Problems preventing the adoption of requirements engineering methods in game development

Miikka Lehtonen

University of Tampere
Faculty of Natural Sciences
Computer Science
Master’s thesis
Supervisor: Timo Nummenmaa
15.6.2018

University of Tampere
Faculty of Natural Sciences
Degree Programme in Software Development
Miikka Lehtonen: Problems preventing the adoption of requirements engineering methods in game development
M.Sc. thesis, 52 pages, 5 index pages, 4 appendix pages
June 2018

As the game industry continues to grow in size and revenue, the cost of creating games increases as well, and the successful outcome of game development projects becomes ever more important.

In traditional software engineering it is generally agreed that a successful requirements engineering process (or lack thereof) has a significant impact in the outlook and outcome of the project. The methods and processes employed in requirements engineering have been discussed, debated and fine-tuned over decades of use, and can be successfully utilized in both traditional waterfall type software development projects, as well as agile and lean projects, which game development usually falls under.

Yet in game development, requirements engineering methods do not seem to be commonly used in any way. As game development is a specialized form of software development, it seems intuitively likely that game developers could benefit from adopting these techniques and processes. Clearly, then, something is preventing them from utilizing these methods and processes which could help them, but what?

This thesis explores the issue by performing a thorough reading of central and current academic research on the topic, and attempts to form a holistic picture of the issues and problems preventing the adoption and widespread use of requirements engineering processes and methods in game development.

In addition to identifying these central problems and issues, the thesis also attempts to verify the existence of these problems by conducting an algorithmic analysis of 315 post-mortems written by game developers and published on industry websites. These post-mortems discuss the factors which contributed to or hindered the successful outcome of these game development projects. They could therefore offer evidence either for or against the significance of these problems and issues.

Keywords: game development, software development, requirements engineering, requirements, post-mortem

Acknowledgements

I would like to thank my instructors Timo Nummenmaa and Zheying Zhang for the guidance and instruction they gave to me during this process. Additionally, I would like to thank Professor Jaakko Peltonen and Chien Lu for their help with the algorithm used in this thesis work. Finally, I would like to thank my partner Kati Alha for the support she has given me.

Table of Contents

1 Introduction

Requirements engineering has been a part of software development for decades, and much has been written on its applications in various domains. Digital game development is no exception. As game development is a specialized form of software development, it logically follows that at least some portions of requirements engineering could be applied to the game development process.

Several articles and papers have been written on this topic, presenting problems, limitations and concerns which need to be addressed if such an attempt were to be successful.

Academic research on the topic covers a wide spectrum from purely theoretical academic works to research which focuses on the developers and their practices and concerns through questionnaires and interviews. This thesis aims to form a holistic picture of this current research, and to tie together knowledge from multiple sources to discover and present central problems and limitations.

To verify the validity of these problems and limitations, 315 developer-written post-mortems were analysed. Post-mortems are a common industry practice where developers reflect on completed software development problems and bring up problems, concerns and issues which affected the outcome of the project, either positively or negatively.

The research questions this thesis seeks to answer are therefore:

1. Based on a reading of current academic research on the topic, can central problems, concerns and issues be identified?
2. Can these findings be supported by analysing developer-written post-mortems?

The 315 post-mortems were analysed algorithmically to determine whether keywords related to the discovered problems appear in them. In addition, a word co-occurrence analysis was conducted to determine the contexts these keywords might be used in. This analysis would help to assess if the problems related to these concepts and keywords are common in the industry, as the expectation was that if game developers are frequently encountering these issues and problems, they would also mention them as contributing factors in their post-mortem writings.

Chapter 2 of this thesis gives a brief overview of the central concepts, theories and phases of the requirements engineering process. Chapter 3 presents a literature review of both traditional requirements engineering literature and current academic research on game development issues and applying requirements engineering concepts to game development, with the focus being on the latter. Chapter 4 presents four issues and problems which were perceived to be central issues. Some of them prevent the successful

application of requirements engineering concepts, while some of them are problems which could possibly be alleviated with them. Chapter 5 presents the study where 315 developer-written post-mortems were analysed to see if the problems and concepts presented in Chapter 4 are present in them. It also introduces the results of this study, and Chapter 6 discusses them in a wider context. Finally, Chapter 7 draws everything together.

2 An overview of requirements engineering concepts and processes

Before any system – be it hardware or software based, or a mixture of the two – can be designed and built, the engineering team needs to understand what it is they are building in the first place. This process of gathering understanding and processing it into an actionable plan is referred to as requirements engineering.

In actuality, what is referred to as requirements engineering is in fact a collection of different phases, processes and methodologies, which seek to take in information from a variety of sources and transform it into concrete requirements, singular and unambiguous physical or functional needs that the product or service must be able to perform. Together, these requirements form the specification of the project, essentially the blueprint the engineers can design according to and refer to when there is ambiguity [Hofman & Lehner 2001].

Additionally, a successful requirements engineering process will give the team valuable data to refer to later, making change management, tracking responsibility and attaching documentation to features easier.

The process of requirements engineering can be broken into four steps: elicitation, modelling, validation and verification, although often the last two form a single step [Hofman & Lehner 2001]. Since each step has its own aims, methods and processes, it is useful to consider them individually. It is also worth noting that although the steps do follow each other, the process is iterative rather than something the team runs through once.

It is typical that after requirements have been gathered, modelled and verified, they can generate new considerations and require further rounds of eliciting, modelling and verifying new requirements. There is no hard rule for the number of iterations that should be taken, as these details vary from project to project depending on the system that is being specified, the team that is gathering the requirements and the methods they use. [Hofman & Lehner 2001].

It is also worth noting that the relationship between requirement quality and the number of iterations is not directly linked. The specification of a system does not automatically become better the more time is spent on them [Hofman & Lehner 2001]. It is easy to get bogged down in trying to model the system at a far too granular level and to get caught up in refining for refinement's sake. The team should rely on their own expertise and prior knowledge to understand when they have reached a sufficient level of specification to begin the actual work of implementing the system [Hofman & Lehner 2001].

2.1 Top-down vs. bottom-up design

A central philosophical divide in software development is the decision between using top-down (diagram 1) and bottom-up design strategies (diagram 2).

In top-down design the system is broken down into ever smaller and more granular pieces, which ultimately end up being the individual features that need to be completed. This is the traditional software engineering model.

Top-down design places a lot of emphasis on pre-planning and trying to map out the system to a significant degree before any actual production work is done. In theory at least, this makes it easier to create a cohesive plan, to see how the individual components and their dependencies work. This makes scheduling and planning easier, but naturally means that there is a longer lead-in period before anything concrete is produced. [Faulconbridge & Ryan 2003].

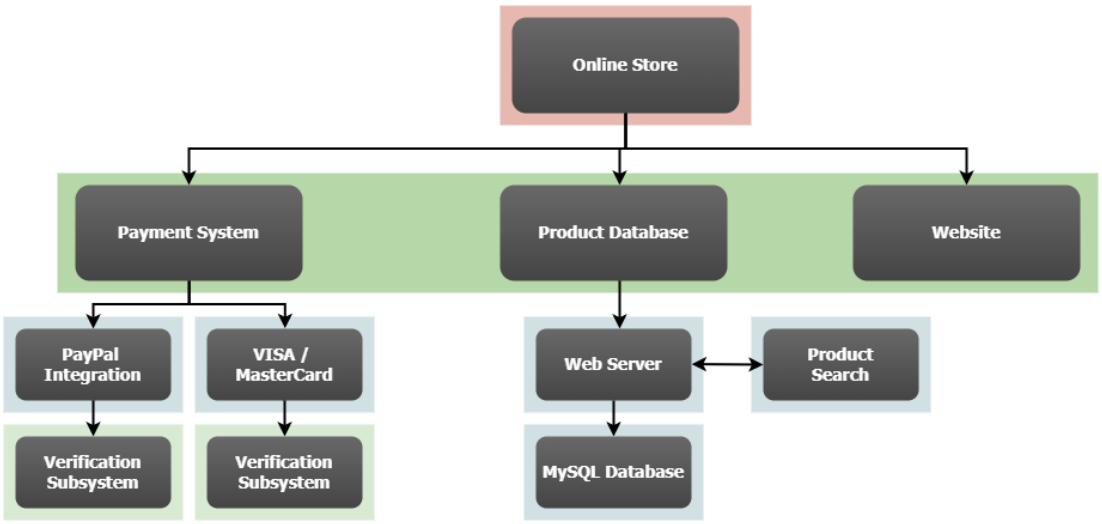

Figure 1 Depicting a partially specified system using the top-down methodology.

Figure 1 depicts a partially designed system using the top-down methodology. Although the specification is by no means complete, it already illustrates the interconnectivity and relations between different components.

As the name suggests, bottom-up design is focused on creating smaller components and trying to get them operational as quickly as possible, then combining them into larger assemblies. This method is well suited for prototyping and is popular in agile development, because the team will quickly get at least something working and can possibly get some idea of how their ideas work in practice. [Faulconbridge & Ryan 2003]

A significant drawback is that what the team built quickly might not be exactly what was needed. With less of an overall plan in place, it may turn out later that something needs to be overhauled to fit in with the larger plan. From a software production perspective, a

bottom-up design process may also create scheduling problems, when unforeseen dependencies mean that some components cannot be completed, because work they depend on has not been completed yet [Faulconbridge & Ryan 2003].

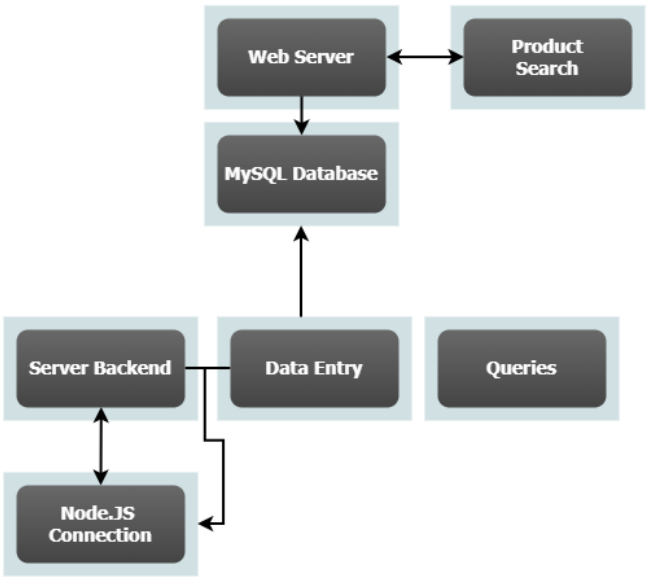

Figure 2 A system designed using bottom-up design.

Figure 2 illustrates the same system as Figure 1, but this time using the bottom-up design philosophy. It lacks the overall structure and clarity of the same system designed with the top down design, but as the design starts from the lowest possible level, individual components can already be worked on in more detail.

Neither system is "better" than the other, and there are benefits and drawbacks to using either. Depending on other philosophies being used in the project (for instance agile development methods), one or the other may be more suitable, but often features from both are mixed together.

2.2 Different types of requirements

Before delving into the different phases and methods of requirements engineering, it is important to understand the different types of requirements and how they differ from one another. Different fields and applications can have their own, increasingly granular categories of requirements. However, a more general division between functional and non-functional requirements usually suffices [Faulconbridge & Ryan 2003].

2.2.1 Functional requirements

Functional requirements describe what a system should do. They are directly related to actions, functions and responses.

- The system shall send an SMS to all registered operators when a system fault is encountered

- Users shall be able to add items to the database
- Users shall be able to delete items from the database
- System operators shall be able to change existing items in the database
- Clicking the “log out” button shall return the user to the front page of the website
- Every page on the website shall have a link to the help section

2.2.2 Non-functional requirements

Non-functional requirements relate to performance characteristics, constraints and environmental concerns. Some examples of non-functional requirements might include

- The time between clicking “log in” and the system processing the action must not exceed 15 seconds
- The system must be usable in both Finnish and Swedish
- The system must be able to function for a minimum of 8 hours in -22 degrees Celsius weather
- The access speed for hard drives must not exceed 15 milliseconds
- The system must be able to handle at least 250 concurrent users
- The system must be compliant with European Union privacy laws, specifically article §42.65
- The system must be able to be extended by connecting further server instances

2.3 Defining good requirements

Regardless of the type of requirement being modelled, the outcome of a successful requirements engineering process is a collection of so called “good requirements” [Hull *et al.*, pp. 73]. This term refers to certain traits all requirements should share in order to be acceptable. As these traits are commonly considered best practices, they should be strived for in all cases.

Additionally, the whole requirements documentation itself should also be held up to certain standards. The goal of a successful requirements engineering process is not to produce as much data as possible, but something that is easily readable, can function as a useful reference tool and ultimately help the team accomplish their goals the in the best way possible. [Hull *et al.*, pp. 73]

To this end, the document as a whole should be consistent, non-redundant and complete. That is to say, all language, terminology and structure should be consistent throughout the document, no requirements should conflict with each other, everything included in the document should be included for a reason, and it should not be missing any information. [Hull *et al.*, pp. 75]

The definition of a good requirement according to Hull *et al.* [2011, pp. 85] is a requirement that is unambiguous, testable, clear, correct, understandable, feasible, independent, atomic, necessary and abstract.

