jamk

Technical Debt in Game Development

Anna Rosputnia

Bachelor's thesis
May 2025
Bachelor of Business Administration, Business Information Technology

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Description

Rosputnia, Anna

Technical Debt in Game Development

Jyväskylä: Jamk University of Applied Sciences, May 2025, 69 pages.

Degree Programme in Business Information Technology, Game Production. Bachelor’s thesis.

Permission for open access publication: Yes

Language of publication: English

Abstract

Technical debt is a known phenomenon in software development, yet its implications in the context of game development are largely underexplored. Game projects are similarly affected by it and by extension experience similar consequences that can potentially harm the development process.

A generalized introduction to the topic was made in order to help educate developers to avoid the worst effects of technical debt. Various sources in software and game development were explored to define and identify technical debt, the causes and effects of it on the development process, and how it can be managed.

To gather understanding of the concept from the perspective of game development, a qualitative survey was conducted to investigate how technical debt is perceived and handled by individuals in the game industry. The results of which were then summarized and analyzed through thematic analysis to provide an overview of some common opinions and experiences with technical debt as reported by various game developers.

Keywords/tags (subjects)

game development, programming, game programming, technical debt, refactoring, qualitative research, thematic analysis

Contents

Terminology

- **Codebase** is the collection of source code in a software, system or project.
- **Refactoring** is the process of restructuring code to improve it, without changing its functionality.
- **Documentation** is text or guidelines to communicate information about software.
- **Optimization** is the process of improving software systems to reduce resource use and latency.
- **Unit testing** is testing made on the smallest, individual parts of a system, such a single function, class or method.
- **Modularity** is the practice of breaking down complex systems into organized, reusable modules.

Introduction

1 The Cost of Development

Technical debt is a recurring phenomenon in both game development and traditional software engineering. While it is sometimes a necessary or unavoidable part of a project, unmanaged technical debt can cause significant setbacks. These include project delays, cancellations, increased costs, developer burnout, and damage to business reputation.

These consequences connect directly to several aspects of sustainable development. Economically, technical debt can threaten project viability and long-term profitability. Socially and individually, it contributes to team frustration, decreased motivation, and stress. Technically, it undermines the longevity and maintainability of systems. For these reasons, it is critical that developers learn to recognize and manage technical debt before it leads to irreversible damage in future projects.

This thesis investigates the concept of technical debt within the context of game development, exploring how it occurs, how game developers perceive it, and how it is addressed in practice. Through qualitative research, this study presents how technical debt is perceived in the game industry by developers of various skill levels and backgrounds. Insights gathered from a survey of game developers reveals common causes, experiences and solutions for technical debt. Time pressures, poor planning, inexperienced leadership and unexpected changes in scope – all were named as causes for complications that lead to increase in technical debt. These factors offer clues to how teams might be able to improve their development process and avoid preventable losses.

1.1 Justification

Understanding technical debt is essential to sustainable development. Technical debt is an inevitable part of production and can affect a game's lifetime at every stage. Education on the topic promotes efficiency and maintainability of the development process, which in turn supports long-term viability of games, teams and studios. Addressing technical debt aligns with the goals of sustainable development in economic, social and technical dimensions.

1.2 Delineation

The topic of technical debt is complex and widely explored in previous research, however studies on its impact in game development are sparse. As such, this thesis provides context through related fields such as software engineering, design principles and game discussions, which help extend the relevance to the phenomenon and give valuable information for this research.

1.3 Definition

The objective of this research is to foster awareness and foundational understanding of technical debt and related concepts in the context of game development, particularly for newer developers or next researchers. Real-world experiences from game developers were gathered to expand the pool of knowledge on the topic of technical debt within game development.

Research Questions

- What is technical debt like in game development?
- How does technical debt accumulate?
- How does technical debt affect development?
- How is technical debt managed?
- How much do various game developers know about technical debt?
- What is technical debt according to game developers?
- What do developers identify as the cause for technical debt most frequently?
- How do game developers manage technical debt?
- Why does managing technical debt in game development matter?

The first four questions are explored through literature and prior knowledge, while the remainder are addressed through a qualitative survey targeting a range of game developers.

2 Knowledge Base

Game development is one of the facets of software development. Software is created to fulfil a purpose, and games are software whose goal is entertainment, creating engaging experiences with art and music. (Bethke, 2003)

As such, programming is a part of the game development process, much like any other software. Games have structure, codebases and various systems. Game programmers are responsible for designing, writing and maintaining the code of the game the same way traditional software developers do. Therefore, during development game programmers are similarly likely to at some point run into issues with their code, such as unintended behavior, using excessive resources or numerous other bugs that temper with the experience of the game.

Programmers of all kinds often understand the concept of “bad code” – systems that are poorly written, flawed, and can create more problems than they solve by being more difficult to understand and maintain. This can take the form of overly complex solutions, poor organization, lack of documentation, repeated code, and other factors that undermine the comprehension of the system by the developers.

Bad code is frequently encountered and fixed during development, which typically is a normal and expected part of the process. However, a far more significant issue may arise when a system’s structure has deteriorated to the point that fixing bad code becomes exceedingly difficult – or, in some cases, impossible. This phenomenon is commonly referred to as “technical debt” in software development circles.

2.1 What is Technical Debt?

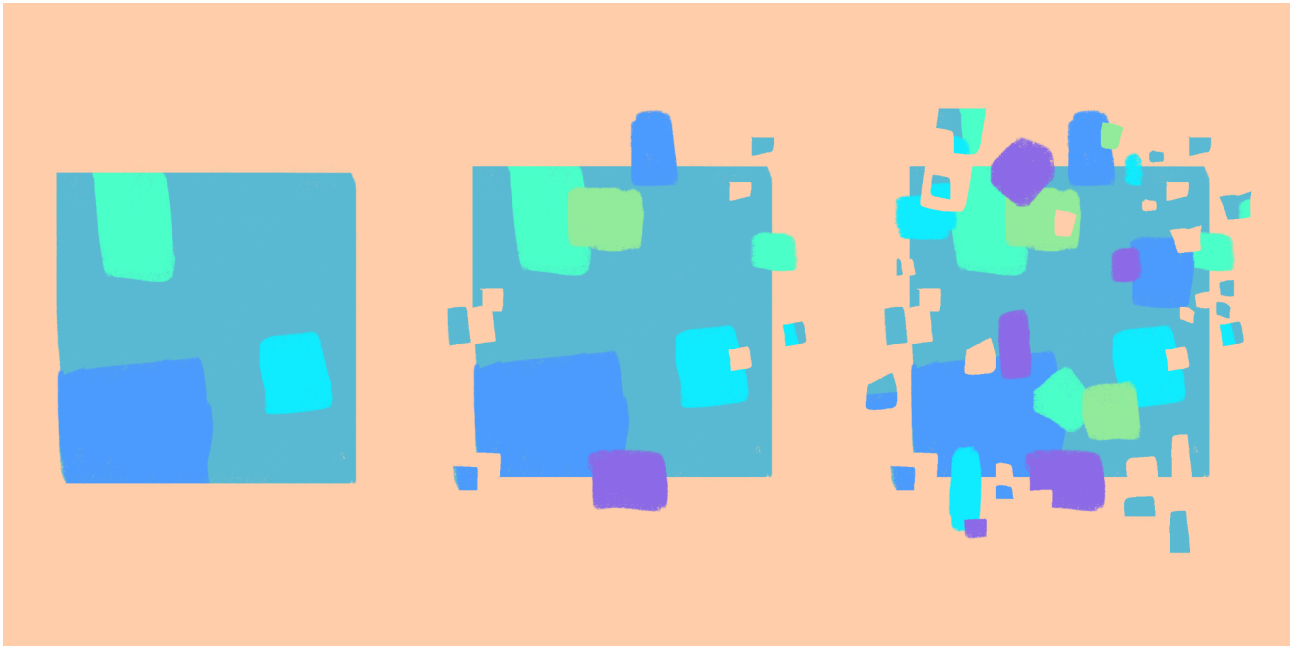
To answer that question, consider the following scenario: development has stalled, and progress has become unbearable. After a quick assessment it becomes evident that there are fundamental flaws in the project – a single, giant script handles player movement, abilities and camera controls all in one. Now, somehow, new features need to be added to those abilities. How could that be accomplished if one wrong move could break all the rest of the mechanics contained in that script? With much frustration, time and patience. Such a flaw could have been avoided with some foresight, but now it has taken far longer to detangle that mess than it would have to simply plan

ahead and separate those mechanics from the start. This type of situation is what technical debt typically encompasses.

Cunningham (2009) originally introduced the **technical debt** metaphor to explain the need for improving code over time by comparing it to financial debt, primarily to non-technical people in the project. It revolves around the concept of taking out a loan which will need to be paid off with interest some time later, else the loan-taker will face consequences from the bank. Similarly, in technical debt the developer takes out a loan when a decision is made for a short-term benefit in the project, whether by choice or accident. In that loan, Interest accumulates over time until that decision is replaced by a better, long-term solution (Suryanarayana et al., 2014). Therefore, if technical debt is not addressed by the development team, it can accumulate to such a degree that deems further development too difficult to continue, causing software to potentially be delayed indefinitely or even cancelled (Extra Credits, 2019). While the metaphor primarily presents a comparison of a phenomenon within software development to financial debt, technical debt is instead more commonly used as a term for the concept itself, foregoing its nature as an analogue. This shift becomes evident as the conversation focuses less on the metaphor itself and more on what that metaphor is attempting to explain.

Figure 1

A visual representation of a project becoming disorganized over time due to technical debt



Note. This image was created by the author. Various colors represent parts of the project, which get duplicated, moved or altered in a way that makes the original design incomprehensible.

Technical debt may arise from a risky decision made in the design process, with the intention to revisit it later – justified by any reason that saves time in the present moment. Rideout (2017, 1:41) proposed that it often comes from the programmer's mindset: "It's not an ideal solution, but it is a solution.", even when that solution has the potential to soon become a problem in itself.

However, that approach to problem-solving is not necessarily a mistake. Farley (2022) claimed that not all debt is detrimental, provided you can repay it in due time. Both Farley (2022) and Nemchinskiy (2024) argued that, in specific cases, technical debt may be necessary for continued development and progress, but while certain types of technical debt are inevitable, some can and should be avoided. Any system will inevitably contain sub-optimal components, and meticulously correcting every single mistake is unnecessary and unrealistic. However, assuming you are able to repay your technical debt, it may be beneficial to embrace it in order to further development and introduce more features. As such, taking risks and prioritizing "completed" over "perfect" is a valuable skill, like any other, to have and use when appropriate.

What counts as technical debt is not entirely set in stone, Martin (2009) expressed that messy code should not be considered technical debt. He debated that risky trade-offs that do not consider the state or needs of the project in the future are the product of unprofessional ignorance that will harm the project no matter what. In the same vein, Suryanarayana et al. (2014) claimed that bugs are not technical debt, they identified visibility as the main difference between the two: bugs and defects are experienced by the users and technical debt is not. The user's experience is prioritized, therefore defects that affect the users receive the most attention, resulting in technical debt being overlooked. As a result, it can be assumed that bugs are a byproduct of technical debt rather than an aspect of it.

To return to the original question of this section, in summary, technical debt can be attributed to anything that significantly slows development (Web Dev Simplified, 2021), and a key cause for that which slows development are the problems that should have been solved earlier.

Overall, the numerous studies conducted on the topic of technical debt describe the phenomenon in various ways, each one discussing facets of code design, project scope and other areas of development that affect the debt both positively and negatively. However, the vast majority of the research done on technical debt and related concepts has been conducted in the context of software development alone, with a lack of studies in game development being noted by Borowa et al. (2021) and Agrahari & Chimalakonda (2020). As such, it is valuable to understand what role the debt metaphor plays in game development and why it is different from traditional software development practices.

