlackBerry App World [6]. Users consume mobile applications for 77 minutes daily on average, 13% more than a year before [1] and the average iOS customer uses more than 100 apps [3]. The mobile app ecosystem is a $10 billion market, growing at 100% per year [13].

1.4 Games

The largest application category is games, currently exceeding 125.000 in total on the App Store alone, as estimated by 148apps.biz [11]. The rate on which new applications are becoming available is also growing. During September 2012 it is estimated that there were more than 24.500 new apps launched on the App Store, 4.200 of which were games, an increase by 20% compared to September 2011. Also, 50% of the free downloads and 60% of the paid downloads are games. Moreover, Farago [1] estimates that 75% of the revenue on iOS Top 100 grossing applications is generated by games and calculates that 31% of the total time app users spend on their devices accounts to gaming.

Apple has paid over $5 billion to developers since the inception of the App Store [3] and there have been great successes in the mobile industry, such as the Angry Birds franchise, which surpassed 1 billion downloads [20], Infinity Blade 1 & 2, reaching $30 million revenue [21], Tiny Tower achieving 2 million free downloads in a week [22] and World of Goo getting over 1 million paid downloads [23].

Mobile stores give thousands of developers the opportunity to distribute their work to millions of people worldwide. The App Store alone provides a customer base of 400 million users, in more than 150 countries [3]. Additionally, many of the most successful

publishers are relatively small, having published less than 10 apps each [13], which suggests that it is not essential to be a large studio in order to succeed.

Finally, the customer base for games in general seems to be growing. The number of USA gamers, not restricted to mobile, has increased by 241% between 2008 and 2011, according to Macchiarella [24]. Macchiarella also mentions that 18% of all gamers download games on their phones, a number 2.5 times higher than in 2008.

1.5 Free-to-play model

Applications that are given as free downloads and include optional premium content or features are often called 'freemium', from the words free & premium. Apple introduced this feature in October 2009 [25], naming it In-App Purchase. The introduction of In-App Purchase made a major difference on the iOS market. According to Flurry [26], in June 2011 freemium games with In-App Purchases represented 65% of the total revenue for top grossing games on the App Store, a number that was at 39% in January 2011. Also, analytics company AppAnnie [27] reports that worldwide freemium revenues apps have almost tripled from September 2011 to September 2012 on iOS, while growing 350% on Google Play from the beginning of 2012. Revenues from paid apps have remained relatively stable during these periods.

In freemium apps, only a small percentage of the users that download them purchase anything; the rest play for free. Flurry says this number ranges from 0.5 - 6% for games, and depends on the quality and core mechanics of the game [26]. However, the amount of money generated by the few users that convert can be significant and can even reach many millions of dollars each month for a single game, based on reports by game developers over 2012 [21], [28].

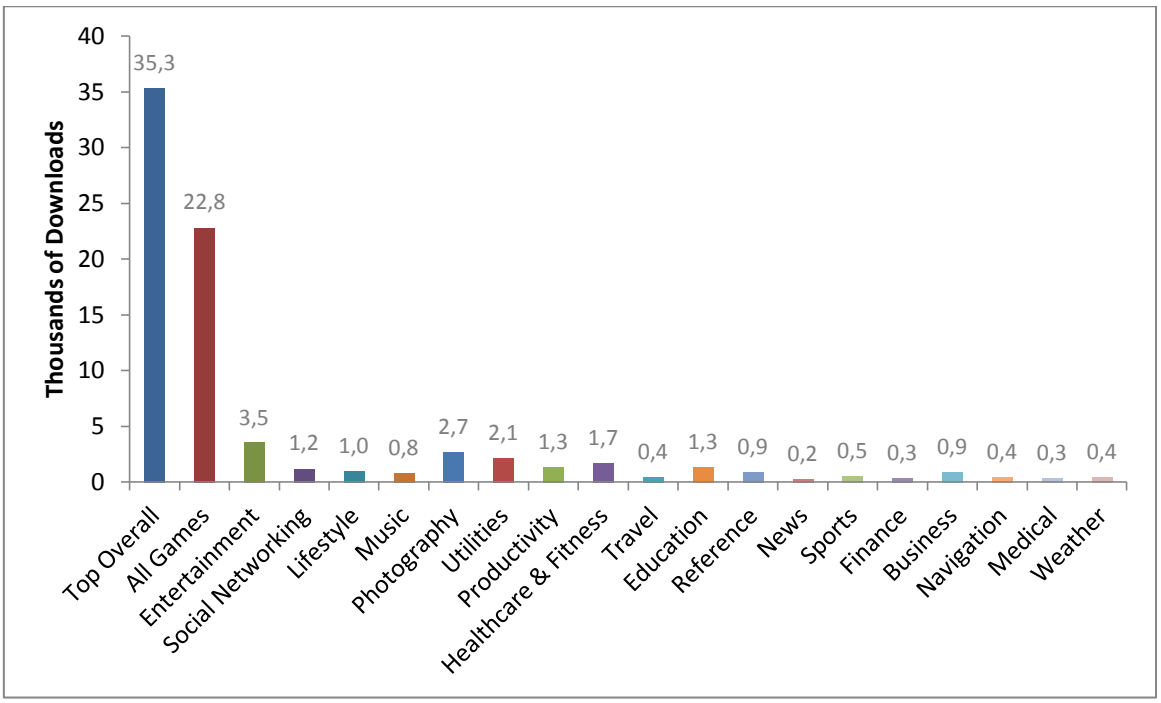
1.6 Success

This might seem as an ideal environment to create commercial software applications. However, creating a successful mobile game is not easy. Apart from the amount of work required to create a quality product, the commercial and financial success is affected by the way the app markets operate. For example, there are roughly 250.000 registered iOS developers in the US, who create applications for the App Store [7]. No matter how small or large, all of them compete for the same spots on the Top 25, which have a significant impact on the applications' revenue.

According to Spriensma [29], reaching the Top 25 paid charts on the USA App Store requires 22.800 sales/day for the Games category, 3.500 sales/day for Entertainment and 300 sales/day for Medical, while Top Overall can be reached when having 35.300 sales/day. Free apps have to achieve even higher download volumes in order to rank high on the charts. Again for the USA App Store, an application needs 25.300 downloads

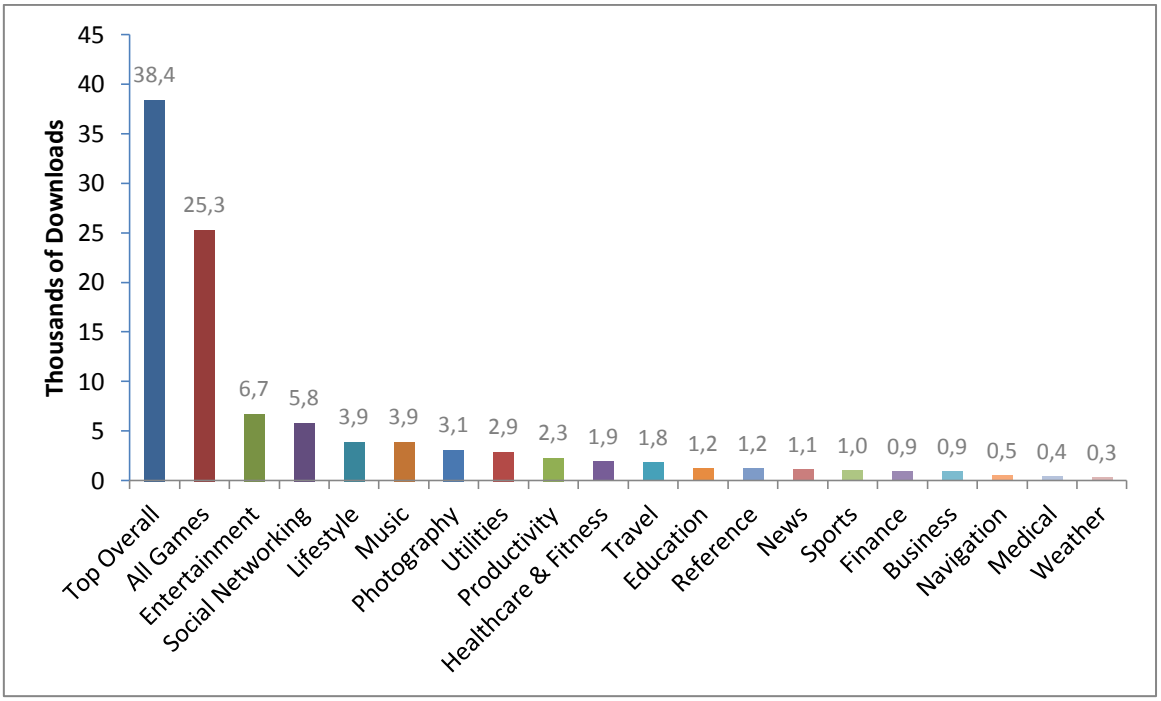
per day to be in top Games. For the Entertainment category it's 6.700 downloads/day and for Medical apps it's 400/day. Reaching the Top Overall free category requires 38.400 downloads / day.

Chart 1.3 Number of downloads needed to reach Paid Top 25 on USA App Store



Source: Distimo [29]

Chart 1.4 Number of downloads needed to reach Free Top 25 on USA App Store



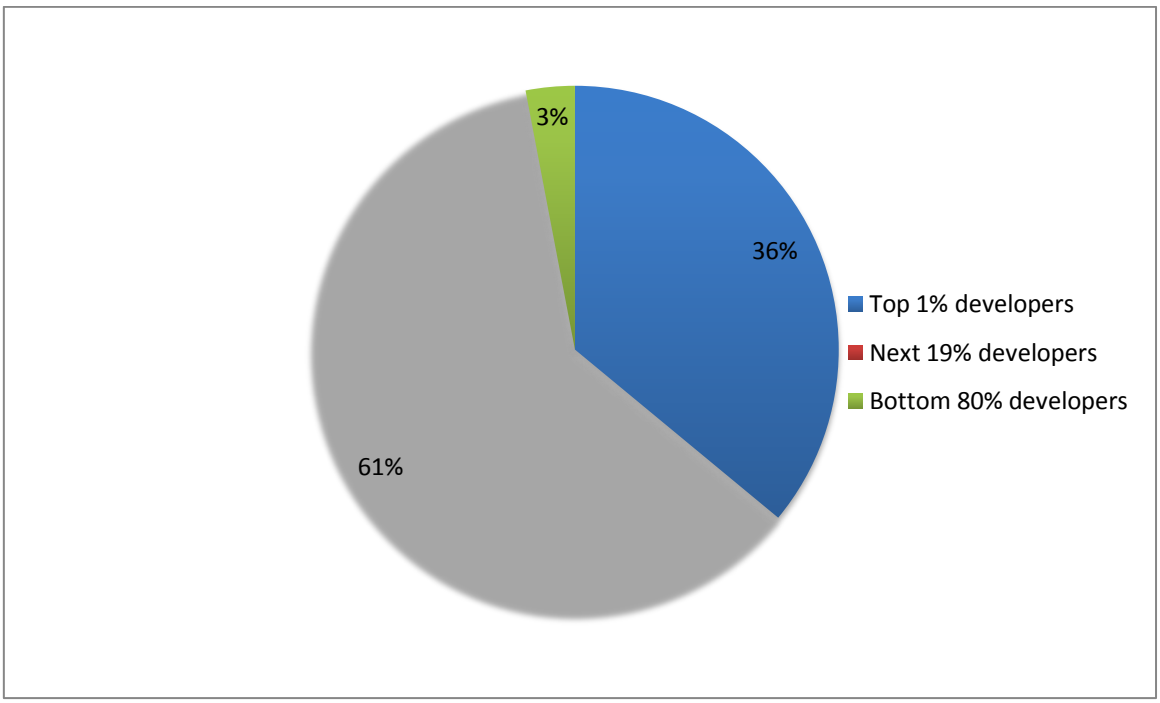
Source: Distimo [29]

A survey conducted by indie developer Owen Goss [30] in 2011 revealed some interesting data regarding the revenue generated by iOS games. Although there may be flaws in the used methodology, as Goss explains, there are not a lot similar studies that explore the revenues of mobile games collectively. On that account, its findings should be examined cautiously. The survey, which collected data from 252 game iOS development studios and individuals, suggests that the arithmetic mean average lifetime revenue for a mobile game on the App Store is $165.000, while the median average is $3.000. This significant difference results from the fact that the values on the high spectrum are many times larger than the values on the low spectrum and that the computed sample standard deviation is 639,966 US dollars. As a result, the median average is more representative in this case [30].

The median shows that half of the developers have made less than a $3.000 lifetime revenue from their apps. Also, 25% of developers have made more than $30.000, another 25% have made less than $200 and 4% have managed to earn over $1 million. Another interesting fact is that the top 20% of developers earn 97% of the total revenue, while the top 1% earn 36% of the revenue [30].

These figures, to the extent that they actually represent the iOS market, paint a non-ideal picture for developers. At least, they suggest that the financial success of a mobile game on iOS is not guaranteed.

Chart 1.5 Revenue distribution among developers



Source: Goss [30]

Marketing firm App Promo conducted another survey, leading to similar findings [31], although the methodology they used has been reviewed with some skepticism [32]. The survey took place in May 2012 and asked for data from developers across various platforms (iOS, Android, BlackBerry, Windows and Symbian), even though it is not stated clearly to what extent developers from each platform responded to the survey. Among their findings is that 59% of apps do not generate enough revenue to break even on development costs, while 80% of them do not earn enough to support a standalone business and 69% of the developers earned $5.000 or less with their most successful app. Also, App Promo continues, the top 12% earning apps have $50.000 or more in revenue and developers of these apps spend 14% of their time on marketing, using $30.000 as a marketing budget on average. Contrary, more than half of the developers in total set aside a marketing budget and spend 5% or less of their time promoting an application, even though 9 out of 10 believe that marketing is necessary for success.

1.7 Conclusions

Judging by their incredible adoption, smartphones and tablets seem to have become a in important part in the everyday life of millions of people. Users access dozens of different apps for completing various tasks, as well as for entertainment. Apple and Google are the definite leaders, but the market is constantly changing as other platforms are gaining or losing market share. The opportunity for developers is great, but it does not come without challenges and obstacles, as most developers cannot sustain a viable business in this market.

Chapter 2

Game development

This chapter begins by describing what makes games different from any other type of software, taking a theoretical approach on key idiosyncrasies of games. The analysis is done from both the player's and the developer's perspectives, outlining what goals a game must fulfill in players, and the way these goals affect the work of the development team. The rest of the chapter deals with the challenges in software creation and the reasons for which software projects, and game projects in particular, often fail. Finally, it presents the development processes built to help engineers and project managers in this regard and outlines the typical stages in game development.

2.1 Purpose of Games

Building gaming software involves everything related to generic software development, but presents some additional challenges. The core ingredient of a game is its gameplay, which can be described as "rules and structures that result in an experience for players" [33]. Gameplay is the product of game design, a process carried out by a game designer, who according to Fullerton [34] is "part engineer, part entertainer, part mathematician, and part social director […] who has to create a set of rules within which there are means and motivation to play",. But there is more to the creation of a game. Will Wright, forwarding A Theory of Fun for Game Design [35], says that "The design and production of games involves aspects of cognitive psychology, computer science, environmental design, and storytelling".

In more than one definitions of *game*, not necessarily a video game, there is the aspect of a lack of obligation; "a voluntary effort to overcome unnecessary obstacles" [36], "we might call it a free activity" [37, p. 4]. Unlike other types of software, the reason for which a user engages with a game is somewhat ambiguous. For example, someone that uses an image processing software may want to do some retouching on old family photos, whereas a user that opens a text editor might need to write a report. These are well-defined goals and are directly bound to the way the specific software works and the tools it provides. The same cannot be said for a strategy game, for example, as the desired outcome of user that plays the game isn't to move around little simulated people, trying to kill other simulated people and create virtual buildings and ships. It is something much more abstract.

A fundamental difference seems to exist between gaming software and non-gaming software. Normal software programs are tools; games are not. According to Malone [38], tools, at least good ones, should be both easy to use and to master. On the opposite side, the founder of Atari, Nolan Bushnell, has been quoted of saying:

"All the best games are easy to learn and difficult to master. They should reward the first quarter and the hundredth." [39, p. 125]

The benefit of using a tool does not come from the tool itself, but from the end result. The tool is not the focus in this case, the result is, and exists outside the tool. The same cannot be said for the use of a game, where the focus of the player's actions perhaps is the game itself. The purpose of a game seems to have to do with emotions. A study made by XEODesign in 2004 [40] attempted to deduce the reasons for which people play games. The research study was conducted with 30 players (equally divided to casual and hardcore) and 15 non-players. Its results revealed that people play games for the experience the games create, rather than for the games themselves. They want to feel challenged, absorbed, accomplished and they enjoy the excitement and pleasure that result from a good game session. They even appreciate a relief from every-day worries [40]. The collection of these feelings and emotions is often described by the game industry people in one word; "*fun*" and it's the goal of game designers and game development studios to create "fun games".

The fact that playing a game is a voluntary activity, means that players are not obliged to return to it. They have to value the experience the game provides more than they value the time they invest in it, otherwise they have no incentive to play again. Games that make players feel this way are commonly called "*addictive*", as they make people want to play again and again. Koster [35] suggests that a game is interesting and fun to a player as long as it provides skill mastery; once the player master the game, it becomes boring. He also advocates that all games are edutainment[3]; they teach something to the players and as soon as there is nothing more to teach, players dismiss them.

Having examined the issue from the perspective of the game's consumers, the players, we can try and approach it from the angle of its creators, the game developers. Judging by the type of professionals needed to make a computer game, namely artists, musicians, writers, animators, in addition to engineers and programmers, we can assume that there is a significant creative aspect in game development.

If we consider the creation of software to be a complex task, we can assume that adding the element of required creativity and the need to bring certain emotions to the players, would only make the task harder to accomplish. According to Cellele *et al*. [41], emotions have a subjective nature, something that makes them inherently difficult to identify, specify and represent. They also point out that game designers have to capture and express emotional requirements, finding out how players are supposed to feel when playing a game. Moreover, Malone [38], using the results of empirical studies he conducted in 1981, comes to the conclusion that the implementation of specific

[3] Edutainment combines the methods of teaching with the form of games, to improve the students' learning interest and make teaching more effective. [72]

representations inside a game, which he calls "fantasies", can play a very important role in creating motivating environments, which players enjoy. He also says that these representations are not perceived equally by everyone, and can have the exact opposite effect on other types of players, reducing the "fun" factor and damaging the success of a gaming experience.

Taking into consideration all the above, we can deduce that the creation of successful gaming development software, in terms of player enjoyment, is indeed challenging and requires work in diverse fields, thus being quite different from traditional software.

2.2 Building game software

Since the appearance of the first commercial video games in the 80's [42], the industry has gone a long way. Nowadays, video games can be large software programs, often having millions of lines of code and are continually increasing in size and complexity [43], or as Blow states, they "have ballooned in complexity" (from 1994 to 2004) [44]. Mobile games in particular, even though they are typically not the largest game systems, present additional challenges for game developers, due to the resource constrains and fragmentation of mobile devices [45]. As discussed earlier in this thesis, a mobile game running on Android alone may have to support a staggering number of 1306 devices with different hardware and software specifications [16].

Managing a complex project is a hard task. Having a team with multi-disciplinary professionals, something required in game development, can add even more to the challenge, as communication among people of different fields can be more problematic. De Marco and Lister [46] state that software projects mainly suffer from management problems, rather than issues related to technology, while Jones [47] advocates that most factors associated with software disasters are directly or indirectly related to project management, often displaying delays and overruns that reach 100%, especially for larger systems. Charette [48] agrees that poor project management is a major factor for which software projects fail, adding bad communication, unrealistic goals and unmanaged risks to the list.

According to Flood [49], all game development post mortems say that the game was delivered with a delay, had many bugs and limited functionality and took much more work and pressure to build than originally planned. Flynt and Salem [50] state the importance of clearly defining the project's scope, saying that often this is the biggest reason for failure in game development. Finally, it is very common to include new features and functionality during the development phase, often because these features were discovered in other games and the development team decided to add them as well [50]. This practice, commonly known as *feature creep*, increases the project's size and makes it harder to manage.

However, since game development is not a linear process [50] and the purpose of the final product -being fun, enjoyable and addictive- is not guaranteed by the initial design and feature set, it can sometimes be crucial to change the design and implement new features towards that goal, even though this may affect scheduling and cost. On the mobile industry in particular, this may be even more prominent.

As discussed in Chapter 1, the growth of the mobile market has been rapid and its ecosystem has been changing constantly and in high speed. New trends appear within a few months, and it is hard to catch up, especially for smaller development teams. The appearance of tablets, the free to play model with In-App Purchases[4], the creation of services like Apple's Game Center[5] and Push Notifications, are just some examples of external changes that may lead developers to implement new features, even though they weren't originally planned. In some cases, these features may be more than a nice addition for players; they can be essential for being able to compete in a fierce market.

2.2.1 Software Development Life Cycle

The concept of Software Development Life Cycle was created in order to define a structure for the development process of software products, and ensure the compliance of the used process [51]. Software methodology can be divided in two major categories, heavyweight and lightweight [52].

The former are traditional methodologies, such as the Waterfall and Spiral model, and rely mainly on extensive documentation and planning, gathering and specifying user requirements upfront and sticking to the plan [51], [52]. However, these approaches have been found to be far from optimal, particularly when there are rapidly changing requirements the development team needs to adjust to. Because traditional methods rely so heavily on specifying requirements and do everything according to the planning made in the early stages, they fail to adapt in such situations and end up costing a lot in time and money [51].

On the other hand, modern, lightweight, methodologies work in a much different way. They focus on short iterative cycles and allow programmers to be faster and more efficient, and respond better to change [52]. These methods promote the inclusion of the customer in the development process, the interaction between the development team and the customer, and the communication between them through frequent face-to-face discussion, instead of documentation [52], [53]. Even though the newer methods perform better, they do not provide an optimal solution for every case. Agile Development, towards which the industry is moving, cannot satisfy all needs of the software industry on its own [53]. For example, Agile methodologies are not ideal for

[4] Applications that are free to download and use, and often provide additional paid content

[5] www.apple.com/game-center

creating reusable code [53]. Thus recent research is trying to combine lightweight processes with other processes, in order to overcome these limitations [53].

2.2.2 Milestones

Milestones are specific stages in a software development project that are used to track progress. Typically used in game development projects as well, they are defined by the producer when the project starts [54]. Some common major milestones used in game development are First *Playable*, *Alpha*, *Code Freeze*, *Beta*, *Code Release*. The contents of each milestone are loosely defined, later to be described in more detail. Sometimes a *Usability* milestone is added after Alpha and a *Gold Master* milestone is added at the end.

A typical breakdown of the major milestones is presented below.

First Playable
The first version that can demonstrate gameplay. This is usually the first time the game is shown to people outside the development team and serves as a proof of concept.

Alpha
Key gameplay functionality is implemented, and there are a lot of placeholder assets in the game (graphics, music, sounds).

Usability
The game's core features are complete and most of the functionality is implemented. A Usability Test is performed, to see if players can understand the controls of the game, perform important tasks and whether they find the software easy to use or not.

Code Freeze
No more features are added after this point, and the focus goes on bug fixing.

Beta
Game code and assets are complete. The game goes for beta testing, either private or public.

Code Release
During this stage the software is considered to be finished by the development team. They provide Release Candidate builds to the QA team, who test them for any missed bugs. When this process is over, the game is ready to ship.

Gold Master
The final build is prepared and shipped. For mobile games this means it is submitted to one or more application stores.

When a game development company is working with a publisher, it is common to have payments scheduled according to milestones. For example, the contract between the developer and publisher may state that an advance payment is to be paid on contract

signing, as soon as the Alpha build is ready, at Beta and at Gold Master, when the developer sends the finished product to the publisher.

2.3 Conclusions

Games appear to have substantial differences compared to other types of software. As studies and researchers suggest, the abstract nature of the goal of playing games is mainly what sets them apart from other types of software. Other than that, game development has all the characteristics of software development, faces the same challenges, and often fails for the same reasons. Additional obstacles that must be overcome include inhomogeneous teams, which make management harder, and the inability to define more solid requirements from the beginning, which can easily destroy estimates and scheduling. The major reasons for which game projects fail to be delivered or get completed way over budget and off schedule seem to be of managerial nature, rather than engineering-related. For modern mobile games this can probably be attributed to the rapidly changing market as well. Various attempts to remedy these issues have been made, and the industry is currently moving towards Agile development processes, although they do not seem to be enough. The current state in game development projects includes heavy use of prototypes, small iterative cycles, less documentation, more testing, and defined milestones from the start to the end of the project.

Part II

A Clockwork Brain for iOS Case Study

This part outlines the development process of Clockwork Brain, describe various important aspects of its development and analyzes several technical and design approaches, as well as decisions made, from the beginning of pre-production until product release and subsequent updates. The following chapters are not structured using a chronological order. The iterative approach in development, as well as the fact that at the time of writing there were already two updates of the product shipped on the market, make a thematic structure more appropriate. The content is divided into thematic chapters, each of them often referring to various time periods of the development. Since the scope of this thesis does not allow the full analysis of development, this case study selectively presents the most important aspects of the process, often in extensive detail.

Chapter 3

Overview, Process and Game Design

The current chapter presents the game for which we are doing the case study, describes its features, lists the team members that took part in the project and describes the development process. A large section is devoted to game design, where the design of the core features is presented in detail. A lot of the gameplay is analyzed, along with meta-game elements and monetization design. The last two sections deal with the process of selecting a development framework and creating prototypes.

3.1 About the game

A Clockwork Brain is a commercial puzzle game, developed by Total Eclipse. The application contains a series of mini-games, where players get to use their cognitive skills, such as visual & spatial ability, pattern matching, logic, language processing, arithmetic and memory.

Figure 3.1 Clockwork Brain as it looks on iOS devices, together with the game logo

Players have 60 seconds to answer correctly as many levels as they can. The levels get harder as they progress. If players performs well, they get score points and a time extension, allowing them to play even more levels. When the time is over, players get rewarded with their final score and a number of virtual coins, called *Sprocket Tokens*, depending on how well they did. The Tokens can be used later to unlock more features in the game.

The application provides global competition through Leaderboards and Achievements, as well as the ability for the players to share their performance with their Facebook friends.

The game was released on the Apple App Store on February 15, 2012 and has exceeded 300.000 downloads before the end of October 2012.

A brief description of the game is presented:

> *"Play fun, innovative puzzles and train your brain with A Clockwork Brain!*
>
> *Discover a series of unique mini-games especially created to test various cognitive abilities such as visual, spatial, logic, language, arithmetic, and memory. Everything in the game has been lovingly hand-painted with influences from Victorian Steampunk and Mayan art.*
>
> *Sprocket, the robot, will be your guide! Let the games begin!"*

3.2 Team

A Clockwork Brain was created by Total Eclipse, a Greek game development studio that was founded in 2004 by Argiris and Dimitrios Bendilas. Prior to Clockwork Brain, the studio had released 6 games in the international market, most of which were for desktop PCs. One of the titles was a mobile game, developed for the iOS and released on the App Store.

The core team that took part in the development of Clockwork Brain consisted of 6 people, often having multiple roles throughout the production. In addition to these, a music composer and a Sound FX studio were hired, as well as two more artists, who helped with some additional illustrations for the game.

Table 3.1 Team members and roles in A Clockwork Brain

Team Member	Main Roles
Argiris Bendilas	Producer, Art Direction, UI Design, Additional Game Design, Management
Dimitrios Bendilas	Software Architecture. Lead Game Design, Programming, Management
Maria Sifnioti	Asset Management, QA, Add. Game Design, Usability Testing Design
Yannis Argiropoulos	Programming, Additional Game Design
Jonatan Iversen-Ijve	Graphics
Nikos Mavros	Asset Management, QA

Development started on January 2011 and finished on February 2012, when the first version of the product was submitted to the App Store. For the first three months, during the preproduction phase, less than half of the team was working on the project;

the rest joined when production began, somewhere around April 2011. All the core team members were working on site, except for the artist. All artists, the composer and the SFX studio worked remotely and they were all located outside Greece, in both Europe and the USA.

3.3 Development Process

The development of A Clockwork Brain followed an iterative approach, using repeated development cycles with pre-defined milestones; Mini game Prototypes, Alpha, Usability, Beta and Gold Master. After creating an original plan, we began the design and implementation of the various features of the software, test and evaluated the state of each section of the game and then repeated the cycle. Based on the evaluation results, we would adjust the requirements, work on the planning and continue with the implementation of any required modifications or move on to the next set of features.

The change in requirements was often driven by the need to adapt in various areas, such as game design, usability or marketing, based on internal or external stimuli. The findings of in-house research, the rapid change of the mobile gaming market at the time, the results of usability and beta testing, and many ideas that emerged during brainstorming sessions, were only a few of the causes that triggered modifications onto the application, compared to the original planning.

The milestones we set were Project Initiation, Alpha, Usability, Beta and Gold Master for the application on its whole and Design, Prototype, Fully-skinned UI, Balanced state and Gold Master for each individual mini game.