The requirement should be unambiguous

The meaning of the requirement should not be open to interpretation, and instead it should only be able to be interpreted in a single way [Hull *et al.*, pp. 85]. Common causes for ambiguity include use of undefined acronyms and poorly formatted sentences. Consider, for example, the following:

NET1: The System should make use of ABR.

Some possible definitions for ABR include "Area Border Router", "Auto Baud-Rate detection" and "Available Bitrate". While it would probably be possible to determine the exact meaning by reading the full requirement text, such ambiguity can lead to problems and is bad practice. Instead, the requirement should be written so that its full meaning is obvious at a glance.

NET1: The system should make use of ABR (Auto Baud-Rate detection).

The requirement should be testable (or verifiable)

Every requirement included in the requirement document should be verifiable in a way that allows the team to determine in a concrete pass / fail fashion whether the feature has been completed successfully [Hull *et al.*, pp. 85]. Ambiguous language and incomplete specification are common points of failure here.

UX4: The user must be able to search for books by author name, book title etc.

Since more than two categories of search terms are desired, they should all be listed out explicitly so they can be verified. As it is currently written, the requirement cannot be completely tested, because all the necessary information is not written out.

The requirement should be clear and concise

Since the requirement document is intended to be a quickly and easily readable reference manual, it should be written in clear and concise language. Avoid unnecessary verbiage and information that is not actually relevant to convey the full meaning of the requirement. [Hooks 1994]

UX8: After entering their user name and password the user can click on the login button and be allowed to enter the website, but only if

their user name and password were correct. Otherwise the system will display an error message and prompt them to enter their user name and password again.

In this case it would be hard to even understand what the actual requirement is about. Does it relate to a successful login action or an unsuccessful one? It might be better to break to break the requirement down further, and generate two new ones – one for each outcome. This way it is immediately much more readable:

UX8: After supplying their correct user name and password, the user can enter the website with the login button.

UX9: If either the user name or the password are not correct, the system shall prompt the user to re-enter them.

The requirement should be correct

It almost goes without saying that any requirement and their relevant restrictions and facts should be correct. If this is not the case, the requirement document cannot be relied upon and everything in it must be checked at every turn, rendering it unusable. Instead, the work of fact checking anything that goes in the document should be front loaded. [Zielczynski 2007]

UX9: The lock screen shall include an UI element which calls the emergency number (112) even when the phone is locked.

The emergency number listed here would be correct for Finland, but probably not elsewhere in the world. The requirement should be rewritten and handled case by case for all markets.

The requirement should be understandable

Since the requirement document is intended to be a working and useful reference material, all language used in it should be easily understandable. Overly complicated language and terminology should be avoided where possible to ensure readability and understandability. [Zielczynski 2007]

The requirement should be feasible

Since the requirement document is intended to be the blueprint by which the final system is built, anything included in it should be a realistic goal for the intended development time. It is naturally extremely difficult if not downright impossible to predict with full accuracy what the team will ultimately be able to implement, but the team should use their expertise to evaluate requirements to ensure nothing obviously unrealistic gets

included in the document. It is better to re-evaluate goals and set them to a realistic level before implementation even begins than to discover later that a core component was never a realistic target. [Zielczynski 2007]

SCOPE1: The game shall include 1000 star systems each with at least 100 procedurally generated planets.

The requirement should be independent

Each requirement should be self-contained. While features will inevitably link together and have dependencies, the requirement texts themselves should not [Zielczynski 2007].

UX12: The lock screen shall display the current time and date in the user's time zone.

UX13: The font shall be user-defined.

While these requirements might be readable at the moment, despite being obviously poor, the document might be reorganized at some later point. New requirements might be introduced and suddenly it will become very difficult to decipher precisely which font is being referred to in requirement UX13.

UX13: The user should be able to define the font used for lock-screen time and date display.

The requirement should be atomic

Every requirement should be a single traceable element [Zielczynski 2007]. Consider the following:

UX3: The main UI should display the calendar, have a list of apps the user can open by clicking on them, have a menu button and a link to the settings app.

This requirement should really be broken down into at least four smaller requirements, one for each of the desired main UI elements. In general, words such as "and" or "but" are good indicators that the requirement could and should be broken down further into smaller components.

The requirement should be necessary

Anything included in the requirement document should be in it for a reason. For example, just because the team thinks the customers might want a certain feature included is not reason enough to include it. Everything in the document should be included because at

least some of the stakeholders need the requirement, and removing it would affect the system. [Zielczynski 2007]

The requirement should be abstract

Requirements should not include design and implementation information.

REQ9: User settings should be stored on the hard drive in an XML file.

When possible, it should be left up to the developers to decide practical issues such as the format of the user settings, since it is transparent to the user.

2.4 The phases of requirements engineering

At the start of the requirements engineering process, the team will only have a very barebones idea of what they are building, commonly possibly only a very high level and informal mission statement which serves as their entry point. This mission statement can describe in rough detail the problems to be solved, or the context in which the system will function. [Zowghi & Coulin 2005].

From this starting point the team needs to understand the system and the stakeholders that are attached to it, as well as the constraints and external factors which will affect the system and their work. As this understanding is gained, the system can be modelled with requirements, which are then verified to be correct and relevant.

The actual requirements engineering process, as illustrated in Figure 3, is often iterative, or at least it should be. It is highly unlikely that the team will manage to produce a functional requirements document with just one iteration of process. What is more likely is that after each iteration the team can understand the system better, and can return to elicit more information, possibly from new stakeholders, and thus produce further requirements, as illustrated below [Zowghi & Coulin 2005].

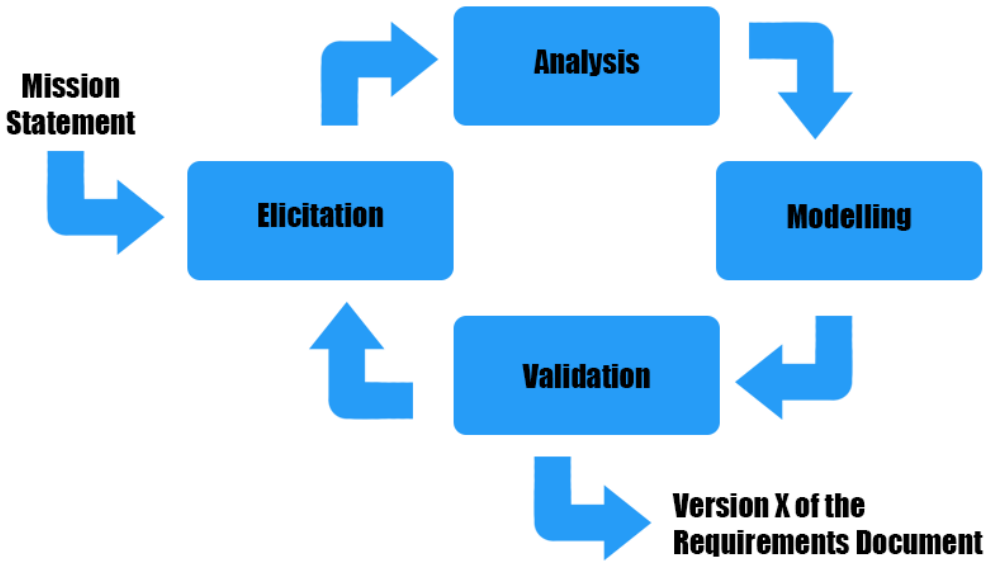

Figure 3 The iterative nature of the requirements engineering process

The end product of this process will the requirements document, which will then serve as the blueprint and a general reference manual for the system that is to be built.

2.4.1 Eliciting requirements

The focus of the elicitation phase is information gathering. Before the system that is being worked on can be modelled as requirements, the team needs to understand what it is they are designing.

According to Zowghi and Coulin [2005], there are five fundamental activities to be conducted in this phase of the requirements engineering process:

1. Understanding the application domain
2. Identifying the sources of requirements
3. Analysing the stakeholders
4. Selecting the techniques, approaches and tools to use
5. Eliciting the requirements from stakeholders and other sources

Understanding the application domain

This activity certainly includes understanding the actual system you are building, but goes beyond that. No system exists in a vacuum, so it is important to consider the real-world situation the system will function in. In addition to describing business goals and issues, possibly already existing solutions (or partial solutions) need to be considered as well as the wider surroundings: political, organizational, social and environmental factors need to be considered as well, to name a few [Zhang 2007, Zowghi & Coulin 2005].

Identifying the sources of requirements

End users and clients are by no means the only source of requirements, because information can be acquired from other sources as well. Especially if the system being designed is intended to replace or supplement an existing solution, its documentation and processes can be valuable sources of information. It can also be worthwhile to examine the wider organizational structures and documentations of the client and the environment in which the system is being developed, and into which it will be deployed. [Zowghi & Coulin 2005].

Additionally, domain experts in areas directly and closely related to the system can produce useful requirements and help in refining them, even if they will not be direct stakeholders in the actual project.

Analysing the stakeholders

It would be easy to understand the concept of "stakeholders" to mean the end users of the project, and the team's customers, but this definition is too narrow.

There is some debate on what encompasses the term. In their 1999 paper, Sharp *et al.* propose a guideline for a more thorough acquisition of different stakeholder groups. They propose four baseline stakeholder groups: users, developers, legislators and decision-makers. Additionally, their method calls for identifying specific roles within each baseline group, then forming relations between those roles and further stakeholders who either supply something to those roles, or benefit from the work those roles do.

Without delving too deep into this vast topic, we can already see that the actual list of stakeholders is quite large. In fact, one of the dangers involved in this process is getting too bogged down in creating an ever-increasing list of sources to be considered [Sharp *et al.* 1999], thus derailing the actual intended role of the elicitation phase: understanding the system that is being developed.

Selecting the techniques, approaches and tools to use

A wide variety of different techniques, tools and approaches exist for requirements engineering. There are no "magic bullets", or in other words single solutions that apply to all situations. Rather, techniques and methodologies should be considered on a case by case basis.

Hickey and Davis [2003] have explored the various factors that contribute to choosing the techniques that will be utilized, and an objective weighing of merits is not the leading factor. Rather, their studies have shown that the chosen technique may be the only one

the analyst knows or their favourite technique, or is recommended in a specific methodology that is being followed in the requirements engineering process.

These methods of choosing are not necessarily incorrect, because familiarity and comfortability with techniques can help produce better results. More important than trying to chase a platonic ideal is remembering to use a variety of techniques, depending on the source of information and the stage of the project. [Zowghi & Coulin 2005, Zhang 2007].

Commonly used techniques include, but are not limited to [Gunda 2008]:

- interviews
- questionnaires
- prototyping
- reusing existing techniques for inspiration
- scenarios
- brainstorming
- use cases
- and user stories.

Eliciting the requirements from stakeholders and other sources

Once the team feels they have a good understanding of the stakeholders that are relevant to their project and have decided on some techniques that would be appropriate for their situation, the actual elicitation process can begin.

Using the agreed upon methods, the team gathers information from their stakeholders and other sources, trying to understand the system they will be designing. What features must it have, should it have and could it have? What shouldn't it have? What constraints will apply to the system and the development process itself? [Zhang 2007, Zowghi & Coulin 2005]

This information by itself is not yet a finished requirement, but it will function as the starting point for the actual process of modelling the requirements.

2.4.2 Analysing and modelling requirements

The analysis and modelling process can be the most technical of the three main processes of requirements engineering. During this process, a lot of information that possibly only exists as natural text or in several different formats is analysed and specified.

During the modelling process the team can make use of a variety of different techniques, depending on their preferences and the demands of the situation. This thesis cannot cover all of them, but will instead aim to give an overview of central concepts and techniques.

These techniques can be roughly divided into four categories, as described by Sannier [2014] in *Modeling Requirements: Requirements Verification and Validation*, from the least formal to the most formal.

Natural language

While natural language is not a formal technique, it can certainly be useful as part of the analysis and modelling process as either a dedicated tool, or more commonly as part of converting original user stories, use cases and other elicited information into more formally specified requirements. Natural language can be used to specify requirements to a desired level of detail [Sannier 2011].

The obvious benefit of using natural language to describe requirements is that it requires no special training and can therefore be accessed and read by almost anyone. However, because there is no universally agreed upon structure or notation for describing and specifying features, natural language is also often ambiguous and vague. [Sannier 2011]

Ad hoc notation

Ad hoc notation, as seen in Figure 4, is also commonly used as part of the analysis and modelling process, often to support more formalized methods or to help translate natural language and ambiguous concepts into more formal structures. The term simply refers to the process of modelling a system on a whiteboard or piece of paper using bubbles, squares, arrows and other simple elements.