2.2 Game Development vs. Software Engineering

Technical debt originates in software engineering, and it is not the only term used to describe the phenomenon of mounting problems in code. For instance, Fowler (2019) called it “cruft”, he defined it as the unnecessary complexity within a system that requires more effort to make changes, and Web Dev Simplified (2021) described technical debt as a library that has become disorganized, with piles of books to search for hours to find what is needed. Not everyone is familiar with the debt metaphor, but many often understand the concept through their own experiences and interpretations. Griffith (2014) stated that this understanding is innate to most software developers, whether they are formally educated on the subject or not. Thus, much of what is described by traditional software developers can be applied to game developers.

This is consistent with the notion that games are software, and in turn they share similarities with traditional software development, namely their need for flexibility in planning and management of the projects. Game development has even adopted development models found in software development, such as waterfall planning or Agile development. Many such models hold value to both software and game development, but are lacking when used in practice. However, the chaotic nature of games drove developers to use increasingly flexible development methods, far more than that in traditional software. (Engström et al., 2018)

That is primarily because the goal of the gaming industry is to provide entertainment, which creates different priorities in the project compared to software development. Borowa et al. (2021) suggest that the gaming industry has different experiences with technical debt than software development. As a result, the nature of game development as an art, business and entertainment

industry creates unique challenges and gives a different environment and development process than that of software development. Engström et al. (2018) stated “video games can not solely be seen and understood as software products, as they can also be characterized as creative products, cultural expressions, or even artwork …”, furthering the idea that games are more than their code. Borowa et al. (2021) claimed that the scope of a game project is unpredictable and ever-changing. That development environment makes it increasingly difficult to plan, test, organize and estimate what will happen during development, and consequently, that adds to the difficulty of managing technical debt within a game project. That is the major difference between general software development and game development – games are always changing.

While technical debt in software development primarily concerns programming, in game development it can extend to other areas such as disorganized level design, unoptimized 3D models that exceed limitations of the engine or outdated art style guidelines that no longer fit the project’s vision. Even simple actions such as time spent locating and managing code, props or models increases difficulty in development exponentially. Such minor inconveniences compound and waste hours of development time as unresolved issues accumulate, often with no guarantee that these problems will be addressed at all due to that same chaotic and unpredictable nature of game development. (Extra Credits, 2019)

That being said, in the context of games, eventually the discussion about technical debt ends and instead shifts to general game design and production planning because programming is only one part of the game development process and every other area of expertise holds equal value to it. However, effective production timelines and game design is a much broader topic in itself, one that will only be touched on lightly, as this thesis primarily focuses on the programming aspect of technical debt in game development.

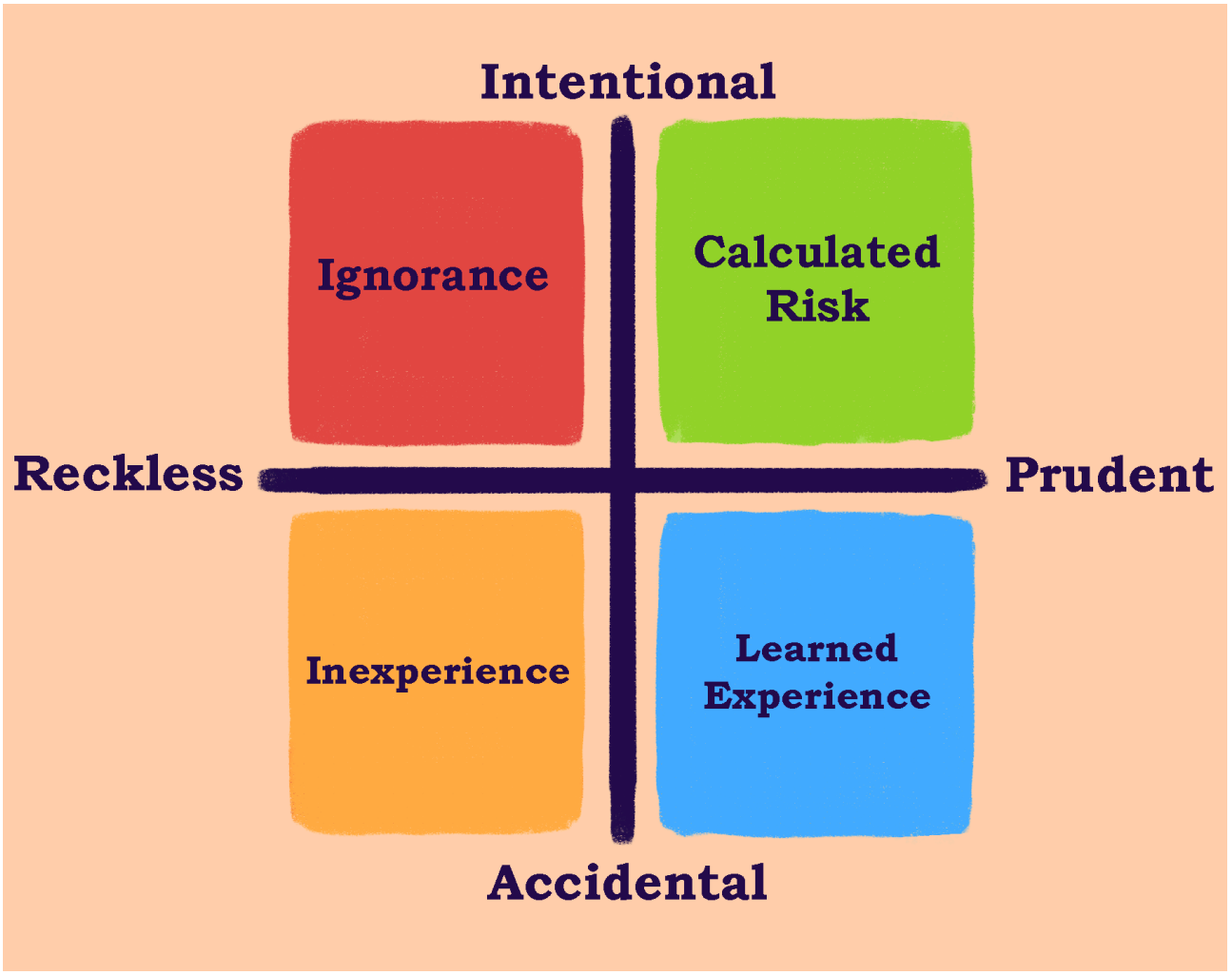
2.3 Causes of Technical Debt

“Technical debt is not a technical problem”as said by Brown (2024), who argued that technical debt represents the trade-offs of short-term benefits for increased costs in the future, where developers deliberately make the choice to accept technical debt for achieving faster development or additional features. Understanding the reason for making such decisions, gives insight into how

technical debt occurs, because each cause for technical debt is governed by a decision made by the developer, some of which are made intentionally and others are not.

Figure 2

Technical Debt Quadrant



Note. This image was created by the author. Adapted from Fowler (2009).

As shown in Figure 2, the technical debt quadrant separates such decisions into four types: Intentional, Reckless, Prudent and Accidental. The image portrays major types of technical debt as described by Fowler (2009), in his post he presented the quadrant to summarize various interpretations of debt made by other figures in the software development community. The quadrant outlines four types based on how the decision to take on debt was made. Technical debt first and foremost begins with a decision – as Fowler stated: “the debt metaphor reminds us about

the choices we can make with design flaws", a poor choice creates a poor solution, but not all choices are made the same. The decision to write everything in one script is *reckless*, because such action is very likely to lead to a lot of problems, but what makes the difference between *inexperience* and *ignorance* is whether the decision-maker knew what the consequences for making that choice would be. In another situation, developers may have spent years working on a project before a new, better solution presents itself for a design that has been established purposefully long ago. This realization indicates the debt was *accidental*, but *prudent* and has served as a lesson for the developers. These distinctions inform the developer on how their debt manifested and how it can be handled next, such as whether it highlights a gap in knowledge in *accidental* debt, or if the action was made by the team to progress on the project quicker through *intentional* debt where the risk was deemed manageable.

But what exactly causes debt? On the surface level, several common causes of technical debt can be identified, as mentioned across multiple sources, communities and individuals opinions. For example, they can include:

- Time pressure
- Shifting schedules and plan changes
- Overcomplicated structures
- Inconsistent coding styles
- Violations of design principles
- Inexperience
- Sub-optimal code
- Delaying refactoring
- Lack of testing
- Lack of documentation

However, the events listed above do not explain the full scope of the problem. Debt accumulation can be caused by both our choices and factors outside the developer's control, which may instead force a choice that was undesirable. Game development is especially susceptible to risky decisions due to the unpredictability of creative production, resulting in more opportunities for technical debt to occur.

Quick Solutions & Deadlines

The gaming industry faces time pressures that tend to drive developers to make quicker, messier design choices in order to add new features and further development (Agrahari & Chimalakonda, 2020). Especially when faced with a problem that does not have a quick and effortless solution, in which case developers may resort to easier, low-resistance solutions instead of working out a better design that would have taken longer to implement (Suryanarayana et al., 2014). In interviews conducted by Besker et al. (2018), most software developers cited time pressure as the main reason for introducing additional technical debt. This pressure affected not only the implementation of solutions but also led to reduced attention for other activities, such as testing and updating documentation.

Typically, the ideal programming process would be to quickly create a solution that works as intended, after which it can be tested and refactored into a polished state. However, there is rarely time and resources to work on a feature in detail during an ongoing project (Nemchinskiy, 2021). Because of high stakes imposed by deadlines, a developer running out of choices may be more willing to take on risks, cut corners or accommodate what is already present in the project instead of fixing the root of the problem or planning ahead. This solves their immediate problem but in reality it only postpones this issue to the future development stage where it will not be as easily solvable (Brown, 2024; Nemchinskiy, 2024).

Games are often rushed to release for specific events or dates with the expectation that once the game is shipped, any particularly detrimental bugs can be fixed later. However, the cost of patching the game after its release may be way more expensive than it would have been had those issues been addressed earlier in development. (Extra Credits, 2019)

But, a buggy game is a released game. As said by DoshDoshington (2024, 17:36): “Sometimes a quick solution is better than an intelligent one”, meaning that there are times where a fast, messy solution must be made for the benefit of production, disregarding the effects it may have in the future. When and why such an approach is taken has to be taken with consideration, else it may yet prove to have been the wrong option.

Prototypes & Feature Creep

During prototyping, the architecture of your initial concept has a major influence on the technical debt of the project. Structure, data storage, which libraries are used, or even how the project files are organized can have an impact on the project once the prototype enters production (Loster, 2023). Furthermore, game ideas often need to be tested quickly and as such prototypes are created with minimal attention given to technical debt because there is no sense in spending additional time polishing a concept that may soon be discarded.

Prototypes frequently serve as a proof of concept, it is possible to develop a game from one, however that may cause issues later on due to their quick and reckless structure. During this concept stage of game development, certain companies abandon prototypes at the start of production, while others have dedicated teams for prototyping that work separately from the rest, eliminating the possibility of messy prototype structure slowing down development. (Extra Credits, 2019)

Inexperience

Every developer has to start somewhere, and as such it is natural for newer developers to accumulate technical debt in their projects. Inexperienced programmers can lack understanding of good programming principles, design patterns, code smells and refactoring, leading them to create solutions of their own that may be overly complicated, lack quality or be difficult to read. Not only that, but they may lead their teammates into following their practices, locking the design further into a poorly made system. (Suryanarayana et al., 2014)

Chernishev (2021) claimed that junior developers tend to overuse design patterns that they have recently learned, often building the structure of the project around them – even when the patterns offer little value to the codebase. The author notes having experienced this firsthand and agrees with the notion: the excitement of a newly learned, clever solution can lead to a desire to apply and understand it in practice, which may bloat or overcomplicate the project at hand unintentionally.