3.4 Concept creation

The original concept for this game was to develop a single player game with a few mini games, featuring Sprocket, a character from Total Eclipse's franchise *The Clockwork Man*. The two Hidden Object & Adventure games of the Clockwork Man series are set in the Victorian era and feature a Steampunk theme, a sub-genre of science fiction where steam power is widely used and many futuristic machines exist, much like in Jule Verne's works. The story of the two games also involves ancient civilizations like the Mayas, where the environment is much more primitive and the use of stone and wood is prevalent. The world of Clockwork Brain is presented in the same setting, combining Steampunk with the Mayan philosophy and aesthetic. Sprocket, a Victorian clockwork, which is presented as an intelligent robot with great abilities in The Clockwork Man, would acts as the players' tutor, guiding them in the game.

As the concept started taking shape, it was decided that mini games would be original and designed in a way that trains the user's cognitive skills, like memory, logic and visual ability and that there would be many more of them than initially planned. The

primary goal would be gaming fun, rather than scientifically-backed brain training. This does not mean that the mini games would not actually hone a player's cognitive skills, but there was no intent to use an actual medical study to test each mini game against its brain training effects nor provide a standardized evaluation like the score given in IQ tests.

On a side note, the game concept we ended up with was a lot different than what we had originally planned, mostly in size and scope. This was due to the potential we saw in the mini game mechanics, as well as the fact that the mobile games market changed substantially within a few months after the project had started. As discussed in Section 2.2**Error! Reference source not found.**, the re-definition of the scope of a project can have a large impact on its scheduling and cost, something that became obvious in our case as well. As soon as the project had a larger scope, we had to re-evaluate most of the scheduling and budgeting, as well as the team members that would be needed for the production.

3.5 Game Design

Game design is the process of designing the rules of a game. This can refer to many areas, such as concept, core game mechanics, difficulty progression, game modes, player controls, scoring system, multiplayer availability, or meta-game elements.

3.5.1 Game Design Document (GDD)

One of the most important documents used in the development of a game is the Game Design Document. The GDD contains a variety of information regarding the game, from concept, characters and story to game play mechanics, level design and technical specifications.

The GDD is a dynamic document, meaning it is constantly updated during the development cycle. Its original form is used as a blueprint for the production. Later on, the document gets modified and extended as some original ideas are rejected, new features are added or various parameters are modified to meet certain needs.

The GDD of Clockwork Brain was drafted from the beginning of pre-production and was continuously updated until the product was released. Even after the release, more information was added, for features implemented on versions 1.1.0 and 1.2.0 of the application, and for future versions that will get released sometime later in the product lifetime, making the GDD a 24-page long document.

3.5.2 Features

The majority of the features were designed during preproduction and drafted in the initial GDD version. Some others, mostly meta-game elements, were decided during production.

The most important features of the application are presented below.

- Bite-sized, fast-pasted game play.
- Mini-games that are easy to grasp and everyone can play.
- Locked content & features that get unlocked with virtual currency.
- The ability to collect virtual currency through playing, depending on the player's performance.
- 4 mini-games available from the beginning, with 3 game packs of 2 mini-games available for purchase.
- An open architecture and UI design that allows us to add more game packs in the future.
- Localization in several languages for the mini games that use words in their game mechanics (9 languages in total).
- Global competition through Leaderboards and Achievements.

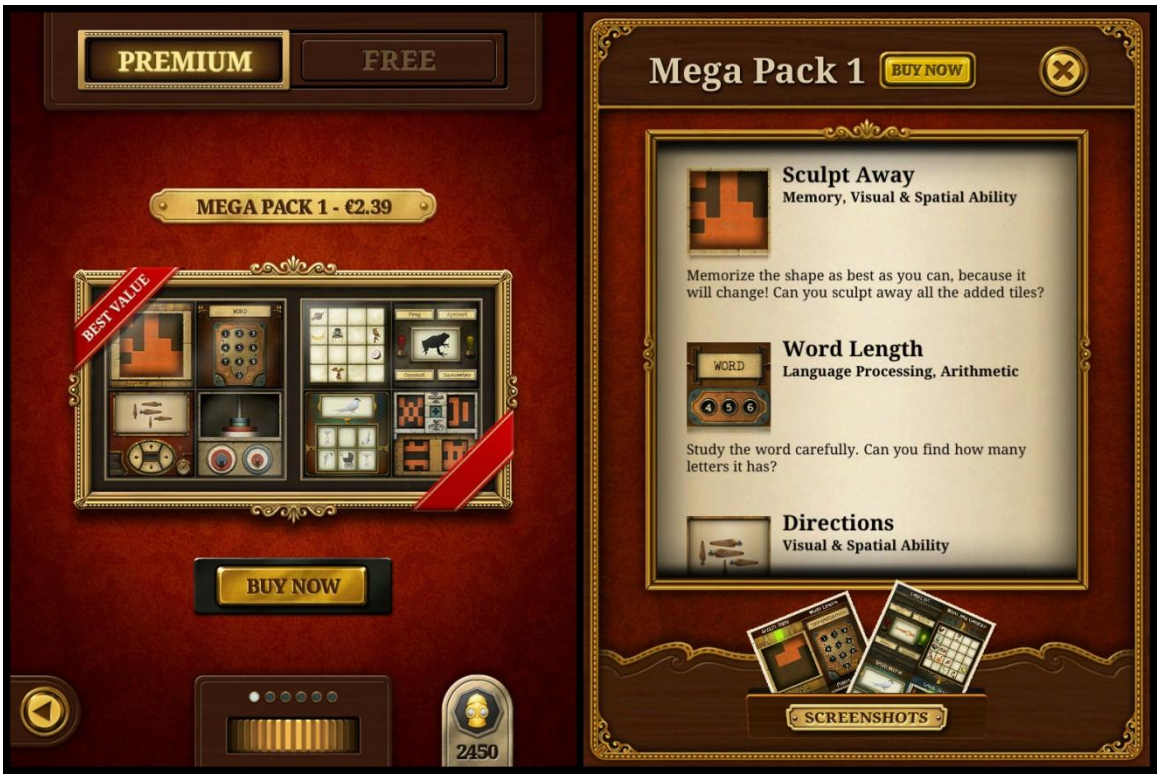
3.5.3 Monetization

When building a mobile game, it is important to define the monetization scheme early. The way the developer plans to make money out of the game can affect a great deal of the production, in almost every aspect: programming, UI design, art, copywriting. At the time the production began, there was a big shift on the App Store in this area; most of the top applications were given as free downloads and included In-App Purchases, which are premium content any player can buy. This trend became even more prominent during the production and statistics show that the top-grossing applications prefer this model, with 70% of the revenue across all apps now coming from Freemium apps (Free + In-App Purchases).

This was the revenue model we chose for A Clockwork Brain as well. Players would download a free application with 4 of the mini games unlocked. Separate game packs of 2 mini games each would be available for purchase, for anyone that wanted to extend the experience.

Later on, together with the first public update of the Free version, we would introduce a paid version, called Premium, that had all mini games unlocked from the beginning, for the small percentage of players that prefer to buy a product at once, rather than paying for In-App Purchases.

Figure 3.2 Users can purchase game packs using real money



3.5.4 Mini game design

As soon as the concept was defined, the design of the various mini games began. At the time, several other brain training games were available on the market, most of which cloned the mini game mechanics from one another. It was decided that all mini games of Clockwork Brain would be original, with the exception of a couple that would be twists of existing puzzles.

All mini games were first designed on paper, explaining their core mechanic and outlining all the variables that affected game play. The concept of each one was intentionally designed to be very simple, so that players could start playing immediately, with no more than a couple of lines of instructions. The simplicity of the core mechanic meant that all variables should be very carefully designed and defined, so that the game would have the appropriate depth in order to be challenging and fun for the players.

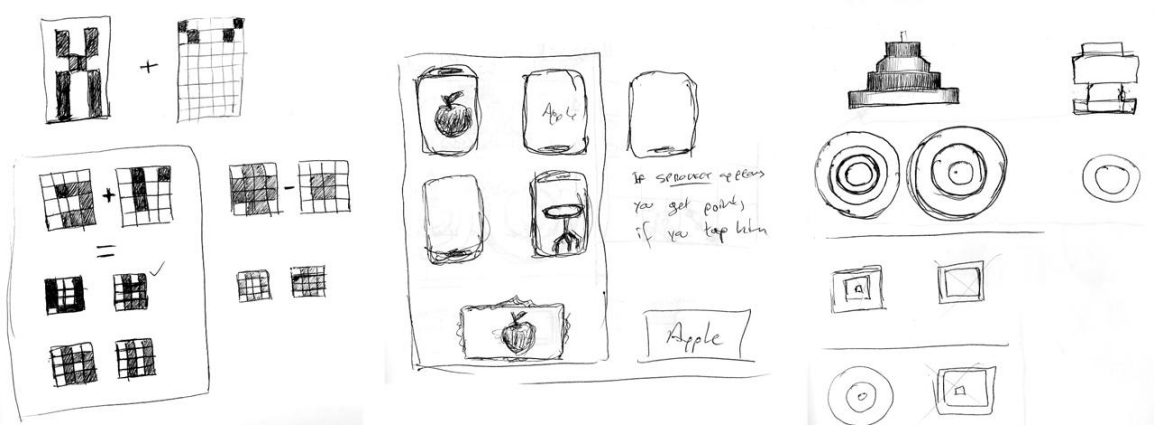
Each mini game would then become a playable prototype, which was tested on a real device, so that we could evaluate how fun and interesting it was, if its difficulty progressed nicely and if its controls allowed it to be fun and engaging. More of this process is described in Section 3.7.

3.5.5 Mini games

The original version of Clockwork Brain contained 11 mini games. In v.1.2.0 two more games were added.

The philosophy of the application called for mini-games that are not hard to play for the common brain, but are challenging when the time factor is added. Everyone can answer correctly in a level if there are no time constrains, but the real value for players comes when they try to pass as many levels as possible in a given timeframe. This allows for fast-paced game play, easy learning curves and a game that is easy to play, but difficult to master.

Figure 3.3 Early game design concepts of Logic Cards, Speed Match and Top View



Each level is created dynamically, so users can play thousands of levels that are different to each other and never know what to expect from the upcoming levels.

The 13 mini games in A Clockwork Brain are Scrolling Silhouettes, Missing Tiles, Anagrams, Chase the Numbers, Sculpt Away, Word Length, Directions, Points of View, What has Changed, Label It! Speed Match, Logic Tiles and Size Matters. Some of them are presented below.

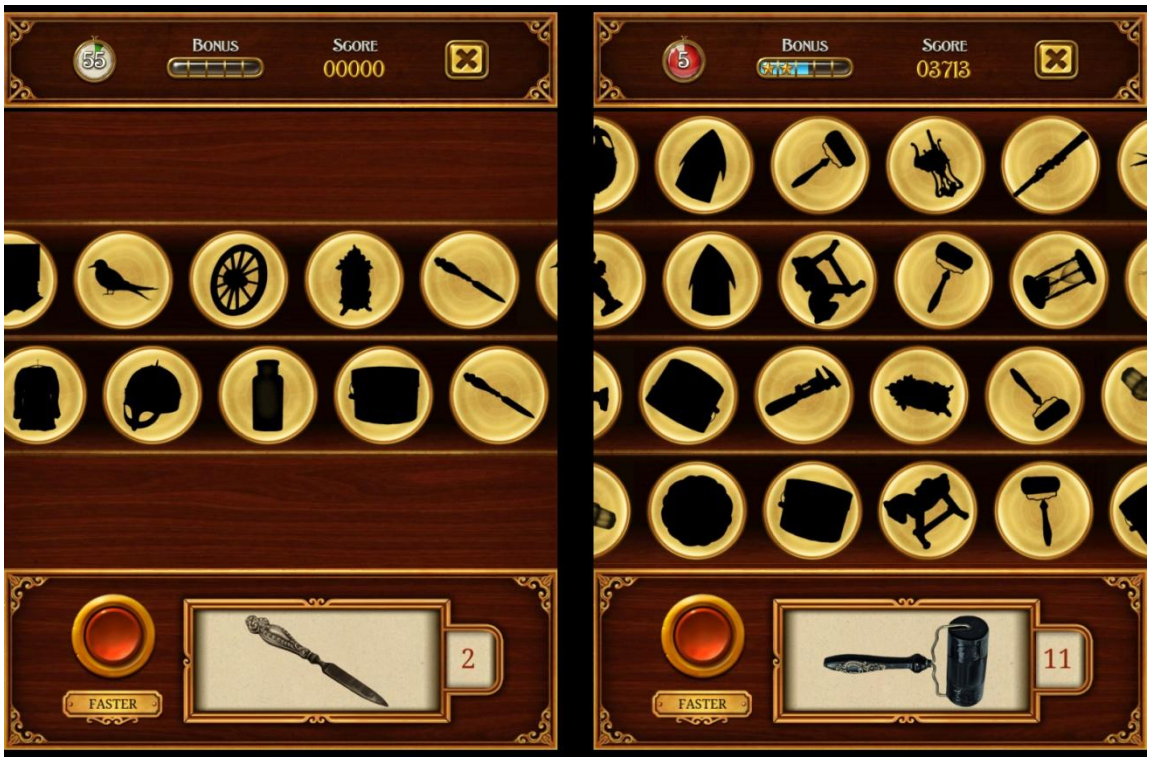
Scrolling Silhouettes

A pattern-matching game, where the player tries to find instances of a certain items among a number of moving item silhouettes of various shapes.

The items appear on rows as black silhouettes (filled with solid black color), which slide horizontally in a continuous movement. The first levels start off with 2 rows and slowly moving items. At the most difficult levels, there are 4 rows and the items are moving faster, appear rotated, scaled and are constantly rotating at the same time. The player is presented with one specific item in full color, and has to find all instances of this on the board.

The game gives bonuses for quick-matching; if the player finds 3 or more items within a few hundred milliseconds apart from each other, she gets extra score and if 5 or more items are matched in the same manner, she also gets extra time.

Figure 3.4 Easiest (left) and hardest (right) levels in Scrolling Silhouettes mini game



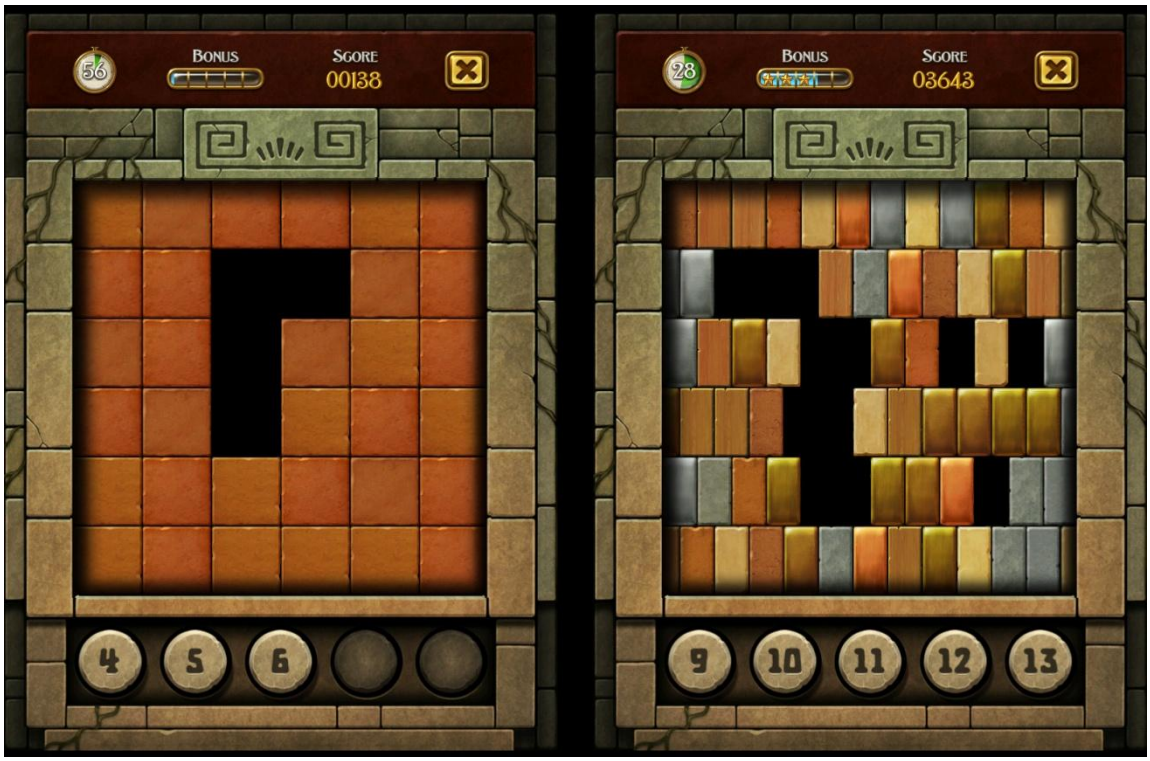
Due to the large number of items in the game (more than 1000 in total), there is a chance that the silhouettes of two different items look identical. For example, the shape of an orange is the same as that of a round shield. When presented as silhouettes, the player cannot distinguish between them and this can result to a mistake, which is not the player's fault. In order to prevent this, all items were categorized in several categories defined by their shape, which were read by the level creation algorithm, in order to exclude similar items in one level.

Missing Tiles

This mini game trains visual and arithmetic skills. Players have to count the missing tiles in a board full of blocks and provide the correct answer.

The variables that define the difficulty of each level are the size of the grid, the number of missing tiles, the number of available answers, the color and shape of the tiles, whether or not they form different clusters or one big cluster and whether they are perfectly aligned to each other or not. Easy levels have large, square tiles of the same color, all belong to one cluster, only a few of them are missing and the available answers are only a few. Harder levels have small, square or rectangular tiles, various colors, form several clusters, are not horizontally aligned, a lot of them are missing and the available answers are more.

Figure 3.5 Easiest (left) and hardest (right) levels in Missing Tiles mini game



In harder levels, the missing blocks usually form several clusters of various shapes, usually consisting of 2-8 blocks. One technique advanced players learn to use by themselves is to perform pattern-matching on these clusters, so that they can count the total number of missing blocks faster. This is where arithmetic comes. Players count the number of tiles in the clusters, cluster after cluster, each time adding the result to the total sum. This calls for basic addition skills, and although it may sound simple, when doing it very quickly it can prove challenging in advanced levels when the clock is ticking.

Top View

This mini game trains visual abilities and logic. In Top View, the player has to think how a stack of concentric, colored disks viewed from the side would look like when viewed from above.

The difficulty factor relies on various parameters, such as the number of disks, size variety, size ordering and total answers. Easier levels use a few disks, varying a lot in size, ordered from smaller to larger when moving from top to bottom and provide only a couple of available answers. Harder levels have many disks, often sized close to each other, with larger disks sometimes being stacked over smaller ones, and provide double the answers. Players need to take extra care noticing the instances where larger disks are placed above smaller ones, so that they do disregard any disk that is hidden under a larger one (see right screenshot in Figure 3.6).

Figure 3.6 Easiest (left) and hardest (right) levels in Top View mini game

Sculpt Away

This mini game tests the player's memory and pattern matching. It is presented in two phases. First, the player is shown a grid with only a part of it being filled with tiles. Then, the image is covered for a second and, when uncovered, the grid has now more filled tiles. The player needs to remember the previous state of the grid and find all newly added tiles.

Figure 3.7 Easiest (left) and hardest (right) levels in Sculpt Away mini game

The size of the tiles on the grid, the grid dimensions, the number of filled tiles, the number of newly added tiles and the amount of clustering all define the difficulty of each level. Easy levels have large tiles on a small grid, only a few are filled, few are added and tiles usually belong to one cluster. Hard levels present small tiles on large

grids, with a large percentage of the grid being filled and many tiles being added at the second phase, often forming one large cluster.

3.5.6 Bonus Levels

In each mini game session there are 5 milestones for which the player gets a bonus. They are called *Bonus Levels*. Every bonus level is presented with a golden star and gives extra score and extra time to the player. The player moves towards the next milestone with every correct answer she gives, and moves one step back for every wrong answer. This provides a visual representation of the players' performance and helps them set specific goals.

For each mini game there are different milestones. For example, the 3rd bonus in Anagram is on level 18 and gives 13 seconds of extra time, whereas the 3rd bonus in Directions is on level 36 and gives 10 seconds of extra time. The adjustment of these values (5 pairs for each mini game) constitutes a large part of game balancing, as explained in later chapters.

Figure 3.8 The bonus bar shows the player's bonus progress

3.5.7 Insane Round

If a player manages to get all 5 Bonus Levels, a special mode is unlocked. All the levels that follow are much harder and require great skill. If the player makes one mistake, the game immediately ends. Every time the player answers correctly, she gets a few extra seconds of time. The time awarded is custom for each mini game and decays as the insane levels progress. The score points earned in each level during an Insane Round are as much as 10 times more compared to the normal levels. Players have to play as many levels as possible, to score the best possible score.

Insane Round was created to give the chance to expert players to distinguish themselves from average players, both in terms of game play challenge as well as positioning in score leaderboards.

The difficulty parameters for insane levels are different for each mini game. For example in Word Length, the player has to find the length of two large words instead of one. In Directions there are more items and the size difference among them is minimal, often as little as 1 pixel. Finally, in Missing Tiles there are a lot of tiles missing, they are very small, not aligned vertically, form large clusters with less distinctive shape patterns and the available choices only differ by 1 point.

Figure 3.9 Insane Round: intro screen, Missing Tiles level and Anagrams level

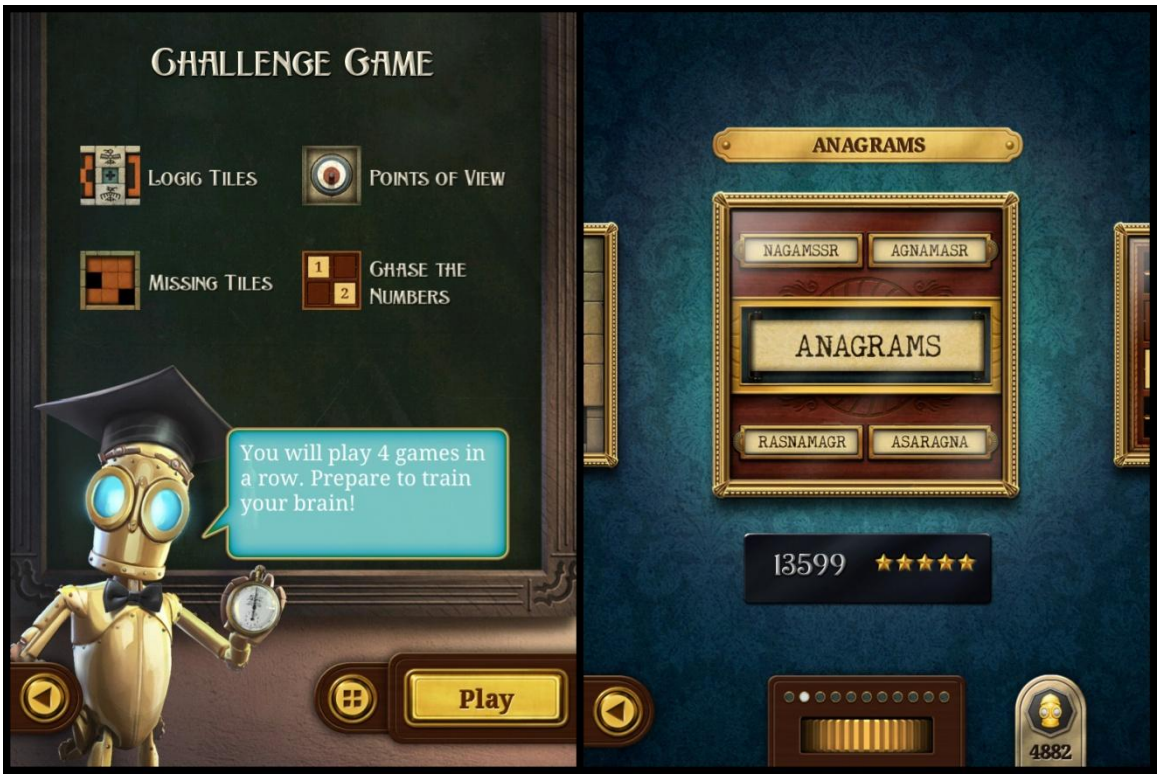
3.5.8 Game Modes

The game provides two different modes; Challenge and Single Game.

In Challenge, the player has to go through a session of 4 different mini-games that are randomly selected in the beginning of each session, from the mini-games that the player has unlocked or purchased. In v.1.1.0 a new feature was added, that allowed players to select which mini-games to play in a Challenge session.

Single Game lets the player choose any one of the available mini-games and play that alone. This is often preferred by the players, because everyone tends to like some mini-games more than others, and wants to play these more often than the rest. This mode is initially locked and players have to earn enough Sprocket Tokens, by playing a few Challenges, in order to unlock it and start using it.

Figure 3.10 User plays 4 random mini games in Challenge, or selects 1 in Single Game



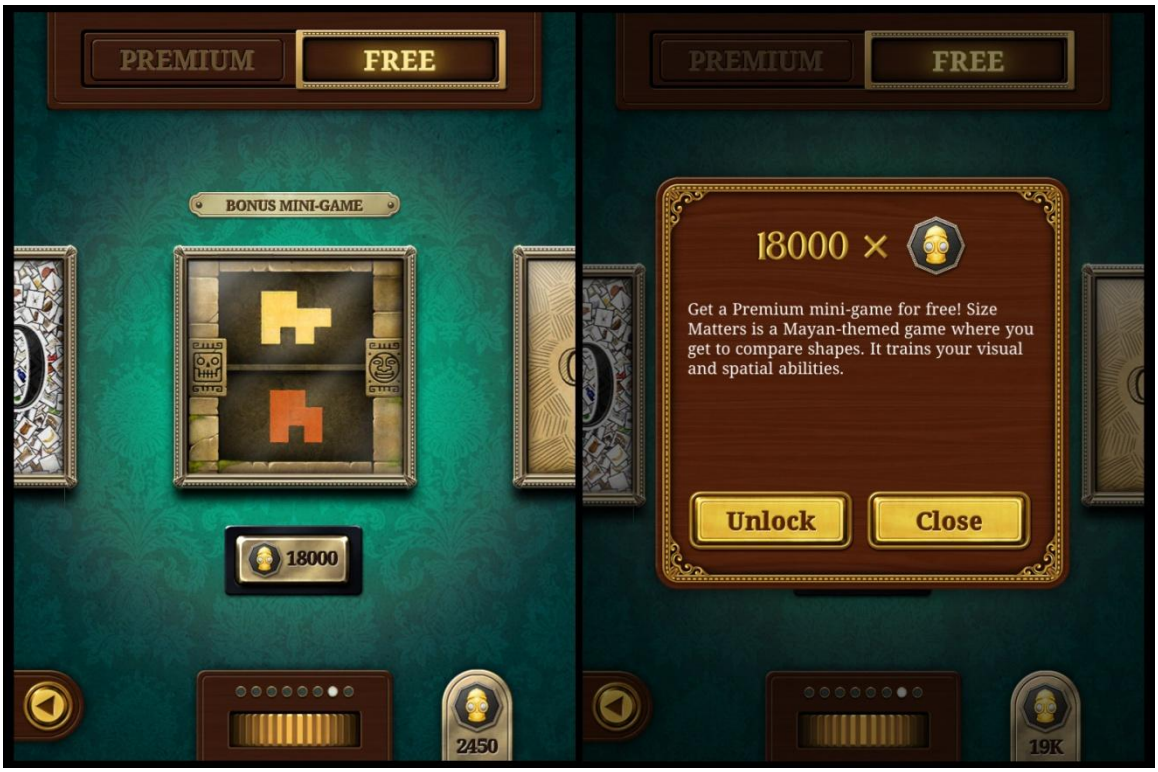
3.5.9 Token System & Free Upgrades

A good example of the use of meta-game elements is the Token System and its combination with the content unlocking mechanism, using Free Upgrades.

The concept of locked content is very common among games, as players enjoy being rewarded progressively with something of value, as they achieve something. A virtual coin system, where coins are earned as a reward for good performance, provides the sense of fulfillment and makes players feel good about themselves and the way they play. This becomes something of a meta-game. They play the normal game and earn something that can be exchanged in order to unlock more content and features in the actual game. This is a fun experience and sort of game in itself.

The Token System was designed having that in mind. The better one's performance is, the more tokens she earns. When she has collected enough, she can unlock various Upgrades, of different value and cost. The Single Game mode for example, is given as an Upgrade, requiring a small number of tokens to unlock (100). Other upgrades are Item Packs -which contain a few hundreds of new items to use in the mini games- a wallpaper, the Create Challenge feature which lets the player choose which mini games to play in a Challenge session and even a whole new mini game, which needs 18.000 tokens to unlock.

Figure 3.11 Users can unlock Free Upgrades using their Sprocket Tokens



Apart from being a fun experience, this also helps the product engage its users more and increases player retention and long-term usage of the application. It gives players a larger goal, an incentive to play more and become better, so that they get rewarded with what they have rightfully earned with hard work and it makes them feel important and successful.