Much like natural language, ad hoc notation requires no special training and can be easily used by anyone. Again, much like natural language, from this it also follows that since there is no formal syntax, ad hoc notation can be vague. Notation can vary from person to person and from one use to another, making ad hoc notation a poor fit for any kind of official or formal models. [Sannier 2011]

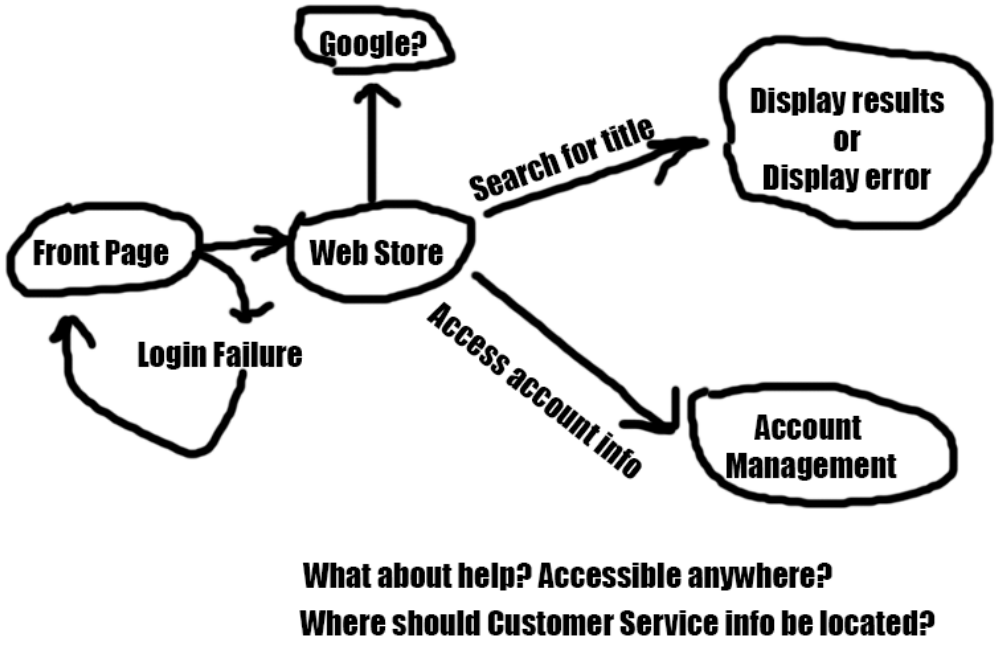

Figure 4 An incomplete and informal ad hoc diagram describing the structure of an online store.

Semi-formal notation

Semi-formal notation is a step up from ad hoc notation. Using an agreed upon and defined notation language such as UML, complex systems can be represented as diagrams. These languages offer well defined and universally agreed upon notation systems and can therefore be used to consistently model and represent systems, and the processes of creating and modifying them can be automated to a degree [Sannier 2011, pp. 18-30].

Semi-formal notation requires some learning, but since the technique is universally used, resources and tools, are widely (and freely) available. The sample image in Figure 5 was created with the online tool LucidChart [LucidChart 2017]. Once learned, these tools and techniques allow for the easy and quick creation of models, which makes languages such as UML very popular and useful in many modelling tasks [Sannier 2011, pp. 18-30].

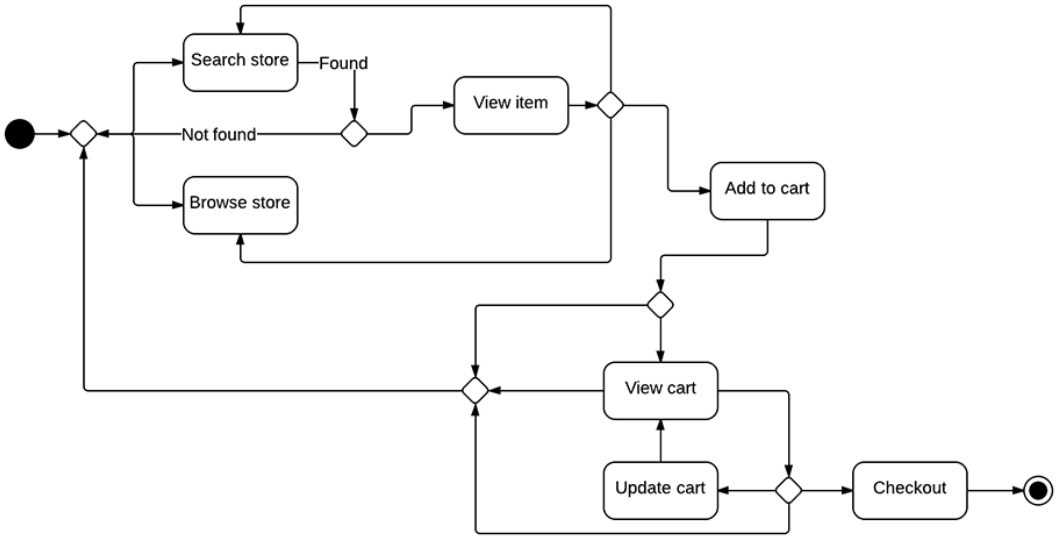

Figure 5 A use case translated into an UML Activity Diagram, which allows designers to follow the logic of a customer trying to find and buy a book at an online store. [LucidChart 2017]

Formal specification and notation

The concept of formal specification and notation covers a wide spectrum of different tools, languages and methods designed and formulated for the specific purpose of specifying and notating complex systems.

Some examples of formal specification and notation methods include complex languages, logical systems and finite state machines. As we can see from their descriptions, these systems must follow formal notation correctly in order to work in the first place, which makes them useful not only for modelling requirements, but also for verifying them. [Sannier 2011, pp. 70]

Complex languages

Complex languages such as Specification and Description Language, or SDL, have been created for the specific purpose of formally representing complex, interactive real-time applications – such as software systems [IEC 2010]. The SDL specification is not the only example of formal languages, but it is widely used and actively maintained by a dedicated specification committee.

It allows for the representation of modern systems in either graphical or textual form, with the intent being that if proper notation is followed correctly, both versions will describe the system identically and can be used interchangeably.

Logic systems

Logic systems have been traditionally used in many computer science applications and notations, because at its essence all computational actions distil down to logical components. While traditional mathematical logic is useful in requirements engineering and in other areas of computer science, expanded systems also exist that have been tailored for the requirements of computer software.

For example, Computation Tree Logic, or CTL, is a system of branching-time logic, which is used to model systems and situations in action [Fourman 2005]. This is in contrast with traditional logic which exists as static statements, the outcomes of which are already known. CTL offers different paths to different futures, any one of which might become the actual outcome.

It is used to model and verify system behaviour and check liveness and safety conditions.

Finite State Machines

Finite State Machines are in a way an extension of complex languages and logic systems. Using predefined notation and language, programmers can create simple representations of complex systems, which can exist in various states. Execution of this code allows the state of the machine to be changed, mimicking user actions, input and output actions and other operations.

A wide variety of dedicated languages and tools (for example, LTSA, or the Labelled Transition System Analyzer [Magee & Kramer 1999]) exist for creating these machines and automating large numbers of operations. This allows designers to test the architecture of the system at a very early phase of the project, ensuring everything works as planned and catching possible design problems, bottlenecks and dependencies early on.

Software tools for requirements modelling and storage

A wide variety of software exists for requirements engineering. Some of it may have originally been crated for other purposes, and some was designed for the specific purpose of creating and storing requirements. These tools offer an object-oriented approach to requirements, allowing users to create discrete objects for individual requirements, cross reference between them, refer to relevant documentation – for instance Wiki pages – and maintain a persistent path of change requests and changes. Sample views from two popular options, Open Source Requirements Management Tool [OSRMT 2017], and JIRA [JIRA 2017], are included as Figures 6 and 7 below.

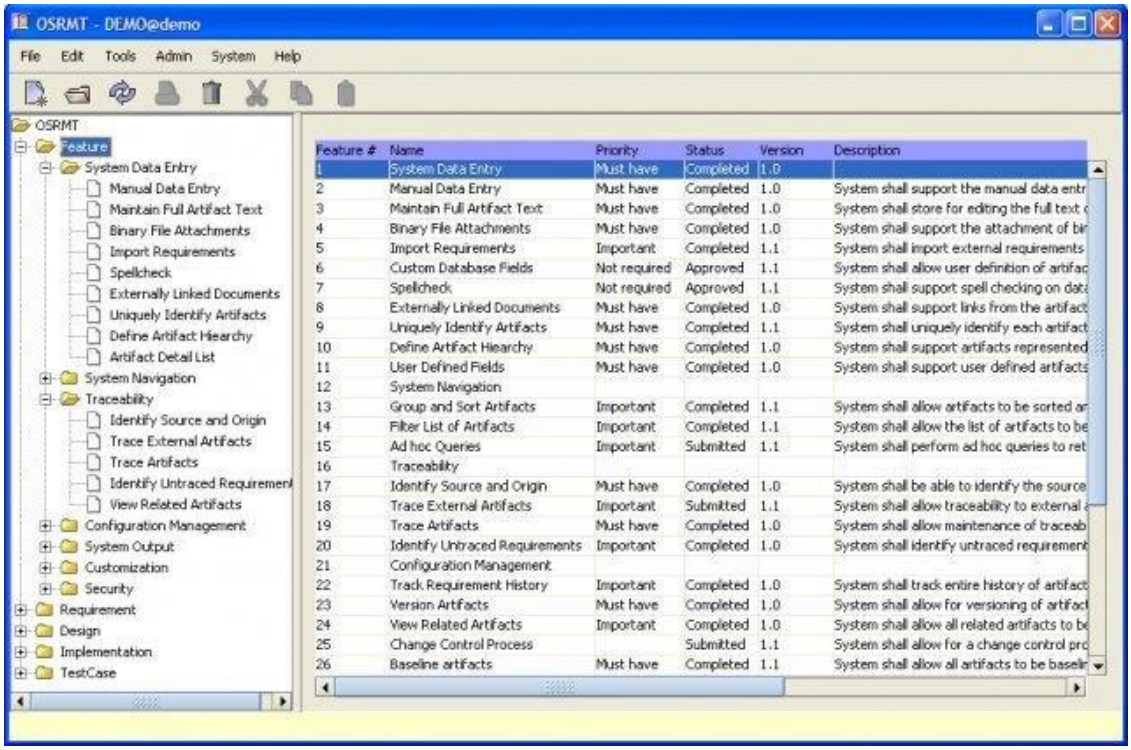

Figure 6 An example view of the free Open Source Requirements Management Tool displaying a list of requirements.

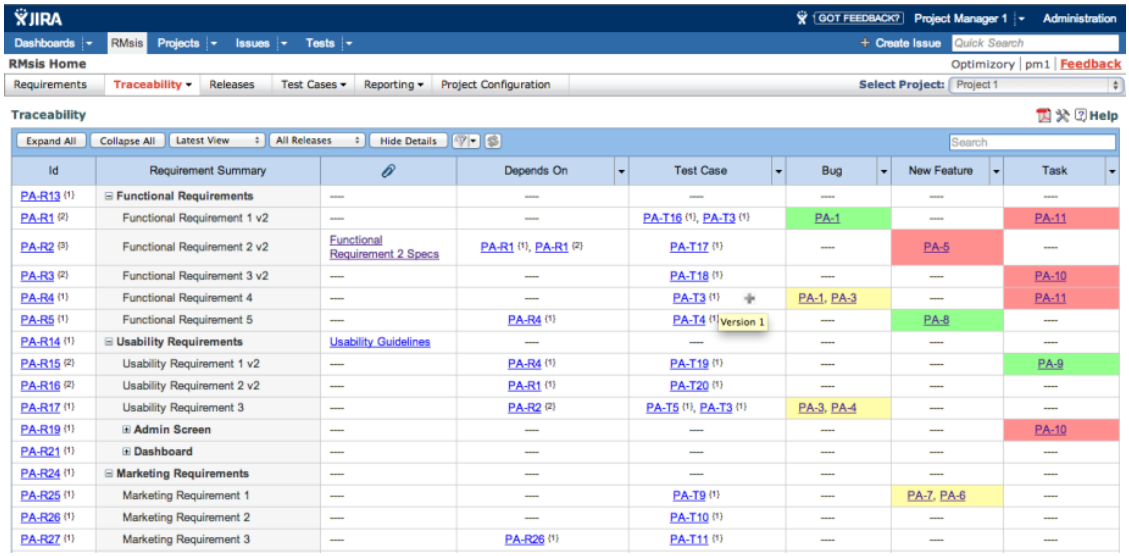

Figure 7 JIRA, a popular issue and project tracking software, can also be extended to support requirements engineering specific features. In this view we see a Traceability matrix, which allows for the easy monitoring of requirements and their statuses and connectivity.

2.4.3 Validating and verifying requirements

Although validation and verification are listed as their own step in the requirements engineering process, they are something that should be done throughout the process. Leaving all validation at the end of an iteration will make it harder to correct errors that could have been caught earlier in the process.