Negligence

While certain debt is accumulated by accident, some developers may make the decision to take on technical debt at the wrong time or place, with the assumption that they will have the opportunity to fix it, only to realize later that they were mistaken. Teams may assume that longevity of the project is unimportant at the time of development and take on technical debt on purpose without thinking ahead (Nemchinskiy, 2024). And in other cases, teams knowingly take shortcuts without ever considering the consequences. As Fowler (2009) puts it, “A team ignorant of design practices is taking on its reckless debt without even realizing how much hock it's getting into.”

However, some decisions that exacerbate technical debt are driven by constraints placed on individuals by the environment they operate in (Brown, 2024). One of the findings reported by Besker et al. (2018) states that *"developers have a higher awareness compared to their managers of how much time is wasted due to TD"* (p. 112). Leadership in a project may prove to be ignorant of the impact technical debt may have on the project when demands are placed on the development team. They may decide that a major game mechanic that has been present in the project for a long time now needs to change entirely, or include additional features it was never designed to have. It may start as a demand in rewriting scripts for programmers, but may extend to others on the team, such as level designers or artists now being required to redo work to fit the new vision better. (Extra Credits, 2019)

Lack of Communication

Documentation is typically amiss due to developers working too quickly (Nemchinskiy, 2024). Documentation and testing are a form of communication within the project between everyone involved in the past, present and future of the project. New people filling positions have no prior knowledge of its development and require time to get accustomed to the project. The absence of tests, documentation or design documents and failure to update them leads to confusion and guess-work, requiring additional time to familiarize with the project and how it functions.

Technical debt can occur when multiple changes are made over a long period of time and the original intention has been lost to time or change in developers who were in-charge previously. However, old code is not necessarily something that should be discarded. Someone’s bad code acts

as the only source of information on how a system works. Old code should only be deleted once it has become clear that this piece of the code is unused and no longer needed in the project. (Nemchinskiy, 2022, 2024)

While many types of technical debt incur a relatively stable or predictable cost over time, some give rise to **vicious circles**—self-reinforcing loops where the consequences of technical debt lead to further accumulation of debt. As the debt causing the cycle continues to remain in the system it incrementally increases additional technical debt surrounding it, which worsens the impact of the debt on development speed and greatly increases the costs of solving the debt (Martini & Bosch, 2015). Existing technical debt in the system can give rise to such cycles.

The list of causes for technical debt is expansive in detail, though it is summarized by what decision has been made and how bad or good the consequences for making that decision are. Therefore, practicing prudence elevates the quality of the project and may safeguard the developers against the worst effects of technical debt.

2.4 Manifestations of Technical Debt

Code design has a major impact on software quality. The root of technical debt is frequently found in poor system structure. The design behind every software project often stems from established practices that have been proven to be effective in development, referred to as design principles and design patterns.

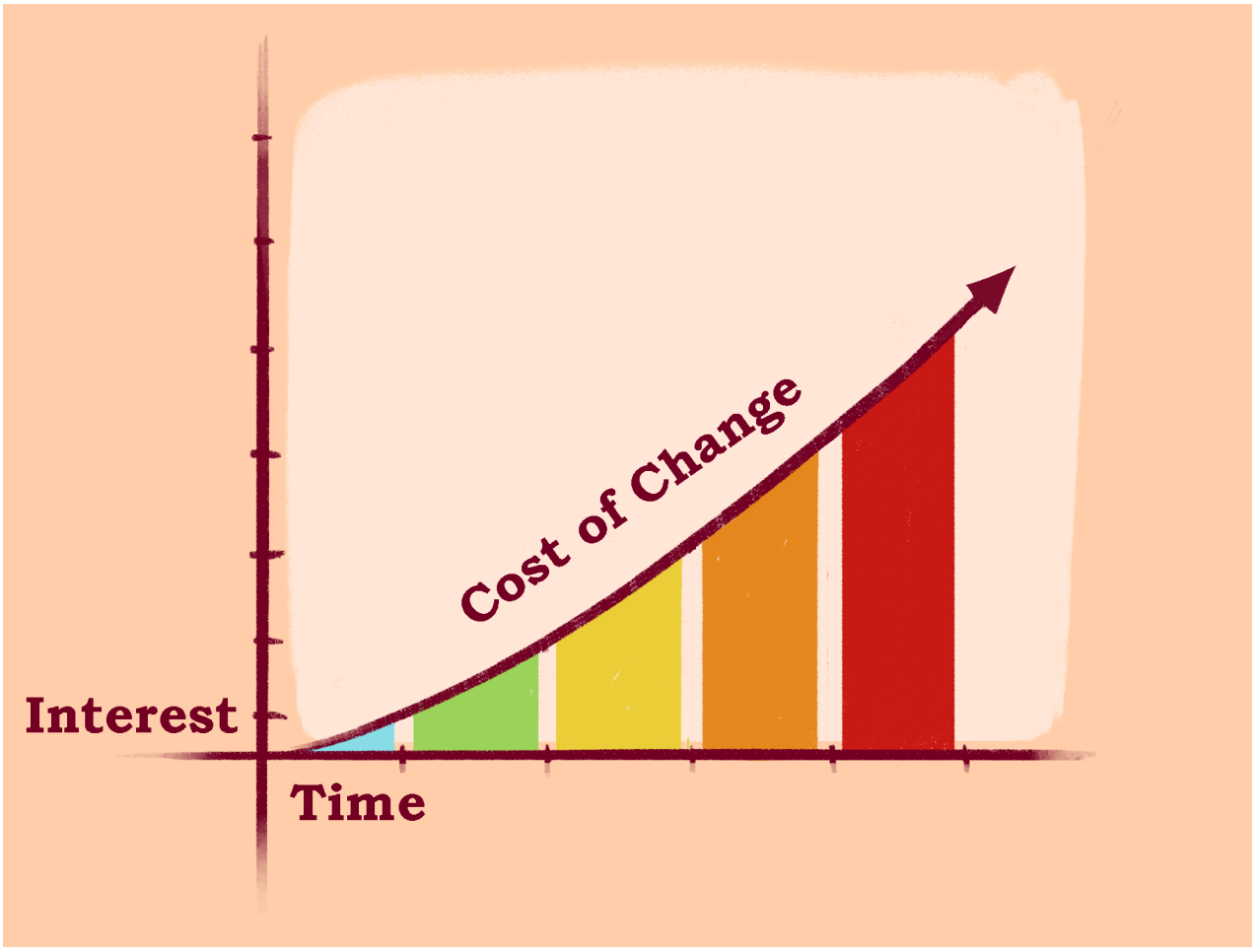
Software has a tendency to build up deficiencies in internal quality (Fowler, 2019). Every piece of code grows in complexity over time due to new features, better technologies or smarter solutions to past problems, becoming fragmented. The longevity of the software is directly influenced by past decisions made in the design of core systems in the project which dictate how much that complexity degrades the code quality. Pizka (2004) claimed that “change is inescapable” and that the accumulation of such changes decreases software quality over time, typically due to time pressure or budget limitations.

On the same note, Farley (2022, 11:22) argued that “systems always getting worse is wrong”, and instead emphasized that software never remains static or reaches true completion; rather, it gains

complexity over time and needs to be maintained to preserve its quality. He also stated that development frameworks that focus on creating or testing new features tend to see more technical debt due to prioritizing those features over clean structure. Similarly, game prototypes prioritize creating and experimenting with various game mechanics, where refactoring would be impractical as mechanics are changed, swapped out or discarded during the design of the game, causing them to potentially establish technical debt before development even begins.

Figure 3

The cost of change raising as interest from technical debt increases



Note. This image was created by the author. Adapted from Beck (2000).

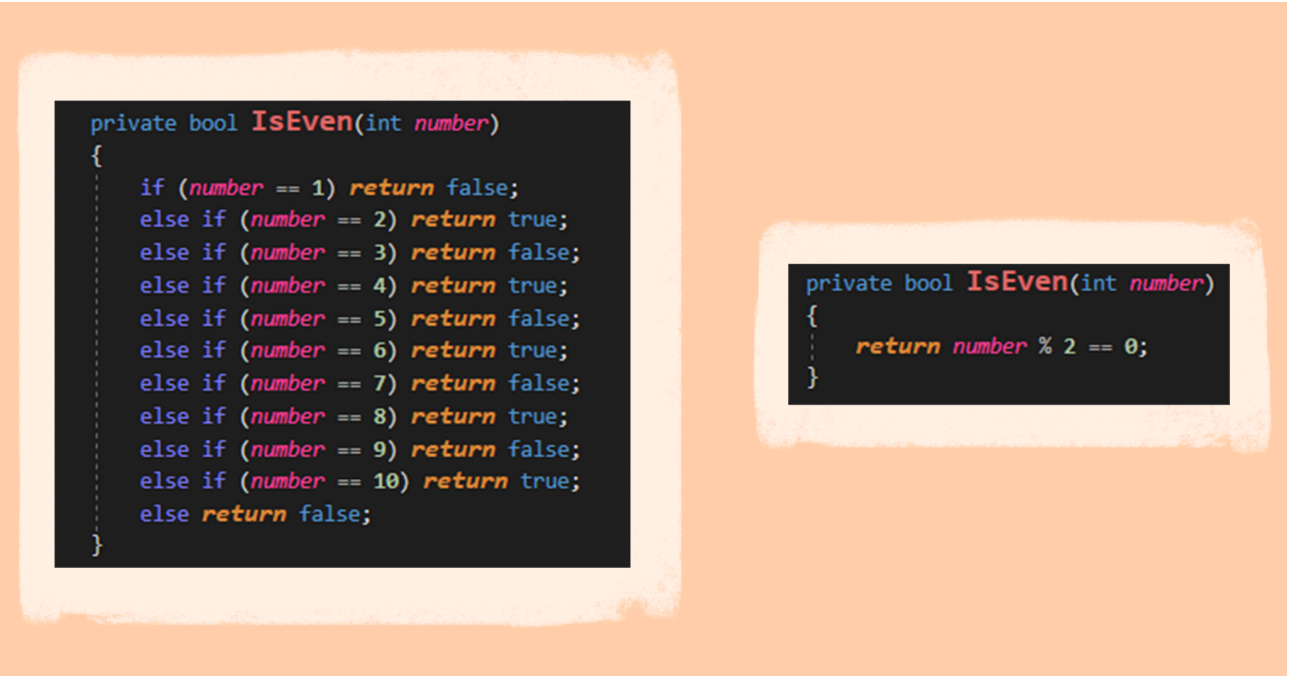
The cost of change increases exponentially as development continues. Addressing the defects made in the project prevents those defects from contributing to technical debt, the sooner a problem is detected and solved, the less it costs to make that change in the long-run. Allowing issues to add-up in the system causes the cost of change to increase dramatically and fast, by keeping technical debt low the project can develop more efficiently. (Beck, 2000)

Code quality is determined by how easy it is to read, understand and modify a given system. Design choices can increase or decrease the quality of the code, depending how well these decisions were executed. Choices that undermine the developer's ability to use and manipulate the project indicate poor code quality, and in turn, changes that elevate the maintainability and clarity of the codebase are a sign of quality programming.

"Spaghetti code" is a common descriptor for a wide variety of flaws in a system, highlighting the messy and tangled nature of bad code. The majority of such code is represented through flaws that appear minor, but can escalate to a debilitating degree. Take for example the code in Figure 5, while on its own that snippet of bad code is harmless, more issues start to arise once that system needs to be expanded. These types of errors typically occur due to laziness, neglect or from inexperienced developers. In relation to technical debt, spaghetti code can be considered its worst offender – the code that clearly should have a better, cleaner solution, yet it has been ignored or missed.