3.6 Framework evaluation

When the most important aspects of the GDD started taking shape, the development team began preparing for the production phase, by examining the technical side of the project. The first task for the engineers was to study all elements of the GDD that had technical implications and think of ways of implementing them in the application and technical obstacles that might occur on the way. The game's features, the overall look and feel, the deployment platforms and the team's prior experience with certain technologies were the most important factors that would determine the selection of the development framework.

The game would be a 2D puzzle game, with emphasis on high quality hand-drawn graphics, few animations and focus on the game play, rather than impressive special effects. It was already decided that the application would run on iOS and launch for both iPhone/iPod Touch and iPad. Up until then, Total Eclipse had released several desktop titles and one mobile, on iOS. The desktop applications were all developed

using Flash and ActionScript, on which the development team had a very advanced know-how and experience.

3.6.1 Laying down the options

Taking all these parameters into consideration, it made sense that one option would be to develop the game in ActioScript, taking advantage of an existing, proven framework that was developed in-house and used in five products. Together with the existing tools, new ones would be built to meet the needs of mobile-specific aspects, and an number of optimizations would be used, in order to adjust to the limited resources of mobile devices. The main concern regarding this option was performance, as Flash had a negative reputation when it came to iOS applications. Even simple applications built with Flash and deployed for the iOS, were known to have major performance issues.

Another option was to make use of an in-house iOS engine we had developed for our previous mobile game. It was built in Objective-C, the native language for iOS applications, but lacked many of the features we would need for this title, as the previous game was much less demanding. However, for that game we had also built a framework that managed most of the application's functions that were not related to the game play itself, such as loading resources, managing In-App Purchases and downloadable content. This framework, named SolarWind, could be used for the upcoming game as well.

The last alternative was to use one of several frameworks available for building iOS games. At the time, one of the most popular was Cocos2D[6] for iOS. Originally built on Python, and later ported to native Objective-C, it displayed a good track record, with dozens of iOS games released, a few of them being the same size as the one we would be building. It had a strong community and was open source and free to use for commercial purposes.

3.6.2 Testing the alternatives

In the past, a hasty decision regarding the selection of the development framework for a game we were building cost the company valuable time and money. Having that negative experience in mind, we now decided to spend time to evaluate our alternatives properly and make the right decision.

In order to create a Flash application for the iOS, Flash Builder CS5 had to be used. Released by Adobe at that time, it feature the capability to compile for iOS as a major selling point. We downloaded the trial edition, which gave us 30 days of free use. Flash had a new option, which exported the application in a native iOS format file, which should be installed to an iOS device for testing on a real environment.

[6] www.cocos2d-iphone.com

The simplest way to see whether Flash was a viable option was to make a very basic test application and see its performance. Then, more elements would be gradually added, until we could evaluate its real potential. To our surprise, the verdict came much sooner than expected. The first prototype we created was a simple sliding puzzle game, where an image is sliced into square tiles, gets scrambled, and the player has to put the tiles in order, to create the original image. Unfortunately, when deployed to an iPhone, this simple game, with minimal use of resources, was very sluggish, to a point that made it unplayable. Reading on various forums and blogs helped us make a few optimizations, but none was enough to make an actual difference. Then, we created an even simpler application, which only showed a ball bouncing in the screen. Even that, achieved no more than 10 fps on the device. As a result, we decided to abandon the idea of building our game with Flash.

Aditionally, the idea of using our existing iOS engine was quickly rejected, as it did not provide much reusability for the type of game we wanted to make.

Our last option was Cocos2D. We started testing the tutorials and examples that came with the installation and they seemed to be working very well. Then, we built a simple prototype. We created the same application we did on Flash, the sliding puzzle, which ran very smoothly. The next step was to create a prototype of one of the mini games of Clockwork Brain. After a few days of work, the prototype was finished and tested fine on an iPhone device, usually running near 60 fps, with the touch controls having a good response time and a smooth feel. At that point we decided that the application would be built using Cocos2D as the core development framework and an extended version of SolarWind for many of the more specific aspects of the application.

3.6.3 Cocos2D for iOS

Cocos2D provided most of the core tools a 2D game application needs, wrapped up in a clean architecture with advanced flexibility. Its main features included

- Scene management
- Sprites and Spritesheets
- Advanced Image Format support
- Fast Textures (compressed formats like PVRTC)
- Advanced Animations
- UI features
- Text rendering support
- Sound support
- Touch (iOS) and Mouse/Keyboard (OSX) support
- OpenGL support
- Integrated Physics Engine

and more.

It came with many examples, which explained the use of its features in detail. The API was easy to use, but the documentation was not very detailed. The large community around Cocos2D was very active and friendly, as was the creator of the framework, who often helped developers facing a hard time.

At the time we started developing the game, Cocos2D was on version 0.8.0. When our game launched, there were already several updates released, most of which we had updated to during development, finally building on version 1.0.0 RC2. Some bugs that arose were fixed either by us, by members of the community or by the framework's creator.

3.6.4 SolarWind for iOS

This framework was developed by Total Eclipse for a mobile dress up game released a few months before the development of Clockwork Brain started. Its main features were:

- Application Setup
- Save/Restore Management
- Analytics management
- Upgrades management
- Crash management & reporting

For the new title, a lot of functionality was added or improved and the framework proved to be a valuable asset of the development team.

By the end of the development, SolarWind altogether composed more than 100 classes, and included a variety of other tools. Major features were:

- Advanced system for managing data externally from an online database
- Complete In-App Purchase support, bound with the downloadable content and upgrades systems
- Bug handling and reporting
- Advertisement management
- Highscores
- SQLite management
- Easing methods
- Various extensions for Cocos2D

3.7 Mini-game Prototyping

As soon as the framework was decided, the development team could start building prototypes for the various mini games.

Our main goal was to see if the design of the mini games, as described in the GDD, would work, resulting in a fun game experience. We would try to make all mini games

work, and we would reject any of them that did not meet our expectations. Games that passed this test would be added to the production pipeline. This would save us time and money, cutting down on the required work from the developers during preproduction. Additionally, the artist would begin working on the game's graphics after the flow of each mini game was outlined in detail, and this would save us even more resources.

In addition to minimizing costs, prototyping helped us in other areas as well. As mentioned before, our team had never worked with Cocos2D in the past. Creating the prototypes helped the developers familiarize themselves with the framework, learn its API, discover its quirks, make mistakes early and experience a smoother transition from previous frameworks, than if we started building a large scale application at once. Also, after 3-4 prototypes were finalized, the software engineer started identifying patterns in the game flow and begun working on the overall architecture of the application.

As it turned out, all mini games were made into prototypes with great success and made it into production. This outcome helped increase the morale and confidence of the team, as things were starting to look nice, even at an early stage.

3.7.1 Finding issues early

Prototyping also revealed some important game play issues that affected future game design decisions, UI control design and art. For example, in Scrolling Silhouettes, initially having the codename *Moving Objects*, the original design called for full-colored objects inside the main screen and only the target object in silhouette form. When testing out the prototype among the team members, it was obvious that the full color of the items was very tiring to the players' eyes and that pattern matching was hard, resulting to a game that wasn't fun. The design solution was to invert the use of silhouettes and have most objects in silhouette form and only the target object in full color. When the artist begun working on the graphics for this mini game, he took this parameter into consideration and managed to save time, creating the art the proper way from the beginning.

Figure 3.12 Finished prototypes of Scrolling Silhouettes, Top View and Missing Tiles

Also, we found out that in harder levels, were there are many different items on the screen, the scrolling speed was somewhat slow and the game felt boring and not responsive enough. On the other hand, if we increased the default speed, it became irritating for some players that could not catch up with all the items moving around so fast. Prototyping helped us identify the problem and try out a solution; adding a *Faster* button, that players could press at will, and make the items scroll faster for as long as they needed to.

Another example of a useful finding during prototyping was in another mini game, Top View. The original design included an additional difficulty parameter, the shape of the items used in the stack. The GDD included both disks and squares. From the side, both shapes would look almost alike, but from the top they would appear different. As it turned out, the rest of the attributes were more than enough to create a challenging experience and the extra one was actually damaging the fun factor, instead of adding any value. Testing the prototypes helped us exclude this parameter, simplifying the code and preparing better specifications for the artist, who would not have to design redundant images in the production phase.

Each of the mini games took around 2-4 days to prototype, depending on its complexity. After having completed 6 prototypes in independent applications, we decided that we had enough to start creating the main application.

3.8 Conclusions

Clockwork Brain was a typical example of a software project that went off schedule. For Total Eclipse it was the most unconventional production in this regard, as the scope of the project changed completely compared to when the project first took off. Also, it was the first time that the market changed so much and in such short time, that we had to re-

evaluate many aspects of the product, including its monetization scheme. We created a free-to-play game for the first time, as we saw a trend emerging in the App Store. By the end of the production this trend had become so apparent that made us realize we had made the right choice.

The framework selection process went smoothly. In retrospect, we could have gone with a cross-platform framework, that would have enabled us to launch on many application stores at once. This would have bloated the schedule and the development cost even more though, and the device fragmentation on other platforms would not have been easy to cope with. Prototyping worked very well for this game and it was executed very effectively. Because the game was made of many different mini games, we could not have proceeded without prototypes and have the same high quality results. On the other hand, we underestimated the time needed to fine-tune the design of so many different mini games.

Chapter 4

Game Balancing

This chapter deals with a very important part of the game development process, the balancing of the game. It approaches the subject from a technical angle, outlining the tools we used to make every mini game fun to play, for both amateur and seasoned players. The concept of easing functions is presented, describing the way they can be an important tool for both game designers and programmers, as they try to adjust the difficulty of a parameter, or a game in its whole.

4.1 Overview

One of the most challenging aspects of this production was game balancing. The team had a lot of experience in puzzle creation. However, this game was unique in the way that it contained 13 different mini games, all of which should be balanced independently and then re-balanced as a whole, so that there were little or no differences among them. All games should feel the same to the average player, in regards to difficulty. This required a lot of work, actually much more than anticipated.

Balancing was executed in many phases, spread out throughout the production. The game designer described in detail, inside the Game Design Document, all the parameters that affected difficulty, for each mini game, thus providing a thorough qualitative analysis. The next part was to define the value ranges for every parameter and the formula to use for changing each value throughout the levels.

4.2 Ranges

In order to create a range for a parameter that is dependent on the current level of a game, two pairs of values are needed; one for the beginning and one for the end. One value defines the current level and the other defines the difficulty attribute. For example, assume that the number of missing blocks in Missing Tiles starts from 2 on Level 0 and gradually increases up to 15 blocks on Level X. To be able to apply a specific formula, the 'X' needs to be defined, so that the formula can work with actual values. The first thing we did was to define a *maximum level* for each mini game. This value would mean that the game would not get any more difficult after that level and that all further levels would have the same difficulty as the maximum level.

Setting up these attributes always requires a trial-and-error approach. The first round relies on the experience of the game designer. Then, minor or major changes are made, many times, after testing the updated values each time. Our goal was to provide a gradual increase of difficulty, without overwhelming the players. Each mini game was

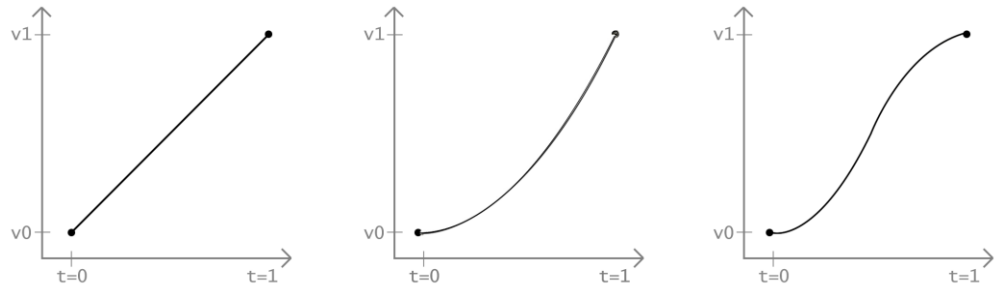
unique in that respect, because the attributes that affected difficulty were never the same among the mini games.

4.3 Easing functions

An easing function determines the way a dependent variable changes within a range. By default, a range uses a linear function, but there are a number of other formulas one can use in game design, such as Sine or Exponential formulas. Consider the following example for the *Word Length* mini game:

The maximum level is set to 12 (zero-based). The attribute we are trying to set up is the length of the word, which increases as the levels advance. The length of the word can be from 4 characters (Level 0) to 16 characters (Level 12). If we use a linear easing, the difficulty increases equally in each step. Implementing another easing function, however, could result in increasing the difficulty less in the beginning and more in the end.

Figure 4.1 Examples of easing curves (linear, easeIn, easeInOut)



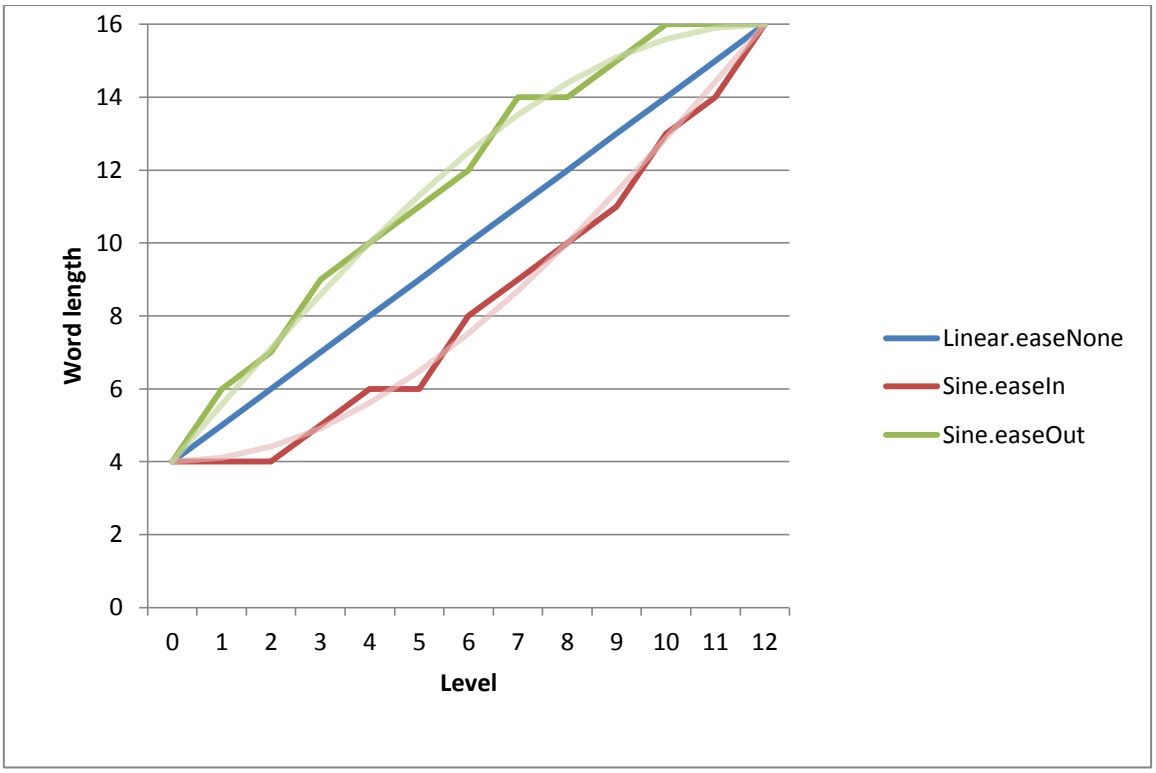
An example is illustrated below, where three different easing formulas are examined together, as seen in the following code:

```
length = round(getValue(level, levMin, levMax, lengthMin, lengthMax, "Linear.easeNone"))
length = round(getValue(level, levMin, levMax, lengthMin, lengthMax, "Sine.easeIn"))
length = round(getValue(level, levMin, levMax, lengthMin, lengthMax, "Sine.easeOut"))
```

Table 4.1 Word length attribute values in each level using various easing functions

	Level												
	0	1	2	3	4	5	6	7	8	9	10	11	12
	Word length												
Linear.easeNone	4	5	6	7	8	9	10	11	12	13	14	15	16
Sine.easeIn	4	4	4	5	6	6	8	9	10	11	13	14	16
Sine.easeOut	4	6	7	9	10	11	12	14	14	15	16	16	16

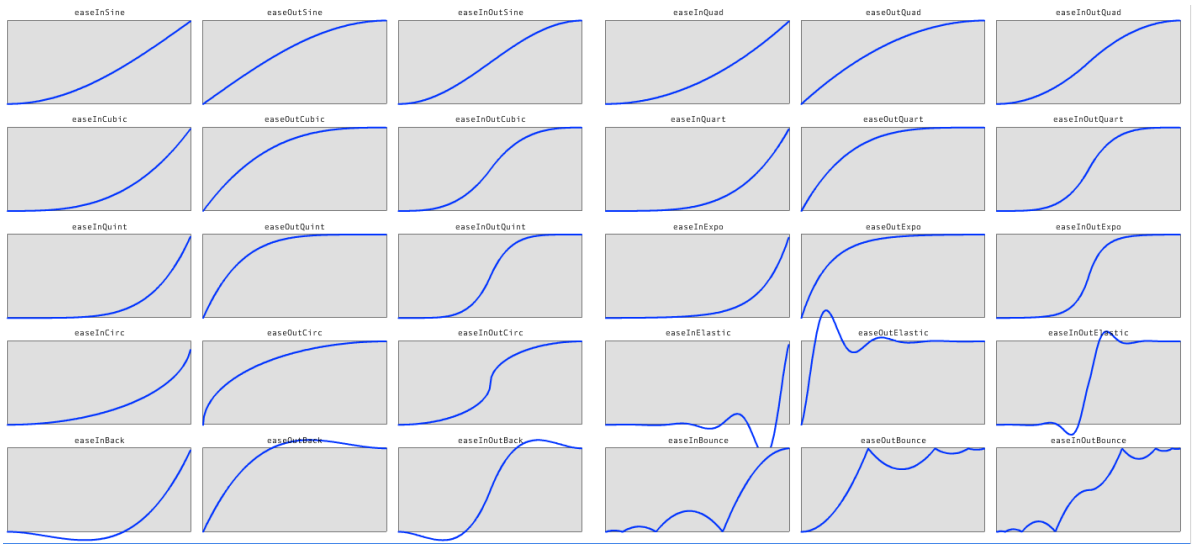
Chart 4.1 Different easing functions on the word length attribute, with raw and rounded values



Both the table and the chart show that the length of the word changes in a much different fashion using the various easing functions. If we want a more gradual increase in difficulty, we can use Sine.easeIn. If a steeper curve is desired, Sine.easeOut works best. Having this method in mind, we can take advantage of many of the easing functions available, such as Quadratic, Qubic, Quintic, Sine, Exponential, depending on how smooth or steep we need to make the change of a certain attribute.

For every mini game in Clockwork Brain, we calculated the minimum and maximum values and the easing functions, based on experimentation and testing among the team members, friends, players during usability and beta. Until the very end of the production, when every aspect of the game was being fine-tuned, these parameters were being modified, until they produced the best possible experience for the majority of players.

Figure 4.2 Various easing functions and their variants



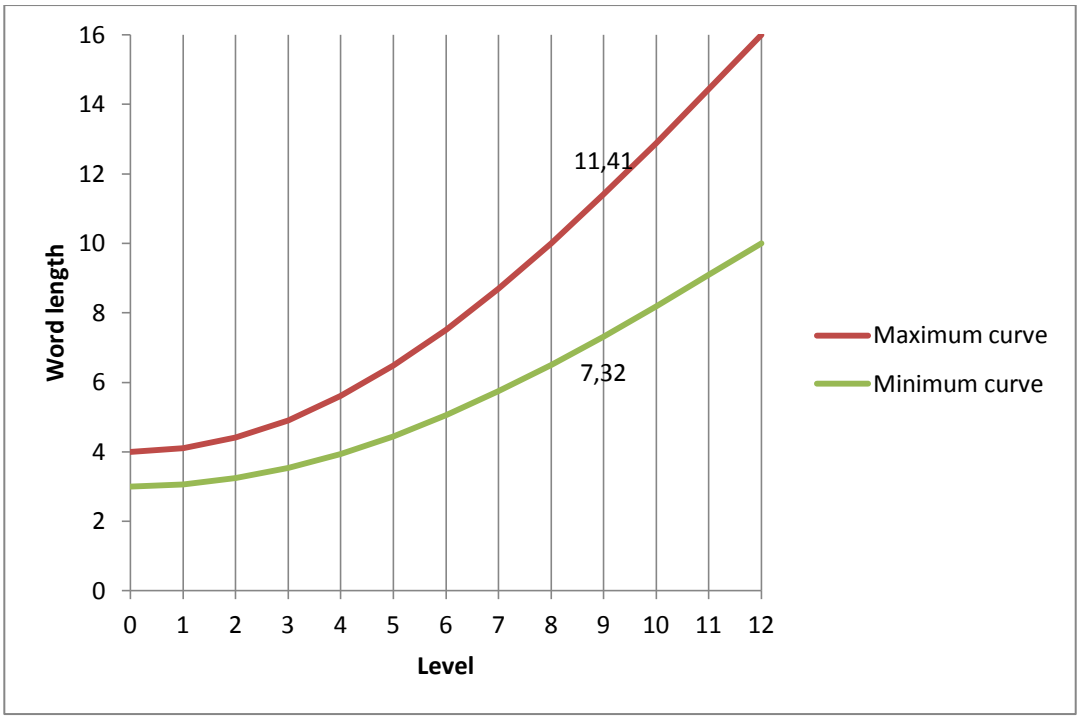
Source: Tweener [55]

4.4 Randomization

Randomization is often used in game design to provide variety. Players do not want to face the same circumstances again and again, nor do they always want to know what to expect next. Randomizing parameters, such as the number of disks in Top View, gives the sense of variety. In some cases though, randomization is more of a necessity, an essential element that is crucial to the very purpose of the game. For instance, in Word Length, if we used the value for the length of the word as derived from Table 4.1, no matter what easing was used, the player would soon memorize the length of the word in each level, or at least know that the word in each level would have more or equal characters, compared to the previous level. This would rule out a lot of the options and would essentially ruin the fun for the player and create a game that wasn't fun, as it would stop be challenging at all.

In order to add randomization in the algorithm, we implemented a different method. We created two different, "parallel" ranges and we used a random number between them, in each level. The first curve would range from 4 to 16, and the second from 3 to 10, both using a Sine.easeIn easing. For each level, we would calculate the values given by both ranges, then create a new temporary range with these values as the minimum and maximum limits and get a random number between these two limits, using a linear easing. The following chart illustrates this method, where the upper and lower limits for Level 9 are marked. In this case, the algorithm for Level 9 would get a random number between 11.41 and 7.32, different each time.

Chart 4.2 Use of dual ranges for randomizing the word length attribute



A similar technique was used in *Missing Tiles*, for both the number of missing blocks and the values of the available options. The "n*umber of blocks"* attribute has exactly the same behavior as the "l*ength of word"* attribute we examined earlier, so the use of a dual-range system was ideal. What was different here was that Missing Tiles provided specific options to the user, rather than displaying a dial and let the user type in the result, as done in Word Length. The values shown on these options are quite important, because if they are not randomized enough, they may hint at the correct answer and help the player win the level. A slightly more complex algorithm was used here, in order to provide options that cannot be easily excluded by the player.

Figure 4.3 The available options in Missing Tiles

4.5 Data-driven balancing

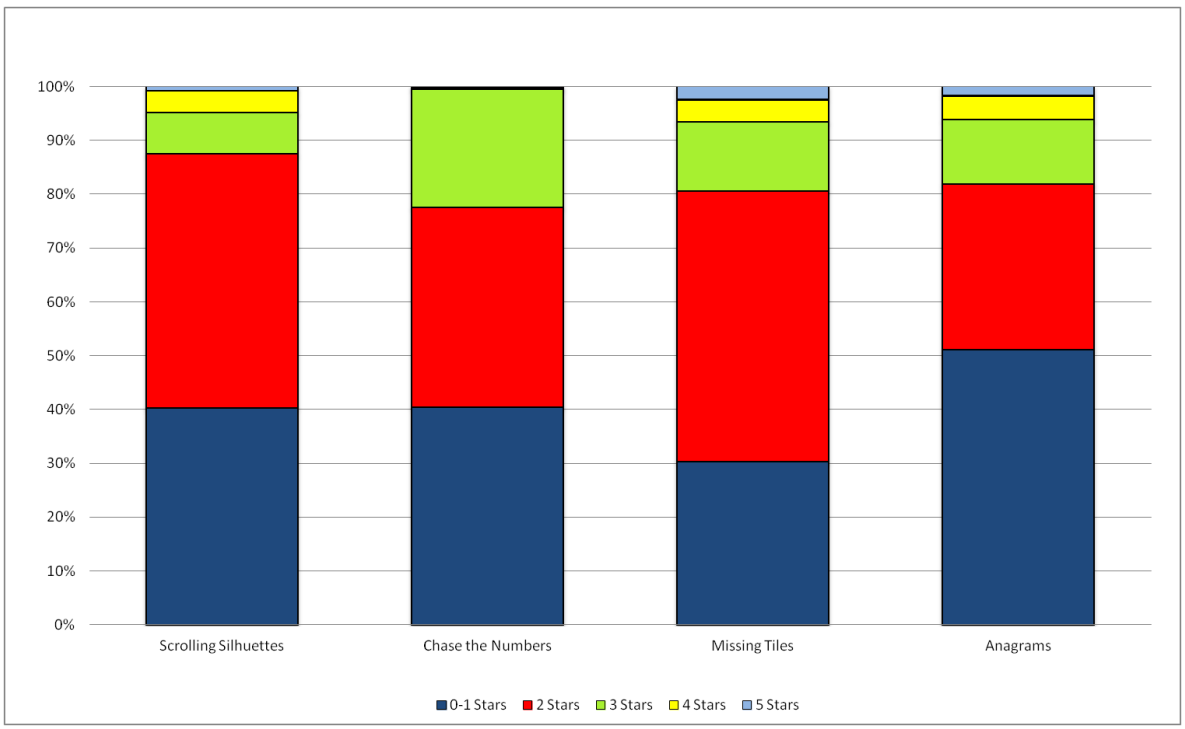
Game designers often base their decisions regarding balancing on experience and intuition. Other times, they rely on observing players. Most of the times though, the collection and analysis of data from real people playing the game is the best way to create a challenging experience for players of varying experience levels.

There are numerous ways to collect players' playing data. In Clockwork Brain we used three different methods. The first method was in-game analytics, which is further detailed in Chapter 8. We implemented analytics during the development of the game and collected data from people on the team, usability participants and beta testers. Another method was an in-game database that recorded session data. The last one was manual recording, where team members would play the game and enter their data on an Excel worksheet. Depending on the types of data we needed and the sample we had to work with, we selected an appropriate method.

The data we collected were analyzed and used to create a number of different charts, which would give us a good idea of the current state of balancing. Then, after necessary changes were made, we tested again and collected new rounds of data, repeating the cycle until no more changes were required.

Chart 4.3 shows the bonus level distribution for the first 4 mini games. The distributions are quite balanced, but it is obvious that players are having trouble getting more than 3 stars in Chase the Numbers. Using the chart, we can adjust the difficulty parameters in this particular mini game, and try to bring it closer to the other games. In a similar fashion, several other attributes of the games were examined and adjusted, such as score, flawless sessions and number of Insane Rounds. This procedure was carried out many times throughout development and took a great number of resources to complete, many more than initially anticipated. It was incredibly helpful though, as it allowed us to balance the mini games very effectively, as the positive reviews and ratings from players showed.

Chart 4.3 Bonus distribution of the 4 free mini games



4.6 Conclusions

Balancing Clockwork Brain proved to be way more challenging and time consuming than expected. A number of tools helped us carry out the balancing process and were invaluable for creating a fun experience for our players. As the reviews and ratings showed, people greatly appreciated the balancing of the game, as there were many comments that said the game made them feel good about their performance from the beginning, but was also challenging and fun to them. All the above tools, ranges, easing functions, randomization and data collection & analysis were instrumental in this regard.

Chapter 5

Game graphics

This chapter discusses the game's visual aspects. The approach is both artistic and technical. It describes the concept behind the game's look and feel, and it analyzes technical aspects of managing image assets. Finally, it presents the effect of visual elements on game design, by showing how changes in the art can disturb the balancing of game attributes.

5.1 Overview

The graphics and overall look and theme were an important aspect of the production. Most of the brain training games on the mobile market focused on a "medical" and scientific look and used graphics that could be described as cold, dry and sterile. Our approach was completely different, as we wanted to provide a great experience to a wide audience, not only those that played brain training games, but also the ones that played puzzle games in general.

5.2 Illustrations

Up until June 2012, six months in development, we used "dummy" or temporary graphics for all mini games, created by the producer and the game designer. It wasn't until mid June when the illustrator joined the team and started working full time on the game's graphics. At that time, most of the mini games were finished to a point that all mechanics were implemented and two rounds of balancing was already complete.