During the elicitation step, it can be a good idea to check and re-check information with sources, to ask clarifying questions, make sure the stakeholders and their needs were

correctly understood and written down during interviews and other more free-form elicitation methods and so on.

During the modelling step, it is a good idea to maintain quality control on the verifications themselves. It may seem redundant, because requirements will be verified during the validation and verification step anyway, the earlier a mistake is caught the easier and quicker it will be to fix [Sannier 2011]. As an example, if someone on the team has not understood notation correctly or is not following style guidelines, it will be simple to correct the problem with some tutoring and instruction during the modelling process, after which they will hopefully produce better work. If the mistake is caught only during verification, a lot of work will have to be redone.

The validation and verification step should focus on answering two questions: has the team written good requirements and is the system actually conforming to the requirements written. For validating requirements, the team should essentially consult the list of characteristics a good requirement has, and go through their list of generated requirements, making sure everything they have generated meets these criteria. The objective of this task is to ensure that every requirement achieves the objectives set out in the initial mission statement, meets the needs of relevant stakeholders and is clearly understood by the developers. [Sannier 2011].

Requirement verification is a process that should be ongoing and repeated as the actual design and development work is undertaken. As with other types of errors, the sooner they are caught, the easier and cheaper they are to fix.

3 Game development from a software development perspective

In the year 2017 the video games industry was bigger than ever. According to market analyst company Newzoo, there are over 2.2 billion video game players across the world. The games industry is expected to generate over 108 billion dollars in revenue, representing a growth of 7.8% of 2016 [Newzoo, 2017].

This growth industry contains innumerable game development studios ranging from lone developers to small companies and large multinational corporations. According to the Entertainment Software Association's 2017 report on the American video game industry, in 2016 there were over 2450 active game companies in the United States along. 99.7% of them qualify as small businesses, meaning they have under 250 employees and less than $7.5 million in annual revenue [ESA, 2017]. Similar numbers have been reported elsewhere in the world, as according to UKIE, there were 2175 active game companies in the United Kingdom alone [UKIE, 2017].

These thousands of game developers are working on varied games ranging from huge titles with budgets in the hundreds of millions to eSports titles, mobile games, small independent projects and everything between.

These factors represent a unique challenge from a software development perspective. Contrary to the more disciplined and theory driven world of traditional software development, games development is a more fractured landscape. Whereas large corporations such as Electronic Arts or Activision, or even larger independent developers, might adhere to traditional software development roles and practices – agile methodologies and Scrum being particularly popular in game development – for smaller independent studios development is probably less regimented and more free form [Koutonen & Leppänen 2013].

Games as a form of software development also have several other unique characteristics. As an example, whereas traditional software engineering teams consist of software developers of various disciplines, a game development project will usually have the normal complement of software developers, but additionally artists, writers and other purely creative people. These disciplines do not often share a vocabulary and might differ widely in their needs, methods and work flows. Yet all these disciplines need to find common ground if the project is to succeed.

Additionally, these very multi-disciplinary teams seek to create software which philosophically differs greatly from traditional software. Traditional software development projects aim to create solutions to discrete problems, whereas games are mass-marketed products aimed to entertain and prompt emotional responses [Kasurinen *et al.*, 2014].

From this it follows that the models and theories which drive traditional software development projects might not be directly and fully applicable to game development. This is also true for requirements engineering.

3.1 Literature in this study

While requirements engineering in traditional software development is a heavily covered field with academic publications, books, magazines and even conferences dedicated to the subject, this is not the case for requirements engineering as part of the game development process.

In order to map the current state of academic writing on requirements engineering and game development, academic search engines such as Google Scholar and the University of Tampere's Tamcat[1] search were utilized. The latter allowed access to various digital libraries, which further widened the field of possible results.

The goal of these searches was to discover peer reviewed articles, academic publications, conference proceedings and published books which dealt with games development and requirements engineering. No specific time constraints were placed on the results. Software development is a field which moves fast, which means that some of the older findings might be outdated. Despite this, they might be used to reveal newer research which builds on their findings or expands upon it. Searches were conducted using a wide array of different search terms in combinations and with wildcards, as shown in Table 1.

"game development" OR "games development" AND ("requirements" or "requirements engineering")
"game development" OR "games development" AND ("software engineering" OR "engineering")
"game development" OR "games development" AND ("R.E")
"game development" OR "games development" AND (“specification” OR “formal specification”)

Table 1 Some of the search terms used to find articles

Together, these searches produced a pool of over 16 000 results. As is to be expected with such broad search terms, most of these results were either marginally related to the actual research question, or not at all related. Even after discarding most of the less applicable results, the pool still contained several hundred articles which might be tangentially related to the research questions. Any articles with titles or abstracts which seemed promising were stored in a separate list to be more carefully examined later.

As these promising articles were read, references that seemed relevant or interesting were noted down and added to the list of articles to be read. The list was also somewhat

[1] https://tamcat.finna.fi/ (retrieved on 21.5.2018)

prioritized based on these findings: articles which were cited by others were given priority.

This strategy proved to be fruitful, and helped avoid a critical problem. A pure keyword search quite probably would have either missed certain central works, or at the very least they might not have been given the weight and consideration they deserved.

Based on this study, the state of research on the topic proved to be rather healthy, if not comprehensive. There is certainly a larger volume of research than anticipated, and requirements engineering proved to be a central topic: as of 2014, 39% of papers submitted on the topic dealt with requirements engineering in some way [Ampatzoglou 2010]. This does not mean there are no gaps to be found in current research.

One notable problem area is that many articles bring up problems in processes and methods, but rarely offer any concrete suggestions beyond vague calls to adapt best practices from the world of traditional software engineering. Due to the central differences between traditional software development and games development, this adaptation would have to be handled with care and consideration, so academic research on the topic would be beneficial.

3.1.1 Traditional requirements engineering

Traditional requirements engineering is also a part of this thesis, as a portion of it is dedicated to explaining the methods and processes, as well as discussing best practices and goals. A central source for this portion of the thesis was Klaus Pohl's *Requirements engineering: Fundamentals, principles and techniques* [2010]. It is still viewed as one of the defining works in the field and is referred to in many articles as well as widely used as a source of information in the industry and non-academic fields.

Additional noteworthy sources for this portion of the thesis include *Requirements Elicitation: A Survey of Techniques, Approaches and Tools* [Zowghi 2005] which, as the name suggests, deals with a wide survey conducted among industry professionals to map which methods and tools they feel are most applicable to each portion of the requirements engineering process. There are almost as many tools and techniques as there are practitioners, so selecting the most fitting one for each task is challenging. Research has shown that many professionals have a habit of falling back on their favourite methods – i.e. methods which they have had success in the past and which they feel familiar with – regardless of how suited those methods are for the demands of their current task, so considering other alternatives is always useful.

Likewise, stakeholder identification is a question that would almost be worth a thesis all on its own. In requirements engineering "stakeholder" does not refer to just the end-user of the product being developed, nor the client. The term covers a vast field of different

possibilities, many of whom should be considered and investigated for a successful requirements engineering process. *Stakeholder Identification in the Requirements Engineering Process* [Sharp *et al.* 1999] was a good refresher on the subject, and offered a lot of practical suggestions on efficient techniques on mapping stakeholders and the dependencies between them.

3.1.2 Game Development literature

There does seem to be a gap in the current academic research on requirements engineering and the software development process as it relates to game development. Namely, while there is a lot of research on the topic, the quality differs greatly.

Some large studies, such as those conducted by Jussi Kasurinen and his team at the University of Lappeenranta, have been conducted on actual industry practices among various groups of developers, and the findings of these studies were extremely useful and enlightening [Kasurinen 2016, Kasurinen & Laine 2014, Kasurinen *et al.* 2014]. They highlight actual issues developers grapple with, as well as the methods and practices used to deal with these issues. On the other hand, some articles and papers, such as Kanode and Haddad's *Software engineering challenges in game development* [2009], present conflicting claims, often with very little to support their contrary claims.

A possible conclusion would be that these articles and papers may be too grounded in academic theory and operate too much in an ideal world, and thus may not be directly applicable in real world situations.

Games as a form of software development

Game development is clearly a part of the larger field of software development, which has been dealing with problems related to the processes, methods and demands of the software development process for decades. It stands to reason, then, that many of these same processes, methods and demands might also apply to game development. However, due to several factors this is not always so easy.

For one, game development is a much more ad-hoc and informal field than traditional software development. Methods, processes and practices which were formalized and honed if not to perfection, then at least to comfortable routine, decades ago in traditional software development might be rarely utilized in game development.

The very act of developing games also differs from traditional software development in many ways. To name a few examples, whereas traditional software development aims to create a solution to a specific problem, game development aims to create entertainment products which are enjoyed for prolonged periods of time by a variety of people. Additionally, whereas traditional software development is very much the world of

engineers and developers, game development projects employ people from a wide variety of artistic fields, which may not be easily compatible with software engineering or even share a common language, as it were. [Kasurinen 2016, Callele *et al.* 2005].

It is therefore natural, that the questions of "to which degree is game development similar to traditional software engineering" and "which methods and processes could carry over to game development, and what could not" are so frequent in academia. It is also good that a lot of the research that is being carried out is being carried out in collaboration with actual game developers.

Not always, though. The article by Kanode and Haddad [2009] is a good example of a paper which seems to go against many of the findings of papers based on actual industry experiences, such as Callele *et al.*, [2005].

The views presented in Kanode and Haddad's [2009] article on the game development document seem to be contrary to evidence presented in other papers. In game development, the game design document is a repository of information about the game. It details the setting, plot, gameplay, characters and themes of the game. It is not a formal document, and as such is poorly suited for actual software development.

One of the problems explored by Callele *et al.* [2005] is that translating this informal document into something resembling a requirements document is a massive and complicated process. Even a short, simple gameplay description in the game design document can generate dozens of pages of requirements, and even more problematically generating those requirements requires unrealistically strong and specialized domain knowledge in many fields.

Based on strong research, studying real-life game design documentation, discussions with actual game developers and observing actual development processes Callele *et al.* [2005] conclude this transition from pre-production to production, i.e. taking informal and often very casual documentation, turning it into a formal document suitable for development, and then beginning to realize the vision outlined in that document, is one of the biggest problems in game development and alone responsible for many project failures. They state that a lot of research and work needs to be done to create more formal and functional processes and methods for generating this documentation, so it is very much a central and unresolved problem in the field.

Considering all this, Kanode and Haddad's [2009] suggestion that a game designer needs to capture all the requirements from a game design document before the actual production work on the game can begin, seems optimistic to say the least.

Some notable research is being carried out at the Lappeenranta University of Technology by Jussi Kasurinen and his team [Kasurinen *et al.* 2014, Kasurinen & Laine 2014, Kasurinen 2016]. They have conducted large scale surveys among Finnish industry professionals, and backed these findings up with several rounds of interviews conducted among project managers, developers, managers and designers. This research has generated several publications.

Kasurinen [2016] explores the very central question of how similar – or different – to traditional software development projects game development is. It was found that there are several similarities, but also meaningful differences which mean that traditional software development methods and lines of thinking will not apply directly.

For example, whereas change to the original specification is something needs to be very carefully managed in traditional software development, in game development changes through iteration are a desired outcome. As the developers try to "find the fun", i.e. create the combination of gameplay and features which makes the game fun, they must be prepared to make even drastic changes late in the project.

Based on these findings, Kasurinen [2016] concludes that while some common traditional software development methods such as Scrum can very easily work with game development, others are not so easily compatible and need special consideration.

Problems in Game Development Methods

Whereas traditional software development and its issues are a topic of some 40+ years of discussion, the same is not true for games. It is generally accepted that there are similarities and unique factors between the two, but concrete solutions and suggestions are few and far between. In recent years more studies have been conducted as to what actual problems game developers are facing, which is a crucial area of research [Kasurinen *et al* 2014, Kasurinen & Laine 2014, Kasurinen 2016, Petrillo *et al* 2008 etc.].

Even if this research does not directly lead to solutions, we must first discover what the issues are. Additionally, understanding these real-life problems and the realities of game development is the ground upon which all meaningful research must be built. What use are solutions which were created in a kind of academic bubble, only to be dismissed out of hand by actual developers because they are not applicable?

Since the development methods and processes employed by especially larger publishers and developers are considered actual trade secrets, they are not something developers can always discuss openly with researchers. This means sometimes the researchers must do a bit of detective work to get results. Petrillo and Pimenta [2008] explored post-mortems published on Gamasutra, an industry-focused website by developers for developers.

Post-mortems are a common practice in the world of software development and are intended to offer candid and honest discussions on what went right and what went wrong on a given project. By parsing these post-mortems, we can learn a great deal about the actual problems faced by real developers.

Petrillo and Pimenta [2008] grounded their research by citing studies which catalogue traditional software development problems, and then compare these with their own findings. The results are not surprising: game development does indeed share many of the same problems as traditional software projects. Some key issues include project management, scope creep (the bad habit of adding new features to a project during production, thus generating new problems and lot of work) and scheduling.