Figure 4

Two versions of a function that achieve the same result



```
private bool IsEven(int number)
{
    if (number == 1) return false;
    else if (number == 2) return true;
    else if (number == 3) return false;
    else if (number == 4) return true;
    else if (number == 5) return false;
    else if (number == 6) return true;
    else if (number == 7) return false;
    else if (number == 8) return true;
    else if (number == 9) return false;
    else if (number == 10) return true;
    else return false;
}
```

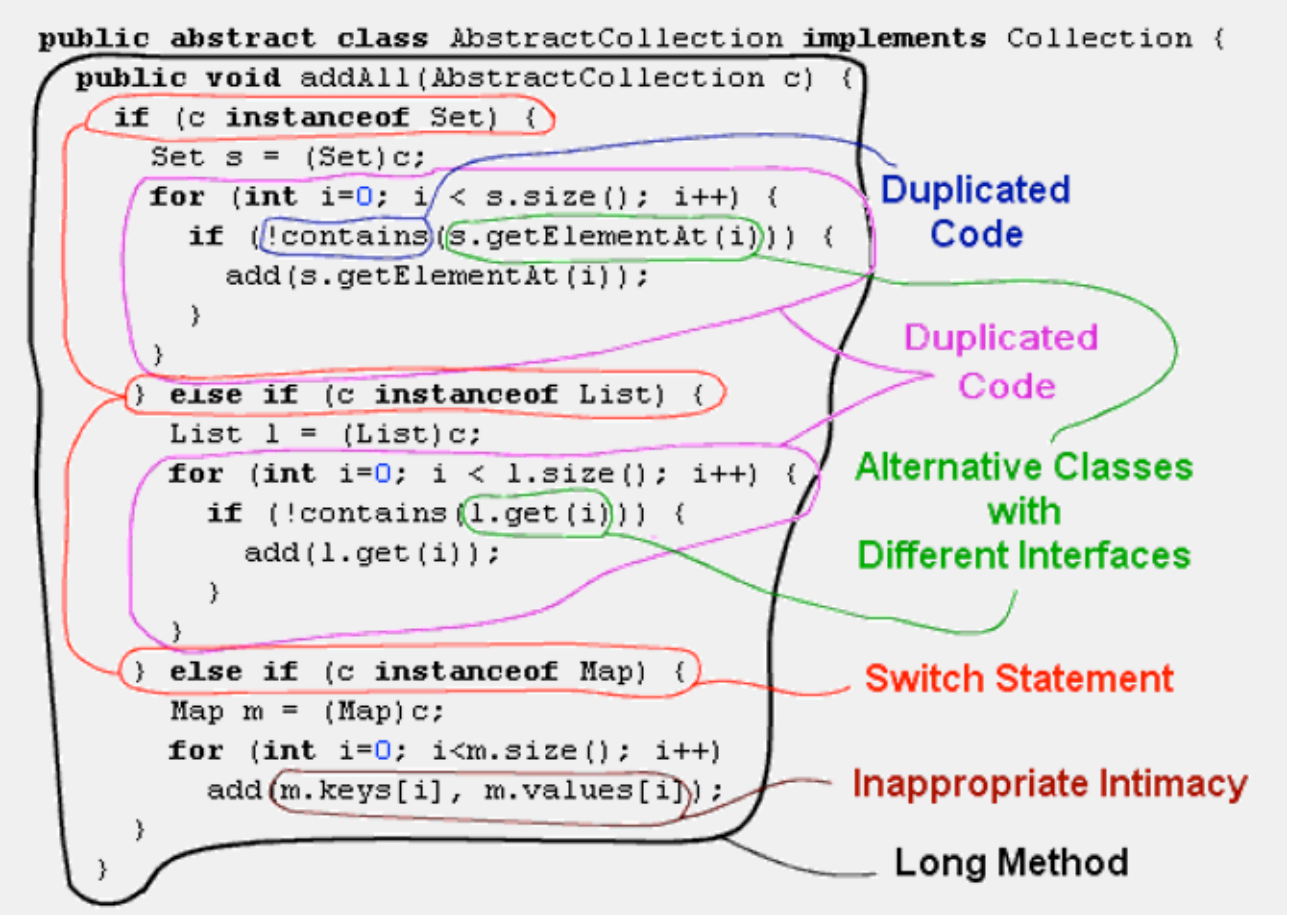
```
private bool IsEven(int number)
{
    return number % 2 == 0;
}
```

Note. This image was created by the author. It shows a large block of code on the left that is overly complicated for its purpose, and on the right is the refactored version that shows that much less work was actually required to solve this.

The next form of technical debt that will be discussed heavily ties to design principles, the common programming practices used by developers. **Code smells** are noticeable indicators of a possible underlying problem in the code, typically caused by poor design applications (Fowler, 2006). Code smells highlight technical debt in the system where design has undermined the quality of the software. Smells frequently manifest in groups, they can affect or amplify the presence and creation of other smells by limiting the design decisions that can be made in the project that contains a lot of technical debt. (Suryanarayana et al., 2014)

Figure 5

Example script with multiple code smells



Note. This image shows various design violations that contribute to technical debt. From "Design Discussion: Code Smell," by A. Bhattacherji, 2024, ITNext (https://itnext.io/design-discussion-code-smell-bb55b19eb1a8). Source of image unknown.

The codebase is not the sole contributor to technical debt, one of the key elements of technical debt is difficulty understanding the project. Loss of information leads to such gaps in understanding, such as when a developer leaves the project with little to no documentation, comments or testing tools, the part of the codebase they had worked on becomes harder to navigate. Every developer needs time to comprehend, learn and conceptualize the system at hand to be able to work on it, and when that process is made difficult or has to be repeated much time is lost on managing information about the project than is spent on actually developing the project. Additionally, inconsistent or undocumented decisions or clarifications made verbally or through informal channels contribute to the constraint on knowledge surrounding the project, limiting it to individuals.

From the game development standpoint, information changes frequently due to mechanics being scrapped or reworked, new ideas flowing in or entire sections of the game being completely rewritten. The practicality of documentation, tests, planning and communication within the project varies on the stage and goals of development, it is likely that information recorded becomes outdated quickly, adding to the issue rather than solving it. However, it can be argued that some information is better than none, as complete blindness to the inner workings of the game destabilizes the organization of the project for everyone involved. During prototyping and pre-production, flexibility often outweighs structure, but as the project matures, a lack of up-to-date documentation and communication becomes a liability.

To reiterate, technical debt manifests as a flaw in code structure or design, caused by mistakes or misuse of common practices made either deliberately or accidentally. Bugs, however, do not fall into the category of technical debt because they represent unintentional behaviour and flaw in the *logic* of the software's functions, not the system itself. By extension, technical debt encompasses more than the code of the software, it is an issue that affects the project's organization at its core.

2.5 Why This Matters

Vidoni et al. (2022) presented technical debt as an inevitable occurrence of the software development process, which will continue for the entirety of development and maintenance of the software. It is something that needs to be considered from the beginning and throughout every step of the project, because it will eventually have an effect on the future state of the software or game. At its worst, the codebase can become so convoluted that it is impossible to neither comprehend its functionality nor identify and resolve bugs (Nemchinskiy, 2021). The longer the debt is left alone, the more daunting the task of refactoring the system becomes, exponentially increasing technical debt that may eventually run the entire system to the ground and be impossible to update without heavy costs.

To work on preventing technical debt is to organize and plan for the future which keeps your development smooth and effective. Certain game companies today race to deliver games at fewer costs which serves to lower the quality of the game as compromises are made for faster releases. That is because technical debt does not remain a self-contained problem within the development team, there are multiple cases in which a game's success suffers from technical debt, a few of which will be discussed next.

Games That Failed Due to Technical Debt

First of all, what defines a "failure" in the context of this thesis is any game studio that has had to cancel, delay, suffer additional costs, lose popularity or experienced anything else that harmed the profits or cultural impact of the game or development studio as consequences for unmanaged technical debt.

A primary aspect of technical debt as mentioned in previous sections is poor software quality. The aftereffects of code negligence can cause various issues in a project as can be explored in these examples. For instance, dyc3 (2020) conducted a video code review of *Yandere Simulator*, a game infamous for its sole developer's poor code base. The review highlights that the game's scripts are long and difficult to read, with thousands of lines dedicated to a single if-statement, which is executed every frame by hundreds of in-game objects. The developer of the game has rejected help on multiple instances, preferring to maintain the state of the project as is, which has had a

major impact on the performance of the game and severely stalled its development. This negligence harbored a highly negative reputation for the developer and the game, many using Yandere Simulator as an example of terrible code.

Another example of technical debt complicating development is *Kerbal Space Program 2* (KSP2), the sequel to a widely acclaimed game, which was developed under intense time pressure in pursuit of an overly ambitious vision. According to ShadowZone (2024), who interviewed multiple individuals involved in the game's production, developers were instructed to reuse the original game's engine and codebase to meet a two-year development timeline. This management decision to build on top of flawed, legacy code without proper refactoring led to extensive bugs, incompatibilities, and significant delays. Compounding the issue, at the start of the project none of the developers were familiar with how the original Kerbal Space Program functioned—further complicating the project and exacerbating the existing technical debt. The game was later released in Early Access in 2023 after several delays in an incomplete and unstable state. The lack of quality in the game caused an outrage among fans, resulting in a high rate of refunds. In the end, the studios behind KSP2 were shut down, following the downfall of the game. The root cause of the game's failure can be tied to multiple mistakes made throughout the development by management of the project, whose oversights worsened the technical debt of KSP2, creating a difficult working environment. It is likely that had the developers been allowed to refactor the project properly as they wanted to, the development could have proceeded much smoother with fewer delays.

Games That Succeeded Despite Technical Debt

Not every game that struggled with technical debt "failed", in many cases it may have been necessary or inconsequential to the success of a game. Every game faces technical debt and needs to work around it to continue development, but the way that debt is approached and handled determines whether the game continues to be a success despite it.

Older titles tend to experience a lot of technical debt in the form of **legacy code**, code that was created a long time ago that has become outdated, but is essential to the inner workings of the project. All codebases eventually become legacy code, and if that codebase was poorly maintained, it may prove to be an obstacle to future content for the game, should the developers ever decide to expand on it.

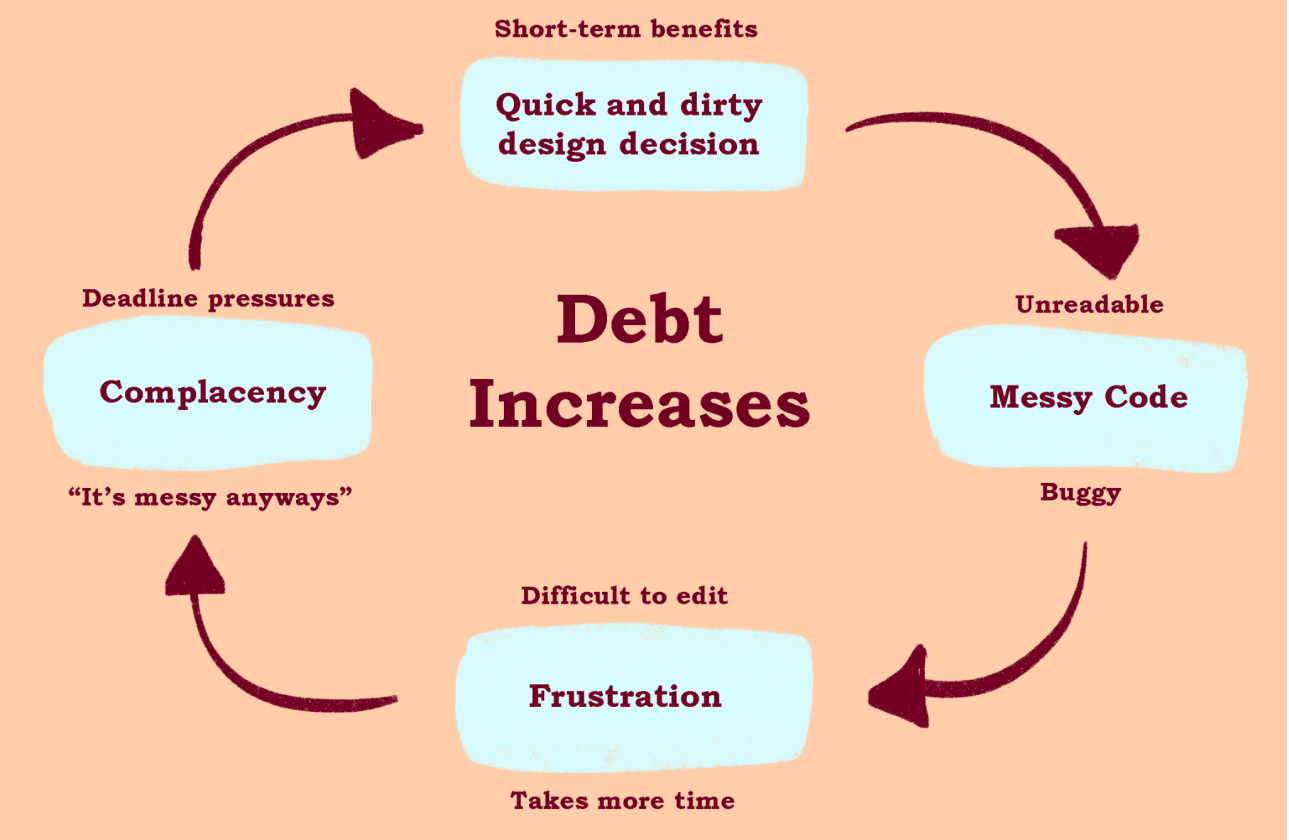
World of Warcraft (WoW) is a large, well-known MMORPG that continues to be played after decades since its release in 2004, with multiple expansions keeping the game alive for all these years. In recent years, WoW has been creating new expansions and revisiting older ones and such an old game has much technical debt built up through the years in its legacy code. One of the issues with old code comes from decisions that were made years ago by people that could never predict how technology would evolve or continue to be used a decade past, requiring developers to update and refactor legacy systems to pave the way for newer, better technologies. (Newman, 2020)