Figure 5.1 Various screens from the game, using a Victorian & Steampunk theme

The concept around the graphics was closely bound to the Steampunk philosophy and Victorian design, which governed the game as a whole. All mini games, as well as other scenes in the game, should be presented as contraptions of that era, and every small component and every animation on the screen should make sense and seem as it could actually belong to a real apparatus. This requirement made things quite challenging. The

producer had to work closely with the artist and the game designer in order to produce such realistic scenes, without diminishing the quality of the game play mechanics in each mini game.

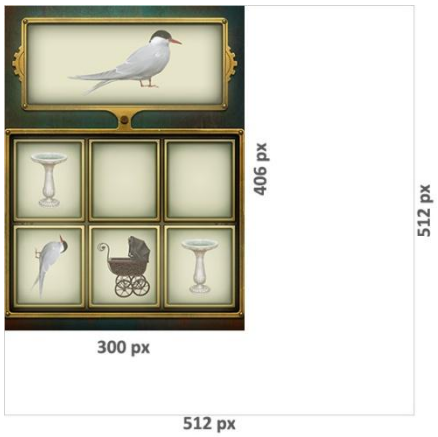
5.3 Memory & performance

One of the biggest technical limitations in game mobile development is the device memory. Modern smartphone devices often have 512 MB of memory. At the time, based on the devices we targeted, the least available memory was 256 MB on iPhone 3GS and iPad and the most was 512 MB on iPhone 4. A game like Clockwork Brain, that uses a lot of graphics, needs good optimization techniques in order to stay in the limit and avoid application crashes due to lack of free memory.

5.3.1 Atlases

One of the most common techniques used in game development is the packing of multiple images in one big image, which is known with the term *sprite sheet* or *texture atlas.* The purpose of this is dual.

Figure 5.2 Unused space in a texture



Textures in graphic cards often require dimensions of the power of 2. When using individual images, regardless of their size, they always take up as much space as the next rectangle that has dimensions of the power of two. When using a lot of images, they can easily take up a lot of memory, while at the same time leaving a lot of unused space, which is "filled" with blank pixels. For example, an image of 406x300 pixels size, will eventually use the same amount of memory as an image that is 512x512 pixels, as illustrated below.

Figure 5.3 Various sprite sheets used in the game

Packing many images together in a large file that has dimensions of the power of two helps take advantage of the blank pixels and save a lot of space.

Atlases can also affect rendering speed greatly. The graphics hardware treats atlases as single units and can quickly render multiple objects that have the same texture, by applying the same image and modifying its coordinates on every object, in order to display the desired sprite of the sprite sheet.

In A Clockwork Brain, all images were grouped in atlases, to take advantage of both optimizations. The process of creating and managing assets in atlases is time-consuming and requires a lot of optimization, as images that belong together semantically often do not fit in the desired dimensions or the sprite sheet dynamics change as more images are added along the way. The creation and maintenance of atlases spanned several months in the development of the game, which in the end used 75 atlases, containing hundreds of sprites in total. All atlases were managed using TexturePacker [56], a great program that creates and manages sprite sheets, providing dozens of features needed for game development .

5.3.2 Image compression

Besides the common PNG and JPG formats, there are other compressed image formats, which can benefit both the performance and memory consumption of an application. One of these formats, PVRTC, uses a lossy compression technique that blends two or more low-frequency signals, in a way that avoids artifacts, provides low decompression cost (faster loading time) and smaller memory footprint. Since its compression algorithm is lossy, images that are exported to this format will have a lesser quality, compared to

the original image. The contents of the image determine the amount of quality that gets lost, and whether the result is or is not acceptable [57].

Complex images, with rich color palettes and heavy gradients can often result to poor-quality PVRTC textures. This is exactly the kind of images A Clockwork Brain has, since its theme is very realistic and relies on colorful graphics with extended use of shadows, highlights and gradients. Nevertheless, the benefits of these textures are very important, so we tried to use them as much as possible. We mostly created PVRTC textures out of background images, that had the largest memory footprint because of their size, as well as other images that appeared to not lose substantial quality. In general, PVRTC was our first choice when exporting atlases, but we often found that the resulting images did not meet the quality standards set for the production. In those cases, we used the PNG format. Overall, of the 75 atlases the game has in total, 28 of them are PVRTC.

5.3.3 Balancing memory consumption and loading time

As discussed, there are ways to reduce memory consumption in an application, and other ways to decrease loading time. In general, the more images loaded in memory, the faster the various scenes of the application will load, because the resources need are ready to use. The downside is that this can increase memory usage. On the other hand, when loading only the images needed in each scene, memory usage stays low but users have to wait each time they move to another scene. Finding the balance between these two situations can prove to be tricky. Sometimes one needs to give up a little on one of the two aspects of the application, sacrificing speed or stability, with the latter often potentially having more serious implications to the user experience. Other times, it is just a matter of finding the optimal balance that produces the best noticeable conditions for the players.

We faced several similar situations during development, one of which was very crucial, because it affected players just after they had launched the game. When someone would initially download the app, only one of the two game modes would be available: the Challenge mode. The scenes that followed after the player had tapped on the Challenge button on the main menu contained a lot of heavy graphics, some of which were used in multiple scenes, throughout the flow of the application.

Figure 5.4 Identical sprites used in Challenge and Results screens

For example, as seen in Figure 5.4, there is a large image of a blackboard as well as Sprocket in the Challenge overview screen, right after the player has pressed the Challenge button. The sprite of Sprocket is composed of 3 separate images, that could change whenever we needed to show him turn his head to the side or hold a stick instead of a stopwatch. The same images were being used later in the game, both at the mini game description scene and the results screen, where Sprocket announced the player's score.

Initially we would load the images in Challenge overview scene, keep them loaded in the description scene and unload them as soon as the memory-hungry mini games begun, in order to reduce memory usage. Then, when the result scene appeared, we would load them again. Even though this helped with the memory limitations, it produced a poor user experience, because of the frequent occurrences of loading, which made players repeatedly wait. The solution we came up with was to optimize these images as much as we could without losing in quality and load them once in the beginning of the application, never unloading them again. The decrease in loading time was so dramatic that a couple of the team members who tested the application without knowing about this modification thought something went wrong and the scene did not load completely.

5.4 Re-balancing difficulty

When moving from temporary/prototype graphics to final graphics, sometimes the game play can be affected, usually because of the extra time it takes for a new animation to play, or because of the difference in contrast some of the new elements might have compared to the old ones. In these cases, after the final graphics have been integrated in the game, some re-balancing must be done.

Figure 5.5 Prototype and final graphics of Directions mini game

In Directions, for example, the prototype displayed the items in front of a patterned background and in every new level the existing items would disappear and the new ones would instantly appear, with no animation at all. When the final graphics were finished, though, items were being presented as painted on an endless piece of paper, which would roll to the left after the end of each level, revealing the items of the upcoming level. After experimenting, the duration of the rolling animation was set to 350 milliseconds. Even though this number may seem small, in a game session of 40 levels, which is what an average player would normally achieve, it adds more than 12 seconds of play time. When the overall time limit for a session is 60 seconds, it is obvious that this small animation can have a great impact on the game play and the perceived difficulty of a mini game.

After this change, we had to revisit various parameters related to the bonus level and difficulty of this mini game. The goal was to implement these animations, and all extra visual elements in general, in a way that made sure they add to the player's experience,

instead of ruining it. The producer worked with the game designer, fine-tuning all the attributes, for each mini game separately, so that the final graphics were integrated gracefully.

5.5 Conclusions

Creating the visuals for Clockwork Brain was not an easy task. The Steampunk and Mayan theme, albeit visually interesting, brought some challenging work for many of the team members. The limited resources in portable devices did not help either, as the game's rich graphics were difficult to manage, making memory management and performance optimization quite difficult. The transition between the placeholder graphics to the final graphics was also time consuming, as it required us to go through an extra round of balancing for each mini game.

Chapter 6

Testing

This chapter analyzes two important milestones of the development process; Usability testing and Beta testing. It explains the process used to carry out both and examines their results.

6.1 Usability testing

6.1.1 Overview

Usability testing is a method used to evaluate products by observing people who are actually using them. Its purpose is to see how real users interact with a product, if they find it easy to use, if they stumble upon any issues and if they can successfully complete certain given tasks. Prior to Clockwork Brain, for all our self-funded productions, we had been testing our games in a much less formal way, by asking friends and family members to play, while we were watching. This allowed us to collect valuable information, but could not be compared to a formal usability testing session. In the three productions for which we had a publishing deal, our publisher always carried out the usability testing and sent us the results, which were always eye-opening. This time we decided to design and execute the usability testing on our own. The usability testing process took one month to complete, ending in mid November, 2011.

Figure 6.1 Room setup for usability testing

During the usability, we would ask people to come to our offices and we would place them in a room, alone, and ask them to play the game. We would be in the next room, monitoring them with the use of two cameras; one pointing at the screen of the mobile device and the other recording the user. This way, we would be able to watch the

application and the player at the same time, thus evaluating her reactions in relation to what was happening in the game, without influencing the player.

The usability sessions took place in the conference room, inside Total Eclipse's offices. The participant would be inside that room and the monitoring team would be outside, connected to the room through the two cameras. The required monitoring software was selected after extensive research and the equipment used to hold the camera pointing the device screen was created from scratch by our team [58].

6.1.2 Session design

Clockwork Brain is a puzzle game that targets players of all ages and experience levels; it's a game for everyone. Since we valued ease of use a lot, we wanted to see how easy it would be for people of different experience with such games to start playing the game. Based on our past findings while testing various games with friends and family, we had seen that inexperienced users often exposed important usability issues, which accustomed users can't. On the other hand, experienced users have a different view, which is also valuable. Thus, we decided to test it on people in a diverse range, including hardcore gamers, causal gamers and people that did not own a touchphone [59]. For recruiting the testers we used Facebook and Twitter.

The application had a large number of features we would like to test, but we knew we could only test the most important ones. We decided to focus on two aspects of the application; the individual mini games and the first impression of the game and a few of its core features. Since the free application comes with only 4 mini games unlocked, and one of the features we wanted to test was the use of Tokens for unlocking game packs, we had a conflict. We needed two distinct usability sessions, with different designs, using a separate group of testers in each [59].

The first session would test the individual mini games. The user would play each mini game 3 times, then complete a questionnaire for each mini game and finally would have a closing discussion with us, in the end of the session [59]. The second session focused on the overall game experience, the first impression of the player in particular, as well as some important features of the game, such as the two game modes and the purchasing process [60].

6.1.3 Results

The usability testing for Clockwork Brain helped us greatly in making a few very important modifications in the design of the game. The feedback we got for the game was very good, and we saw that players really enjoyed playing. We managed to collect valuable feedback for the application flow, as well as the individual mini games and gain insights that helped us improve the game substantially.

For example, we found out that players were having trouble with two of the mini games, Sculpt Away and What has Changed, which both have some similarities in the mechanics used when changing levels. Many of the participants could not understand when a level was changing, something that led to a poor performance and made them feel bad. Most players never managed to get a grasp of these games, even after playing each 3 times in a row. The changes we made on the two mini games, due to the findings of the usability, had a great impact.

Another important decision that came from the usability was the naming of one of the game modes. When the usability took place, the two game modes were called *Challenge* and *Practice*. When watching users play the game, we noticed that some of them did not unlock the Practice Upgrade, even though they earned enough tokens and visited the Upgrades section. Others did unlocked the upgrade, but did not attempt to play Practice sessions, or at least took them a long time before they did. When discussing the naming of the two game modes with players, it became clear that the name of the Practice mode confused them, as they thought that they would not play "real" sessions in there. A few players even suggested names for this mode, according to what would make sense to them. Finally, we decided to change the name to Single Game, after the end of the usability session, and that's how the product was shipped on the market.

Finally, we discovered flaws in the game flow, as players would not perform some actions we intended them to do, like visiting the Upgrades screen sooner, after having earned enough Tokens to unlock an upgrade. The information we gathered helped us create hooks in the game, providing better feedback to the players and making them understand how the application worked in a much more effective way.

Figure 6.2 First-Token message added in results screen after Usability findings

6.2 Beta testing

Just like with Usability, this was the first time Total Eclipse conducted an official Beta Test for a mobile game. Our previous mobile title, Maya's Dress Up, was only tested by friends and relatives on our devices. This time we wanted to do it properly and have a formal Beta Test, with as many participants as possible.

6.2.1 Goals

We started working on the Beta in the beginning of January, 2012 and sent it to the users on January 16. Before starting the implementation, we defined our goals, stating what exactly we wanted to accomplish with the Beta. We had 3 main goals

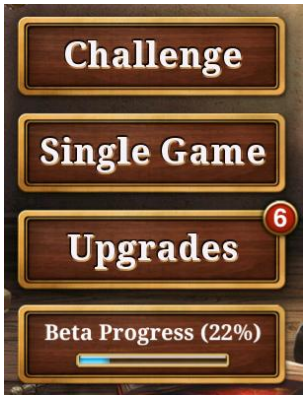
- Collect bugs & issues
- Get feedback for the game itself
- Collect player performance statistics, to use for fine-tuning the balancing of the mini games

The goals we defined helped us streamline the whole process and select the appropriate tools to use.

6.2.2 Design

From a user's perspective, this is what we wanted the Beta to be like: install the application, play and explore the game, give feedback. We wanted players to provide us with their feedback after having spent some time on the application, so we decided to make it easier for them, by displaying their progression on the beta, so that they knew how much more they needed to play before submitting their feedback.

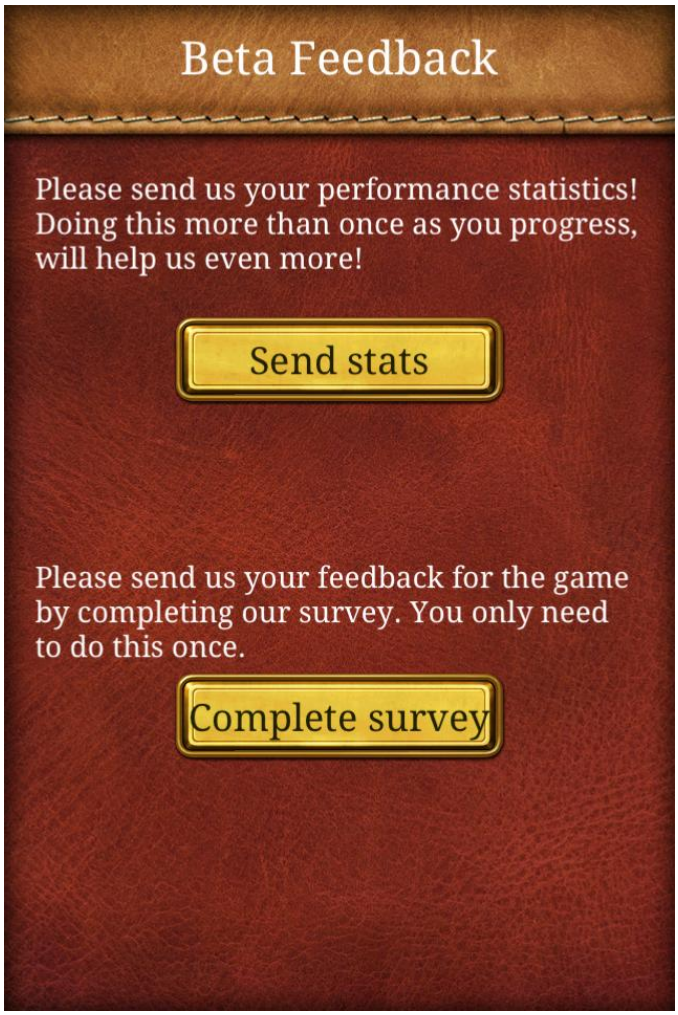
Figure 6.3 Indicator of the player's progress in Beta on Main Menu screen



When they reached 100%, we automatically showed them a screen with 2 buttons, asking for their feedback. This information was critical to us, so we did not want to give them the opportunity to procrastinate or probably forget to send their feedback altogether. As a result, there was no close button on this screen and if they restarted the app without having submitted their feedback, the submission screen would pop up automatically.

We needed players to test every aspect of the game, so naturally we wanted them to play all mini games, even the locked ones, and go through the process that allows players to purchase Upgrades as well. Normally, premium Upgrades are bought using real money, through In-App Purchases. Of course, we would not want to charge our beta testers, but the IAPs would probably not work on an Ad Hoc distribution[7] anyway. In order to be able to achieve both goals, we enabled a simulation of the buying process, which worked exactly like the real one, except it did not ask for the player's login details and did not use a credit card. This way we were able to test how usable was the purchasing system we had designed, and get feedback on all mini games as well. The only mini game that we did not test was Size Matters, the bonus mini game that required 28.000 Sprocket Tokens at the time, in order to be unlocked. We knew that there was no way players could earn that amount of tokens during the limited beta time, but it was not a large issue, because we valued the feedback on the "earn tokens-spend them on Upgrades" mechanism more.

Figure 6.4 Player Submission screen for Beta version



[7] Ad Hoc distribution allows developers to share an app with up to 100 iOS devices, via email or the developers' server [73]

6.2.3 Tools

Distribution

The iOS is a closed and proprietary platform. Its DRM allows only for applications purchased through Apple's App Store and all applications follow a centralized approval by Apple personnel before being made available for download. The only exception is Ad Hoc distribution, which allows developers to share an application with up to 100 iOS devices. This number is considered as small by the developer community for beta testing purposes, but it's a limitation we have to comply with, as there is no way to circumvent it. The procedure for sending an Ad Hoc application to multiple users requires several steps, such as

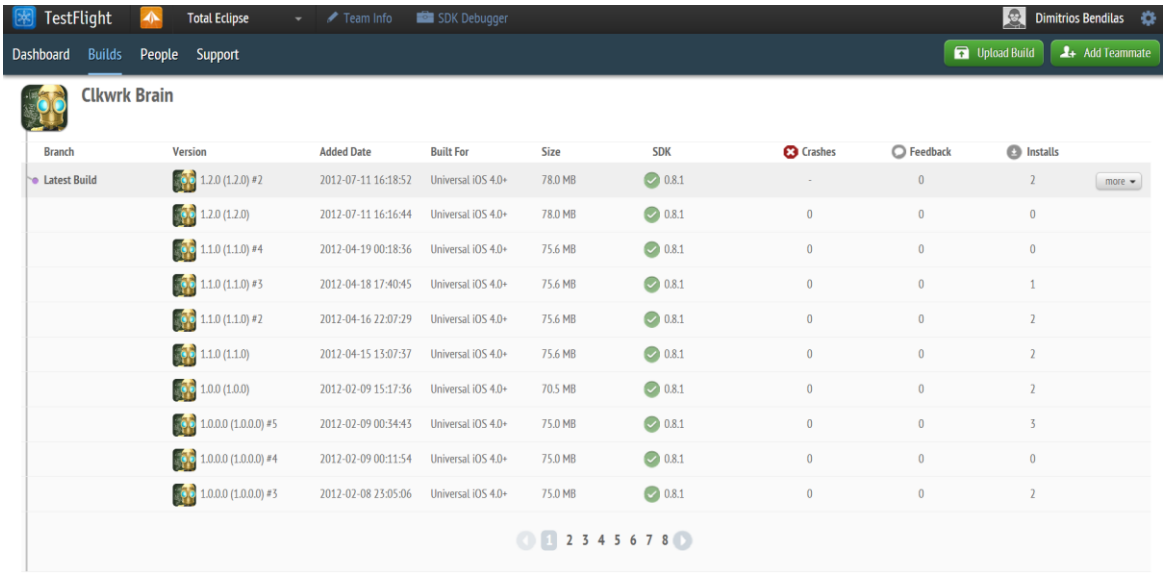
- acquiring the Device ID of each user
- adding all IDs to the Apple provisioning portal
- creating a provisioning profile
- sending this profile to all users and explaining how they should install it on their device
- create a special build of the application and send it to the users, so that they install it on their device

There are a number of software solutions that can do much of this work, more transparently and in a more usable manner for beta testers.

The most popular of these solutions is TestFlight[8], a free service, which we used for the beta of Clockwork Brain. We had been using TestFlight extensively before the beta, for sharing builds among the development team. In general, there are two ways of installing an application on an iOS device, other than downloading them from the App Store; by connecting the device to the machine where the application is being developed and installing the app directly through XCode or iTunes, or by creating an Ad Hoc build and sending it to the users remotely. The later is much more convenient, so we used it extensively throughout the development process, whenever there was a new build we wanted to test internally, among the team members.

[8] www.testflightapp.com

Figure 6.5 A list of builds distributed during development with TestFlight



Analytics

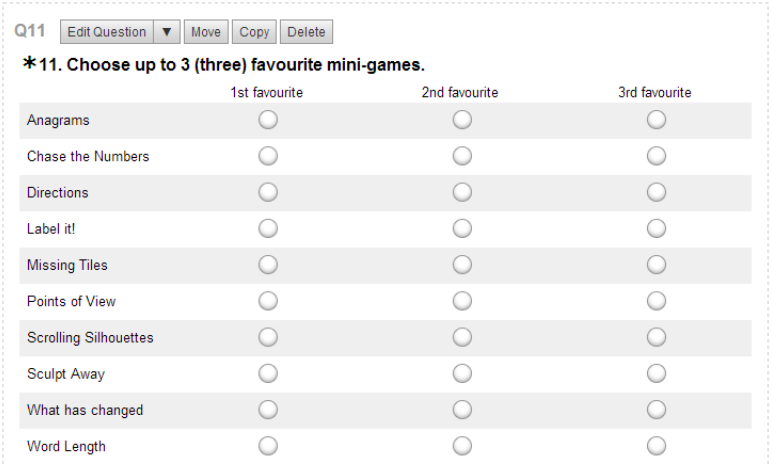
In order to collect the performance data from beta players, we used two sources. The first was the local database where we kept all information regarding the mini games player, score collected and other performance statistics for each user. In order to collect this, we created a script that attached the database on an email message and sent it to us. This allowed us to have data from individual players. In addition to this, we used Flurry, a mobile analytics platform, that collected many game events and provided useful reports. More on Flurry can be found on Chapter .

Survey

The last method we used for collecting data was an online survey. For this we used SurveyMonkey[9], one of the most well-known digital survey tools. This allowed us to gather information on general game aspects, rather than performance-specific ones, such as if players enjoyed the game, which mini games they liked the most, or how often they play mobile games.

[9] www.surveymonkey.com

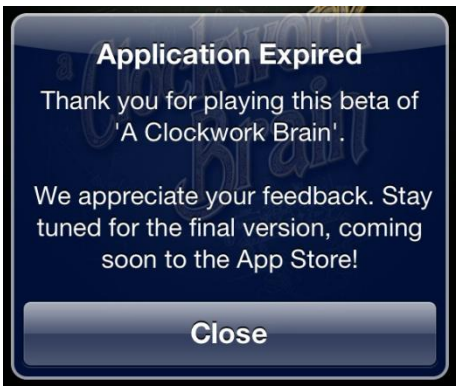
Figure 6.6 One of the 22 Beta survey questions of Clockwork Brain on SurveyMonkey



Expiration

The beta version we sent to the players had to expire on a given date, so that they could not continue using the application after the beta was over. In order to do that, we set a hard-coded expiration date inside the code and performed a check as soon as the app launched. We checked the current date, based on information collected live from online time servers, to stop players from cheating by modifying their device date, and if the expiration date had passed, we displayed a message and stopped the application from going further.

Figure 6.7 Beta expiration message



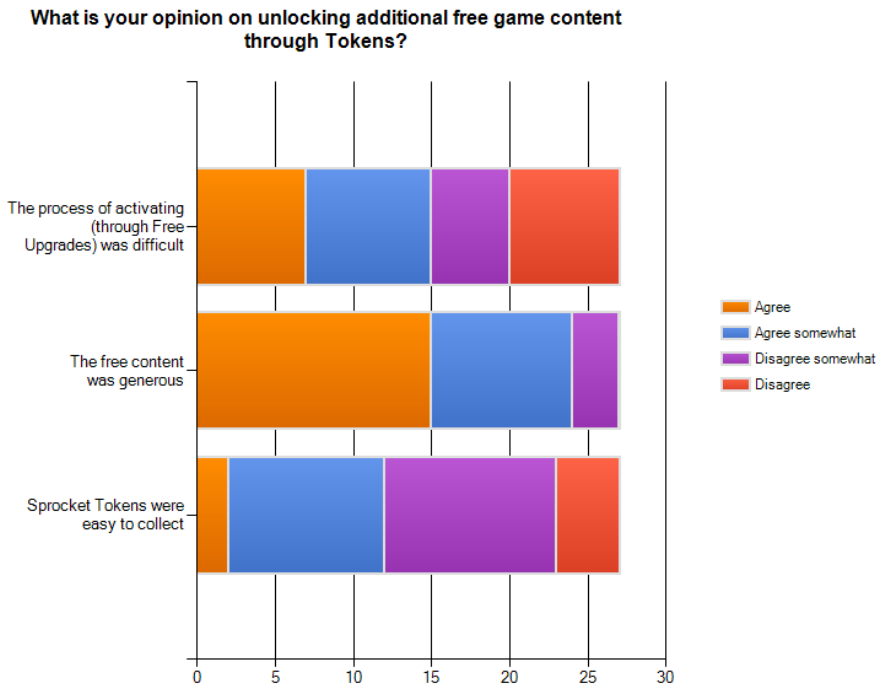
6.2.4 Creating a user base

In order to find players for beta testing the game, we mainly used social media and word of mouth. We posted on the company's Facebook page and twitter account, as well as a couple of gaming fora. We made sure to get both friends, who would most definitely spent some time on the game, as well as strangers, who tend to be less biased and sometimes provide more sincere and meaningful feedback. In the end, we managed to enlist around 70 players.

6.2.5 Results

The beta version was sent out on Monday, January 16, 2012 and gave players until Friday, January 20 to play and send their feedback. As it turned out, some of the players that had played the game, did not have enough time to finish the beta, and did not make it until that deadline. This lead us to sending out an updated version on Saturday, that altered the expiration date, allowing for submission until Sunday, 22nd of January. The new version also contained some of the latest features we managed to complete during the beta week, such as the ability to view local High scores and global Leaderboards through Game Center and OpenFeint[10], various minor features and some optimizations.

Figure 6.8 Player responses on a certain survey question for Clockwork Brain's beta



Of the 70 people that signed up for the beta, 49 of them actually downloaded and installed the application and 44 of them re-installed the updated version we uploaded after the expiration of the first. We got 27 completed surveys, 35 database submissions and 1133 analytics sessions of users playing 2,500 mini game sessions, for more than 100 hours in total, providing us with data for 110 different game events.

Overall we considered the beta testing to be a great success. We managed to collect large amounts of data, regarding the opinion of players for the game, as well as valuable analytics which helped us fine-tune the balancing of the individual mini games. Also, we caught a critical bug that we would not have found out about in any way just by testing on our devices, as the settings we had did not let the bug appear. One thing we would do better in future betas would be to give more time to the testers to complete the beta

[10] www.openfeint.com. Company now named GREE

and send us their feedback. We would at least include one full weekend, because during weekends people tend to be more relaxed and find time for things like beta testing.

6.3 Conclusions

This was the first time we conducted Usability and Beta testing ourselves. Our previous experience from working with a publisher, as well as the extensive research we did helped us design and execute both with great success. Even though they pushed the schedule back by several weeks, their results were invaluable and contributed immensely in the quality of the final product. Conducting these sessions by ourselves provided us with a lot of experience we did not have before. Some mistakes we made were identified and recorded, so as to be avoided in future projects.

Chapter 7

Technical Details

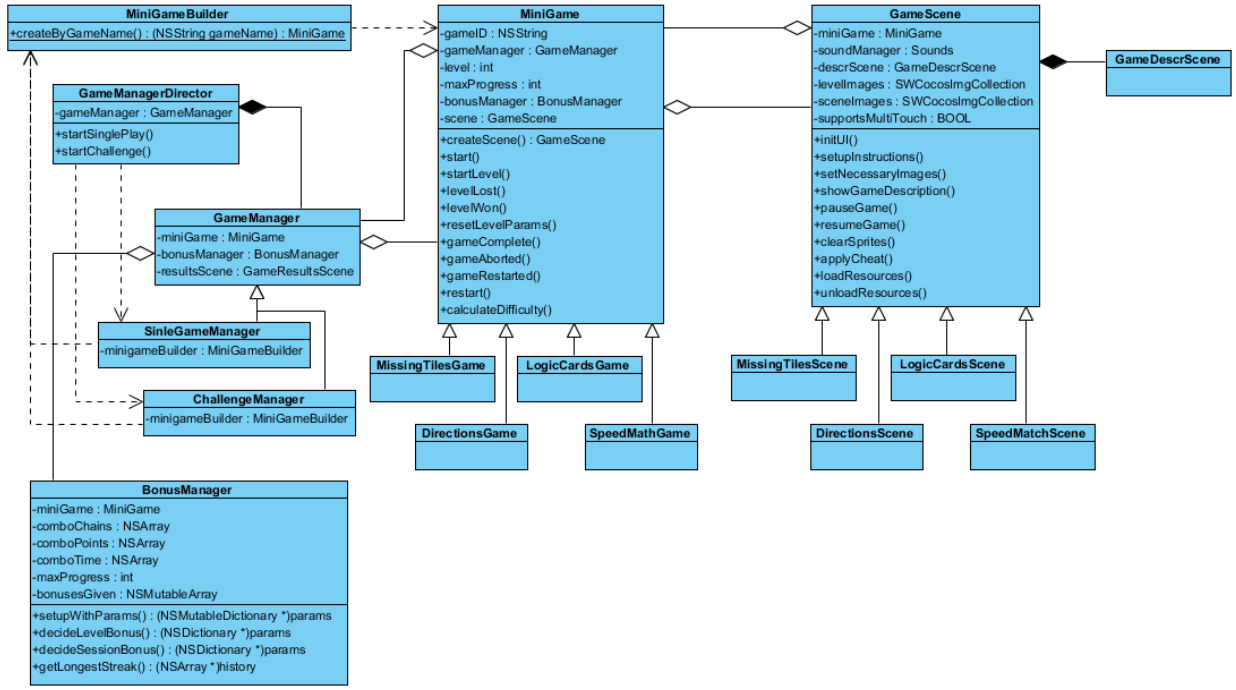
This chapter contains various aspects of a technical nature, that constitute an important part of the game's production. Most of them probably concern all game development teams that work on mobile platforms. They following sections present specific problems we had and illustrate the solutions we implemented, often using diagrams and code samples. Issues discussed are remote configuration & caching, backwards compatibility, In-App Purchases and building for multiple devices.