Rather dishearteningly despite 40+ years of discussion and research, these issues still plague traditional software development, so the hopes of game developers solving them any time soon, especially when keeping in mind the peculiarities of game development, are not very strong.

One of the central problems in game development is the composition of teams. As discussed earlier, a game development team is composed of engineers, managers and other technical personnel as well as artists of various types. Even finding common ground and a common language to effectively communicate between these teams can be extremely challenging [Callele *et al.* 2005], but an even bigger problem is scheduling.

Game development is a vastly complicated process where dependencies between drastically different and often seemingly fundamentally incompatible workflows and processes must be created, maintained and accomplished. As games development moves from traditional waterfall type development models towards various agile philosophies, these problems are compounded even further. The act of generating artistic assets and preparing them for inclusion in a game is a slow and complicated process, which is by nature incompatible with the sprint-driven iterative world of Scrum.

Musil *et al.* [2010] propose a unified development model consisting of essentially three phases: pre-production, production and closure. This model bears a strong resemblance to Scrum, upon which it is built.

The ideas presented in the paper are interesting, especially the suggestion to make the pre-production phase longer than it often is now, and to have it be a period of strong iteration and rapid prototyping, aiming to not only produce a rough game concept but to also iterate upon it and generate a semi-formal design document based on this. Of note is also the authors' suggestion to more formally include testing in every phase of production, and to implement the receiving and processing of testing feedback at all levels.

Despite this, the article has some omissions. The main suggestion presented in the article for fixing the incompatibility between different types of sub-teams in the development team is to essentially run separate and independent homogenous teams of artists, writers and developers, and to sync their work between sprints. This seems impractical at best, and Clinton Keith [2010] seemed to have the better idea in suggesting that because asset generation is by nature unfit for iterative work, it should be separated into its own pipeline, which could use other agile methods such as Kanban to function.

Agile Methods in Games

Game development is becoming increasingly agile. Historically games have been developed with some variations of the traditional waterfall model, i.e. a structured and phased development process, where the whole project moves from one stage to the next. This has largely been replaced by various agile methods [Leppänen *et al.* 2013].

For instance, a study was conducted in Finland among game developers [Leppänen *et al.* 2013] to see how they applied agile methods and practices. These findings, which proved to be quite typical, indicate that the clear majority of developers utilize agile methods, typically Scrum. Scrum is the most popular agile methodology in traditional software development, so this is not surprising. Additionally, some practices and methods such as extreme programming (or XP for short] and Kanban are employed partially.

However, according to Leppänen *et al.* – and this view is backed by for instance Petrillo and Pimenta's similar study [Petrillo & Pimenta 2010] – most teams actually practice some type of "adapted Scrum", i.e. they have cherry picked some of the ceremonies, roles and methods of Scrum and then apply them to an ad-hoc agile development process. According to the results of the survey, teams self-reported for instance only following one process from what was intended to be a pairing or supporting practices and methods.

These findings are not surprising as such. Project management in general is a problem in game development [Callele *et al.* 2005] especially in smaller companies, because there simply are not that many trained and experienced project managers available. Instead, these positions are filled by people who were assigned or promoted to these roles for other reasons and therefore may lack the understanding and experience to best make these decisions.

Related to this, Stacey and Nandhakumar [2008] conducted a survey among three well known game development studios. The researchers hypothesized that many studios might be agile in spirit even if they might not be intentionally observing all the ceremonies of, say, Scrum. They had open ended discussions during nearly 40 interview sessions regarding the work practices of developers. Based on their discussions and findings they make four concrete recommendations for fostering a more agile atmosphere at a company.

Their findings are quite interesting from a requirements engineering perspective, as their findings deal with creating an atmosphere where soliciting and receiving honest suggestions and feedback is as easy as possible. For example, they prompt developers to seek and accept feedback from non-traditional sources, citing an example of a studio having company-wide milestone testing sessions, during which the company secretaries ended up making suggestions which ended up improving the quality of the game.

Requirements engineering and games

The place and role of game development in the hierarchy of software development is an interesting topic of discussion, as is that of the role of requirements engineering in game development. Kasurinen *et al.* [2014] explored the actual practices employed by several Finnish developers and companies. Based on the interviews they discovered that most Finnish developers seem to use iterative, agile methods to create games, whereas a few holdouts still use methods which bear a strong resemblance to older waterfall models. Interestingly the authors found that despite their actual work practices, all interviewed developers self-reported using agile methods and iterative practices. The authors created two rough schools of thought based on the actual methods and processes used by Finnish developers, and then considered the role requirements engineering plays in each.

They discovered that the specifics vary quite a lot. In traditional waterfall-type development (i.e. where development moves from one phase of the project to the next in a linear fashion) requirements engineering is closer to that found in traditional software development. Emphasis is placed on creating specifications at the beginning of the project, and then to sticking to these specifications. Of course, this is not always realistic, as all game development is iterative and has more change than traditional software development. As Callele *et al.* [2005] also illustrated, creating this document is no easy feat. This problem is heightened in iterative development, where change is an outright goal.

The authors discovered that the requirements engineering processes used by Finnish companies are in general very informal and relaxed, and proposed that game developers could probably benefit from a more formalized and structured requirements engineering process, but that the current models and methods used in traditional software development are not directly applicable, and therefore these formal processes and structures need to be created before they can be used.

One interesting common theme in the discussion of requirements engineering and games is the unique nature of requirements in games. Traditional software development places a heavy emphasis on functional requirements (i.e. concrete features in a project), whereas in game development these are almost standardized among games of the same genre.

Instead, the differences between games come largely from non-functional requirements, which play a heavier role. Of special interest are so called affective, or emotional requirements [Callele *et al.* 2005].

Games are intended to prompt emotional responses in their players, and these should also be modelled through requirements engineering. However, the tools and techniques to do so are still in their infancy and a discussion is ongoing as to how best accomplish these goals.

The role of non-functional requirements is an interesting and open field of research, but a few studies have been conducted. Notably Paschali and Ampatzoglou [2014] conducted a study into the role non-functional requirements play in the player experience. They explored various game genres and what types of non-functional requirements players of that genre found important. The results are not hugely surprising, as for instance players of role-playing games or adventure games place more emphasis on characters and story coherency than those of sports games. Further research would be called for, as the study conducted for this article seems a bit too superficial and generalized.

4 Key differences and problems

Game development and traditional software development methods and tools, for instance requirements engineering, are not fundamentally incompatible. There is evidence that game developers make use of these methods, and get benefits from them [Kasurinen *et al.*, 2014].

That being said, there do seem to exist some fundamental differences and problems, which make adapting these traditional processes and methods to game development difficult. Game developers do seem to suffer from many problems which could be alleviated or eliminated through better requirements engineering processes and methods. For example, in post-mortems published on Gamasutra.com, game developers cite factors such as "inadequate planning", "underestimating the scope of tasks" and a schedule that was "too aggressive" [Callele *et al.* 2005] as aspects of the project which went wrong and hindered them.

It is worth noting that these findings are not universal. Game development is a wildly varied field, with studios ranging from one-person teams to massive international companies. Many developers, especially larger companies, tend to regard their methods and practices as trade secrets and are not open to discussing them with journalists or academics. Despite this, from merely reading recruitment posts and requirements for open positions, it is clear that at least larger companies do value degrees and formal training when seeking to hire developers.

It is also worth noting that these issues are heavily linked and could also be thought of as different aspects of the same problem. After all, any differentiation between "a lack of formal processes and methods" and "poor change control" is going to be somewhat arbitrary, as the latter could easily be considered a part of the former.

What, then, could be some of these key differences that need to be considered, and key problems that need to be overcome?

4.1 Emphasis on non-functional requirements and affective requirements

In traditional software engineering, the emphasis is very much on functional requirements. They describe the key features of the system to be implemented, and are what ultimately distinguishes it from its competition and allows it to fulfil its stated and desired goals.

In game development, non-functional requirements are considered much more important. In what is called "horizontal differentiation", it is claimed that the functional requirements for games of a particular genre of game are often quite similar to begin with, and non-functional requirements make the difference and help distinguish a game from its peers [Paschali *et al.* 2014, Callele *et al.* 2005].

Additionally, unlike in traditional software engineering, more and more game developers are using pre-made game engines such as Unity[2], Unreal Engine[3] or CryEngine[4], which further removes emphasis from functional requirements, as these requirements are already fulfilled by the pre-packaged engine. [Kasurinen *et al.* 2014].

This in and of itself might not be a problem, as tools for capturing and modelling non-functional requirements have existed for decades. In game development, however, non-functional requirements deal with more difficult concepts. In traditional software engineering, requirements generally refer to concrete and measurable real-world conditions, whereas game-domain specific requirements are more abstract and harder, if not impossible, to measure [Kasurinen & Laine 2016]. Requirements related to concepts such as fun, storytelling, aesthetics and so on are key concepts in video game projects, but of course not at all relevant in traditional software engineering [Callele *et al.*, 2005].

These requirements also vary from genre to genre [Paschali *et al.*, 2014]. What is important in a racing game might not be at all relevant in a puzzle game, or an adventure game.

Unlike traditional software projects, games are intended to produce emotional responses in their users. Requirements relating to these emotions are referred to as emotional, or affective, requirements and they are viewed as a key component in creating an engaging gaming experience [Callele *et al.*, 2005].

Sadly, the tools and techniques for capturing and modelling these requirements either do not exist, or are not as developed as they should be. Additionally, validating these requirements is also extremely difficult, as they deal with highly subjective concepts. Traditional validation methods such as testing are not easy to implement or very reliable [Callele *et al.*, 2005].

4.2 The incompatibility of the game design document and requirements engineering documentation

In traditional software engineering projects which utilize requirements engineering methods and processes, a common guideline for the design work is the requirement documentation. It is essentially the blueprint against which the product and its features are compared for specifications and verification.

In game development, a similar role is played by the game design document [Callele *et al.*, 2005]. While its contents and size vary from team to team and project to project, commonly it includes descriptions for plot, characters and events as well as gameplay

[2] https://unity3d.com/ (retrieved on 21.5.2018)
[3] https://www.unrealengine.com (retrieved on 21.5.2018)
[4] https://www.cryengine.com/ (retrieved on 21.5.2018)

mechanics, puzzles and so on. Much like the requirement document, the game design document is often created during pre-production [Callele *et al.*, 2005].

While these two documents share a similar role, they are not stylistically equal or even similar. A game design document is usually more free form and written in natural language [Callele *et al.*, 2005].

Since it is the primary design document for game development, it stands to reason – and has been proposed – that the game design document would also be a major source for requirements [Callele *et al.* 2005, Kanode & Haddad 2009]. This is logically sound, after all if the document contains descriptions of gameplay mechanics and elements, it stands to reason that requirements could be generated from these descriptions. In fact, some have gone as far as stating that all of the game design document should be captured as requirements before production should start [Kanode & Haddad 2009].

Evidence has shown this to be an unrealistic expectation, however. Even a single paragraph length description of a game design element from the game design document could end up producing several pages of requirements. Even worse, many of these requirements are merely implied, and capturing them requires high level domain knowledge in game design, genre conventions, technical matters and many other fields [Callele *et al.* 2005]. A skilled and experienced game developer will be able to pick up on some of these cues and implications, depending on how well versed they are in the different disciplines of game development (e.g. programming, art and sound design, writing), their team's own culture, the capabilities, features and limitations of the game engine the team is using, and the genre of the game they are working on.

Expecting this kind of expertise from a single person is unrealistic, as is the expectation of being able to generate good requirements based on heavily implicational natural language. The latter half of the problem could possibly be alleviated by employing technical writers, who are skilled in writing precise and unambiguous language, but they would probably not have the required domain knowledge. The common feeling is that it is "easier to do it myself than to explain it to someone else" [Callele *et al.* 2005] which may be true, but does not help eliminate the problem.

Even if suitable candidates could be found, or if the job of capturing the implied requirements were divided among a versatile group of skilled developers, the process would be extremely time consuming. Game development projects are usually executed under extremely tight, publisher-driven deadlines, and extending the pre-production phase to accommodate a lengthier requirements engineering process would probably not be welcomed [Callele *et al.* 2005]. For example, according to a study conducted in

Finland, most Finnish game development projects last under 12 months, with many of them lasting less than 6 months [Koutonen & Leppänen 2013].

It would therefore seem that there is a base level incompatibility between traditional requirements engineering documentation and the artefacts of game development.

4.3 Iteration, scope and change management

Change is an inevitable part of almost any software product. No matter how thorough the pre-production planning, how well executed the requirements engineering process and how accurate the model, something will eventually change. Change control and management are considered essential parts of the requirements engineering process, and significant work both during pre-production and production is carried out to ensure changes can be tracked and managed as efficiently as possible [Pohl 2010, Paetsch *et al.* 2003, Cao & Ramesh 2008].