In another example, *Team Fortress 2* (TF2) reuses legacy code from the Source engine, which was originally tailored to a previous game from the same developers. In his video, shounic (2024) explains the cause and absurdity of a bug-fix recently made by Valve. The bug occurred due to an old, unused check in the logic that tried to find “Gordon Freeman”, the player character from the Half-Life series, and when that check failed (because that character does not exist in TF2), the game would crash (Valve, 2024). This is a check that has been left in the game for over 10 years since its release. While this type of bug is relatively harmless, it highlights that the Source engine contains technical debt that occasionally affects the updates of games built on it.

And yet, these games are popular, played and modded to this day. This shows that while technical debt can cause huge setbacks for the development and continued presence of the game, it occurs in every game and its effects can be lessened through proper management and future-thinking.

Figure 6

The cycle of technical debt



Note. This image was created by the author. It empathizes that existing technical debt can lead to further debt down the line as it gets ignored in favour of project progression, deeping the issue with each new feature.

Effect on the team

Technical debt primarily affects the team and the system they work with, it is in the developer team's and company's interest to have a smooth development process. Research conducted by Besker et al. (2018) suggests that six activities account for the majority of developer actions in response to technical debt: additional testing, code analysis, refactoring, searching for documents, work-arounds and communication. By maintaining clean code, sufficient documentation and refactoring consistently, the project can continue its development at a steady pace. Eliminating uncertainties and improving workflow. (Suryanarayana et al., 2014)

Complacency. When defects are introduced into the system, it makes the system more difficult to work with and the team may grow hesitant to make changes, preferring to add features wherever possible with little regard for structure. The more defects accumulate and are ignored, the worse the debt. New changes become exponentially difficult to implement with increasing interest and may even encourage or require developers to continue making sub-optimal decisions, because the code is nearly impossible to understand and edit already. This ultimately leads developers to make the same decisions in the future, increasing the debt of the system (Suryanarayana et al., 2014; Farley, 2022). Tight schedules and deadline pressures may have programmers resort to quick and easy solutions to get the job done in time. The system may function as intended and show no glaring issues, hiding the underlying problems and postponing them for later. However, because the consequences of those quick solutions are not evident yet, the decisions made feel worth it and extend the issue, creating confusing code that can break at any small change made at a later date. Companies want to get their product on the market sooner and at a reduced cost, that leads to an expectation on the development team to work faster, but this pressure leads them to make worse design decisions than they would make normally. Small poorly made design choices add up and as development continues, these issues become more prominent and degrade the overall quality of the software. (Suryanarayana et al., 2014)

Frustration. Poorly written code is not only difficult to work with, but it is also frustrating. “Constantly working with dirty code may reduce team morale, reducing the level of happiness in the programmers (and by extension the wider team)” (Roberts, 2020, p. 6). It directly affects the developers work ethic and their motivation. As technical debt increases with it increases the difficulty and frustration of the working environment of the team. Difficult systems stall production as developers struggle to make sense of their mess and not lose their minds over it, leading to stress and lost time (Suryanarayana et al., 2014).

Effect on the game

Community. When a game is released with poor framerate, unmanageable crashes and unavoidable bugs, the player base naturally becomes frustrated or even outraged, as seen in game releases like Kerbal Space Program 2. It not only frustrates players, but also damages the reputation of the developers. Studios that fail to release a stable game and are unable or unwilling to resolve their technical debt to improve their game lose their players’ trust, which can affect the

success of future titles. Additionally, a poorly developed game that upsets the players may cause outclash that hurts the morale of the team behind it. Among other things, modding communities are also affected by technical debt. Players often create their own content for the game through mods, which can keep the game alive and active for many years after its release. A game that is difficult to mod is unlikely to gain a large modding community, which may subtly hurt the lifetime of the game, and if not, it is still likely to frustrate the player base.

Development Cost. Besker et al. (2018) found that software developers reported an average of 23.1% of their development time being wasted due to technical debt, with significant variation between responses. The increased time spent managing neglected technical debt amounts to bigger cost of development and slower releases. It diverts resources from updates or features that companies want to deliver soon, but are forced to dedicate time to reorganizing and patching the game instead. Unsolved debt continues to linger in the project for the entirety of its lifespan, which impacts what the future of the game might be.

2.6 Sustainability: What Happens After

Software systems grow in complexity over time and require constant maintenance to remain sustainable over a long period of time, as Venters et al. (2018) state: “Sustainable software requires a solid foundation that allows efficient and effective maintenance and evolutionary change over its entire life-cycle”. They state that design decisions can drastically affect software architectures, which serve as the foundation of the project, leading to lower software quality and increased costs of development. These challenges are present in game development and will have various levels of consequences as a result of poorly maintained codebases.

Technical debt is closely tied to a project's overall sustainability. Because time and resources are limited, decisions made during development directly influence long-term outcomes. A project with unmanaged technical debt is likely to face increased development times, inflated costs and reduced maintainability – undermining the **economic** sustainability of the project. Studios may resort to focusing time and resources towards patches or updates after the release of a game, which are harder to deliver due to the unstable state of the project, further increasing costs.

A project cancelled due to the inability to continue it without greater costs represents a significant loss of time and effort, and possibly investor fundings and team morale. Stressed, demotivated developers may experience burnout and contribute to higher turnover in the industry as a result, which shows implications for **social** and **individual** sustainability.

Lastly, code quality affects the **technical** sustainability of the project, poorly structured systems are difficult to maintain and update, which limits the game's potential, particularly – its longevity.

Longevity

The quality of the structure of the game lends to its potential lifespan. Flaws in implementation of game mechanics harm the game experience, such as gaps in game logic allowing the player to break or cheat the game (Agrahari & Chimalakonda, 2020). But while most bugs that affect gameplay can be fixed, structural issues stemming from technical debt are much harder to deal with, leading developers to leave some problems in their games unresolved. These issues may lead to poor stability, loss of community engagement and modding support.

However, indie games, prototypes and other small titles often have little need to be carefully crafted if the developers expect to release a game and leave it behind quickly. Therefore, how much attention should be paid to the quality and organization of the game project should be proportional to the expected longevity of the game.

Legacy Code

Any project that has been in development for several years is expected to be messy due to the codebase naturally deteriorating through issues accumulating over time, no matter how effective the development team were at managing their technical debt. Improper management, however, can affect the future of the project, such as new features or expansions being too difficult to implement because the present legacy code had gone on poorly maintained, being essential yet posing a significant obstacle to any changes. For example, porting systems and tools can prove to be a challenge due to legacy code, old rendering systems encounter compatibility and optimization issues when adapted from older technologies, requiring additional work and cost (Pedroso, 2025).

Legacy code creates an environment in which developers are hesitant to act and may be forced to implement quick and hacky solutions to continue development, because the state of the project is too big, too difficult or too unknown by current developers. A finding reported by Besker et al. (2018) claims that a quarter of technical debt encounters reported by developers in their study forced the introduction of technical debt in addition to the debt already present in the system, causing what they call **contagious debt**. As a result, any technical debt that is unmanaged contributes to the complexity and poor quality of legacy code.

How this affects games is seen in older titles such as previously mentioned World of Warcraft and Team Fortress 2, among any other games that continue to be popular today years after their release. Namely, online games, which aim to continuously bring new experiences to the game and require continual updates, are at risk of straining due to unmanaged technical debt in their legacy code, which Borowa et al. (2021) claimed to experience different and possibly more significant consequences from technical debt due to being sold as an online service.

Developers

Unmanaged technical debt can lead to not only cancelled games, but entire studios and companies falling apart. It primarily slows development and impedes implementation of new features, which has a ripple effect on the cost of production that may ultimately end in the worst case scenario.

Additionally, burnout as a present issue in the game industry can affect many developers, and technical debt may serve as one of the causes for it. Working in a project affected by it has been shown to not only be frustrating, as findings from the study conducted by Jesper et al. (2021) shows, but also detrimental to the project. Developers in the study tended to be reluctant to work on projects with high debt, experiencing anxiety in relation to it and jeopardizing task prioritization which caused projects to stall or lose focus.

Environment

Although the consequences of technical debt on the developers and the game are evident as discussed in previous sections, the impact of technical debt may extend beyond development challenges and into environmental concerns. Agrahari and Chimalakonda (2020) suggested that poorly optimised games contribute to more power consumption and encouraged developers to reduce power usage in their games. By extension, efficient development practices that minimize

delays and prioritize managing technical debt could help reduce excess energy consumption and thereby lower the environmental impact of gaming. And with the growing use of AI in modern games as indicated by Mason (2025), unoptimized AI usage and poor planning may require retraining models, something that could be mitigated by careful debt management, reducing development time and overuse of resources that can have impact on the environment.

2.7 Managing & Mitigating Technical Debt

Addressing technical debt is an ongoing task throughout software development, because it is not possible to eliminate it for good. Suryanarayana et. al (2014) claimed that it is impossible to entirely avoid technical debt, only manage it.

Managing technical debt involves two primary goals: prevention and repayment. When design decisions are being made, they must be made with the goal of minimizing the increase of technical debt to prevent future workload from becoming difficult or overwhelming. Then, these decisions and the technical debt that accompanies them must be tracked and paid off periodically through refactoring, documentation, mentoring and better practices. Good methodologies, code styles and code reviews are all solutions that reduce technical debt. Taking additional time to clean and organize code and assets can significantly improve the structure of the game project and benefit the workflow. (Suryanarayana et al., 2014; Extra Credits, 2019)

In his book, Roberts (2020) makes an argument that writing clean code reduces the workload on the person reading the code. Understandable code allows the reader to make changes much quicker and easier, something that is in the interest of both the development team and the users of the software. However, he also questions the requirement of clean code for software that is not expected to be accessed again, especially if said software is already carrying out its purpose effectively.

Refactoring

Tripathy (2014) defines refactoring as “performing changes to the structure of software to make it easier to comprehend and cheaper to make subsequent changes”. He expresses that the purpose of refactoring is to maintain the readability, understanding and structure of a project. Easy

comprehension of the system allows the programmer to make assessments, predictions and problem-solve more efficiently.