7.1 Software Architecture

7.1.1 Overview

The architecture of the application was based on Object Oriented design. A number of Design Patterns were implemented, to improve maintainability, extensibility and code reuse.

Figure 7.1 Class diagram of the core game play classes



Some of the Design Patterns used in the application are listed below, along with a brief description of the issue they solve.

7.1.2 Template Method

Used in several instances throughout the architecture, the most characteristic being the model and view classes of the mini games. All mini games have common a behavior in

many aspects. For instance, the visible flow is exactly the same in all of them; the game is initiated, the description scene is shown, the first level begins and then new levels are presented as players make the correct or wrong choice each time. Additionally, much of the inner workings of a mini game are repeated; the calculation of difficulty for each level, the reset of parameters when players restart a mini game, the broadcasting of analytics in certain important events and many more. Furthermore, each mini game has some special features and often needs to modify the implementation of the super class.

The basic, reusable functionality exists in the `MiniGame` and `GameScene` classes and the individual implementations and special features are integrated in their subclasses, such as `MissingTilesGame`, `DirectionsGame`, `MissingTilesScene` and `DirectionsScene`. The structure was created as soon as the first prototypes were finished, and was continuously built upon when new functionality needs were presented during production.

7.1.3 Builder

The two subclasses of `GameManager`, `SingleGameManager` and `ChallengeGameManager`, access an instance of `MiniGameBuilder`, which creates the appropriate `MiniGame` subclass. This helps decoupling the individual mini game subclasses from the game managers, which only need to be aware of the `MiniGame` superclass. The active manager subclass calls the `createByGameName` method of `MiniGameBuilder`, providing only the string ID of the mini game, and the builder class creates and retirns the specific mini game class that corresponds to the ID.

7.1.4 Command

In version 1.2.0 of the game we implemented *Push Notifications* [61], a feature Apple introduced in 2009 that allowed developers to send messages to all users of their application[11]. Common uses of the service are sending promotions to users, or notifying them about new important features or news regarding the application.

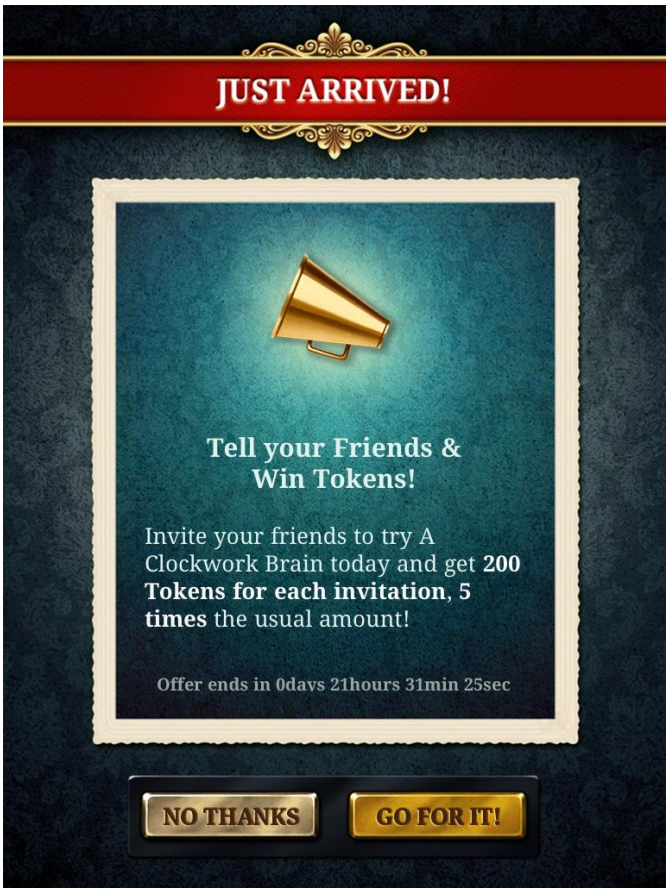
In A Clockwork Brain we used this feature, mainly for sending promotion messages, by integrating a third party service called Warp.ly[12]. One of the promotion messages notified users that they could rate the application and get 400 Sprocket Tokens, instead of 200, which was the usual amount. When users viewed the promotion message, they could either select to rate the app, or close the promotion screen. The action that opened the Rate screen was implemented in a Command object. The parameters for the action were send from Warp.ly's servers and were stored in the command object.

[11] Users are prompted to accept notifications for an application the first time it launches. If they don't accept, they don't receive notifications for this application.

[12] www.warp.ly

The object was used in two different ways. First, when the notification arrived, a validation object was created for the particular command type and decided if the notification should be displayed. For example, we would not show a notification for an upgrade to a user that already had this upgrade installed. Then, the command was kept by the notification manager and was executed as soon as the player decided to apply the promotion.

Figure 7.2 Promotional screen shown to users through Push Notifications



7.2 Evaluation

The Object Oriented approach and the use of design patterns were important characteristics of the application's architecture. Even though they demanded additional work in the beginning of the project as well as during the production, they proved to be invaluable in expanding and maintaining the software more effectively after its first release. Also, they allowed us to be way more flexible in adding new features and new mini games on the way. In retrospect, it was a valid decision to invest in good architecture, like we had been doing in all our previous projects.

7.3 Managing live parameters remotely

7.3.1 Overview & purpose

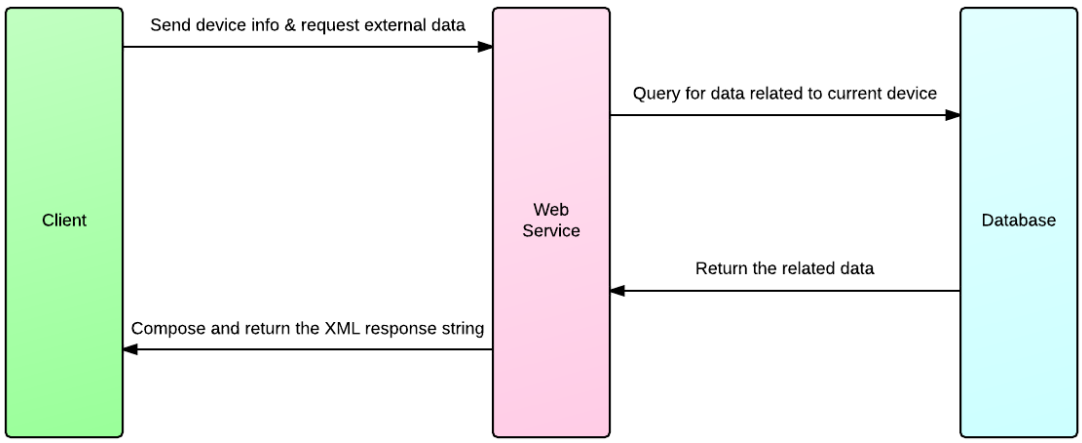
One of the parts in software development lifecycle is maintenance, which has a number of purposes. A software system may need error diagnosing and fixing, feature addition or improvements, adaptation due to changes made in the software environment (operating system, devices, integrated third-party libraries and frameworks) etc. In addition to these technical issues, there are often requirements of a different nature, such as modifying game play parameters, improving the monetization of the product or introducing new content. A way of executing this type of maintenance actions easily and with reduced cost is modifying the behavior of the application live, without having to produce a new version of the product. This provides many benefits, as it reduces development time, maximizes adoption, gives the ability of testing multiple variables and provides a lot of flexibility. On the other hand, there is some overhead in time and cost, as the system that handles external modifications needs to be developed upfront, before the application is shipped.

A similar system was used in A Clockwork Brain, for several elements of the application. The core of this system was already created for a previous game developed by Total Eclipse and was included in SolarWind, later to be refactored and expanded for the needs of A Clockwork Brain.

7.3.2 System architecture

The system consists of 3 main components; the client, the web service and the database. The client requests information from the web service, which connects to the database, acquires the information and replies back to the client, as seen in the diagram below.

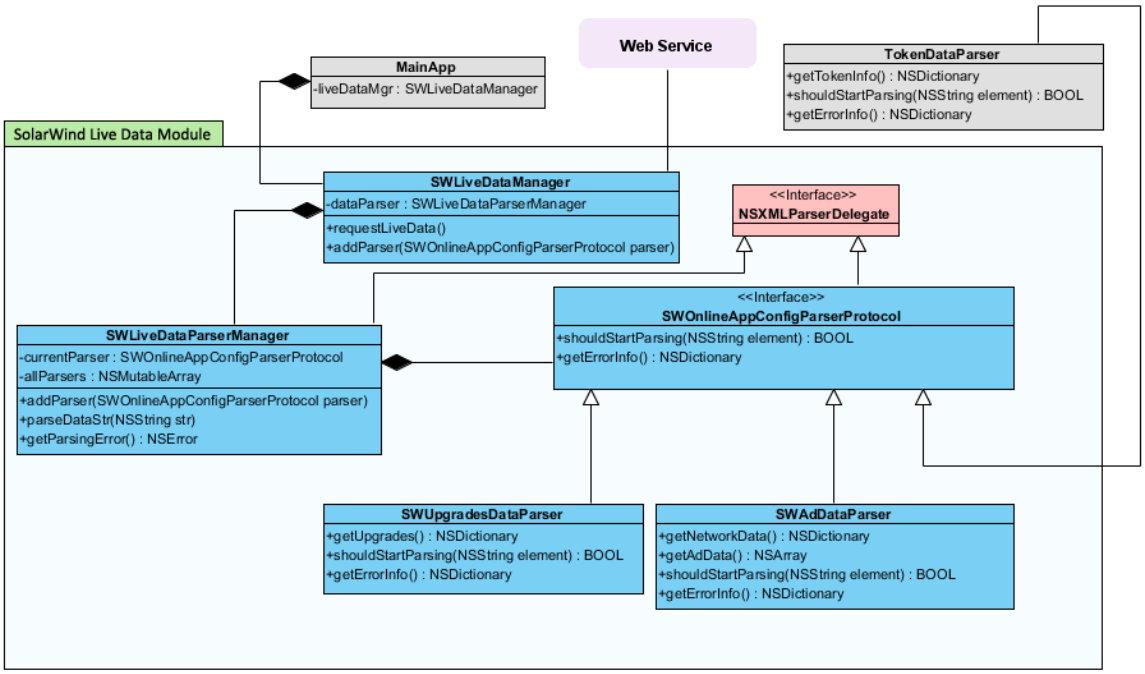
Figure 7.3 Live Parameter Management Model in Clockwork Brain



The client communicates with the web service, while sending some information regarding the current device. This is related to the functionality of the Upgrades mechanism, which needs to be aware of the device type (iPhone or iPad) and the

availability of Retina display, so that it can return the URL to the proper image resource packages that contain the files suitable for this device. The client sends an encrypted request, asking for all external information and expects a reply as a serialized XML string.

Figure 7.4 Live Parameter Management Client Model



The web service, which is a PHP script located on an online server of Total Eclipse, receives the request and performs a few security checks to ensure the authenticity and the integrity of the client request message. Additionally, the communication occurs over a secure socket, using SSL. The script connects to a database and collects the required data. Then it composes the XML string and replies back to the client. In case there is an error, the data returned contains the error information, so that the client can display a useful message to the user.

7.3.3 Client architecture

The architecture of the client side system that handles the initiation and parsing of the live parameters consists of several classes.

The core of the client's module for live data management is `SWLiveDataParser`, which is the interface that connects the application to the live world. The main application calls `requestLiveData` on its `SWLiveDataManager` instance asynchronously, and waits for a response. `SWLiveDataManager` connects to the web service and when it receives a reply, it delegates the parsing of the received XML string to an instance of `SWLiveDataParserManager`, by calling `parseDataStr:`. The parser manager, using a Strategy pattern scheme, described below, delegates the parsing of each XML node to

the corresponding parser. When every node has been processed, it executes a callback on `SWLiveDataManager`, which in turn informs the main app that the operation is complete.

7.3.3.1 Setting up the module

When `SWLiveDataManager` is initialized by `MainApp`, it creates an instance of `SWLiveDataParserManager` and keeps a reference of it. `MainApp`, which acts as the client to the Live Data Module, then adds a number of parsers that are needed to handle the functionality of the current application, which in case of A Clockwork Brain is advertisement management, upgrade management and token information management.

Part of the code for this module is provided below.

```
////////////////////////////
// MainApp.m
- (void) initLiveDataManager {
    liveDataMgr = [[LiveDataManager alloc] init];
    [liveDataMgr addParser:[[SWUpgradesDataParser new] autorelease]];
    [liveDataMgr addParser:[[SWAdDataParser new] autorelease]];
    [liveDataMgr addParser:[[TokenDataParser new] autorelease]];
}

- (void) requestLiveData {
    [liveDataMgr requestLiveData];
}

////////////////////////////
// SWLiveDataManager.m
- (void) init {
    parserManager = [SWLiveDataParserManager new];
}

// Adds a new parser
- (void) addParser(id <SWOnlineAppConfigParserProtocol> parser) {
    [parserManager addParser:parser];  // delegate to the parser manager object
}

////////////////////////////
// SWLiveDataParserManager.m
// Adds a new parser
- (void) addParser(id <SWOnlineAppConfigParserProtocol> parser) {
    [allParsers addObject:parser];
}
```

7.3.3.2 XML string sample

A sample of the server's response can be seen below.

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
<addon id="GamePack1">
  <title><![CDATA[en‡‡Game Pack 1]]></title>
  <desc><![CDATA[en‡‡Memorize the shape as best as you can, because it will change! Can
you sculpt away all the added tiles?|Study the word carefully. Can you find how many
letters it has?]]></desc>
  <apple_pr_id>#########</apple_pr_id>
  <avail>2011-06-22</avail>
  <device></device>
  <type>non-consumable</type>
  <min_app_ver>1.0.0.0</min_app_ver>
  <visible>1</visible>
```

```
  <order>1</order>
</addon>
<addon id="GamePack2">
  <title><![CDATA[en‡‡Game Pack 2]]></title>
  <desc><![CDATA[en‡‡Can you locate the smallest object? Be quick, and find the direction
it's pointing at!|Take a good look at those stacked disks! How would they look, if viewed
from above?]]></desc>
  <apple_pr_id>##########</apple_pr_id>
  <avail>2011-06-22</avail>
  <device></device>
  <type>non-consumable</type>
  <min_app_ver>1.0.0.0</min_app_ver>
  <visible>1</visible>
  <order>2</order>
</addon>
<current_app_ver val="1.2.0" />
<success val="1" />
<ads>
  <network id="tapjoy" enabled="1" tracking="1" />
  <network id="chartboost" enabled="1" tracking="1" />
  <ad placement="MainMenu_Full" network="tapjoy" type="featured_app" options="" odds="1"
/>
  <ad placement="MainMenu_Full" network="chartboost" type="featured_app" options=""
odds="1" />
</ads>
<tokens>
  <event id="CHALLENGE_SESSION_GAME" tokens="12" />
  <event id="PRACTICE_SESSION" tokens="9" />
  <event id="FLAWLESS" tokens="10" />
  <event id="FB_LIKE" tokens="80" />
  <event id="RATE_GAME" tokens="100" />
  <event id="TELL_A_FRIEND" tokens="40" />
</tokens>
</root>
```

7.3.3.3 Parsing the XML

`SWLiveDataParserManager` keeps a reference of all registered parsers. It begins parsing the provided XML string node by node, deciding which is the suitable parser instance for each xml node. To accomplish this, it queries all parsers, until one of them claims "ownership" of the current node that is being parsed, based on the node's name. For example, the Ad parser will claim that the part of the XML document belongs to it, when an `<ads>` node is met, while the Token parser will do the same when asked for a `<tokens>` node. For each individual xml node, the parser manager will execute `findParserForElement:(NSString *elementName)`, which in turn will query all registered parsers, calling their `shouldStartParsingElement:(NSString *elementName)` method until one parser is matched. When a parser is selected, the manager keeps using it, until another parser requests to be used.

```
//////////////////////////////
// SWLiveDataParserManager.m
// Parses the provided serialized XML string
- (void) parseDataStr(NSString *str) {
       NSXMLParser* parser = [[NSXMLParser alloc] initWithData:[dataStr
dataUsingEncoding:NSUTF8StringEncoding]];
       [parser setDelegate:self];
       [parser parse];
       [parser release];
}
```

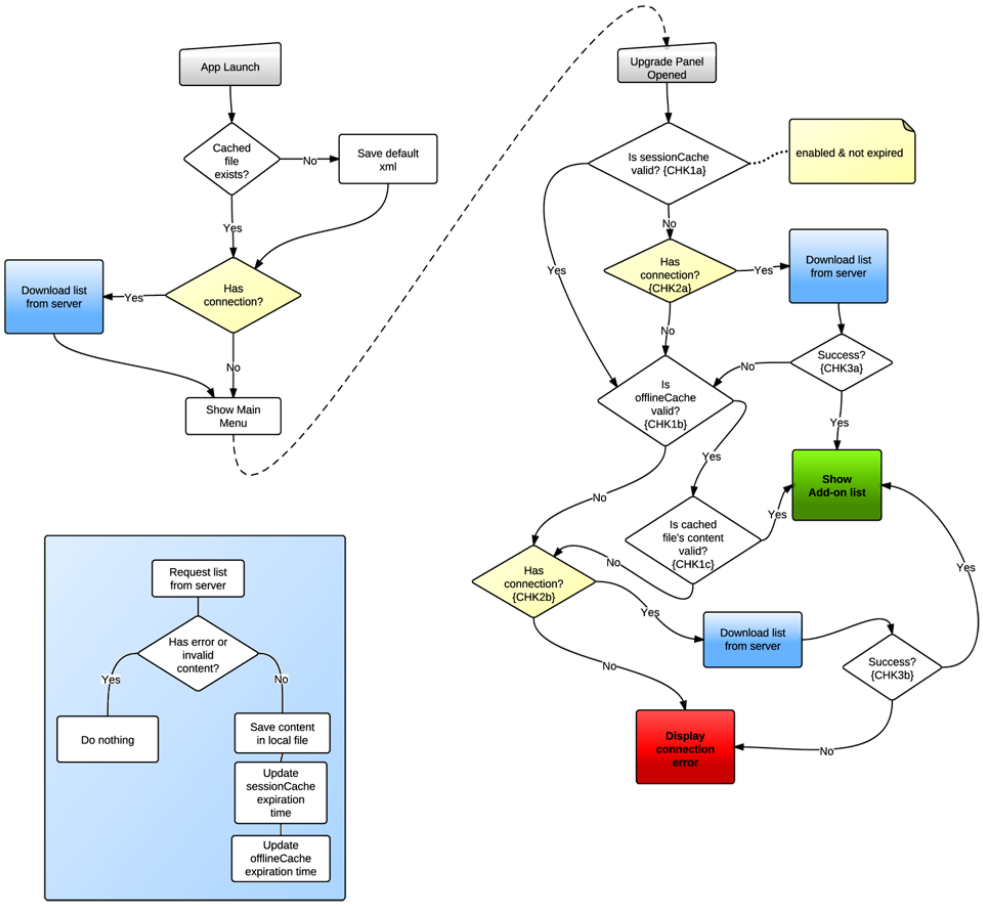
```
// Method of the NSXMLParserDelegate protocol - Gets called for every new XML node found
- (void)parser:(NSXMLParser *)parser didStartElement:(NSString *)elementName
attributes:(NSDictionary *)attributeDict {
    [self findParserForElement:elementName];    // updates currentParser
    [currentParser parser:parser didStartElement:elementName attributes:attributeDict];
}

// Finds the parser instance that should parse the current XML node
- (void) findParserForElement:(NSString*)elementName {
    for (id <SWOnlineAppConfigParserProtocol> parser in allParsers) {
        BOOL shouldUseThisParser = [parser shouldStartParsingElement:elementName];
        if (shouldUseThisParser) {
            currentParser = parser;
            break;
        }
    }
}
```

7.3.4 Caching system

Having these parameters online provides a lot of flexibility, but it does not come without of issues. The application requires an active Internet connection, which brings up the question “What happens if a user is not connected?”. Does the application work or does it prompt the user, saying that an active connection is required? The latter would seem very harsh for such an application, where the online connection is only required for obtaining up-to-date information regarding non-essential game features. However, there are several decisions to be made, such as what to do when no connection is available, so that the application falls back gracefully in such cases, without compromising the necessity of updating the external data in a timely manner.

Chart 7.1 Flowchart of Caching System for live parameters



Additionally, even if users do have a connection, it may be of low quality, or perhaps the server may be experiencing issues and not be able to respond quickly. The app needs to detect such a case and act accordingly. To resolve all these issues, a caching system is required. An advanced caching system was created for A Clockwork Brain and integrated in SolarWind. Its functionality can be seen in Chart 7.1.

This system consists of 3 main caching levels. One occurs on the server and the other two on the client. The main principle is to download the updated information from the server whenever a connection is available and use the cached content in case there is no connection. The system also provides security, by confirming the integrity of the cached content, so that players cannot cheat by modifying the cached parameters in their favor. Overall, the concept behind the caching system is to prioritize player convenience and try to display the content in almost every case. It will stop displaying content only if absolutely necessary, i.e. if it has been too long since the last time a player downloaded fresh content from the server. In all other cases, the system will use the most recent content available.

App Launch

When users enter the Upgrades screen, we want the upgrades to be available immediately, so that the screen loads fast and users can make their purchases or unlock upgrades quickly. Since the upgrade information is downloaded from an online server, there may be a delay from the time the request is made until the response arrives, if it ever does. This means that the upgrade information needs to be available as soon as the application reaches the Main Menu screen, so that it is instantly available if a user enters the Upgrades screen.

The system attempts to connect to the online server while preloading the game assets. If the information is downloaded successfully, it is cached locally and the app proceeds to the Main Menu. If there is no connection or there is a timeout within 8 seconds, the app once again continues and opens the Main Menu screen, without having downloaded the updated information. The top priority is user experience, so we do not want to keep them waiting more than a few seconds, because of a bad connection or a server-side issue. In case the user enters the Upgrades screen later, the system will attempt to connect to the server again.

Session cache

Every time the Upgrades screen opens, a number of checks is performed. First, we check if the session cache is valid. The session cache expires every time the application shuts down or whenever a few hours pass while the app is still active. Its purpose is to prevent the app from trying to connect to the server every time the user enters the Upgrades screen and thus save time and frustration. In addition to that, the session cache is not valid throughout the session, because a user may not close the application for many hours or even many days in a row. In that case, the online content may have changed significantly and it would be wise to attempt to download a fresh set of parameters.

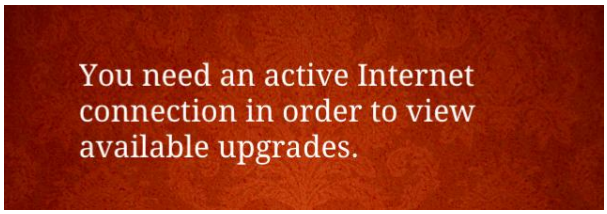
If the session cache has expired, the application will attempt to download the updated content from the server. If successful, the chain ends here and the new content is displayed. If, however, this fails, the system will fall back to the locally saved content, or offline cache.

Offline cache

Every time the content is downloaded from the server, it is cached in a local file and the exact date and time is stored in a secure field in NSUserDefaults, together with a hash string that derives from the downloaded content. Since the offline cache is the last resort when no connection is available, its duration is quite long, as it is set to expire by default in 30 days. This practically means that a player can view and unlock all free upgrades in the app for a whole month, even if she doesn't have an online connection.

When the system resorts to the offline cache, it does 2 different checks. First, it examines if there is a local cache file and if it has expired. If this check passes, it validates the integrity of the saved file, using the hash that was saved when the file was first created and saved in NSUserDefaults and comparing it with a newly created hash of the local file. If the two hashes match, the contents of the local file have not been altered and are safe to use. In that case, the content is displayed. If, on the other hand, the file has been tampered with, the system requests fresh content from the server. If successful, the content is displayed. If not, an error is displayed, stating that an online connection is necessary in order to display the upgrades.

Figure 7.5 Connection error on Upgrades screen



Server cache

When the client requests the contents from the server, it calculates and sends along with the request a hash, a checksum of the local file. The server generates the online content string and calculates the checksum, using the same algorithm as the client. If these two strings match, instead of the full XML response, only a 304 Not Modified header is returned. The client identifies the header id and uses the cached content as if it were fresh.

Handling cache expiration

Every time fresh content is downloaded from the online server we perform 3 distinct actions. We save the content in a local file, we create a checksum for this file and store it securely and we refresh the expiration dates of both the session cache and the online cache.

First launch

The first time a user launches the application, she may not have an active connection. In this case we still want to be able to display the upgrades content normally, so we have included a default XML string inside the code (so that there are no security issues) and we use this as a fallback.

7.4 Backward compatibility & migration system

Software that gets updated from time to time may face the issue of backward compatibility. A system like A Clockwork Brain, that outputs data and then uses that same data as input may have to perform special actions in order to ensure the old data does not become useless when an updated version of the system is used. The required

changes may be the result of new features, bug fixes or optimizations of the software, or even changes in the operating system on which a software runs.

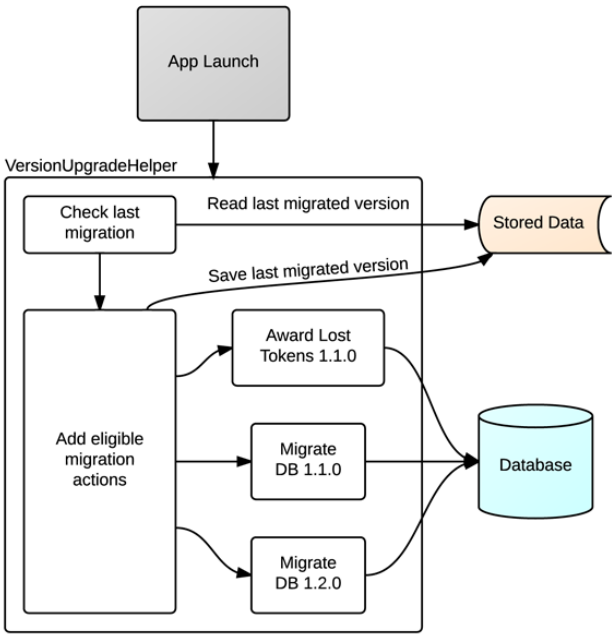
In Clockwork Brain, for the two updates we released after version 1.0.0, we had to implement a few migration mechanisms in order to take care of such issues. These issues were related to both the operating system (new iOS versions) and internal systems that loaded and saved data.

7.4.1 Migration mechanism

As soon as the application finishes preloading all necessary resources, the `VersionUpgradeHelper` class is initialized and starts checking if any migration actions are needed. Based on the last migrated version saved in Stored Data (`NSUserDefaults`), the class decides which migration actions to execute.

The order in which these actions must be carried out is important, as every action has a specific order index. For example, the action for awarding the Lost Tokens must be executed before db 1.1.0 migration, whereas the db 1.2.0 migration must be executed last. The order is critical to ensure the system remains valid at all times. Every time a migration action is executed, the value in stored data is updated, so that this action won't ever run again in the future. When the migration mechanism is finished, the flow returns back to the main application and the main menu is presented to the player.

Figure 7.6 Flow chart of the migration mechanism in Clockwork Brain



7.4.2 Lost Tokens

Version 1.0.0 of the game had a nasty bug that we were not aware of, when we released the game on the App Store. In some rare circumstances, there was a malfunction when

the application was terminated while being in the early stage of preloading its core assets and all Token information of the player was overwritten. In other words, players suddenly lost all their Sprocket Tokens, which they had been collecting by playing the game. We started getting reviews on the App Store of people giving us 1 out of 5 stars rating, saying that the game is horrible because they lost all their tokens. Some of them reported having lost as much as 5.000 or 10.000, which corresponds to many hours of playing.

Of course, this was a big disappointment to our users and gave a bad impression for the game and the company that had built it. As soon as we discovered what was happening, we started working on fixing the issue right away, as soon as we found out what was happening. After a couple of days, we were able to reproduce the bug and find a solution. This was the easy part. The hard part, that required more work and creativity, was to find a remedy for the damage that had been done, both in public and individually for the players that suffered by this bug. We discussed this among the team members and decided to give the tokens back to the players, plus a little bonus, in the next version update.

For the players that got this bug, the application had the secure token field reset to the value of zero and there was no way to find out what the value was before. Since every user could have played any number of games, with a varying token reward rate, we had to find a way to calculate the number of tokens each player had lost. Luckily, most of the parameters that took part in the algorithm that determined the number of tokens for a mini game session were saved in the database.

For any mini game session in Single Game mode, we used the following algorithm:

$$\begin{aligned} \textit{Tokens Awarded} = \quad & \textit{bonusLevel\_tokens} + \textit{insaneRoundLevels\_tokens} + \\ & \textit{flawless\_tokens} + \textit{singleGame\_tokens} \end{aligned}$$

The `insaneRoundLevels_tokens` parameter is calculated by taking into account the number of insane rounds a player achieved in a mini game. Unfortunately we did not store this information in the database, so there was nothing we could do about it. But the rest of it was stored for each mini game entry, so we could easily make a good estimate for the number of tokens a player might have lost.

The method we created, that finds and rewards lost tokens is the following:

```
int const minLostCoinsToAward = 100; // Only give back if player lost more than this
float const roundLostTokensRange = 50.0; // Round upwards to the next 50 tokens
float const kLostTokensAwardRatio = 1.15; // Give 15% extra tokens than what they lost

- (void) giveLostTokens_v1_1_0 {
    NSMutableDictionary *params = nil;

    // See how many tokens the player actually has
    int currentTokens = [[PlayerManager instance] getCurrentPlayerAvailableTokens];
```

```
    // Query the database and estimate the tokens the player should have collected
    // for her history
    NSDictionary *results = [[SW statsmanager] performCustomAction:kTokenAwardingEvents];
    int estimatedTokensCollected = [self getTotalTokensForResults:results];

    // Add any tokens earned for one-time rewards, such as liking our page on Facebook
    // or rating the application
    estimatedTokensCollected += [self getTokensEarnedForOneTimeEvents];

    // See how many tokens the player has spent (on Free Upgrades)
    int tokensSpent = [self getTotalTokensSpent];

    // Estimate how many tokens the player should have
    int supposedRemainingTokens = estimatedTokensCollected - tokensSpent;

    // Estimate how many tokens the player has lost
    int estimatedLostTokens = supposedRemainingTokens - currentTokens;

    float roundedLostTokens = estimatedLostTokens;
    if (estimatedLostTokens >= minLostCoinsToAward) {
        roundedLostTokens = estimatedLostTokens * kLostTokensAwardRatio;
        float normalized = roundedLostTokens / roundLostTokensRange;
        roundedLostTokens = ceilf(normalized) * roundLostTokensRange;
        float normalized2 = estimatedLostTokens / roundLostTokensRange;
        estimatedLostTokens = ceilf(normalized2) * roundLostTokensRange;
    }
    else {
        roundedLostTokens = 0;
    }

    if (roundedLostTokens > 0) {
        [[PlayerManager instance] addEarnedTokensToCurrentPlayer:roundedLostTokens];
        [AppCommon broadcastAnalytics:acbAnalytics_LostTokensAwarded withParam:
roundedLostTokens key:acbAnalytics_param_Tokens];
    }

    // Save the information that the lost tokens issue in version 1.1.0 was resolved
    // so that we never attempt to give this player lost tokens
    [self lostTokens110resolved];
}
```

We tried this method with data collected on our devices and we adjusted the constants until the estimated result was very close to the actual lost tokens. When the update was released, players were excited to see that they got their tokens back, plus some extra ones. Some players even wrote about this in the App Store review section, giving us a 5/5 stars rating. The analytics data we collected showed that around 3% of all players had lost their tokens, most of them 200-300 and in rare occasions 5.000 tokens or more.

Figure 7.7 Message shown to players that lost Sprocket Tokens

7.4.3 Database

The two updates, v. 1.1.0 and v. 1.2.0 required a few changes in the local database scheme, used to store the player's history. First of all, learning from our mistake regarding the Lost Tokens issue, we decided to include two extra fields in the MiniGame table; *tokens* and *insaneRounds*. The first field records the number of tokens awarded for each mini game, a value that would have saved us a lot of trouble had we been storing it since version 1.0.0. The second field contains the number of insane rounds achieved in each mini game session, if any. This would be an interesting metric for having in the future, perhaps for displaying statistical information to the user. After adding these two columns on the table, we populated them with dummy values or -1, so that we could be aware that they did not contain real values, and be able to calculate them in the future, if needed.

Another modification was required because of a bad decision the programming team made when designing the database in the first place and setting internal conventions regarding the ids of a certain value; the id of each mini game type. The MiniGameType table associated each mini game name with a numeric id. For example, Scrolling Silhouettes was 1, Missing Tiles was 2 and so on, up to number 11 for Size Matters. Unfortunately, the id we selected for the Challenge session, was 12, the next number right after the id of the last mini game. What we did not think about at the time was that in later versions, we would be adding more mini games, which would have to be associated with IDs larger than 12. There would be no actual problem with this approach, but it would look untidy, so we decided to fix it. We created a query that shifted by one all ids that were lower than 12 and set Challenge type to be associated to number 1. Thus, in future versions, when more mini games would be added, they would take the numbers from 13 and forth, while Challenge would always be at number 1.

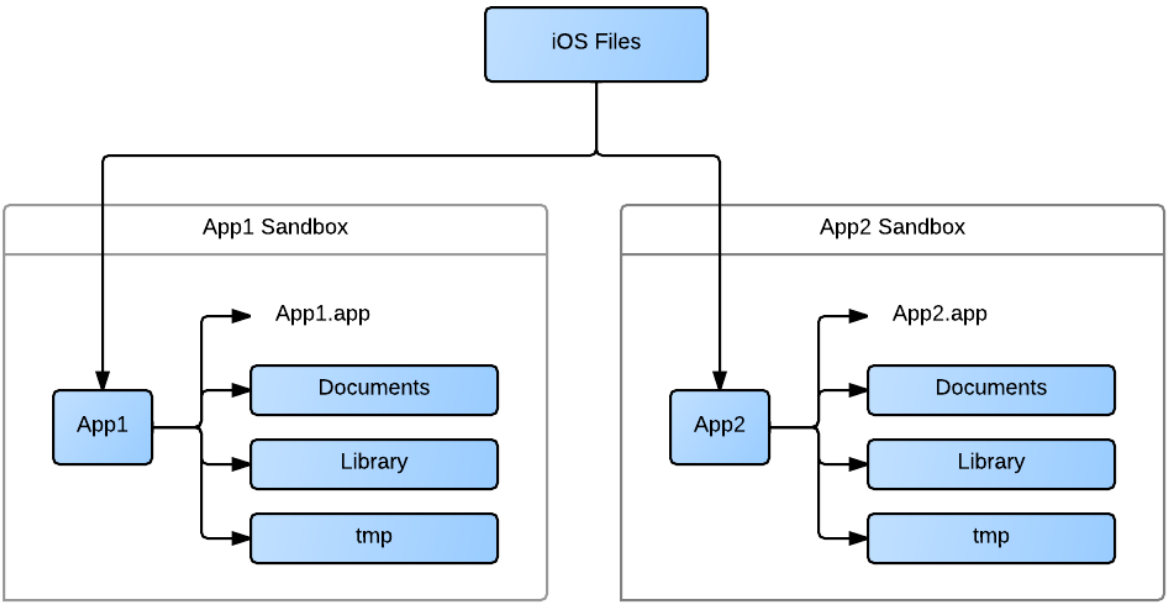
The last migration action we designed regarding the database was in v.1.2.0, where we added two more games, Speed Match and Logic Cards. For this modification, we had to run a query and add the associations for these new mini games, as seen in the code below. The id field of the table is auto-increase, so we just insert the names of the mini games.

```
NSString *q1 = [NSString stringWithFormat:@"Insert Into game (name) Values ('%@');",
kGameNameSpeedMatch];
NSString *q2 = [NSString stringWithFormat:@"Insert Into game (name) Values ('%@');",
kGameNameLogicDots];
[self execute:q1];
[self execute:q2];
```

7.4.4 Downloadable Upgrades

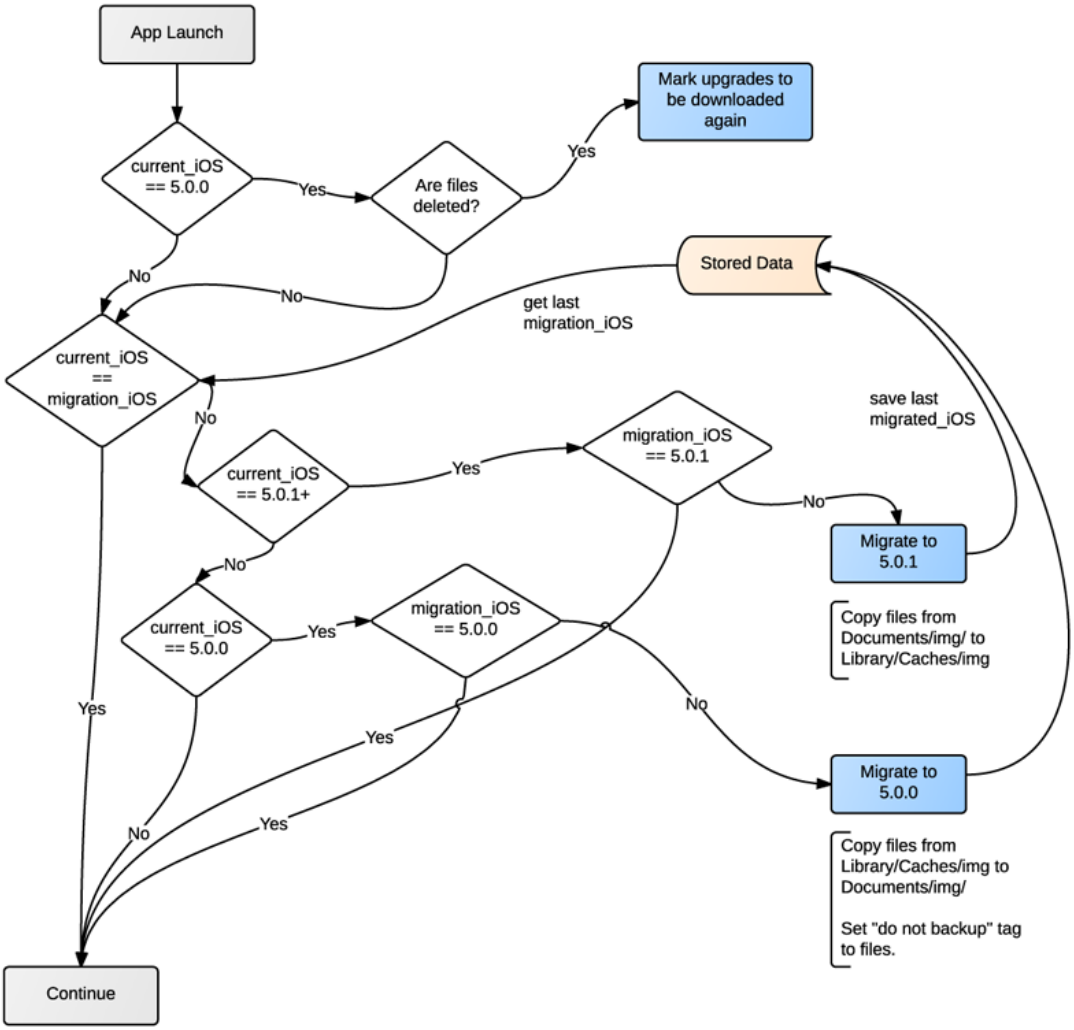
Each application on iOS has a sandbox; a folder where it has permissions to write, delete and modify data. Outside the sandbox, the application has no permissions at all, with a few exceptions such as the user's Music folder.

Figure 7.8 Each iOS app operates within its own sandbox



This is one of Apple's mechanisms for securing iOS applications and decreasing the risk of allowing malicious actions to be performed by an application. Inside this folder, there are some predefined subfolders, which have varying purposes and behavior, when it comes to whether they are backed up by the operating system. The contents of the `Documents/` folder are never deleted by the iOS and they are backed up on iCloud, Apple's cloud storage service. On the contrary, files in the `Library/Caches/` folder are not backed up and may be purged by the iOS without any warning.

Figure 7.9 Content migration system for Clockwork Brain



When iOS 5.0.0 was released, it modified the data storage guidelines, and then iOS 5.0.1 modified them again. From 5.0.0 and on, the `Documents/` folder may only contain user-generated data, which is data that the application cannot recreate. The reason for this is space saving, since the information in the `Documents/` folder is backed up on iCloud. An exception to the rule (valid for all folders, including `Documents/`) exists for files that can be recreated by the application, but users expect to be present in offline mode, even in low storage situations. When iOS 5.0.1 was released, it provided a special "do not backup" tag, which can be applied to any file or directory that does not need to be backed up on iCloud. This tag does not exist on iOS 5.0.0.

Unfortunately, iOS 5.0.0 was released a few weeks after A Clockwork Brain launched on the App Store. The application makes use of these folders, mainly for storing the downloaded contents of the Item Pack upgrades. These upgrades contain hundreds of files and weigh many megabytes in size and these updates on the operating system

affected the functionality of the game. In order to provide a solution, we studied the storage guidelines in all iOS versions that matter for Clockwork Brain carefully.

Prior to iOS 5.0.0
Upgrade files were saved in `Documents/img/`.

iOS 5.0.0
Upgrade files should be saved in `Library/Caches/img/`.
The operating system can purge these files at any time, without warning.

iOS 5.0.1
Upgrade files can be saved in `Documents/img/`.
The `img/` folder must be assigned the "do not backup" tag.

The migration mechanism for the downloadable content was based on the above rules. The result was a rather complex mechanism, that required four full days for its design, implementation and testing.

One thing to note is that this system does not support iOS downgrading. If, for instance, a user installs iOS 5.0.1, runs Clockwork Brain and then installs iOS 5.0.0 or lower, the migration mechanism will not be triggered and the upgrades will not work. This was a conscious decision we made in order to save time, since this scenario, although possible, is highly unlikely to occur in real life.

`ContentCompatibility`, the class responsible for implementing this mechanism, checks the last migration iOS version saved in `NSUserDefaults` and decides if there is any migration action that needs to be executed. Additionally, if the app is running on iOS 5.0.0, it finds if any upgrade files were deleted by the operating system. In case there are such files, it queries the `UpgradeManager`, in order to mark the deleted upgrades for re-download.

7.5 In-App Purchases

7.5.1 Overview

In late 2009 Apple introduced In-App Purchases on the App Store. Until then, there were free and paid applications and developers often offered both versions for an application. The free version, sometimes called Lite, had limited features or content and would act as a trial version. Users who liked what they saw in a free version could purchase the full-featured paid application.

In-App Purchases (IAP) provide the flexibility of supporting multiple business models on iOS applications. By using In-App Purchases, developers can offer their customers additional content or functionality, in the same application. Game developers started using them in both free and paid apps, providing extra content or features. For example,

in games like Smurfs and Temple Run players can purchase virtual currency, which can later be used to acquire content and features.

There are 5 types of In-App Purchases.

Consumables
Correspond to items in the application, that are consumed when used. Users need to purchase additional units if they have used up all instances of an item. For example, a consumable item is a magic potion in a game, various supplies like ammunition, health kits etc.

Non-Consumables
These IAPs need to be purchased only once, and then they are available forever to the user. They are usually associated with features or content that does not get consumed, like additional levels in a game, or a new game mode, or a unique ability of the player.

Auto-Renewable Subscriptions & Non-Renewing Subscriptions
Allow users to acquire access to dynamic content for a fixed duration time.

Free Subscriptions
Work like Auto-Renewable Subscriptions, except they do not cost anything to the end user.

The introduction of In-App Purchases made a major difference on the iOS market. Applications that are given as free downloads and include In-App Purchases are often called 'freemium', from the words free & premium. According to Flurry [26], in June 2011 freemium games with In-App Purchases represented 65% of the total revenue for top grossing games on the App Store, a number that was at 39% in January 2011. Also, analytics company AppAnnie [27] reports that worldwide freemium revenues apps have almost tripled from September 2011 to September 2012 on iOS, while growing 350% on Google Play from the beginning of 2012. Revenues from paid apps have remained relatively stable during these periods.

In freemium apps, only a small percentage of the users that download them purchase anything; the rest play for free. Flurry says this number ranges from 0.5 - 6% for games, and depends on the quality and core mechanics of the game [26]. The amount of money generated by the few users that convert though, can be significant and can even reach many millions of dollars each month, based on reports by game developers over 2012.

7.5.2 In-App Purchases in A Clockwork Brain

A Clockwork Brain was designed as a freemium application. It includes 4 mini games for free and the rest are available to acquire through In-App Purchases. There are 4 game packs (the last was introduced in version 1.2.0), with 2 mini games each. When a player

buys any game pack, she owns it forever and can play its mini games for as long as she likes (non-consumable In-App Purchases).

The game also has a form of virtual currency, Sprocket Tokens. The only way to earn Tokens, though, is by playing the game and performing some specific actions, like rating the application or liking the company's page on Facebook. In the future, we could add consumable In-App Purchases that give players Sprocket Tokens, which they can later spend in the game. In fact, this is one of the major updates planned for future versions of A Clockwork Brain.

7.5.3 Adding In-App Purchases to an iOS application

Before a developer can add In-App Purchases to an application, there are several steps that need to be taken care of. One has to do the following:

- Agree to the latest Developer Program License Agreement with Apple.
- Complete the contract, tax, and banking Information on the Apple developer account.
- Configure an App ID for In-App Purchase in the iOS Provisioning Portal
- Create, download, and install a new Development Provisioning Profile that uses the App ID enabled for In-App Purchase .
- Ad In-App Purchase products in the application, through iTunes Connect.
- Create a test user account on iTunes Connect, in order to be able to use a testing environment called "sandbox", which simulates the purchases process, without occurring any financial charges.
- Code-sign the application using the correct provisioning profile.
- Integrate Apple's Store Kit framework in the application.

These steps will not be further explained here, as they are standard procedure for any iOS developer. Details on them can be found on Apple's iOS Developer Library website [62], [63].

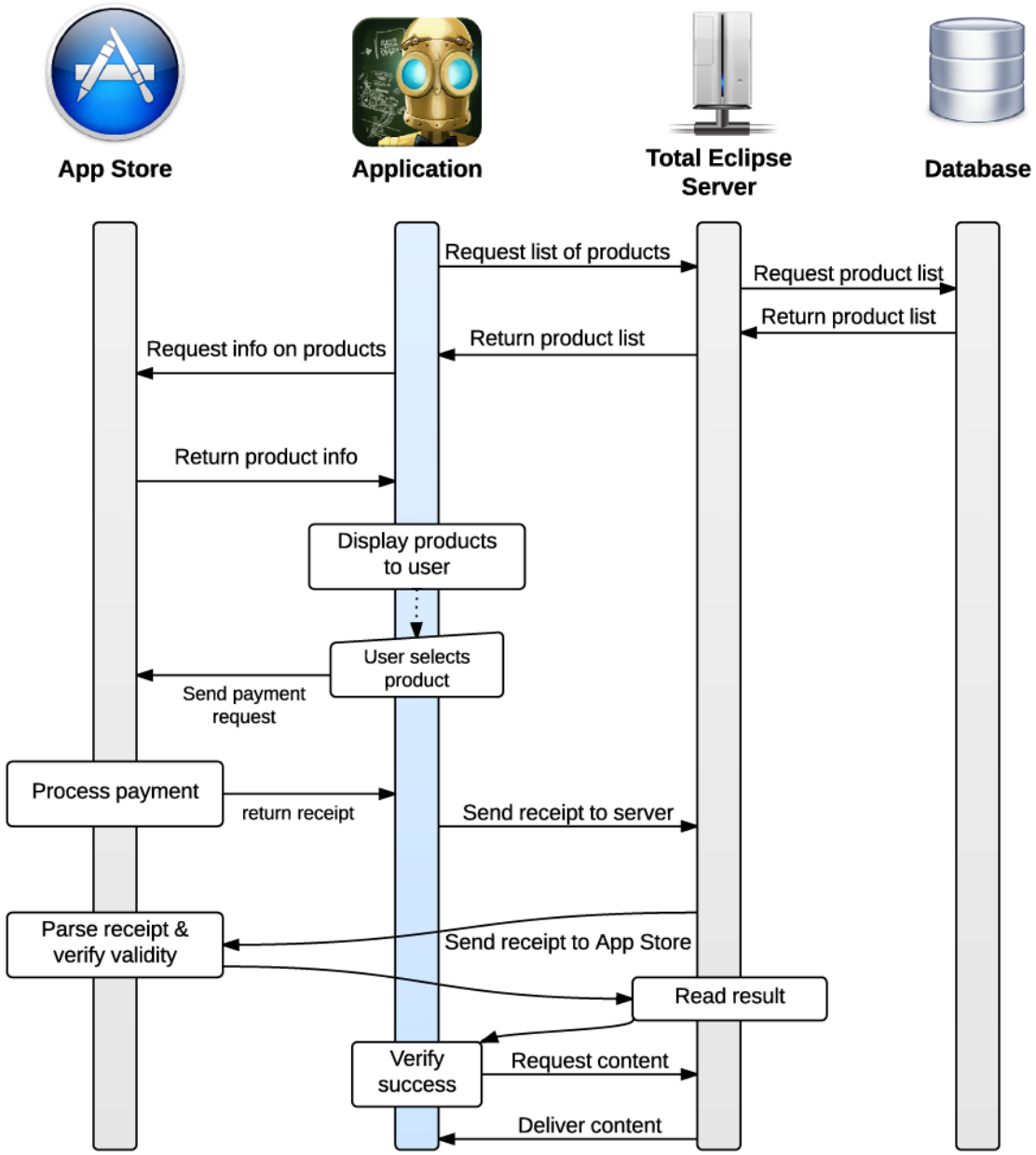
7.5.4 Store Kit integration

In-App Purchases are implemented in an iOS application by using the Store Kit framework, developed by Apple [64]. Store Kit connects to the App Store on the application's behalf to securely process payments from the user. Store Kit asks for the user's credentials, in order to authorize the payment, and then notifies the application regarding the success or the failure of the transaction, so that it can provide the purchased items to the user.

It is solely the application's responsibility to provide any content or features to the user. If there is content that needs to be downloaded, the application must use the developer's web servers, as Apple does not host files for In-App Purchases. An exception is content for non-consumable products, which as of mid 2012 Apple can provide free hosting for.

In-App Purchases do not provide the ability of patching the application binary. If the application bundle needs to be updated for a purchased feature to work, there must be an updated version of the app on the App Store and the user needs to download that version.

Figure 7.10 In-App Purchase Product model for Clockwork Brain



Server Product Model

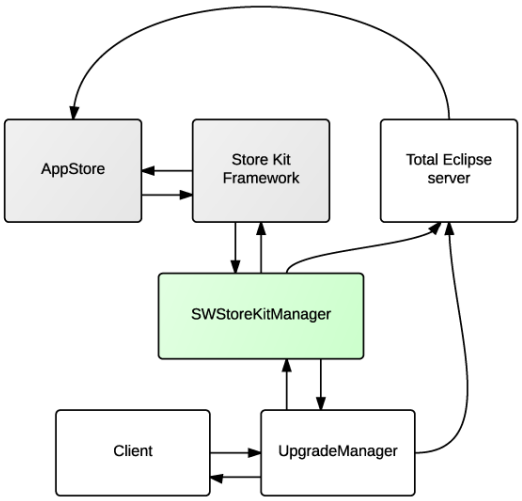
As seen on Figure 7.10 , the information of In-App Purchase products exists on a web server, just as explained on Chapter 2.11. This is the method Apple recommends, because it allows developers to add and remove products without updating the application and re-submitting to the App Store.

SWStoreKitManager

In A Clockwork Brain, and in the SolarWind framework in general, there are several instances where we create Facade classes, to wrap the functionality of third-party frameworks or libraries and usually add some extra features as well. This way, we manage to make the frameworks easier to use, more consistent with the SolarWind framework style, and reduce client dependencies on other frameworks. Additionally, we make the system more flexible to future changes, as well as portable, so that it can be used with other clients.

The same applies for Store Kit. The `SWStoreKitManager` class acts as an intermediate and becomes the only interface the client needs to communicate with, wrapping the Store Kit integration into a black box.

Figure 7.11 SWStoreKitManager model



One of the features we added to `SWStoreKitManager` was related to error handling. When the application launches, before reaching the main menu screen, we communicate with the Total Eclipse server, ask for the list of In-App Purchase products, and then tell `SWStoreKitManager` to retrieve the product information from the App Store, so that we can display the available products to the user, as soon as she enters the Upgrades section. There are cases when communication between the Store Kit framework and the App Store is bad or there are delays due to a low quality Internet connection or an internal server issue at the App Store. In such cases, the Store Kit framework does not respond with an appropriate event, or at least not consistently. In some cases we got a "fail" response after 45 seconds, while other times we did not get a response at all for as long as a few minutes.

All that time, the application would still be in the preloading phase. This, of course, is not an acceptable behavior for users, as they have to wait for the game to load for no obvious reason. Our solution was to add a timeout timer on `SWStoreKitManager`, which can be set externally by the client, and defaults to 8 seconds. If that time passed,

`SWStoreKitManager` broadcasted a time-out event, which would get propagated all the way to the client (main application), notifying it that the App Store server could not be reached because of a time-out, so that the client could continue the application flow.

7.5.5 Issue handling

Store Kit integration is quite straightforward, until there is an issue with invalid product IDs. When a request is made to the App Store, asking for information regarding In-App Purchase products, the App Store server responds with detailed information for all valid products. If any product is invalid for any reason, the response contains an `invalidProductIdentifiers` array, with the string IDs of all products that did not pass validation.

As there are many reasons a product is not considered valid, one would expect that the response would contain information, stating the reason each product failed. Sadly, that is not the case. The only information a developer gets is the ID of the product that failed to validate. Sooner or later, probably every developer implementing In-App Purchases will bump into this issue once, at least during the first time integrating them. The same happened to us, in multiple occasions throughout the production. Since Apple does not provide any feedback at all, we had to do research online, and see how other developers had solved these issues. After reading dozens of posts on fora and developer blogs, we stumbled upon a blog post [65] that contained a "troubleshooting" list, created with collective work, from various iOS developers.

The troubleshooting list features a series of questions:

- Have you enabled In-App Purchases for your App ID?
- Have you checked Cleared for Sale for your product?
- Have you submitted (and optionally rejected) your application binary?
- Does your project's .plist Bundle ID match your App ID?
- Have you generated and installed a new provisioning profile for the new App ID?
- Have you configured your project to code sign using this new provisioning profile?
- Are you building for iPhone OS 3.0 or above?
- Are you using the full product ID when when making an SKProductRequest?
- Have you waited several hours since adding your product to iTunes Connect?
- Are your bank details active on iTunes Connect?
- Have you tried deleting the app from your device and reinstalling?
- Is your device jailbroken? If so, you need to revert the jailbreak for IAP to work.
- Have you restarted your device?
- Have you agreed to the latest Developer Program License Agreement with Apple?

- Have you completed the contract, tax, and banking Information on the Apple developer account?
- If the answer to any of these questions is "No", this is probably the cause.

This list proved to be incredibly helpful, as we faced about 4-5 of these issues during development, some of them in several occurrences. This is by no means an exhaustive list, as there may be other causes for getting an 'Invalid product ID' error. It is simply the best reference we were able to find and, to our experience, allowed us to resolve any issue that came up during In-App Purchase integration.

7.5.6 Fake StoreKit

In order to test In-App Purchases on a device, one needs to log in with the credentials of a Test User, when the Store Kit framework prompts the log in panel inside the application. Test users are created on iTunes Connect, the Apple developer's portal where all iOS and Mac OS applications are set up and are valid for all applications of a certain developer. During testing, there are occasions where the players that are testing the application do not have access to these credentials. The most common case is when they are people outside the development team, like in a Usability session or a Beta testing.

For these cases, it is good to have an alternate system, which simulates the functionality of Store Kit, but does not rely on App Store communication and Apple user accounts. This way, developers can make sure that users experience the same functionality as the real product, without the restrictions that come with using systems that require real user authentication. In the case of a Beta test for example, there was no other solution, as the only alternative would be to create a Test user and give its credentials to all beta testers. This user would have access to all iOS applications of Total Eclipse. Also, once a player made an In-App Purchase, this transaction would be recorded in the App Store and any other user trying to purchase the same game pack would automatically get a different message for Apple's Store Kit framework, saying that the user already has purchased this upgrade, but not downloaded it yet. This would definitely confuse players.

In order to solve these issues, we created the `SWStoreKitManagerFake` class. It extends `SWStoreKitManager` and modifies only a few of its methods, in order to simulate the normal purchase flow of the Store Kit framework. The class implementation is listed below:

```
//////////////////////
// Provides a simulation for IAP buying procedure, mimicking the way Store Kit works.
@implementation SWStoreKitManagerFake

- (id) init {
    return [super initWithVerificationWebserviceURL:nil];
}

// Normally restarts any purchases if they were interrupted last time the app was open
```

```
- (void) startListeningForTransactions { // Do nothing }

// Normally sends a request to Store Kit framework, to purchase this product
- (void) purchaseProduct:(NSString*)productUniqueID {
    currentAddon = [[SW addonmanager] getAddonByAppleProductID:productUniqueID];
    NSString *msg = [NSString stringWithFormat:
                        @"Do you want to buy one %@ for %@?\n\n(Test transaction - will
not use a credit card)",
                        [currentAddon.title translate],
                        [currentAddon getLocalizedPrice]];
    NSString *title = @"Confirm your In-App Purchase";

    UIAlertView *alertSaved = [[UIAlertView alloc] initWithTitle:title
                                                         message:msg
                                                        delegate:self
                                               cancelButtonTitle:@"Cancel"
                                               otherButtonTitles:@"Buy", nil];
    alertSaved.tag = ALERT_CONFIRM_PURCHASE;
    [alertSaved show];
    [alertSaved release];
}

// User confirmed or rejected the purchase
- (void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:(NSInteger)buttonIndex {
    switch (alertView.tag) {
        case ALERT_CONFIRM_PURCHASE:
            if (buttonIndex == 1) {
                [self provideContentForAddon:currentAddon];
            } else {
                [[NSNotificationCenter defaultCenter]
                   postNotificationName:STOREKIT_MANAGER_TRANSACTION_CANCELLED
                   object:self];
            }
            break;
    }
}

// Notifies the client that the addon was purchased successfully
- (void)provideContentForAddon:(AddonData *)addon {
    if (addon != nil) {
        NSDictionary *data = [NSDictionary dictionaryWithObject:addon
                                                         forKey:kStoreKitManagerProductID];
        [[NSNotificationCenter defaultCenter]
                      postNotificationName:STOREKIT_MANAGER_PRODUCT_PURCHASED
                                    object:self userInfo:data];
    }
}

// Normally removes the transaction from Store Kit's queue
- (void) finishTransactionForProductID:(NSString*)uniqueProductID {
    // Do nothing
}
@end
```

When we needed this functionality, all we had to do is use the `SWStoreKitManagerFake` class instead of `SWStoreKitManager`.