Despite this, change is not seen as an outright goal, and instead more of an unavoidable necessity. This is in contrast with game development, where change is often outright desired. Game development is a heavily iterative endeavor, as the developers try to find the magical formula of features and gameplay executed just right to make the game as fun as possible [Kasurinen & Laine 2016, Stacey & Nandhakumar 2008]. This will inevitably lead to many and in some cases quite drastic changes to the design and scope of the project.

As discussed in Chapter 4.1, due to the emphasis on non-functional and affective requirements, change is also often the outcome of testing. A version of the game is given to testers, and based on their feedback changes can be made. Sometimes these changes can be quite drastic, and in many cases these iterations will carry on quite late in the actual development phase of the game and changes will occur very close to the end of the project. This is in part due to the fact that this user-driven testing is not only a tool for validation, but also defining the quality of the product [Kasurinen & Laine 2016, Stacey & Nandhakumar 2008].

With this in mind, it would stand to reason that game development could benefit from more robust change management procedures and methods. A common problem in game development is scope management. The game will be designed to have a certain set of features, and time and resources are budgeted to fulfil these design criteria in the available time.

During development features get added either due to outright planning, because testing suggested they might work well in the game, or sometimes even because individual developers felt they were “cool”. Suddenly there are no longer enough resources or time to finish the game as specified, and sometimes the revised and changed version of the

game no longer works as well as originally planned. This process is referred to as “feature creep”, and according to some sources, it is one of the biggest problems in game development [Petrillo *et al.*, 2008].

Feature creep is seen as a large problem not only because it creates scheduling problems, causes games to be delayed and costs money, but also because of its human cost. Game development is a massive industry, and publishers will often not be willing to delay projects significantly. Instead what happens is, game developers will work longer and longer days as deadlines approach. There are stories of people literally living at work, sleeping under their desks for a few hours when they can [IGDA 2004]. Burnouts and people quitting the games industry inevitably follow because of these heavy periods of crunch, as it is called.

However, at the same time, this iteration and change is both desired and necessary. Often developers will “find the fun” quite late in the development process, which means that if change and experimentation were to be avoided, these games might never have been finished, or at least not in their final conditions.

This issue is compounded by game development being notoriously difficult for scheduling in general. Evidence suggests many possible factors as the reasons. One popular suggestion is the multidisciplinary nature of games development. Different types of developers (e.g. artists, coders, writers, designers) have different workflows and different types of “production pipelines”, which can cause delays when some parts of the development team must wait for dependencies to be completed [Petrillo *et al.*, 2008].

In traditional software development, several processes and methods exist for managing changes and maintaining scope and product integrity despite changes. Therefore, it seems that game development could benefit from more robust change and scope control and scheduling mechanisms. Unfortunately, it seems that right now these mechanisms either do not exist, or are not utilized frequently, in game development.

4.4 Lack of formal methods and processes

According to two studies conducted in Austria [Musil *et al.* 2010] and Finland [Koutonen 2013], game developers do not make good or widespread use of typical methods and processes. In Finland, 61% of the respondents to the survey indicated that they did not use any systematic development methods. In Austria, 23% of respondents indicated they did not use any kind of formal methods or processes.

Even those who did self-report using formal methods and processes mostly used adapted and flexible processes which were “comparable to Scrum and XP” [Musil *et al.* 2010, Kasurinen 2016]. Further, according to the Finnish study, developers do not collect metrics or document their activities [Kasurinen 2016].

This kind of laissez-faire approach permeates all levels of development. For instance, developers prefer to not engage with traditional requirements engineering activities and instead prefer the approach of "test and tune" to replace it. As established, this testing is largely user-driven, as feedback received from users is used to gauge quality and drive development. Despite this, the feedback is not commonly collected in any kind of formal or systematic fashion [Kasurinen *et al.* 2014].

Some of this approach can be explained by base level incompatibilities in game development and traditional software engineering. Whereas traditional software engineering projects are launched to answer specific problems, game development can be iterative even at the ideation stage. It is common for developers to briefly explore tens of ideas initially, but only choose a few for detailed implementation, at which point the project has already moved to informal production and formal methods and processes are incompatible [Kasurinen *et al.* 2014].

There could also be other explanations. A lack of formal training and the tendency to promote from within could play a role. If a project manager does not have any training or knowledge about formal methods and processes, how could they hope to make use of them?

Despite this, there is evidence to support the claim that game development could benefit from adopting formal methods and processes. According to research conducted in Finland, many developers do already utilize some aspects of project management processes, but do so informally and in an ad-hoc fashion [Kasurinen *et al.* 2014]. This would seem to indicate that the need for these processes and their benefits exists within the developer community. The problems formal processes and methods are intended to fix are observable within the game development community: difficulty transitioning from pre-production to production, difficulty in capturing requirements, difficulty in change and scope management and so on [Petrillo *et al.* 2008].

According to research, game development falls into two broad and informal categories. Larger, more traditional developers still make use of more linear processes which bear a strong resemblance to the traditional waterfall model, whereas increasingly especially smaller developers are making use of agile and flexible methods. These agile methods are often self-created to some degree and might mostly draw inspiration from more formal schools of thought such as Scrum, Kanban and XP [Musil *et al.* 2010, Kasurinen *et al.* 2014].

Both styles of development could benefit from requirements engineering processes. For the more traditional project style, structured requirements engineering processes could be utilized in largely the same way as in traditional software engineering projects, hopefully

with similar results. Even the more informal projects, which are driven by iteration and user feedback, could benefit from structured processes and methods to capture and document this feedback and the requirements it generates [Kasurinen *et al.* 2014].

5 Post-mortem analysis

Based on the reading conducted on 15 current academic articles and white papers on game development, certain key problem areas and problems could be identified. As some of these academic writings leaned on industry-focused studies and were based on the thoughts and opinions of actual game developers, it can be assumed that these problems do in fact exist in actual game development at least to some degree. It was felt, however, that it would be beneficial to get more context for these findings. How common are these problems in actual game development?

Ideally the matter would have been studied with a larger study, perhaps one where industry professionals could have been interviewed or surveyed on these topics, but schedules did not allow for such an extensive study. Developer-written post-mortems on websites such as Gamasutra.com[5] and Gamecareerguide.com[6] offer some insight to actual industry professionals' opinions and thoughts on game development. It was felt that they could provide an adequate alternative option.

Post-mortems are a common industry practice, where a developer who served a central role in the project is invited to reflect on their project. According to Gamasutra.org's instructions [Shirinian 2011], each post-mortem should include a few aspects that went right in the project, as well as a few aspects that went wrong. These should be unique to the project, and should offer concrete thoughts other developers can learn from.

Due to their nature, these post-mortems were assumed to provide a valuable and reliable insight to the pros and cons of a wide variety of game development projects. They were therefore fetched and analysed algorithmically using custom Python and C# classes and scripts, as well as the statistical programming language R. The purpose of this analysis was to see if key topics and words related to what were perceived as central problems in the field, were present in these post-mortems.

The tests were conducted to test two assumptions.

1. If, for instance, requirements engineering methods and practices are not widely used in game development, keywords related to the topic would not appear frequently (or at all) in post-mortems.
2. If game development could benefit from requirements engineering methods and practices, common problems believed to be alleviated using these methods would appear at least relatively frequently.

This approach does have some limitations. As each developer is instructed to only include a few problems in each article, post-mortems are not exhaustive. Problems may not have

[5] https://www.gamasutra.com/features/post-mortem/ (retrieved on 21.5.2018)
[6] https://www.gamecareerguide.com/archives/post-mortems/1/index.php (retrieved on 21.5.2018)

been brought up among the few listed in a post-mortem despite influencing the actual development process.

An actual interview or even a survey would give more focused information on the topics of the thesis, but this does not necessarily have to be a weakness. As these post-mortems are not guided or directed by research questions or prompts, they do offer a view into what the developers themselves viewed as central and significant factors in the success or failure of their games.

Additionally, it is worth noting that correlation does not necessarily equal causation. Even if both assumptions turned out to be true, it does not automatically mean that all these problems are caused by the lack of requirements engineering methods and processes, nor that would they be fixed merely by adopting these methods and processes. Again, a much more exhaustive study would be required for any conclusive results, but it was felt that even a simple study like this might produce valuable findings.

5.1 Data gathering

As of March 2018, Gamasutra.com offers 218 separate post-mortem articles, but no easy way to access them beyond individual hyperlinks on a catalogue page. There is no way to access the plain text of each article, or even to access the articles themselves without all the surrounding, non-relevant website elements.

To get around these limitations, the HTML files for each article were retrieved using a custom script and the Linux program wget. Each file was saved locally as a PHP file, which contained the article text as well as non-relevant other data. These HTML files were then parsed with a custom Python script, which extracted the plain text using the Beautiful Soup library and exported it into TXT files.

Additional post-mortems were retrieved from GamecareerGuide.com [2018], which offers their own selection of unique post-mortems written in the same style as those found on Gamasutra.com. This second source produced 129 additional post-mortems.

The 347 post-mortems retrieved range from 1997 to 2018, and cover everything from small independent teams to large studios, and everything from small browser games to large, big budget productions. Games from a variety of different genres are included. Not all the post-mortems are suitable for this study, as some of them are small “post cards” from industry events. After eliminating these obviously non-related articles, there were 315 post-mortems left for analysis. These post-mortems cover roughly 300 unique games, as a few projects were discussed from different perspectives, such as general design and audio design.

5.2 The first study

Based on the central problems in game development presented in Chapter 4, a list of keywords was created. These keywords, which can be seen in Table 2, were thought to be related to these central problems based on existing domain knowledge on the topic. There was no specific methodology for creating this initial list of keywords, and instead it was always intended as a simple jumping off point which would hopefully generate interesting and promising articles, based on which additional keywords could be discovered.

The initial intent was to narrow down the list of 315 post-mortems to find which post-mortems should be studied more closely, and which could be discarded, as analysing all the post-mortems would not have been practical and quite probably also not useful. Therefore, the intent was to prioritize the post-mortems based on how many of these keywords appeared in them.

Project Management	Methods and processes	Requirements Engineering
crunch	Agile	requirement
schedule	Process	emotional
management	Method	affective
overtime	Scrum	game design document
estimation	Kanban	pre-production
feature creep	Engineering	production
creep	Development	requirement engineering
feature	transition	requirements engineering
scope	extreme programming	specification
communication	backlog	
multi-disciplinary	formal	

Table 2 Keywords used in the searches, divided into three broad categories

This analysis was conducted using custom built tools. First, a custom Python script, again using the Beautiful Soup library, iterated through all 315 post-mortems. The script searched for instances of keywords, noting down the articles in which they appeared, and exporting the results into a separate file. This file was then input into a custom C# application, which combined the results into custom objects, each representing the search results for a given article. These results could then be analysed and manipulated as desired.

As these custom tools were created and refined, the study evolved beyond simply trying to narrow down the list of post-mortems. It became apparent that getting statistical

information about how often given keywords appeared in articles would be easy, and the focus was shifted towards this approach.

This approach has some obvious problems and limitations. The first of these is the list of keywords used. If some relevant or useful term was not thought of, it would not be included on the list of search terms. As this study was created and conducted by a single person, albeit with some supervision, it is quite probable that something was overlooked. This problem was probably compensated at least in part for by repeated versions the keyword list and repeated analysis of the subject text. The list of keywords grew significantly over time as additional terms were discovered through further readings of the source texts, or derived from results of earlier iterations of the analysis.

Additionally, this approach offers next to no context. While the algorithm will find all instances of, say, the keyword “scope”, it has no way of knowing the context the term was used in. Did the article refer to the scope of the project, or was the author talking about a physical scope item in the game? Many of the terms used have multiple meanings, only some of which are relevant to this thesis, so this could have been a real problem.

To compensate for this lack of context, a second test was devised and run.

5.3 The second study

Whereas the first test relied on self-created Python and C# classes, the second test made use of R, a free software environment for statistical computing. A researcher from the University of Tampere who specializes in text analysis was asked to create an R algorithm suitable for the purposes of this thesis. This algorithm was then fine-tuned over time in collaboration with the researcher.

The algorithm first breaks the input text into smaller, sentence length chunks. Next, the text was lemmatized (i.e. the inflected forms of each word were grouped together in their dictionary form), and so called “stop words”, or common, short function words such as the, is, that and which, were removed. After these steps the remaining text was analysed. Sentences which contained words from the keyword list were kept, while the others were discarded. The remaining sentences were analysed for word co-occurrence, producing a list of found search terms as well as lists of words they appear together with. This would then give context to these results.

Due to the way the algorithm parses words, it will distinguish between multiple word keywords such as “feature creep” and individual components of the keyword, in this case “feature” and “creep”. Thus, the algorithm will not produce skewed false hits for these component words.

As with the first test, this test was also run several times. The original list of keywords grew and changed after each iteration as new keywords were discovered externally, prompting repetitions of the first study as well. Additionally, the results of this test also helped refine the list of keywords, as interesting or relevant terms appeared in co-occurrence with original keywords and were subsequently included as keywords themselves.