In contrast, Loster (2023) argues that refactoring is not always necessary, because at some point, the time it would take to refactor a system may be equivalent to the time it would take to rewrite it entirely. Premature optimization and technical debt is a delicate balance, while messy code brings all sorts of difficulties with it, an overly optimized game is unlikely to be ever released (DoshDoshington, 2024). Games such as Undertale, Celeste, Project Zomboid, Cyberpunk 77 are all examples of games with poor code design decisions that baffle onlookers – and yet these games are very successful.

According to Nemchinskiy (2022), refactoring bad code is best done when new functions are not possible to add without refactoring part of the code, if there are bugs that cannot be solved without refactoring, or if the developer understands the codebase and is ready to remake it into a usable state. He follows “the rule of three tries”, if a piece of code needs to be edited once, twice, on the third time it becomes clear that this section of code is due for some refactoring at large. As such, refactoring is a valuable investment, but requires careful consideration to avoid doing it unnecessarily.

Design Principles & Patterns

There are numerous design principles and patterns known and practiced within programming circles. In his book, Chernishev (2021) discusses a wide variety of these patterns and principles, such as SOLID, KISS, DRY, YAGNI, GRASP and GoF Design Patterns. Each acronym encompasses a pattern of principle: set of rules, guidelines or recommendations that assist development. For instance, SOLID is an acronym of five common design principles:

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

SOLID, among many other design principles and patterns used in game development, is grounded in the object-oriented programming (OOP) paradigm. As discussed by Blaschek (2012), OOP emphasizes organizing code around objects and the relationships between them where different kinds of objects represent different kinds of data. While the objects themselves appear simple at the surface level, complexity quickly rises as various objects containing their own structure, data and behaviours begin to add up.

OOP covers four core concepts: Encapsulation, Abstraction, Inheritance and Polymorphism. The purpose of encapsulation in OOP is to group and combine functions and variables into a reusable unit that can be exchanged with one another without disrupting other parts of the system. Abstraction focuses on hiding details that are not needed for using part of the system, leaving only the most necessary elements available for manipulation. By simplifying the system, it reduces the side-effects of developers making changes to the code. Inheritance allows you to eliminate redundant code, allowing similar classes to share required functions and data. Polymorphism works to simplify long if-else statements or switch cases by defining functions with custom parameters, which allows the system to run regardless of which class is making the call and with what. (Hamedani, 2018)

Chernishev (2021) describes the purpose of design patterns as a tool used to provide a general solution to frequently encountered problems that typically require similarly structured solutions, hence the name – patterns. Design patterns carry out a specific purpose in the code, such as the Factory pattern that handles creation of classes, or the Singleton pattern that acts as a global access point for the project. Patterns are partially built on object-oriented programming and rely on pattern-based coding found in object-oriented languages, like encapsulation, abstraction and more. They are tools that improve the efficiency and quality of a project, as well as teaching developers new skills. By creating common solutions to problems, other developers that are familiar with those patterns may be able to recognize them, making their job easier and more organized (Lasater, 2006).

Patterns and principles can be thought of as tools in a toolbox – when faced with a problem, instead of engineering a solution single-handedly with materials at hand, a tool specifically tailored to that problem can be applied instead.

However, patterns are not the ultimate solution to every problem, Chernishev (2021) empathizes that patterns do not need to be used in all situations due to each project being unique, making it unnecessary to fit a pattern where it does not belong lest it leads to creation of anti-patterns.

Anti-patterns are the inadequate attempts at utilizing design patterns which result in creating more problems than they solve. They occur when a design pattern is wrongly applied to a given situation, typically through inexperience or misunderstanding of the pattern. While the solution presented by the anti-pattern has a negative impact on the project it can be refactored into a better solution that uses the pattern appropriately and serves as a learning experience for the developers. (Brown et al., 1998) Game engines may contain anti-patterns that can be inherited during development by the game itself (Agrahari & Chimalakonda, 2020).

Planning

Fowler (2019) finds the metaphor of technical debt appealing, because it changes how developers think about code deficiencies. Instead of making predictions on the value of refactoring versus adding a new feature in a poorly made system, it can be treated the same as financial debt and paid off slowly, allowing teams to plan out which debt takes priority. As a result, documented debt becomes part of the project (Nemchinskiy, 2022). Then, time can be allocated to mitigating technical debt, which in turn will decrease the impact of the debt in future changes.

Awareness

One of the most effective ways of managing technical debt is through awareness and understanding. Developers cannot predict every issue that will arise in a project, but by recognizing signs and common causes of technical debt allows the team to work around it more effectively through better planning and designs (Extra Credits, 2019). Whether the concept is understood through the metaphor of “debt”, or learned through experience, the key is that knowledge allows developers to strategize and make better decisions during development. When programmers, designers and management alike understand the underlying nature of technical debt, they are more likely to make informed decisions that balance short-term benefits of technical debt with long-term project sustainability. (Suryanarayana et al., 2014)

Importantly, tracking technical debt is the shared responsibility of the team as a whole. Project leaders must actively recognize its significance and allocate time towards managing it. When technical debt is identified, it must be acknowledged soon, ensuring it is documented and visible, so that it may be solved at an appropriate time. Treated this way, technical debt represents part of the development process that needs to be addressed, allowing the team to better anticipate and plan towards long-term challenges. (Nemchinskiy, 2024)

Ultimately, one of the simplest but effective solutions to mitigating technical debt is through education. When developers are taught what technical debt is, how to recognize it and respond to it, they are better prepared to avoid preventable pitfalls and contribute to sustainable, long-term success of a project.

Games are a chaotic environment that demands frequent changes, making it crucial for developers to acknowledge ways the development process can be improved. However, even the worst consequences of technical debt can be negligible at the release of a game in the right situation, with the right decisions. Smooth and successful production can be achieved by developers that are able to establish balance through predicting, planning and strategic decisions surrounding technical debt in game development.

2.8 Information Retrieval

Sources searched for on: Janet Finna, Google Scholar, Theseus, ChatGPT, YouTube

Search terms:

- Legacy code in game development
- Technical debt game development
- Spaghetti code in game development
- Refactoring game development
- Rewriting code in game development
- Code smells in game development
- Technical debt maintenance in game dev
- Code maintenance in game dev
- Техдолг в геймдев (Techdebt in gamedev)
- Технический долг в геймдев (Technical debt in gamedev)
- Development hell in gamedev
- [Game title] technical debt
- Games that failed to technical debt
- Game technical debt

- Gamedev technical debt

Material types found: Articles, journals, conference proceedings, books, videos, blog posts, theses, devlogs.

Criteria for selecting sources: Topic relevance and adjacent concepts (software engineering, game development and existing games), credibility (academic information takes priority over blog posts etc.)

Key findings: Majority of technical debt research is done on Software Engineering, not Game Development, though both share similarities.

Challenges: Difficult to find relevant game development-centric research.

2.9 Source Material Analysis

Borowa et al. (2021) claim that at the time of writing their article, no research has been done on technical debt from the perspective of game development. Additionally, Agrahari and Chimalakonda (2020) note that facets of technical debt such as anti-patterns and code smells are not well documented within game development, despite multiple studies having been made on the subjects. They state that the study surrounding games focus on topics such as psychological, social or educational factors of games, rather than the software quality.

The majority of research, thesis, articles and books found on the topic of technical debt covers it from the perspective of general software engineering, with far fewer talking about it in the context of game development. The material was reviewed from the side of game development, which brought some insight into what ways they overlap and differ from traditional software.

However, due to the lack of relevant research, much of the writing in the knowledge base of this thesis was done through a patchwork of information found in general software engineering circles, as well as covering game code reviews, news articles and design guide books to create a picture of what may actually be going on when the game industry faces technical debt. Technical debt in game development is a broader topic than what is talked about in research done on software engineering circles as it ties heavily into production and design and the experiences in both branches of development do not always match-up. That means that the value of the information found for this thesis varies in the context of video game production, but the thesis attempts to

bridge that gap and deliver information that holds value to any developer, while providing insight and experiences from the game industry.

2.10 Base concepts

Technical debt is a concept based in software development to describe the gradual deterioration of a project's codebase quality. This phenomenon can occur from a multitude of causes, such as time pressures, bad code design, violation of design principles, poor planning, inexperience and more. In game development, the concept can extend beyond programming to other roles and may describe project quality as a whole. Not much previous research exists on technical debt in games, as such the majority of sources available are catered to software engineers.

Projects that experience severe technical debt are more likely to be struggling to update, with several examples of games failing to profit due to complications from technical debt. This debt not only shows poor quality in the code, but makes development longer, more complicated and frustrating for developers, with added stress from deadlines and community backlash. This is reflected in the cost of change for each next feature, as the project accumulated technical debt, further development becomes difficult, if not impossible. However, in certain cases, technical debt may be necessary to proceed with the project, making it important to know when it is prudent to take on technical debt and how to manage it afterwards.

To prevent projects from failing or older projects from becoming too difficult to navigate after a few years of development, multiple solutions have been presented. Clean code based on design principles, patterns and practices such as SOLID help improve the quality of the codebase. Refactoring projects is effective at minimizing the effects of technical debt, although the time spent on refactoring and writing clever, clean code must be balanced against the expectations and timeline of the project, as there is no need to carefully craft the perfect system that is unlikely to be used in the future. Lastly, education serves to raise awareness of the phenomenon to aid developers have better, smoother development processes.

3 Research Implementation

The data gathered includes experiences and insights from communities of game developers and/or students obtained through a qualitative survey with multiple open-ended questions to bring out various perceptions of the topic of technical debt in the gaming industry.

3.1 Research Methods

The original intent was to use a combination of an open-ended survey and interviews for a qualitative approach, however, none of the respondents of the survey elected to continue with an interview. Nonetheless, an online qualitative survey was conducted to gather detailed information through open-ended questions. Qualitative surveys assist in identifying themes by collecting detailed responses in long-form written format, which can give reason to further expand research related to the topic, often serving as a precursor to interviews (Deakin University, 2025). In the case of this thesis, the survey aimed to explore the general impression on the term “technical debt” and opinions surrounding the concept, with the purpose to provide possible avenues of next research topics that build on it further. Quantitative data through multiple-choice and ordinal scale questions was also gathered to establish common identifiers for the purpose of analysing long-form responses.

This mixed approach was selected to explore and understand what is known and can be learned about the concept of technical debt in game development. The data collected through people’s experiences provided answers to the questions of the thesis by showing perspectives from various developers, helped explain and describe the phenomenon, and showed what is perceived by people in the industry.

3.2 Data Procurement Design

Data was collected through Google Forms survey responses from game developers and/or game development students. All participants were informed of how the data was to be handled at the front page of the survey, detailing from which date the responses would be available in the form of thesis research analysis, where it could be found and which university is responsible for the evaluation of the research.

Information gathered was stored on Google Forms and later summarized and synthesized in Google Docs and Google Sheets. Forms, Sheets Docs data will be deleted 6 months after the thesis was completed and submitted. Questions presented in the survey are available in Appendix 1. Summarized responses, tables and additional charts from the results of the survey are present in Appendix 2, 3 and 4.

3.3 Justification of Implementation(s)

Surveys are a simple way for people to communicate large ideas without the need for personal meetings for those that prefer that. Due to the target group being any game developer from various communities, the ease of access of an online survey was considered sufficient for this research to be able to reach wider audiences. Interviews were presented as an option to have an in-depth conversation on the topic, giving detailed responses that can present a more direct insight from the individuals experiencing it. However, it should be noted that no interviews were held due to lack of interest from respondents. Qualitative design aims to understand complex, subjective phenomena (Deakin University, 2025), which aligns with the broadness of the topic of this thesis and can provide detailed, valuable information on a loosely-defined concept that is not well documented within the gaming industry. Borowa et al. (2021) held similar research for their article, who contacted people within the gaming industry and software development community with questions similar to that of this thesis. Following that, a few questions for this thesis' survey were inspired by their questionnaire.