```
//////////////////////////
// App.m
- (void) initializeUpgradeManager {
    UpgradeManager *upgradeManager = [SW upgradeManager];
    [upgradeManager setStoreKitManagerFake];// Use setStoreKitManagerReal in final product
}
```

```
//////////////////////////
// UpgradeManager.m
- (void) setStoreKitManagerFake {
    storeKitManager = [[SWStoreKitManagerFake alloc] init];
    [self setupStoreKitObservers];
}
```

7.6 Multi-build handling

A common practice nowadays is to develop cross-platform mobile software, using one framework and exporting for many devices or platforms. For example, there are frameworks that can create applications for both iOS and Android, or even desktop and mobile. There are many implications and roadblocks someone has to overcome, such as supporting different resolutions and overall device specifications, different UI requirements, or even varying monetization models, as the ecosystem in software distribution are hardly ever the same across different platforms.

Additionally, there are times when the same application in the same platform needs to be exported in multiple builds, in order to meet specific demands of the production cycle or the final product. Our game was developed for iOS, so it is not cross-platform, but we did have the need to create more than one build during the development. Some of the builds were only used internally, whereas others were integrated in the final product.

7.6.1 Multiple devices and Retina display

Our application targeted both mobiles and tablets . Specifically, we would ship the product to the App Store, for iPhone & iPod Touch, as well as the iPad. When we were about 4-5 months in development, there were various available iOS devices that had a significant market share. Based on these numbers and the memory requirements of the app, that were starting to take shape, we decided that we would support iOS 4.0 and above. The supported devices with their respective resolutions can be seen in Table 7.1.

Table 7.1 Target iOS devices for Clockwork Brain

Device	Screen Resolution	Retina Display
iPhone 3GS	320 x 768 px	No
iPhone 4	640 x 960 px	Yes
iPhone 4S	640 x 960 px	Yes
iPod Touch 4G	640 x 960 px	Yes
iPad	1024 x 768 px	No
iPad 2	1024 x 768 px	No

Retina Display is a technology introduced by Apple, which allows for very small pixel sizes and thus big screen resolutions. Retina screens essentially doubled the resolution of their predecessors.

When supporting the devices mentioned above, there are two different issues that need to be taken care of, the different aspect ratio and the different actual size in pixels. This means that all images should be exported in multiple sizes and that there should be a resource management part of the application that handles the loading of resources automatically, depending on the specifications of the current device. The positioning of the images on the screen should be also be handled in a graceful way, independently of the resolution and the aspect ratio.

Loading resources

The aspect ratio on the iPhone & iPod Touch (3:2) is different than the one on the iPad (4:3). Normally, this would require different UI design for the two devices, which would create a significant overhead in graphic creation, UI design and asset management. In order to minimize the workload, we decided to share the design of the retina resolution of the iPhone screen (480 x 960) with that of the non-retina resolution of the iPad (768 x 1024 on portrait mode), essentially setting up the interface for the iPad's screen and using a cropped rectangle of that for the iPhone retina. Using this method, most images would be the same for the iPhone retina and iPad, with only a few exceptions.

Regarding the handling of retina and non-retina devices, Apple has created a system for making the management of resources easy, by creating a naming convention for retina images. Retina images are recognized by the suffix *@2x*. For example, consider two files, main_menu.jpg and main_menu@2x.jpg. Inside the code of the application main_menu.jpg is loaded. If, however, the application runs on a retina device, the large-sized file main_menu@2x.jpg will be loaded instead. Cocos2D has a similar system, using the naming convention *–hd*, e.g. main_menu-hd.jpg.

The system we used was a hybrid, taking in advantage the naming convention of Cocos2D and combining it with the iPhone retina/iPad UI convention we created, ending up with four distinct folders, where all image resources were placed.

Figure 7.12 Organizing image resources in device-specific folders

The folder named *Resources-hd* held images that were common to iPhone retina and iPad screens. *Resources* had all images for the non-retina iPhone and iPod Touch and the other two folders, *Resources-iPad* and *Resources-iPhone*, held the few images that were different between iPhone retina and iPad.

The resource management class we built examined the type of the current device and automatically loaded the images from the proper resource folder. This system has multiple benefits. It saved us a lot of work in asset management, as there were less files to arrange and maintain. Also, it created a much smaller file size, because iPhone retina

and iPad versions shared most of their resources, something that is crucial when dealing with downloadable applications. Note that because the application is universal[13], there is only one executable for both iPhone and iPad, which includes the files for both screen resolutions. The system also provided endless flexibility for cases where the UI needed a different design on the iPad. This helped us create a better product, taking advantage of the diversity of the various devices, and helping the UI designer and the artist deliver a better experience to the end user.

Figure 7.13 UI differences between iPad (left) & iPhone (right) for Speed Match

7.6.2 Free and Premium builds

Sometime after the first version of the application was released on the App Store, the management team decided to launch a separate application using a different monetization scheme. Up until that time, the game was available as a free download with 4 mini games and players could purchase additional mini games. Now a Premium version would be released at a fixed price, having all mini games unlocked from the beginning. This called for several changes in the UI and code. For example, the Upgrades screen of the Premium version should not feature a section where players could purchase additional game packs. Also, the application's icons should be different and the web service the app connected to should be on a different URL. All the rest of the features and the code would be exactly the same. The changes are so few that it would be very inconvenient to have to create a new project and manage two different projects at once, one for each application. The best way would be to change a certain

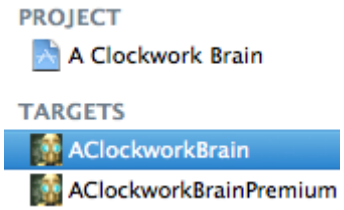
[13] Applications that run on both iPhone and iPad devices are often called "universal"

variable in the same project and alter the behavior for the two applications. XCode provides tools for this kind of requirement.

XCode Targets and Preprocessor macros

XCode has the concept of Targets. A target allows the developer to change any of the build settings for a product, include different files, frameworks or libraries and produces a single product. For Clockwork Brain, we created a new target, which would produce the Premium version.

Figure 7.14 XCode Targets of A Clockwork Brain



The C and C++ languages have the concept of Preprocessor macros and Objective-C allows the use of both C and C++ code. XCode has integrated support of preprocessor macros, which gives the ability of conditional compilation. For each target, any number of macros desired can be set. For the Premium target we created a macro called *PREMIUM_VERSION*. Then, using the `#ifdef` directive, we could differentiate the behavior between the two targets. The code below illustrates this technique.

```
// Beginning of the program
#ifdef PREMIUM_VERSION
[App setPremiumVersion:YES];
#endif

…

// Premium Upgrades only available in Free version
if (![App isPremiumVersion]) {
    [self showPremiumUpgrades];
}

// All mini games should be available in Premium version
if ([App isPremiumVersion]) {
    [self makeAllMiniGamesAvailable];
}
```

7.6.3 Developer and Release builds

During development it is essential that developers, test players and QA staff are able to use certain shortcuts in order to advance faster in the game or toggle various features. For example, it helps game designers to enable cheats in the game, so that they can easily test the difficulty of advanced levels, without having to worry about making mistakes while playing and never getting to reach these levels. The final build of the product,

though, should not expose this feature to the players. In order to differentiate this behavior, we added a flag, a Boolean variable that enabled or disabled the *Developer Panel*, as we called it.

This time we did not use the same technique as for Free and Premium versions, because we did not need a new product. Maintaining multiple products means handling dozens of settings and it can cause issues. The developer release was approached in a simpler way. In the beginning of the application's code, we set the value of the aforementioned flag depending on the build we wanted to create each time.

```
////////////////////////////////////
// IMPORTANT! Set to NO for final builds!
// Toggle developer mode
[App setDeveloperMode:YES];

// Dev panel button pressed – check to display the panel
- (void) devButtonTapped {
   if ([App developerMode])
      [self showDevPanel];
}
```

The Developer Panel provided important shortcuts that assisted the team in many ways. The Game Designer could enable cheats, force the game to enter Insane Round and toggle the speed bonuses, in order to see if the constants fed in the speed bonus calculation algorithm were producing good results. The developers could toggle highscore and achievement submission, add or remove Tokens easily to be able to check the Upgrades system and debug various cases. Testers could enable and disable all Upgrades, force the Insane Round mode to test it thoroughly without having to play a full 60" session, and so on.

Figure 7.15 Developer's panel (left) and a cheat-enabled session of Directions (right)

It is never too early to create these shortcuts and it is often overlooked by the developers. Our experience has shown that it is best to create a Developer Panel early on, providing only a handful of features and then build them up as the application grows. At the end of the production, it will have saved the team hundreds of hours of extra work and a lot of frustration.

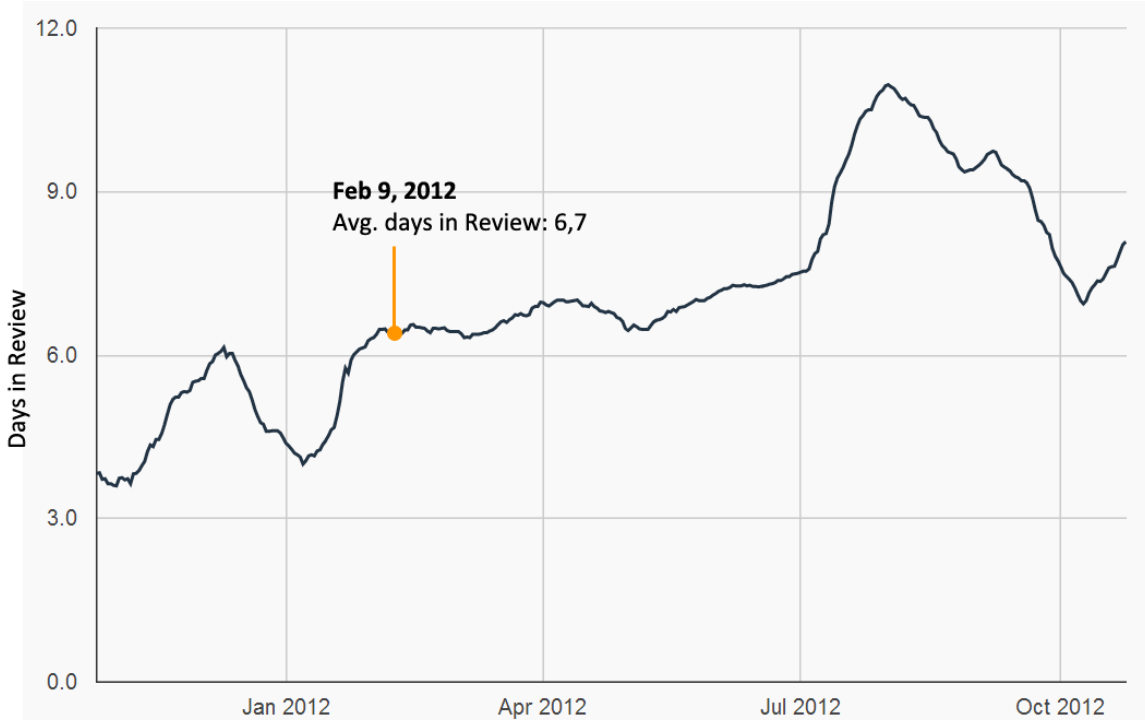
7.7 App Store submission

As soon as the GM version was ready, after having completed all testing, we prepared the final build to submit to the App Store. It was February 9, 2012, a couple of hours after midnight. When a build is submitted, it gets uploaded to Apple's servers and when the upload is complete the developer receives either a confirmation that everything is OK with the bundle, or an error report stating anything that needs to be fixed. When the bundle is successfully uploaded, Apple notifies the developer that the state of the application has changed.

In our case, the state changed to "*Waiting for Export Compliance*". This is a necessary step for any application that uses encryption. A Clockwork Brain uses SSL for communicating with our server, for downloading external information, as well as other standard cryptography methods for securing local data. In such cases, the developer needs to register with the Bureau of Industry and Security, U.S. Department of Commerce, to get an Encryption Registration Number by creating an online account and completing a form with necessary details about the company and the project [66], [67].

Four days later, the app's state moved to "*Waiting for Review*", which means that the export compliance was validated. The next day, the app went "*In Review*", which meant that someone from Apple's team began the review process. Finally, on February 16th, the application's state changed to "Ready for Sale" and became available on the App Store. This process is described in such detail because it is one of the most anxious feelings waiting for an application to be accepted or rejected by Apple, as many developers admit. There have been reports of developers who waited for many weeks, stuck in the "Waiting for Review" or "In Review" state, without having any power to speed the process up. In fact, one of the developers created an website that collects information about the review time of other developers' apps, and reports a daily average.

Chart 7.2 App Store Average Review Time - Annual Trend Graph



***Source**: shinydevelopment.com [68]*

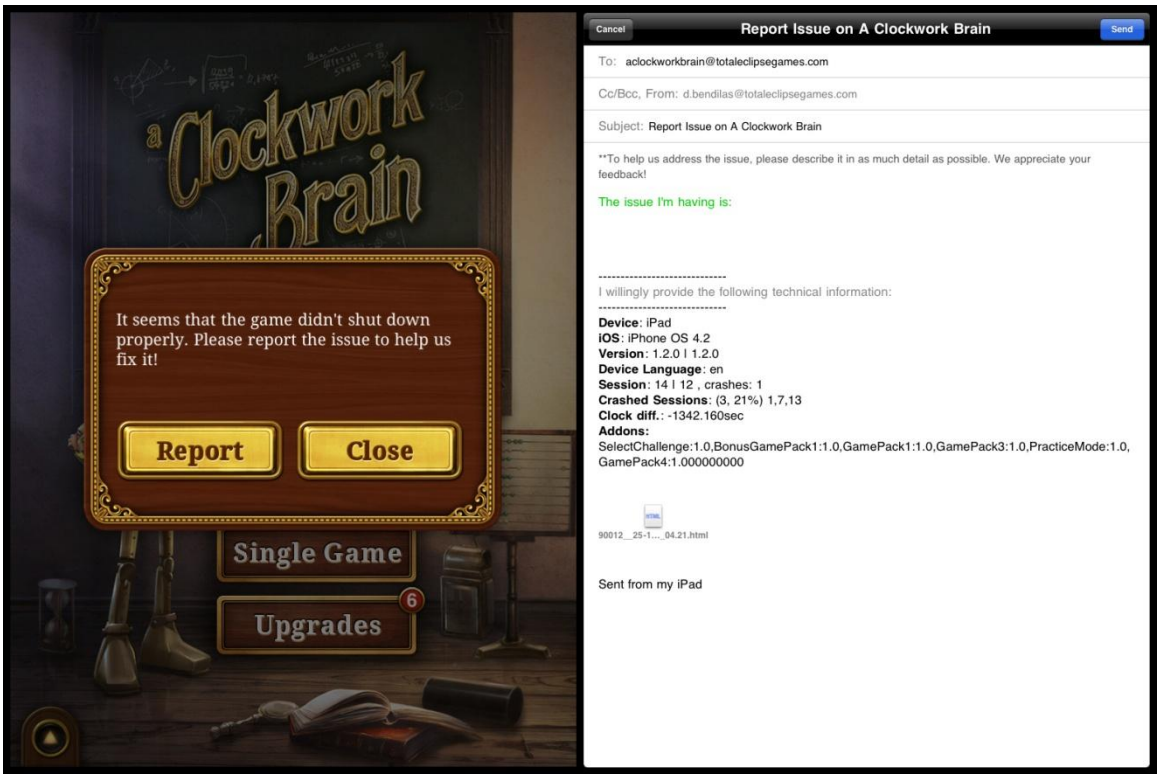
In our case, the actual time it took the app to be reviewed fell right on the average; in 7 days it was cleared for the App Store. On our subsequent updates, version 1.1.0 and 1.2.0, it was 9 and 11 days respectively.

7.8 Support

As we have learned during the past 8 years of running a game development studio, something that never ends is the need for support for a released product. It depends on how popular a title is and how many years have passed since its release, but, in general, the need for support can exist for several years.

In Clockwork Brain we implemented a Crash Report tool, to help us provide better support to our players. Every time there was a crash, a crash log was created by SolarWind's Crash Manager, and was saved locally. The next time the user launched the application, a panel would appear, asking the user to report the issue to us, as seen in Figure 7.16. If the user chose to report the issue, an email form was opened, filled with a lot of information that was useful to us, together with an attachment of the crash log.

Figure 7.16 Crash Report Tool – Prompt and E-mail form



The player could type a description of the issue they had faced, or simply send the email. The contents of the report were optimized by us many times, each time adding more information or formatting the existing data in a more readable or useful way. On version 1.2.0 the report contained a list of the Upgrades the player had purchased or unlocked, how many times the application launched, the session IDs of all the times the application crashed, the device language and a log of all the important actions that took place during the session that crashed.

These emails would arrive in our inbox, and then we would evaluate each one and reply or simply examine the crash information, in case the user had not written anything. In October 2012 we integrated ZenDesk[14], a customer service platform, which allowed us to provide much better support to our players.

[14] www.zendesk.com

Offering a way to report an error inside an application is not uncommon. Apart from the obvious benefits for the user, which can affect their satisfaction, in mobile applications there is one more reason to include such a feature. When players have trouble with an app, at least on iOS, they often visit the App Store and write a negative review for the app, sometimes giving a rating of 1 out of 5 stars. These ratings are public, so they can really harm the reputation of an application and therefore affect its sales or downloads. Having an issue report feature shows users that you care for them and usually makes them express their problems to you directly, instead of going public and exposing a bad critique to everyone.

7.9 Conclusions

The technical aspects described above were an important part of this project. In our experience, the handling of these issues is often more time consuming than the creation of the core game itself. Being integral components of every production, technical matters require as much attention as everything else, because sometimes they can affect the quality and success of a product in major ways. We have found out that it is wise for studios to invest in solving these issues by creating well-designed, reusable libraries and frameworks. This way they can use them in more than one project, helping reduce the cost and development duration by a large factor. Even if the code itself cannot be reused, because the new platform does not support the same programming language, a solid architecture can make it much easier to port the functionality into new projects.

Chapter 8

Analytics

Chapter 9 describes the use of in-game statistical data and its use for various purposes. It outlines the importance of analytics in modern applications and provides tips about their implementation in a game. It also includes examples of data-driven analysis we conducted in A Clockwork Brain and the decisions we made using that data, in order to improve specific aspects of the game.

8.1 Overview

Game analytics is the data collected when users interact with a game application. This data is used by developers in order to understand the behavior of players, detect issues in game design and user interface design, find opportunities to improve on various aspects, such as better monetization and balancing, or simply analyze usage data for statistical reasons with multiple uses.

8.2 Importance

Data collection can be extremely helpful in many areas. First of all, it allows developers to know how often and in what fashion players are using the application. In our case, for example, we can find out which mini games users play more, if they prefer Single Game or Challenge mode, if they share their scores on Facebook and how often they visit each section of the application. Regarding game play, analytics can work as a continuous, real-time balancing testing session. There are many variables that can be reported, such as the score and bonus level achieved in each mini game, the number of flawless sessions or sessions reaching Insane Round. This data can be later analyzed and used to deduct useful conclusions. For instance, we can find out if the scores and bonus levels that players achieve in each mini game are balanced or not among the various games. In fact, this analysis proved that a few mini games were easier than the others. This could not come up during Usability and Beta testing, where only a few dozens of people took part, but it became clear when we gathered information from hundreds of thousands of players.

Another type of data that made a great impact was related to In-App Purchases. We created a number of analytics events that had to do with the process of purchases, from viewing a certain game pack, until the actual purchase had been made. The data we collected painted a surprising picture, showing that players that buy a game pack have only played for a few days before completing their first purchase, as seen in Chart 8.3. This information was very important, as it helped us shape our strategy regarding the monetization of the application.

8.3 Analytics software

There are various analytics software solutions in the market. When we were developing A Clockwork Brain, one of the most popular was Flurry [15], a free analytics service with many features. Currently serving more than 70.000 companies with 200.000 applications on iOS and other platforms, Flurry is a powerful tool for recording player behavior data, and providing advanced features such as funnel analysis, conversion tracking and custom segmentation. Their SDK is very straightforward and easy to integrate in any application.

Figure 8.1 Flurry's geography reporting, using sample data

Source: Flurry.com

The Total Eclipse team had worked with Flurry during a previous production, so we were already experienced with the integration and the capabilities of the platform.

8.4 Gathered information

In total, we report more than 140 different events in A Clockwork Brain. Some of them are plain events, such as *Credits*, which tells us that a user has visited the Credits page, whereas others are fairly complex, such as the *UpgPurchased_1st*, which is called when a user first buys a game pack and is recorded together with 8 different parameters that describe the circumstances under which the upgrade was bought.

Some of these events are presented in Table 8.1, along with a few of their parameters.

8.4.1 Events explained

A few of the events used in the game are described below.

[15] www.flurry.com

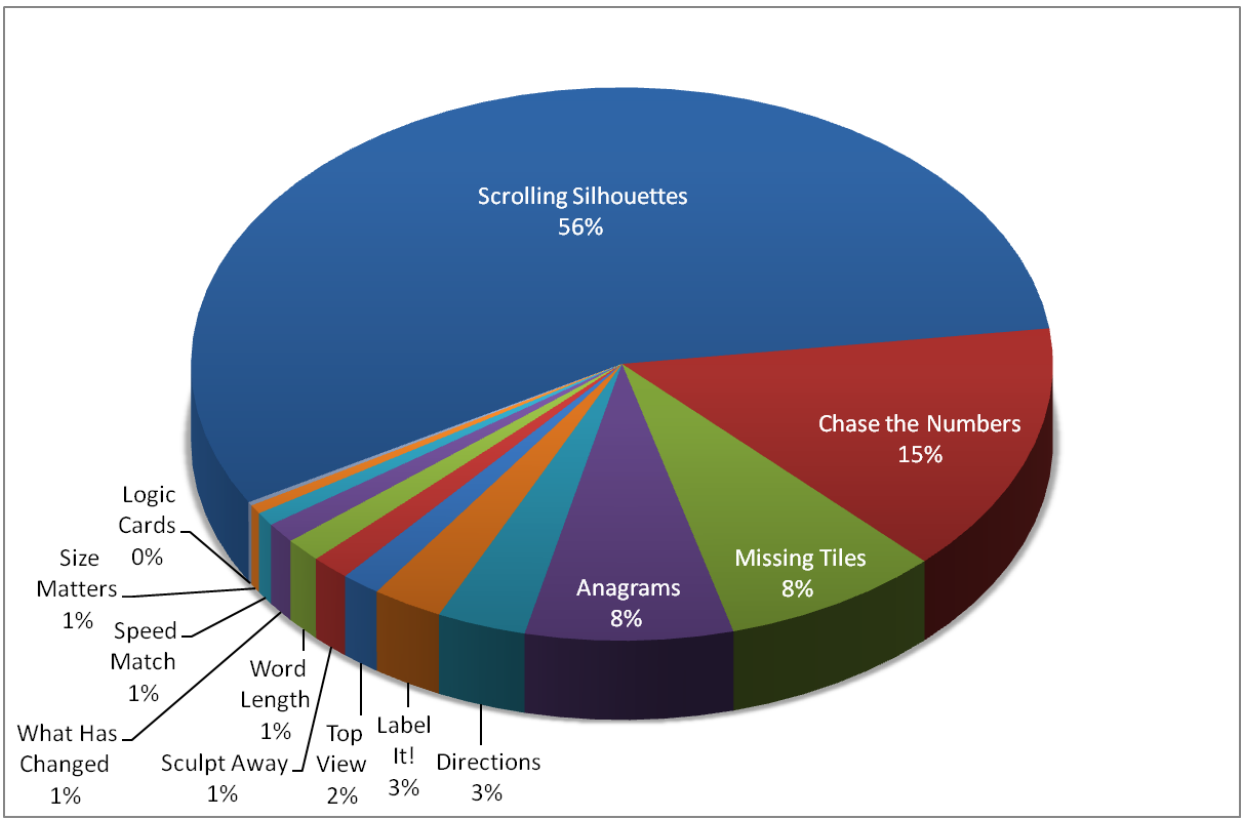
GameComplete

One of the most basic events records the name of the mini game in each completed mini game session. This allows us to know which games users play the most and which the least. Chart 8.1 shows that players display a strong preference towards Scrolling Silhouettes, as this mini game alone accounts for more than half of the total mini game sessions ever played.

Table 8.1 Some of the analytics event in Clockwork Brain

Event	Event Description	Param	Param description
Bonus_Anagram	Bonus level achieved in Anagrams.	level:float	Bonus achieved
Bonus10_Anagram	Bonus level achieved in Anagrams, only in the first 10 mini game sessions.	level:float	Bonus achieved
FBShare	Player pressed the Facebook share button.	game:String	The mini game played
FBShareDone	Player actually completed sharing.	game:String	The mini game played
Flawless	Player achieved a flawless victory in a mini game session.	game:String	The mini game played
GameComplete	A Single Game session was completed.	game:String	The mini game played
GameInChallenge	A mini game was completed, inside a Challenge session.	game:String	The mini game played
		createChallenge:Bool	Create Challenge upgrade available or not
		singleGame:Bool	Single Game unlocked or not
InsaneRound	Player reached Insane Round in a mini game.	game:String	The mini game played
InsRnds_Anagram	Player reached Insane Round in a mini game.	levels:int	Insane rounds achieved
LostTokens	Player had lost some Tokens and now she got them back.	tokens:int	Number of tokens awarded
TellAFriend	User suggested the app to a friend via email.	-	-
TokensTotal	Number of total Tokens a player has collected.	tokens:int	Number of tokens
TryStarted	A player has just started a Try game session.	game:String	The name of the mini game
		ratio:int	Tokens available / tokens needed
TryNotStarted	A player has just decided not to start a Try game session.	game:String	The name of the mini game
		ratio:int	Tokens available / tokens needed
UpgPurchased	Player just purchased an Upgrade.	upgrade:String	The purchased upgrade
		tries:int	Try sessions for this game pack
		otherPackTries:int	Try sessions for other game packs
UpgPurchased_1st	Player just purchased her first Upgrade.	tries:int	Try sessions for this game pack
		otherPackTries:int	Try sessions for other game packs
		daySpan:int	Days since first launch
		totalGames:int	Total mini games played
		avgBonus:int	Average bonus level in all sessions
WordLangChanged	Player just changed the language	language:String	The language selected

Chart 8.1 The distribution of played mini games in GameComplete event



To better interprete this diagram, one needs to consider that by default only 4 games are available on the free application; Scrolling Silhouettes, Chase the Numbers, Missing Tiles and Anagrams. The rest are locked and can be unlocked via In-App Purchases, or Sprocket Tokens (Size Matters). Having that in mind, it makes sense that these four games appear to have far more sessions than the rest.

This information can be very valuable. We can study it carefully, analyze the attributes of each mini game, and try to understand what types of games our players like to play the most. This way we can design mini games that share the same attributes as the most popular ones in the future, when introducing new game packs. Additionally, we can use this information when promoting the game, in marketing copy, video trailers, screenshots or any other type of media, or even promote the most popular games more heavily.

TryStarted vs TryNotStarted

When a user enters the Single Game screen, she is presented with all mini games, both the ones that were purchased and the ones that are still locked. As of version 1.1.0, players can try any locked mini game, using some of their Sprocket Tokens. This gives them the opportunity to play the game and decide if they want to purchase it or not, instead of relying on a 3-line description and a few screenshots.

Figure 8.2 Confirmation panel for trying a mini game

When the user reaches the Description screen of the mini game, the *Play Game* button has been changed to *Try game*. If the player taps on the button, a pop-up window appears and asks for confirmation. Depending on the button the player taps, a analytics different event is fired, TryStarted if Play is pressed, or TryNotStarted if Cancel is pressed.

These two events can help us calculate the conversion rate of this screen.

*Conversion Rate % = 100 * TryStarted / (TryStarted + TryNotStarted)*

Later on, this screen can be modified, probably by using A/B Testing techniques, and see if the conversion rate increases or decreases.

UpgPurchased_1st

This event is called when a player purchases an upgrade for the first time. This is one of the most important events in the game, because it defines the point at which a player becomes a paying customer. From a monetization standpoint, it is probably the most crucial, as at this point a free product manages to generate revenue. Thus, it is critical to know under which circumstances a player converts for the first time. The use of analytics can help create the profile of buying customers provide valuable information, which can later be used from the sales and design teams to further improve monetization.

There are several parameters that are recorded along with this event; `daySpan`, `totalSessions`, `totalGames`, `avgBonus`, `bestBonus`, tries and `otherPackTries`. The first 3 show how much the player has interacted with the game, measured in days, sessions (app launches) and mini game sessions. The next two demonstrate the player's skill and the last two show whether the player has used the *Try Game* feature.

As it turns out, most users of Clockwork Brain make their first purchase very early, after having played only a few mini game sessions, with 66% of them typically within a couple of days after downloading the application. This valuable piece of information is

currently being evaluated by Total Eclipse, to be used for improving the conversion rate of the application.

Chart 8.2 First In-App Purchase in UpgPurchased_1st event in app version 1.2.0

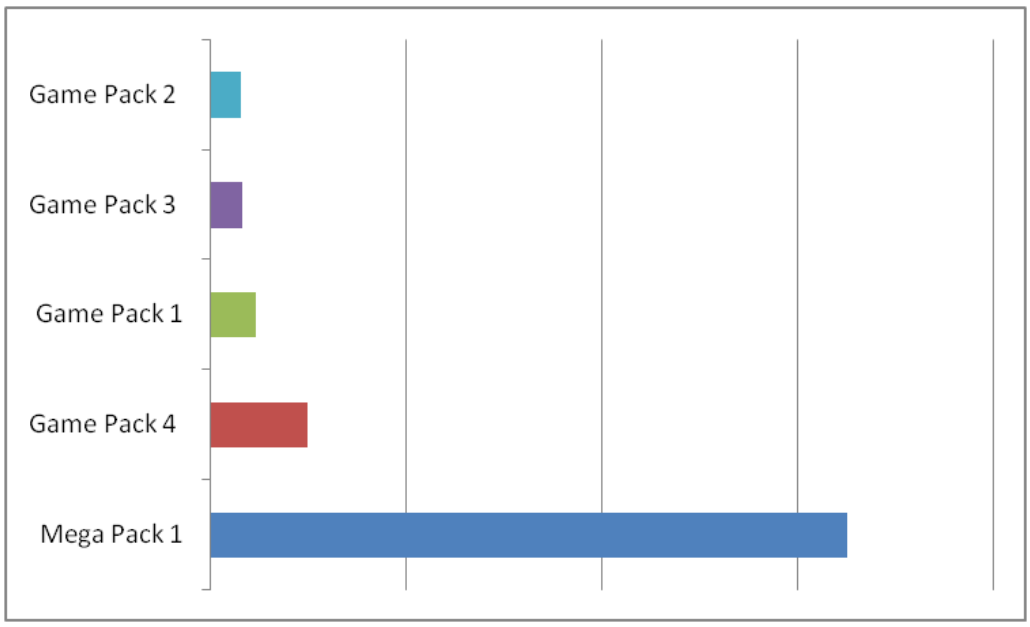
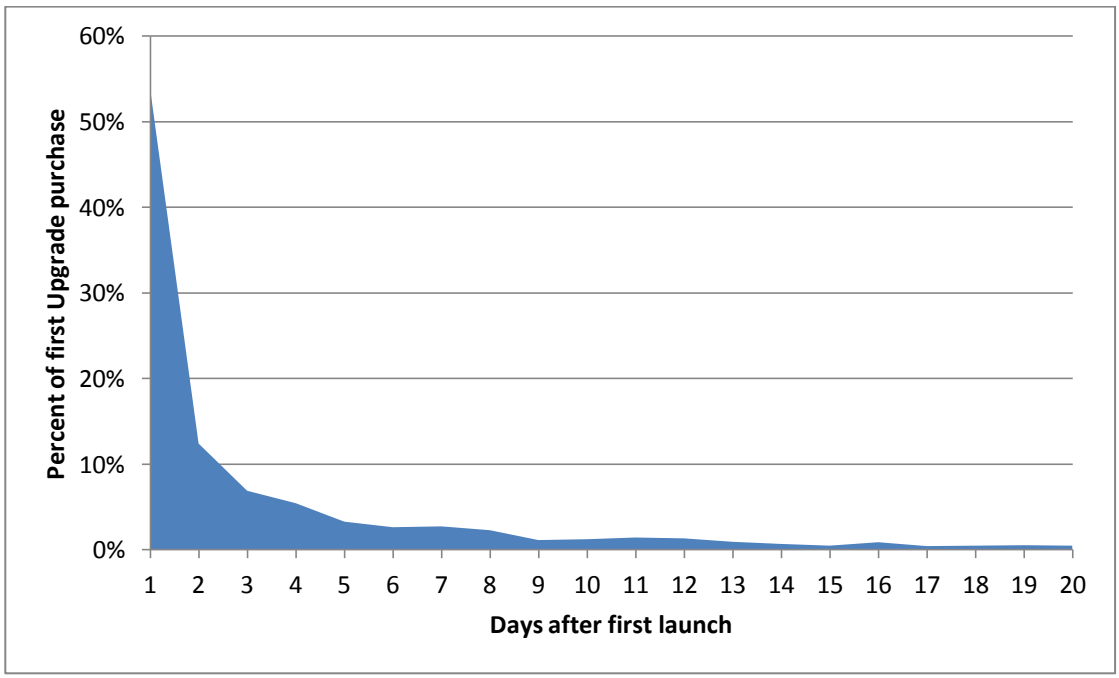


Chart 8.3 Days after downloading the app when users purchase their first Upgrade



Bonus_Anagram

This event records the Bonus Level (number of stars) a player achieves in a mini game session. Similar events were created separately for every mini game, e.g. Bonus_Directions, Bonus_ScrollingSilhouettes etc. The results were used to compare and

improve the balancing of the mini games. In a similar fashion, the events Bonus10_Anagram, Bonus10_Directions etc, record the bonus level during the first 10 sessions of a mini game only, as we wanted to provide extra care for the players that tried a mini game for the first few times, making sure the learning curve is not steep and making the players feel good about their performance from the very beginning.

8.4.2 Mistakes

A Clockwork Brain was the first mobile production of such scale for Total Eclipse and the first game of the company that was distributed as a freemium application. The team had no prior experience with managing so many and so complex analytics events. Naturally, there were a few mistakes made, which are explained below.

Proper binding

Clockwork Brain contains some mini games that rely on words. These games are localized in 9 languages. Users can select any of these languages in the Settings screen.

Figure 8.3 Language selector for word games

When the first version of the application shipped, we had bound the WordLangChanged event with the selected button callback. Every time players tapped on a language button, the event would fire. In case users played around with the selector, changing multiple languages in a row, until deciding which one to select for real, there would be a number of analytics events recorded, that did not provide an accurate representation of the actual language use in the game. This erroneous approach was fixed in version 1.1.0, by firing the WordLangChanged event only when the player closed the Settings screen, and not each time a language button was selected. This allowed us to get a better understanding of how often users change the language and what languages are mostly used.

Value aggregation

When using an analytics software, it is important to understand its limitations. In our case, Flurry had such a limitation, related to numeric values. As we discovered during development, Flurry cannot accurately display a large number of distinct numeric values and it automatically creates a group of values named *Other*, if its count reaches a certain point Flurry has defined internally. The table below illustrates this behavior. Its values are not real, because we would need a table of triple or more lines, but for the sake of argument it is able to illustrate the limitations of Flurry in this regard.

Table 8.2 Limitations of numeric value reporting in Flurry analytics

Score	Count	Percent
Other	250	54%
1380	49	11%
2380	35	8%
4280	33	7%
2830	25	5%
23420	23	5%
5830	12	3%
1720	8	2%
2980	8	2%
11280	5	1%
49380	4	1%
4380	3	1%
85730	2	0%
54990	2	0%
58740	1	0%

For each score event , we submit its score value to Flurry, e.g. 1380 points. When Flurry creates a report, it aggregates all entries that have the same value, in order to display meaningful data. For example, it shows that there are 49 entries with the value of 1380. The issue arises when there are many different values. In that case, Flurry creates a cluster of all entries that their value appears only a few times and displays them together with the value *Other*. Even if you try to download a CSV file with all the values of the table, from Flurry's dashboard, the values are saved in an aggregated form. If the count of these events is large, you may end up having a significant percentage of unknown values. In the example above, 54% of the reported values are unknown, which makes the data much less valuable.

The solution is to create groups of values and report rounded numbers instead of the actual numbers. That is exactly what we did in Clockwork Brain, when, during development, we found out that the score values we were submitting created large blocks of unknown values. We created a few ranges for the score values and set different rounding rules for each one. Scores from 0 to 10.000 points would be rounded to the next 200 points, scores from 10.000 to 100.000 points would be rounded to the next 1.000 points and scores larger than 100.000 points would be rounded to the next 100.000 points.

For example, a value of 514 points becomes 600, 2780 becomes 2800, 24.380 becomes 25.000 and 178.200 becomes 200.000. This type of reporting makes sense for us, because we do not require more accuracy than this. Depending on the range a value can take,

one may need to define even larger rounding ranges, so that there are not too many different values reported.

The code we used for modifying the actual score value to one that is safe for reporting is the following:

```
+ (NSString*) getScoreRangeStringForAnalytics:(int)score {
    float rangeSize = 200.0;
    if (score > 100000) {
        rangeSize = 10000.0;
    } else if (score > 10000) {
        rangeSize = 1000.0;
    }
    float normalized = score / rangeSize;
    float scoreCategory = ceilf(normalized) * rangeSize;
    return [NSString stringWithFormat:@"%.0f", scoreCategory];
}
```

Completeness

As mentioned before, we created 140 different analytics events for Clockwork Brain. Not all of them were created from the beginning though, as we did not think of them in time, before the initial application launch. For example, even though we had the UpgPurchased_1st event, we only submitted one parameter, the name of the game pack the player had just purchased. It wasn't until version 1.1.0 that we realized we should take advantage of this event and record the circumstances under which the purchase was made, adding 7 more parameters that were very important, as explained earlier.

Until that time, we had more than 100.000 downloads and many purchases that we did not collect this valuable information for. As a result, we lost an opportunity to understand our customers earlier and improve the monetization of the application. This made us realize that it is of great importance to create as many events as possible from the beginning, even ones that may seem redundant, and decide later which to keep and which to remove.

8.5 Segmentation

Flurry allows developers to create custom segments, which can be used to filter data and provide reports with certain attribute specification. For example, one can create a segment for USA users and view all events of the app, only for users from USA.

In Clockwork Brain we created a number of custom segments, which helped us track the behavior of our users, based on their geography, loyalty and application use. Some of these segments can be found in the table below.

Table 8.3 Custom analytics segments in Clockwork Brain

Segment	Description
USA users	Users from USA.
UK users	Users from UK.
Greek users	Users from Greece.
Paying customers	Users that have bought at least one game pack.
Returning customers	Users that have bought at least two game pack.
Users that tried	Users that have used tokens to try at least one mini game.
Frequent users	Users that play at least once a day.

8.6 Conclusions

Analytics is a powerful tool for mobile applications. It can provide insights on many areas regarding the way people use an application and can offer important data that can be used to improve the application. Designing the analytics events for A Clockwork Brain was not an easy task. Although we had a lot of experience from our previous mobile title, the differences between the two games and the fact that this was our first free to play game, led us to us omit important events and parameters. Data analysis we conducted after the launch, showed us what we could improve upon and helped us create a better application.

Conclusion

The first part of the thesis dealt with the mobile game development industry. As demonstrated in the introduction, research in the smartphone and tablet market has shown that it is currently growing rapidly; this is unlikely to change in the next few years. Hundreds of millions of people are using these devices, making them part of their everyday life. The industry is practically in its infancy and there are several major companies investing in it. Modern mobile devices come with a live ecosystem of thousands of applications, which take advantage of the powerful features the hardware provides and offer real value to consumers. A significant portion of these applications are games, accounting for the lion's share in generated revenue. The current trend in mobile games is freemium apps, which are free for the users to download and use, but include premium items that can be purchased at the user's will.

Even though developers have earned billions of dollars collectively from selling their applications, the success of a mobile app is not guaranteed, something that can be attributed to several reasons. The fact that there are over 1.5 million applications on major platforms combined, all competing for the same top ranks, makes it difficult for new applications to be seen by prospective customers, especially if they are published by small development teams with low marketing budgets. Also, device fragmentation in some platforms seems to be a major barrier for developers that want to enter the industry.

Games in particular have additional reasons to fail, as their purpose greatly differs from other types of applications, which are mainly used as tools. Studies and research show that compared to non-gaming software, the goal of a game is much more abstract, reaching for the player's emotions, rather than being used as a means to produce a tangible result. Along with the fact that games need a far more diverse team of professionals, and require better devices in terms of performance and memory, make both the project management and the development of games harder than other types of software. Agile development processes are helping developers overcome some serious obstacles, but are still not enough to make all challenges disappear.

The second part presented a case study for mobile game software. The development of a commercial freemium game, A Clockwork Brain, was described in detail. Many of the design, development and management aspects of the production were presented, for the initial release as well as subsequent updates on Apple's App Store.

As with most software projects and games in particular, A Clockwork Brain proved to be challenging; much more than initially anticipated. The original plan was modified and the schedule had to be re-evaluated and changed multiple times during development. The use of non-linear processes and the implementation of iterative cycles

during development helped the team create a high quality product. However, the important changes that occurred in the app ecosystem during development did not allow for a very smooth production. The implementation of the technical aspects of the application proved to be equally time-consuming as the core game itself. At the same time, the balancing of the game was one of the most challenging tasks and required the coordination of many of the team members, as well as various methodologies, in order to be carried out.

Testing was a major milestone for the team, as it was carried out in-house for the first time. The findings of both Usability and Beta testing were eye-opening and contributed a lot to the success of the game. Also, it became clear that it is not difficult for a small studio to execute official testing of this kind. The valuable experience gained can be used in future projects as well.

The end product was very well-received by players, managing to exceed 300.000 downloads within 8 months and getting great reviews and ratings on the App Store. The clean software architecture allowed for faster development times for the new updates of the application, as well as for easier code maintenance. The extensive use of analytics and the data-driven decisions followed after the first release, helped the application improve its performance, in terms of both revenue for the developers and gameplay for the players.

References

[1] P. Farago, "iOS and Android Adoption Explodes Internationally," 27 Aug 2012. [Online]. Available: http://blog.flurry.com/bid/88867/iOS-and-Android-Adoption-Explodes-Internationally.

[2] IDC, "IDC - Press Release: Android- and iOS-Powered Smartphones Expand Their Share of the Market in the First Quarter, According to IDC," 24 May 2012. [Online]. Available: http://www.idc.com/getdoc.jsp?containerId=prUS23503312.

[3] T. Cook, *WWDC June 2012*, 2012.

[4] R. Blandform, "100,000 apps published to Windows Phone Marketplace," 5 Jun 2012. [Online]. Available: http://allaboutwindowsphone.com/news/item/14960_100000_apps_published_to_Windo.php.

[5] J. Rosenberg, "Google Play hits 25 billion downloads," 26 Sep 2012. [Online]. Available: http://officialandroid.blogspot.gr/2012/09/google-play-hits-25-billion-downloads.html.

[6] A. Saunders, *BlackBerry 10 Jam*, 2012.

[7] AppleInc., "iOS Developer Program," [Online]. Available: https://developer.apple.com/programs/ios/. [Accessed Oct 2012].

[8] Google Inc., "Android Developers," [Online]. Available: http://developer.android.com/. [Accessed Oct 2012].

[9] Apple Inc., "Creating Jobs through Innovation," 2012. [Online]. Available: http://www.apple.com/about/job-creation/.

[10] P. Farago, "The Truth About Cats and Dogs: Smartphone vs Tablet Usage Differences," Flurry, 29 Oct 2012. [Online]. Available: http://blog.flurry.com/bid/90987/The-Truth-About-Cats-and-Dogs-Smartphone-vs-Tablet-Usage-Differences.

[11] 148Apps.biz, "App Store Metrics," 10 Oct 2012. [Online]. Available: http://148apps.biz/app-store-metrics/.

[12] Intel, "The Personal Computer Turns 25: Then and Now Technology Timeline," [Online]. Available: http://www.intel.com/museum/archives/pctimeline.htm. [Accessed Oct 2012].

[13] M. Meeker, "Distimo Presentation From Festival Of Games," 26 Apr 2012. [Online]. Available: http://www.distimo.com/blog/2012_04_distimo-presentation-from-festival-of-games/.

[14] M. Meeker, "Top Mobile Internet Trends," 10 Feb 2011. [Online]. Available: http://www.slideshare.net/kleinerperkins/kpcb-top-10-mobile-trends-feb-2011.

[15] Wikipedia, "List of iOS devices," [Online]. Available: http://en.wikipedia.org/wiki/List_of_iOS_devices. [Accessed 22 Oct 2012].

[16] Google Inc., "Supported Devices," 10 Mar 2012. [Online]. Available: http://support.google.com/googleplay/bin/answer.py?hl=en&answer=1727131.

[17] Wikipedia, "List of Windows Phone devices," [Online]. Available: http://en.wikipedia.org/wiki/List_of_Windows_Phone_devices. [Accessed 22 Oct 2012].

[18] N. Luckyanova, "Twitter post," 27 Mar 2012. [Online]. Available: https://twitter.com/nattylux/status/184689222135975936.

[19] M. King and S. Ellison, "Q1 2012 Mobile Developer Report," Appcelerator / IDC, 2012.

[20] Rovio, "1 Billion Angry Birds Downloads!," 5 Sep 2012. [Online]. Available: http://www.rovio.com/en/news/blog/162/1-billion-angry-birds-downloads.

[21] Epic Games, "Epic Games and ChAIR Entertainment Announce Earnings from Infinity Blade Franchise Exceed $30MM," 5 Jan 2012. [Online]. Available: http://epicgames.com/news/epic-games-and-chair-entertainment-announce-earnings-from-infinity-blade-fr/.

[22] I. Marsh, 3 Jul 2011. [Online]. Available: https://twitter.com/eeen/status/87545357478793216. [Accessed 19 Oct 2012].

[23] 2D Boy, "One Million Downloads!," 9 Jan 2012. [Online]. Available: http://2dboy.com/2012/01/09/one-million-downloads/.

[24] P. Macchiarella, "Trends in Digital Gaming: Free-to-Play, Social, and Mobile Games," Parks Associates, Dallas, Texas, 2012.

[25] J. Kincaid, "Apple Announces In-App Purchases For Free iPhone Applications," Techrunch, 15 Oct 2009. [Online]. Available: http://techcrunch.com/2009/10/15/apple-announces-in-app-purchases-for-free-iphone-applications/. [Accessed 29 Oct 2012].

[26] J. Valadares, "Free-to-play Revenue Overtakes Premium Revenue in the App Store,"

Flurry, 7 Jul 2012. [Online]. Available: http://blog.flurry.com/bid/65656/Free-to-play-Revenue-Overtakes-Premium-Revenue-in-the-App-Store. [Accessed 1 Oct 2012].

[27] AppAnnie, "Freemium Apps are Exploding, Japan and China among growth leaders," 26 Oct 2012. [Online]. Available: http://www.appannie.com/blog/freemium-apps-ios-google-play-japan-china-leaders/.

[28] D. Takahashi, "NaturalMotion's iOS racing game generates $12M in revenue in first month," VentureBeat, 15 Aug 2012. [Online]. Available: http://venturebeat.com/2012/08/15/naturalmotions-csr-racing-ios-game-generates-12m-in-revenue-per-month/. [Accessed 29 Oct 2012].

[29] G. J. Spriensma, "Quora Answering Series; Download Volume Needed To Hit Top 25 Per Category," 16 May 2012. [Online]. Available: http://www.distimo.com/blog/2012_05_quora-answering-series-download-volume-needed-to-hit-top-25-per-category/.

[30] O. Goss, "Results: iOS Game Revenue Survey," 28 Sep 2011. [Online]. Available: http://www.streamingcolour.com/blog/2011/09/28/results-ios-game-revenue-survey/.

[31] App Promo, "Wake Up Call – If You Spend It, They Will Come," 2 May 2012. [Online]. Available: http://app-promo.com/wake-up-call-infographic/.

[32] C. Foresman, "iOS app success is a "lottery": 60% (or more) of developers don't break even," 4 May 2012. [Online]. Available: http://arstechnica.com/apple/2012/05/ios-app-success-is-a-lottery-and-60-of-developers-dont-break-even/.

[33] K. Salen and E. Zimmerman, Rules of Play: Game Design Fundamentals, The MIT Press, 2003.

[34] T. Fullerton, Game Design Workshop-A playcentric approach to creating innovative games, Morgan Kaufmann Publishers, 2008.

[35] R. Koster, A Theory of Fun for Game Design, Paraglyph Press, 2004.

[36] B. Suits, The Grasshopper: Games, Life and Utopia, Broadview Press, 2005.

[37] R. Caillois, Man, Play and Games, University of Illinois Press, 2001.

[38] T. W. Malone, "Heuristics for Designing Enjoyable User Interfaces: Lessons from Computer Games," Xerox Palo Alto Researcher Center, Palo Alto, 1981.

[39] I. Bogost, How to Do Things with Videogames, U of Minnesota Press, 2011.

[40] N. Lazzaro, "Why We Play Games: Four Keys to More Emotion Without Story," XEODesign Inc., 2004.

[41] E. N. K. S. David Callele, "Emotional Requirements in Video Games," in *14th IEEE International Requirements Engineering Conference*, Saskatoon, Saskatchewan, Canada, 2006.

[42] S. D. Bristow, "The History of Video Games," in *IEEE Transactions of Consumer Electronics*, Sunnyvale, California, 1977.

[43] E. Bethke, Game Development and Production, Wordware Publishing, 2003.

[44] J. Blow, "Game Development: Harder Than You Think," Queue, 2004.

[45] P. C. R. E. Fadi Chehimi, "Evolution of 3D mobile games development," *Personal and Ubiquitous Computing,* vol. 12, no. 1, p. 19–25, 2008.

[46] T. L. Tom DeMarco, Waltzing With Bears: Managing Risk on Software Projects, Dorset House, 2003.

[47] C. Jones, "Patterns of large software systems: failure and success," *Computer,* vol. 28, no. 3, pp. 86-87, 1995.

[48] R. N. Charette, "Why software fails," *Spectrum, IEEE,* vol. 42, no. 9, pp. 24-49, 2005.

[49] K. Flood, "Game Unified Process," 14 May 2003. [Online]. Available: http://www.gamedev.net/page/resources/_/technical/general-programming/game-unified-process-r1940. [Accessed 22 Oct 2012].

[50] J. Flynt and O. Salem, Software Engineering for Game Developers (Software Engineering Series), Course Technology PTR, 2004.

[51] S. Soundararajan, J. D. Arthur and O. Balci, "A Methodology for assessing Agile Software Development Approaches," in *2012 Agile Conference*, Dallas, TX USA, 2012.

[52] R. J. Q. U. A. K. Asif Irshad Khan, "A Comprehensive Study of Commonly Practiced Heavy & Light Weight Software Methodologies," *IJCSI International Journal of Computer Science Issues,* vol. 8, no. 4, pp. 441-450, 2011.

[53] S. Singh and I. Chana, "Enabling Reusability in Agile Software Development," *International Journal of Computer Applications,* vol. 50, no. 13, pp. 33-40, 2012.

[54] H. M. Chandler, The Game Production Handbook, Second Edition, Jones & Bartlett

Publishers, 2008.

[55] Z. Fernando, "Tweener Documentation," [Online]. Available: http://hosted.zeh.com.br/tweener/docs/en-us/misc/transitions.html. [Accessed 30 Oct 2012].

[56] "TexturePacker," [Online]. Available: http://www.codeandweb.com/texturepacker. [Accessed 09 2012].

[57] S. Fenney, "Texture compression using low-frequency signal modulation," in *ACM Siggraph/Eurographics conference on Graphics hardware*, 2003.

[58] M. Sifnioti, "Game Usability Testing for Indies: It's Easier than you Might Think! (Part 1)," Total Eclipse, 25 Jan 2012. [Online]. Available: http://blog.totaleclipsegames.com/game-usability-testing-for-indies-its-easier-than-you-might-think-part1/.

[59] M. Sifnioti, "Game Usability Testing for Indies: It's Easier than you Might Think! (Part 2)," Total Eclipse, 16 Jul 2012. [Online]. Available: http://blog.totaleclipsegames.com/game-usability-testing-for-indies-its-easier-than-you-might-think-part-2/.

[60] M. Sifnioti, *Game Usability Testing for Indies: It's Easier than you Might Think! (Part 3),* Unpublished manuscript, 2012.

[61] Apple Inc., "Local and Push Notification Programming Guide," Apple Inc., 8 Sep 2011. [Online]. Available: http://developer.apple.com/library/mac/#documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/ApplePushService/ApplePushService.html. [Accessed 12 Oct 2012].

[62] Apple Inc., "In-App Purchase Programming Guide," 19 9 2012. [Online]. Available: https://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/StoreKitGuide/Introduction/Introduction.html. [Accessed 20 10 2012].

[63] Apple Inc., "Technical Note TN2259 - Adding In-App Purchase to your iOS and Mac Applications," 22 2 2012. [Online]. Available: https://developer.apple.com/library/ios/#technotes/tn2259/_index.html. [Accessed 10 10 2012].

[64] Apple Inc., "Store Kit Framework Reference," 19 9 2012. [Online]. Available: https://developer.apple.com/library/ios/#documentation/StoreKit/Reference/StoreKit_C

ollection/_index.html#//apple_ref/doc/uid/TP40008300. [Accessed 20 10 2012].

[65] T. Brant, "Invalid Product IDs," 17 Jan 2010. [Online]. Available: http://troybrant.net/blog/2010/01/invalid-product-ids/. [Accessed 10 Oct 2012].

[66] U.S. Government, "SNAP-R," [Online]. Available: https://snapr.bis.doc.gov/snapr/exp/UserLoginLoad. [Accessed 15 Oct 2012].

[67] U.S Government, "How to file an encryption registration," [Online]. Available: http://www.bis.doc.gov/encryption/question3_sub.htm. [Accessed 15 Oct 2012].

[68] Shiny Development., "iOS App Store- Rolling Annual Trend Graph," [Online]. Available: http://reviewtimes.shinydevelopment.com/ios-annual-trend-graph.html. [Accessed 23 Oct 2012].

[69] C. Newark-French, "Mobile Apps Put the Web in Their Rear-view Mirror," 20 Jun 2011. [Online]. Available: http://blog.flurry.com/bid/63907/Mobile-Apps-Put-the-Web-in-Their-Rear-view-Mirror.

[70] J. Sauro, "http://www.measuringusability.com/sus.php," 2 Feb 2011. [Online]. Available: Measuring Usability With The System Usability Scale (SUS).

[71] J. S. J. N. Lennart Nacke, "Gameplay experience testing with playability and usability surveys – An experimental pilot study," in *In Playability and player experience: Proceedings of the Fun and Games 2010 Workshop* .

[72] Q. Wang, "Research and Design of Edutainment," in *Information Technologies and Applications in Education, 2007. ISITAE '07*, 2007.

[73] Apple Inc., "iOS Developer Program - 3. Distribute," Apple Inc., [Online]. Available: https://developer.apple.com/programs/ios/distribute.html. [Accessed 29 Oct 2012].