5.4 Findings

The two studies produced several outputs: a full list of all 315 post-mortems, and the keywords which appear in them, the total count of how often any keyword appears in each post-mortem, a list of all the keywords and the most common words that appear near them, and statistical information about the total number of occurrences for each keyword across all 315 articles (Table 4), as well as the percentage of articles each keyword appears in (Table 3).

Keyword	Total occurrences
development	2476
feature	1178
process	969
production	960
schedule	532
document	480
communication	322
management	290
scope	261
engineer	259
method	218
crunch	149
requirement	134
engineering	118
pre-production	102
emotional	79
discipline	69

Keyword	Total occurrences
scrum	65
creep	64
formal	64
transition	52
agile	43
feature creep	39
overtime	38
specification	30
game design document	21
estimation	14
backlog	6
multi-disciplinary	3
affective	2
extreme programming	1
kanban	0
requirement engineering	0
requirements engineering	0

Table 4 Total occurrences across all articles for a given keyword.

It becomes apparent that some terms were too broad especially for the initial intent of the studies even from a cursory glance at the list of keywords. The words “development”, “feature” and “process” appear in almost all of the articles. However, due to the word co-occurrence analysis (full findings in Appendix I), it is apparent that they do not appear

without context and were as such deemed interesting enough to be left in the pool of keywords.

The word co-occurrence analysis produced a list of each keyword and the most common words they appear together with in the analysed material (see Figure 8). The number of times each word appeared together with the term is also included, and if any other search terms appeared together with each search term, this is also indicated with italics. From the example in Figure 8 we can see that the term *schedule* appears together with terms like *project* and *time* 116 times, and with the search term "development" 79 times. Based on this example we can assume that at least 116 times this term was included in the source material in the desired context, ie. that of the project's schedule, instead of for instance the schedule for cleaning the studio's coffee machine.

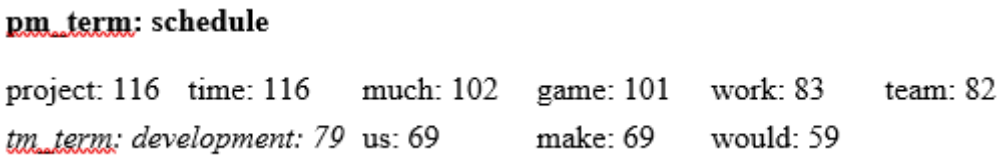

Figure 8 An extract from the word co-occurrence analysis results. Each search term is listed along with the words it appeared together with, as well as how often each co-occurring word appeared. If another search term appeared as a co-occurring word, it was indicated with italics and an identifying label signifying whether the keyword was related to project management, requirements engineering or methods and practises.

In general, terms thought to be related to the requirements engineering process and its methods appear either very rarely or not at all. "Requirements engineering" (and its alternative spelling "requirement engineering") do not appear once. The broader keyword "requirement" appears in 28.3% of the articles, but it is practically always used in the non-requirements engineering sense. "Affective" is used precisely once, and while the keyword "emotional" does appear in 11.1% of the articles, it is not used to talk about emotional requirements.

Terms related to agile methods and Scrum appear quite rarely. "Agile" is used in 10.2% of the articles, "Scrum" in 10.5%. "Extreme programming" is mentioned once. Formal methods and specifications in general do not seem to be a frequent topic in post-mortems, as the keyword "formal" is used in 13% of the articles. Context analysis suggests that when the term is used, it is rarely used in the context of formal processes: it appears three times close to the term "process", and three times close to the term "development". "Specification" is used in 6.3% of the articles, and is usually used in the context of formal design methods.

These findings would seem to back up the arguments presented in current academic research, and suggest that formal methods and practices, requirements engineering and other accepted industry best practices are not widely used in game development.

The problems these methods and processes are thought to alleviate appear in the post-mortems quite frequently. The keyword “crunch” appears in 26.3% of the post-mortems, and when it appears it is often mentioned several times in the same post-mortem. Additionally, the term “overtime” appears in 7.6% of the post-mortems, usually in the context of having to work overtime.

“Feature creep” is used in 9.8% of the post-mortems, and additionally “creep” is used in 15.9% of the articles, often in a context which suggests it is used to describe feature creep rather than an action by a game character.

Terms such as “schedule” (56.8%), “management” (43.8%), “communication” (47.3%) and “document” (69.2%) appear very often, both in positive and negative contexts, indicating they are factors in the successes or failures of game development projects.

6 Discussion

The topic of requirements engineering and game development is by no means a new one. As game development is a specialized field of software development, and requirements engineering is an accepted and commonly used part of the software engineering process, the assumption that game development could benefit from requirements engineering processes and methods is only natural.

Along with this long-standing interest in the topic comes a lot of previous research. This body of work varies greatly in scope and style. As game development is a practical real-world problem, it stands to reason that for it to truly be useful, research carried out on the topic should be conducted with the realities of the discipline in mind, if the goal is to solve real problems faced by developers.

With this in mind, this thesis set out to explore two research questions:

1. Based on a reading of current academic research on the topic, can central problems, concerns and issues be identified?
2. Can these findings be supported by analysing developer-written post-mortems?

6.1 Background and context

Key problems and issues were identified based on academic research conducted through interviews and studies conducted among game developers and game publishers. Some of these problems make it harder to adopt requirements engineering processes as a part of the game development process, while some are areas where game development could clearly benefit from adopting these processes.

It is worth stressing that the findings in this thesis apply mostly to smaller independent developers, the most common type of developer for instance in Finland. Larger and more organized studios probably have their methods and processes for dealing with these issues and approach the development process much in the same ways as a traditional software development project would, but as these larger studios and corporations tend to regard their practices and methods as trade secrets, little information is available on the subject.

Of the problems discovered, the general lack of formal processes and methods among developers seems to be the most fundamental one. While the emphasis on non-functional requirements and the lack of tools for capturing and modelling affective requirements are also significant problems, they can be overcome with work.

That work will not be conducted if developers are not interested in utilizing formal methods and processes, or applying requirements engineering techniques to their work. The reasons for this perceived lack of interest and its remedies are beyond the scope of a thesis work, and a large survey would be needed to chart attitudes and problems before

even any educated guesses could be made. It could be that developers are interested and willing to utilize more formal methods, but do not have the knowledge and skills required, or they might not even be aware of the possibility, having grown used to doing things their own way.

It is worth noting that these are by no means the only significant challenges or problems game developers are facing, nor are they the only factors making it harder to adapt requirements engineering methods and processes to game development. As an example, game development is a much more multi-disciplinary activity than normal software development.

Game development teams employ software engineers, designers, producers, project managers and other computer science professionals just like traditional software engineering teams, but additionally make use of different types of artists (e.g. writers, graphical artists, musicians, animators, sound technicians) and others.

It was briefly mentioned earlier in Chapter 4 that merely finding common vocabulary among these wildly varied disciplines can be challenging, but their variety alone introduces difficulties into the requirements engineering process. Capturing and modelling requirements specific to each of these disciplines requires strong domain knowledge.

Beyond the need for specialized knowledge, all the disciplines of game development may have their own considerations that need to be taken into account, and scheduling can also be challenging. Not all of these components might even be actively worked on during the pre-production phase, where most of the requirements engineering work takes place.

While a significant problem, this is not unique to game development, as traditional software development projects need specialized domain knowledge for requirements engineering work as well. For instance, experts on legal concerns, data privacy or sociology might have specialized domain knowledge needed in the project.

As so many different problems could be discovered so easily, the topic is clearly rife for further research and discussion. The scope of this thesis could have been far larger, as many of these problems would have been interesting topics of research and discussion. It is clear that there is much work left to be done in this area.

6.2 Contributions

To evaluate if these identified key problems are actual issues for game developers, 315 post-mortems were analysed. The results of this analysis are not conclusive by any means, but do seem to give validity to the idea that these are real issues and factors affecting actual game development projects. This is not surprising, as the reference materials used

for this thesis leaned heavily towards more practical studies [Kasurinen & Laine 2014, Kasurinen *et al.* 2014, Kasurinen 2016, Koutonen & Leppänen 2013, Petrillo *et al.* 2008] and therefore were already based on the experiences and opinions of developers.

From the perspective of this thesis, it is especially interesting to note the almost complete lack of keywords relating to requirements engineering. The topic itself was not mentioned once in the 315 post-mortems, which include everything from big budget PC games to smaller indie products, games created using traditional waterfall methods to agile projects and so on.

As post-mortems deal with factors which contributed, positively or negatively, to the outcome of each individual project, the total lack of mentions could mean that requirements engineering is simply not a concern to any of these developers. This result is not conclusive, of course, as post-mortems are not all-inclusive lists of all contributing factors. However, the fact that no developer mentioned requirements engineering as a factor – positive or negative – in the outcome of the project, does give validity to the claim that game developers do not utilize, nor even think to utilize, requirements engineering methods or processes.

As there is next to no discussion on keywords related to requirements engineering, this study did not reveal any conclusive evidence for or against the incompatibility between requirements engineering and the game design document. Keywords such as scope (41.9%) and document (69.2%) are often mentioned in post-mortems, so clearly some kind of issue exists, but based on this study little can be said on the topic.

The keyword "crunch" appeared in 26.3% of the articles, and the clearly related term "overtime" appeared in 7.6% of the articles. The algorithm does not guarantee that there is no crossover between these results, so both keywords could appear together in at least some of the post-mortems. Phrases such as *"It was an expensive lesson, given the amount of overtime we had to work to finish the game", "building several levels, working a tremendous amount of overtime"* and "*others were totally fried from the tremendous amount of overtime"* indicate that "overtime" usually appears in the intended sense rather than describing, for instance, a system working overtime.

This figure seems low, as crunch is generally considered an extremely widespread problem in the industry. According to a 2016 survey conducted among the International Game Developers Association members, 65% of developers reported having experienced crunch, with 52% reporting having experienced it more than twice in the previous year [IGDA 2016].

This inconsistency could be explained by several factors. The post-mortems deal with individual projects, rather than individual developers, the contrary of which is true on the

IGDA survey. Thus, even a project where multiple developers reported experiencing crunch would only represent a single item in the post-mortem data. The post-mortems also include many smaller indie projects, which might be more loosely scheduled and could afford to postpone the project rather than crunch to finish it on an external schedule. Finally, the post-mortems include material from 1997 to 2018, and it could be that in the earlier material crunch simply was considered an inevitable part of working in the game industry and not worth reporting as a factor.

Terms related to formal project management processes and methods appear in the post-mortems quite often, and in contexts which relate to project management.

- document (69.2%)
- production (63.5)
- schedule (56.8%)
- management (43.8%)
- communication (47.3%)
- scope (41.3%)

This means that these issues were considered by developers to be a key factor in the success of the project, whether a positive or negative one. This would seem to be in line with Kasurinen's claim that game development would benefit from more formal, commonly used methods and practices, as they are generally agreed to improve and facilitate these key areas of the development process [Kasurinen *et al.* 2014].

These findings demonstrate that there is clearly need for further and deeper studies on the issue. Game development is a growth industry where ever-increasing amounts of money are on the line, depending on the successful outcome of large, expensive and extremely complex software development projects. It is clear that game development could benefit from additional formalization, but in order for that to happen, several hurdles need to be crossed.

Developers need training, and methods and processes need to be adapted and created to better suit the needs of the industry. While these initiatives probably need to be driven by developers themselves, academic research has an important role to play as well. Studies conducted by academics could hopefully breach the wall of secrecy surrounding many developers and help discover both the causes and eventual fixes for these problems.

7 Conclusions

This thesis explored the question of adapting requirements engineering methods and processes to game development projects. Based on a thorough reading of state-of-the-art academic research, key problems and limitations were identified. These included

- a general lack of formal processes and methods in game development
- the emphasis on non-functional, affective requirements, which traditional requirements engineering methods and processes are not well suited to
- emphasis on change as a central development tool, and the need for better change control, which requirements engineering could provide
- the incompatibility between the requirements document and the game development document, central artefacts in requirements engineering and game development respectively.

To study the validity of these claims, 315 developer-published post-mortems were analysed algorithmically, using custom programs created for the purposes of this thesis. Keywords based on academic findings were searched for, and their total number of appearances, as well as the frequency of these appearances, were noted. Additionally, they were analysed for word co-occurrence to discover, which words the keywords commonly appeared with.

While the findings of this analysis were not conclusive, they did seem to offer support to the key problems identified in Chapter 4. These problems would therefore seem to be real issues being faced by game developers. Much work remains to both discover, and more importantly implement, best practises and guidelines for solving these issues.

References

[Ampatzoglou & Stamelos 2010] Ampatzoglou, A., & Stamelos, I. (2010). Software engineering research for computer games: A systematic review. *Information and Software Technology*, *52*(9), 888-901

[Callele *et al.* 2005] Callele, D., Neufeld, E., & Schneider, K. (2005, August). Requirements engineering and the creative process in the video game industry. In Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on (pp. 240-250). IEEE.

[Cao *et al.* 2008] Cao, Lan, and Balasubramaniam Ramesh. "Agile requirements engineering practices: An empirical study." *IEEE software* 25.1 (2008).