Although this research took the qualitative approach, an argument can be made for utilizing a quantitative survey instead to more easily target a large audience for a more detailed view into the relationship of different types of game developers with the concept of technical debt.

3.4 Research Ethics Review

To conduct research ethically, the following principles were applied: Participants were informed about the purpose of the study and how their responses would be utilized before they took part in the survey. A brief introduction was provided on the objectives of the research and the topic of the thesis, voluntary participation in further interviews (which had gone unused) and anonymity.

Participants had no incentive to take part, and could choose to skip questions or disclose as much or as little information as they deem appropriate.

All responses were collected anonymously and stored on private cloud services, which were then deleted no later than six months after the data had been acquired. No personal data was collected at any point. Any names or personal information gathered was omitted from summarized data.

All data was considered and reviewed carefully to avoid misinterpretation, misinformation or biases. Questions of the survey were made to be as neutral and non-leading as possible, however it is possible the responses were unintentionally affected by descriptions given alongside questions. The full set of questions is listed in Appendix 1.

4 Results

Below are summaries and charts compiled based on primarily open-ended responses gathered in the survey. Each question is marked by section number and question number (from total question amount) as it appears in the survey. The questions of the survey in the corresponding order can be found in Appendix 1.

Q1.1 - Experience with Game Development

The majority of respondents picked Hobbyist, SoloDev, IndieDev, Student or Professional as a way to describe their experience with game development. Out of the 26 responses, the choices made were:

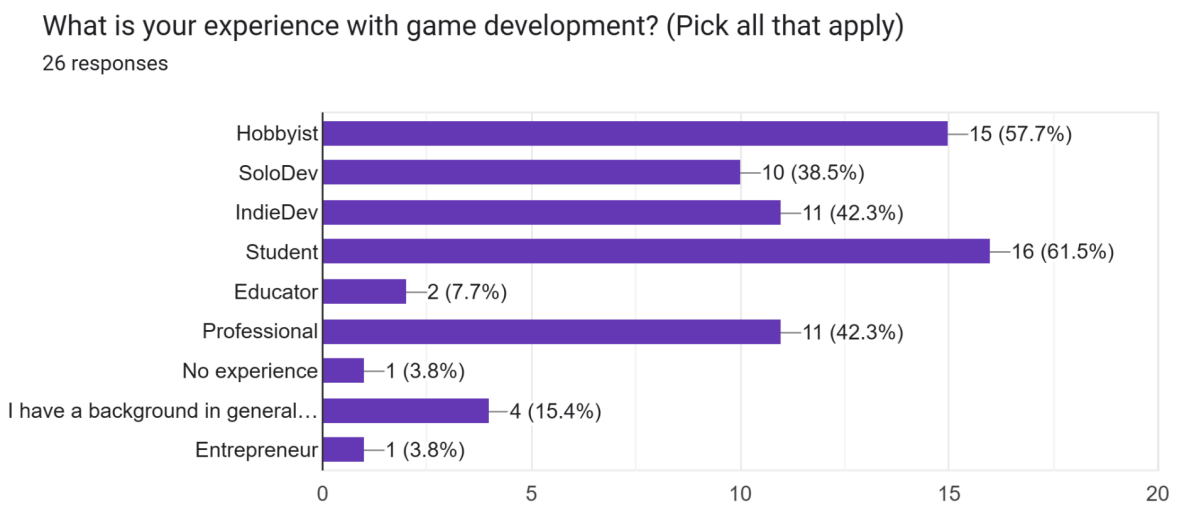
- 1 Hobbyist, IndieDev
- 1 Hobbyist, IndieDev, I have a background in general software development in addition to the other answer(s)
- 1 Hobbyist, IndieDev, No experience
- 1 Hobbyist, SoloDev
- 1 Hobbyist, SoloDev, IndieDev, Educator, I have a background in general software development in addition to the other answer(s)
- 1 Hobbyist, SoloDev, IndieDev, Student, Professional, I have a background in general software development in addition to the other answer(s)
- 1 Hobbyist, SoloDev, Student, I have a background in general software development in addition to the other answer(s)

- 1 Hobbyist, SoloDev, IndieDev, Student, Entrepreneur
- 1 Hobbyist, SoloDev, Student, Professional
- 1 IndieDev, Educator, Professional
- 2 Hobbyist, SoloDev, IndieDev, Student
- 2 Hobbyist, SoloDev, IndieDev, Student, Professional
- 2 Hobbyist, Student
- 2 Student, Professional
- 4 Student
- 4 Professional

The response amount per choice can be seen in Figure 7. One person added an additional custom "Entrepreneur" choice in addition to other selections. Almost no educators or people with experience in software engineering outside game development responded to the survey.

Figure 7

Section 1, Question 1 of the survey.



Note. Response summary chart taken from Google Forms. Respondents were allowed to pick more than one option, with the last being custom written. The option that is not fully visible says "I have a background in general software development in addition to the other answer(s)".

Q1.2 - Roles in Game Development

Total of 15 responses picked Programmer (among additional roles), with 11 non-programmers. The roles selected were:

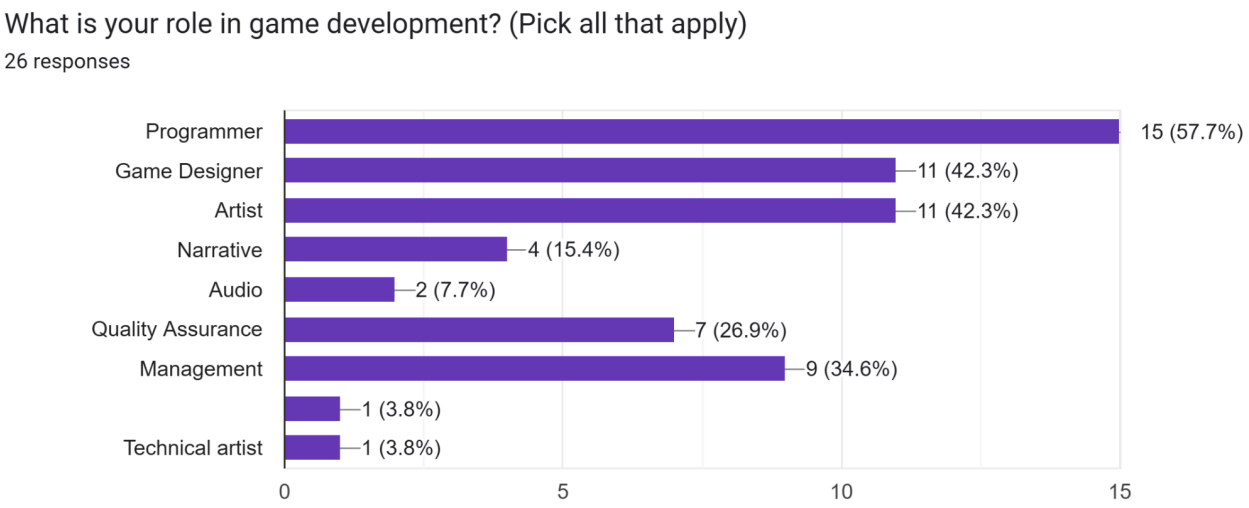
- 1 Quality Assurance, Management
- 1 Programmer, Quality Assurance, Management

- 1 Programmer, Quality Assurance
- 1 Programmer, Management
- 1 Programmer, Game Designer, Narrative, Management
- 1 Programmer, Game Designer, Management
- 1 Programmer, Game Designer, Audio, Quality Assurance, Management
- 1 Programmer, Game Designer, Audio, Quality Assurance
- 1 Programmer, Game Designer, Artist, Narrative, Quality Assurance, Management
- 1 Programmer, Game Designer, Artist
- 1 Programmer, Game Designer
- 1 Game Designer, Artist, Narrative, Quality Assurance, Management
- 1 Game Designer, Artist, Management
- 1 Game Designer, Artist
- 1 Game Designer
- 1 Artist, Technical artist
- 1 Artist, Narrative
- 4 Artist
- 5 Programmer

Total amounts per role are seen in Figure 8, with one respondent adding “Technical Artist” as a custom role in addition to other selections. Nearly no narrative or audio inclined developers responded to the survey.

Figure 8

Section 1, Question 2 of the survey.



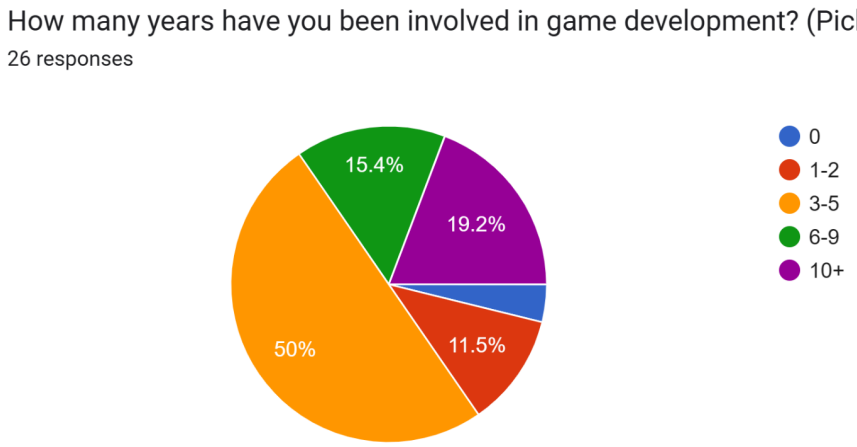
Note. Response summary chart taken from Google Forms. Respondents were allowed to pick more than one option, with the last being custom written.

Q1.3 - Years of Experience

According to the results in Figure 11, half of the responses gathered claimed to have 3-5 years of experience in game development. With only one response claiming to have 0 years of experience.

Figure 9

Section 1, Question 3 of the survey.

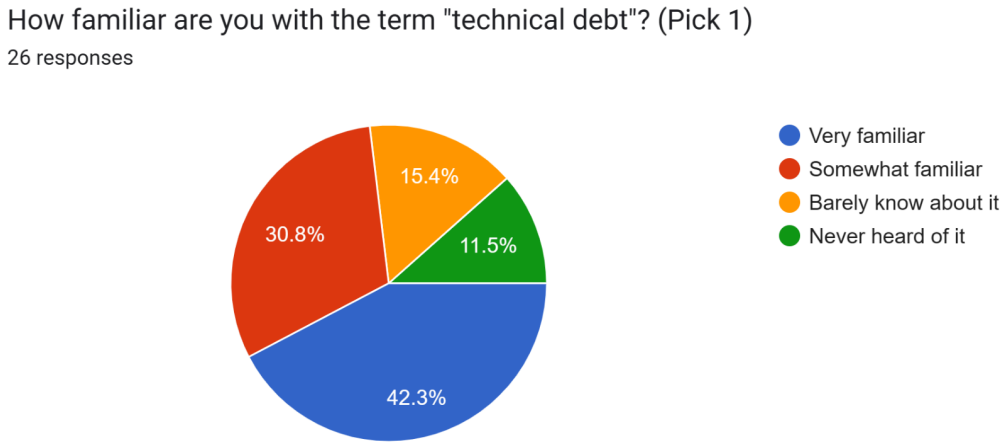


Q1.4 - Familiarity with Technical Debt

Out of the 26 respondents, 19 of them were very or somewhat familiar with the term “technical debt”, with only 27.9% of the respondents being vaguely or not at all familiar with the topic as seen in Figure 12.

Figure 10

Section 1, Question 3 of the survey.



Q1.5 - What is Technical Debt to you?

When asked to explain technical debt (TD), developers most commonly attributed it to short-cuts – easy short-term solutions made instead of a hard long-term one. A simpler decision typically made during strict time frames that undermines the quality of the project. These decisions may be made in response to changing plans, business or management actions or time pressure. In fewer cases, developers appointed outdated tools and libraries, inexperienced and stubborn developers, or early prototypes to the cause of TD.