[ESA 2016] Entertainment Software of America: *Analysing the American Video Game Industry 2016.* http://www.theesa.com/wp-content/uploads/2017/02/ESA-VG-Industry-Report-2016-FINAL-Report.pdf (retrieved on 20.12.2017)

[Faulconbridge & Ryan 2003] Faulconbridge, R. I., & Ryan, M. J. (2003). Managing complex technical projects: A systems engineering approach. Artech House.

[Fourman 2005] Fourman, M (2005). Propositional methods: Computational Tree Logic [PowerPoint sliders]. Retrieved from http://www.inf.ed.ac.uk/teaching/courses/propm/papers/CTL.pdf (retrieved on 25.4.2018)

[Game Career Guide] Game Career Guide: Features: Post-mortems. Retrieved from https://www.gamecareerguide.com/archives/post-mortems/1/index.php

[Gunda 2008] Gunda, S. G. (2008). Requirements engineering: elicitation techniques.

[Hickey & Davis 2003] Hickey, A.M., Davis, A.M. (2003): Elicitation Technique Selection: How Do Experts Do It?, Proceedings of the Eleventh IEEE International Requirements Engineering Conference, pp. 169-178, September 8-12, Monterey Bay, CA.

[Hofmann & Lehner 2001] Hofmann, H. F., & Lehner, F. (2001). Requirements engineering as a success factor in software projects. *IEEE software*, *18*(4), 58.

[Hooks 1994] Hooks, I. (1994, August). Writing good requirements. In *INCOSE International Symposium* (Vol. 4, No. 1, pp. 1247-1253).

[Hull *et al.* 2005] Hull, E., Jackson, K., & Dick, J. (2005). *Requirements Engineering 2nd Edition*, Springer Publishing Company, Incorporated

[IGDA 2004] International Game Developers Association. *Quality of Life in the Game Industry: Challenges and Best Practices.* IGDA 2004.

https://www.igda.org/resource/collection/9215B88F-2AA3-4471-B44D-B5D58FF25DC7/2004_IGDA_QualityOfLife_WhitePaper.pdf (retrieved on 23.5.2018)

[IGDA 2018] International Game Developers Association. *Developer Satisfaction Survey 2016*. IGDA 2018. http://www.igda.org/resource/resmgr/files__2016_dss/IGDA_DSS_2016_Summary_Report.pdf (retrieved on 29.4.2018)

[JIRA 2017] JIRA Project Management Software, https://atlassian.com/software/jira (retrieved on 21.12.2017)

[Kanode & Haddad 2009] Kanode, C. M., & Haddad, H. M. (2009, April). Software engineering challenges in game development. In Information Technology: New Generations, 2009. ITNG'09. Sixth International Conference on (pp. 260-265). IEEE.

[Kasurinen 2016] Kasurinen, J. (2016, June). Games as Software: Similarities and Differences between the Implementation Projects. In *Proceedings of the 17th International Conference on Computer Systems and Technologies 2016* (pp. 33-40). ACM.

[Kasurinen *et al.* 2014] Kasurinen, J., Maglyas, A., & Smolander, K. (2014, April). Is requirements engineering useless in game development?. In International Working Conference on Requirements Engineering: Foundation for Software Quality (pp. 1-16). Springer, Cham.

[Kasurinen & Laine 2014] Kasurinen, Jussi, and Risto Laine. "Games from the Viewpoint of Software Engineering." *Proc. of the Federated Computer Science Event* (2014): 23-26.

[Keith 2010] Keith, C. (2010). Agile Game Development with Scrum (Adobe Reader). Pearson Education.

[Koutonen & Leppänen 2013] Koutonen, J., & Leppänen, M. (2013, June). How are agile methods and practices deployed in video game development? A survey into Finnish game studios. In *International Conference on Agile Software Development* (pp. 135-149). Springer, Berlin, Heidelberg.

[LucidChart 2017] LucidChart Online Diagram and Visualization Solution. http://lucidchart.com (retrieved on 21.12.2017)

[Magee & Kramer 1999] Magee, J., & Kramer, J. (1999). *State models and java programs*. Wiley.

[Musil *et al.* 2010] Musil, J., Schweda, A., Winkler, D., & Biffl, S. (2010, September). Improving video game development: Facilitating heterogeneous team collaboration through flexible software processes. In European Conference on Software Process Improvement (pp. 83-94). Springer Berlin Heidelberg.

[Newzoo 2017] Newzoo: *Report on the Global Games Market*. https://newzoo.com/insights/articles/the-global-games-market-will-reach-108-9-billion-in-2017-with-mobile-taking-42/ (retrieved on 20.12.2017)

[Nuseibeh & Easterbrook 2000] Nuseibeh, B., & Easterbrook, S. (2000, May). Requirements engineering: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering* (pp. 35-46). ACM.

[OSRMT 2017] Open Source Requirements Management Tool, https://sourceforge.net/projects/osrmt/ (retrieved on 21.12.2017)

[Paetsch *et al.* 2003] Paetsch, F., Eberlein, A., & Maurer, F. (2003, June). Requirements engineering and agile software development. In Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003. WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on (pp. 308-313). IEEE.

[Paschali *et al.* 2014] Paschali, M. E., Ampatzoglou, A., Chatzigeorgiou, A., & Stamelos, I. (2014, November). Non-functional requirements that influence gaming experience: A survey on gamers satisfaction factors. In Proceedings of the 18th International Academic MindTrek Conference: Media Business, Management, Content & Services (pp. 208-215). ACM.

[Petrillo & Pimenta 2010] Petrillo, F., & Pimenta, M. (2010, September). Is agility out there?: agile practices in game development. In Proceedings of the 28th ACM International Conference on Design of Communication (pp. 9-15). ACM.

[Petrillo *et al.* 2008] Petrillo, F., Pimenta, M., Trindade, F., & Dietrich, C. (2008, March). Houston, we have a problem...: a survey of actual problems in computer games development. In Proceedings of the 2008 ACM symposium on Applied computing (pp. 707-711). ACM.

[Pohl 2010] Pohl, K. (2010). Requirements engineering: fundamentals, principles, and techniques. Springer Publishing Company, Incorporated.

[Sannier 2011] Sannier, N (2011): Modeling Requirements Requirements Verification and Validation. *EDF R&F*. https://nicolassannier.files.wordpress.com/2011/04/3-modeling-requirements-requirements-validation-and-verification-sannier.pdf. Retrieved on 20.11.2017

[Sharp *et al.* 1999] Sharp, H., Finkelstein, A., & Galal, G. (1999). Stakeholder identification in the requirements engineering process. In *Database and Expert Systems Applications, 1999. Proceedings. Tenth International Workshop on* (pp. 387-391). Ieee.

[Shirinian 2011] Shirinian, A (2011). Dissecting the Post-mortem: Lessons Learned From Two Years of Game Development Self-Reportage. Retrieved from https://www.gamasutra.com/view/feature/134679/dissecting_the_post-mortem_lessons_.php

[IEC 2010] The International Engineering Consortium (2010). Specification and Description Language (SDL), http://www.sdl-forum.org/SDL/Overview_of_SDL.pdf (retrieved on 28.11.2017).

[Stacey & Nandhakumar 2008] Stacey, P., & Nandhakumar, J. (2008). Opening up to agile games development. Communications of the ACM, 51(12), 143-146.

[UKIE 2017] UKIE: *The games industry in numbers*. (https://ukie.org.uk/research) (retrieved on 20.12.2017)

[Zielczynski 2007] Zielczynski, P. (2007). *Requirements management using ibm® rational® requisitepro®*. IBM press.

[Zhang 2007] Zhang, Z. (2007). Effective requirements development-A comparison of requirements elicitation techniques. *Software Quality Management XV: Software Quality in the Knowledge Society, E. Berki, J. Nummenmaa, I. Sunley, M. Ross and G. Staples (Ed.) British Computer Society*, 225-240.

[Zowghi & Coulin 2005] Zowghi, D., & Coulin, C. (2005). Requirements elicitation: A survey of techniques, approaches, and tools. In *Engineering and managing software requirements* (pp. 19-46). Springer Berlin Heidelberg.

Appendix I: Co-occurrence of words within post-mortems

Each keyword is listed along with the top words they appeared in close proximity to. Any other keywords appearing as co-occurring words are italicized.

pm_term: crunch

time: 33	project: 29	much: 27	mode: 27	team: 24	game: 19
period: 19	us: 18	month: 17	want: 16		

pm_term: schedule

project: 116	time: 116	much: 102	game: 101	work: 83	team: 82
tm_term: development: 79	us: 69	make: 69	would: 59		

pm_term: management

project: 58	game: 51	team: 49	much: 29	*tm_term: development: 27*
time: 27	good: 25	work: 24	make: 23	problem: 21

pm_term: overtime

work: 20	team: 8	project: 8	game: 7	go: 7	get: 6
amount: 6	hour: 4	new: 4	us: 4		

pm_term: estimation

time: 1	year: 1	fact: 1	meet: 1	target: 1	poor: 1
magazine: 1	profi: 1				

pm_term: feature-creep

end: 5	much: 4	us: 4	project: 4	problem: 4	make: 4
game: 3	*tm_term: development: 3*	avoid: 3	experience: 3		

pm_term: creep

time: 4	*pm_term: scope: 4*	still: 3	long: 3	*pm_term: feature: 3*
much: 2	work: 2	team: 2		

pm_term: feature

game: 409	much: 204	time: 174	new: 172	add: 162	would: 147	make: 144
good: 103	use: 98	implement: 95				

pm_term: scope

game: 77	project: 41	much: 37	time: 24	make: 23	would: 19
design: 19	good: 18	also: 16	team: 15		

pm_term: communication

team: 71 good: 39 game: 32 much: 29 problem: 28 work: 24
project: 20 *tm_term: development: 20* make: 17 design: 17

pm_term: discipline

project: 12 team: 11 game: 11 much: 9 *tm_term: development: 7*
work: 6 good: 6 high: 5 *tm_term: process: 4* help: 4

tm_term: agile

tm_term: development: 6 *tm_term: scrum: 6* use: 5 make: 4
tm_term: process: 3 methodology: 3 version: 2 work: 2
team: 2 game: 2

tm_term: process

game: 210 *tm_term: development: 173* much: 117 design: 114 time:
109 make: 94 good: 92 team: 77 would: 74 work: 72

tm_term: method

use: 52 game: 33 much: 23 good: 22 work: 21 make: 21 us: 15
animation: 15 project: 13 render: 13

tm_term: scrum

tm_term: development: 16 project: 14 game: 9 use: 9 team: 8
plan: 7 much: 6 us: 6 time: 6 good: 6

tm_term: engineering

team: 23 game: 18 design: 18 much: 16 work: 15 project: 15
software: 15 tool: 15 art: 14 us: 12

tm_term: development

game: 814 team: 379 time: 379 much: 309 work: 236 project: 233
make: 228 month: 218 good: 179 *tm_term: process: 173*

tm_term: transition

game: 18 much: 10 make: 10 us: 7 good: 7 project: 6
time: 6 would: 6 something: 5 original: 5

tm_term: extreme-programming

also: 1	program: 1	hall: 1	discuss: 1	com: 1	pair: 1	http: 1
cgi: 1	larry: 1	wiki: 1				

tm_term: backlog

team: 2	us: 2	also: 2	add: 2	fix: 1	even: 1
report: 1	always: 1	organize: 1	case: 1		

tm_term: engineer

game: 36	artist: 34	team: 24	designer: 23	audio: 22	work: 20
time: 20	much: 19	*tm_term: development: 19*		engine: 19	

tm_term: formal

team: 5	make: 5	game: 4	sure: 4	design: 4	much: 3
new: 3	*tm_term: process: 3*		*tm_term: development: 3*		without: 3

re_term: requirement

game: 51	team: 26	gameplay: 21		work: 19	design: 17
much: 16	time: 16	system: 15	base: 14	model: 14	

re_term: emotional

player: 34	state: 32	use: 30	game: 28	form: 28	would: 25
music: 20	relationship: 19		prototype: 18		art: 18

re_term: affective

experience: 2	story: 2	player: 2	creation: 2	game: 1	focus: 1
good: 1	tell: 1	think: 1	create: 1		

re_term: game-design-document

game: 12	play: 10	build: 10	designer: 10	world: 10	even: 9
though: 9	everyone: 9	separate: 9	would: 9		

re_term: document

design: 101	game: 68	project: 45	time: 29	way: 27	write: 23
detail: 23	much: 22	team: 21	record: 19		

re_term: pre-production

month: 41	game: 32	time: 29	much: 26	concept: 22	project: 21
long: 21	prototype: 19		part: 18	course: 18	

re_term: production

game: 187	much: 109	team: 109	time: 104	work: 82	project: 73	
good: 72	design: 71	go: 61	make: 59			

re_term: specification

design: 12	game: 11	work: 7	team: 7	make: 6	would: 6	good: 6
project: 5	also: 5	mean: 5				