Q3.6 - Survey Definition of Technical Debt

Some have disagreed with the definition for technical debt given in the survey. Namely, one developer said: “technical debt is not necessarily a bad design”, that rather it stems from change in design from the initial goal, where no matter how great the design was at the beginning, change of plans renders that design unfit for the new direction of the project.

Q3.11 - Change of Plans

Developers reported that unexpected changes had caused an uptick in workload, adding days or months to development time. These changes resulted from a variety of reasons, such as business decisions, lack of concrete design or incompatible tools that needed to be reworked or scrapped.

One developer notes that decoupling, the act of separating logic within the codebase so they may be independent of one another, is an effective way to adapt to any change of plans more easily. Another developer empathized that technical debt is not hard to avoid if people are aware of it, as most of it is caused by time pressure that leads to poor planning rather than being caused by it. This point is reiterated by multiple other developers, who stated that change of plans is not the root of TD, but may often increase the risk of it.

Q3.14 - Time Pressure

Time pressures were stated most frequently to be the cause of shortcut solutions. Majority of the developers mentioned that suboptimal design decisions were abandoned due to lack of time to develop them further properly, leading to them being left in the system, which added additional work or optimization issues in the project. At times, adding a new feature to a system that was developed poorly at the start which would require rewriting the system, at which point developers would cut the feature or apply more shortcut solutions to implement it, evidently adding more (contagious) debt to the system. Developers that reported having not experienced significant time pressure claimed that had that been the case, multiple features would have needed to be cut.

Q3.17 - Testing

For testing, respondents were split on whether unit tests and other dedicated tests were useful in game development. Some claimed that unit tests were of great help, namely automated tests saved a lot of time that would otherwise be spent by testing teams. While others argued that tests in game development are rarely used or not necessary, as logs and internal testing proves to be enough testing. In another contradiction, one developer argued that meaningful TD does not occur at the level of testability, stating: “The real problem isn't ‘*is this code doing what it intended,*’ it's ‘*is the intent of this code the right thing at all?*’”. Meanwhile, the next respondent said that testing itself can become a source of TD if not maintained properly, requiring good source code management and pull request reviews to prevent this.

Q3.20 - Documentation

Documentation appears to be regarded as quite useful, with a few developers noting that it may help with better design decisions and prevent misunderstandings of goals by logging important conversations. Several responses mention undocumented projects or systems being difficult and confusing to work with, especially when the person responsible for a system is no longer in the project. However, despite the usefulness of documentation, it is argued that it costs too much time to create and maintain properly, with one developer stating that it can take upwards of a third of development time on documentation alone. Documentation is communication and at the lack of it, developers resort to in-person conversations, in which individuals may spend hours explaining the project to keep everyone aware and on track with the project. This, of course, poses an issue as knowledge that is not documented in any way can easily be lost due to misremembering or somebody leaving the team.

Q3.23 - Legacy Systems

Technical debt from legacy code is considered by respondents to be primarily caused by poorly designed systems that had never been refactored, which at the current stage of development are left untouched due to bad code breaking new features easily. One developer stated that modularity eliminates almost all issues with legacy systems. Tools and assets brought into the project are typically not modular or outdated, contributing to TD of the project. The untouchable nature of legacy systems also appears to be worsened due to the person in charge of said system no longer being on the team, making it difficult to modify due to lack of knowledge.

Q3.25 - Inexperience

The general impression of respondents on inexperience being a cause of TD appears to be that inexperienced developers typically work without concrete planning, organization or proper design, which may contribute to TD in the long-run. Such missteps are not made intentionally, one respondent expressed that skill is improved over time and developers will find their previous work to be flawed in a couple of years, no matter how good it is currently. Some people may be more organized than others, but experience does help developers to be able to predict and prepare when creating solutions for the project. Lastly, a single developer had an interesting perspective,

stating: “most juniors are not productive enough to contribute enough code to be a problem”. It is possible that larger projects are less affected by inexperienced developers.

Q4.29 - Success of a Game

None of the 26 responses thought that TD has no effect on the success of a game, with over half of the participants believing it **depends on the type of game**. Three respondents chose to argue that TD always occurs in a game and success likely depends on the type and amount of TD, because in some cases it may be necessary or inconsequential.

Q4.30 - Acceptable Technical Debt

The majority of developers in the survey considered it most forgiving to take on technical debt **during prototyping**, with many accepting TD in time pressure situations, such as **meeting a deadline** or **ensuring a game’s release on time**. The responses that commented in detail argued that some types of TD are always acceptable, especially for prototypes. In cases where a system is not essential, or in a project that is expected to be made with a minimal budget, such as shovelware.

Q4.31 - Investing Time to Manage Technical Debt

The investment in managing technical debt faces mixed opinions. TD is primarily stated to be inevitable, but crucial to be balanced. Programmers with professional experience had a more pragmatic approach to the question, in the best case scenario it would be ideal to have perfect code and excellent TD management, but most of the time that option is not available. Instead, it is a matter of priority, not all TD needs to be fixed and some shortcuts are acceptable for the sake of a game’s release, because it is expected that either the development team will no longer be working on the game once it is out, or the team expects to patch slowly after release. A few have claimed that often TD does not need to be addressed, specifically relying on business decisions to determine its priority. Contradicting that, multiple developers lamented the loss of quality caused by technical debt, believing it is important to address TD in cases where it may affect the user experience and performance, additionally some negatively noted the pressures management or

publishers may put on developers to make decisions surrounding TD, with one developer simply stating: “We already have enough buggy games”.

Q4.32 - Learning About Technical Debt

After having learned about technical debt, developers most frequently note taking greater care to future-proof plans and designs, relying on past experiences to have smoother development and avoiding common pitfalls that were encountered in past projects. One respondent expresses working differently in legacy and newer projects, expecting older ones to have far more significant TD, thus increasing the workload.

Q4.33 - Causes of Technical Debt

Nearly every developer that responded to the survey considered poor planning and bad leadership as the main cause for technical debt. Related to planning – feature creep, sudden design changes, lack of or unrealistic scope and overcomplication were all mentioned multiple times as the defining factor that introduces TD. While many agreed that inexperience contributes to TD, a few developers highlighted that inexperience is rarely at fault compared to other causes, although inexperienced leaders were considered to more significantly increase the risk of the project losing direction and devolving into TD. The responses were in general agreement that a poorly defined goal or vision of the project leads to unexpected changes or unplanned additions that cannot be supported, which leads to poor communication between management and developers, as well as worsens the pressure of deadlines, increasing the workload for the developers.

Q4.34 - Managing Technical Debt

When it comes to managing TD, emphasis has primarily been put on modularity and good documentation, communication or planning. The purpose of creating modular designs serves to ease refactoring that will need to be done later, simplifying systems and easing the workload for developers to keep the development flexible and able to adapt to unexpected changes better. The majority of respondents state that proper documentation, testing and code reviews help eliminate TD effectively, additionally highlighting the importance of good communication between team

members. Lastly, a point has been made to acknowledge that managing TD is a time investment and requires a dedicated timeframe to resolve well through regular refactoring.

Q4.35 - What Developers Wish to Know about Technical Debt

In terms of what developers wished to better know about technical debt, only a few responses were received. One of the developers was interested in a tool or measurement to calculate how much addressing certain TD will improve game performance or development, since such a tool would be beneficial for deciding how much time should be dedicated to resolving TD. Lastly, another developer simply wanted specifics on how detailed a technical documentation should be.

4.1 Abstraction of Results

Game developers that responded to the survey primarily named shortcuts made during time pressures as the definition of technical debt (TD), and when asked to describe what causes it, the vast majority took to blame poor planning and leadership as the main source of TD, which is worsened by deadlines and inexperienced developers. Developers reported most frequently that TD is managed by regular refactoring, good planning and modular systems.

Below are tables detailing the most and least common responses given per open-ended question, frequency is defined by the amount of responses mentioning each theme identified in the data for a rough estimate of rarity. For viewing the thematic categories highlighted for each question in detail visit Appendix 4.

Table 1

Common response themes ranked by frequency for Q1.5 “What is technical debt to you?”

Frequency	Themes
Common	Shortcuts, Design Quality, Time Pressure
Uncommon	Additional Work, Poor Planning, Outdated Tools
Rare	Individual’s Choice, Business Reasons, Prototypes, Inexperience

Table 2

Common response themes ranked by frequency for Q3.11 "Can you give an example of how change of plans influenced technical debt in your projects?"

Frequency	Themes
Common	Additional Work, Change or Lack of Plans
Uncommon	Shortcuts, Design Quality
Rare	Prototype, Inexperience, Legacy Tools, Misunderstandings or Knowledge Loss

Table 3

Common response themes ranked by frequency for Q3.14 "Can you give an example of how time pressure influenced technical debt in your projects?"

Frequency	Themes
Common	Time Pressure, Shortcuts, Design Quality
Uncommon	Change of Plans, Additional Work or Complications
Rare	Prototype

Table 4

Common response themes ranked by frequency for Q3.17 "Can you give an example of how testing influenced technical debt in your projects?"

Frequency	Themes
Common	Testing
Uncommon	*none*

Rare	Additional Work, Change of Plans, Individual's Choice, Time Pressure, Management, Design Quality

Table 5

Common response themes ranked by frequency for Q3.20 "Can you give an example of how documentation influenced technical debt in your projects?"

Frequency	Themes
Common	Documentation, Communication, Knowledge Loss
Uncommon	Additional Work
Rare	Design Quality, Shortcuts, Time Pressure, Individual's Choice

Table 6

Common response themes ranked by frequency for Q3.23 "Can you give an example of how old/legacy systems influenced technical debt in your projects?"

Frequency	Themes
Common	Legacy Systems, Outdated Software, Design Quality or Modularity
Uncommon	*none*
Rare	Additional Work, Knowledge Loss

Table 7

Common response themes ranked by frequency for Q3.25 "Can you give an example of how inexperience influenced technical debt in your projects?"

Frequency	Themes
Common	Inexperience

Uncommon	Additional Work, Poor Planning or Organization
Rare	Management

Table 8

Common response themes ranked by frequency for Q4.31 "Do you think game developers should invest more time in preventing and managing technical debt, even if it means delaying the game's release? Why or why not?"

Frequency	Themes
Common	Design Quality or User Experience, Business, Investors, Management, Planning or Organization
Uncommon	Shortcuts, Additional Work or Complications
Rare	*none*

Table 9

Common response themes ranked by frequency for Q4.32 "Has learning about technical debt changed how you approach development, design or planning? If so, how?"

Frequency	Themes
Common	Planning or Organization, Communication, Documentation
Uncommon	Design Quality
Rare	Additional Work, Legacy Systems

Table 10

Common response themes ranked by frequency for Q4.33 "In your opinion, what causes technical debt?"

Frequency	Themes
Common	Change or Lack of Planning, Feature Creep, Organization, Leadership
Uncommon	Additional Work, Deadlines or Time Pressure, Inexperience
Rare	Prototype, Business Decisions, Legacy Systems, Shortcuts

Table 11

Common response themes ranked by frequency for Q4.34 "How do you or your team manage technical debt?"

Frequency	Themes
Common	Design Quality, Refactoring, Modularity, Documentation, Code Reviews, Testing, Good Planning
Uncommon	*none*
Rare	Additional Work, No Shortcuts

4.2 Collected Data

Research was conducted through a survey which focused on opinions given by game developers, where a total of 26 developers responded. The survey was sent to various game development communities on Discord, a few content creators found on YouTube, as well as to game development students at Jamk University of Applied Sciences in Finland. All data gathering in Discord communities was done with explicit permission from moderators. Emails sent to content creators were addressed to public emails which said creators encouraged their viewers to use. The teachers of Jamk university approved sending an email to all current students in the Business Information Technology (Game production) program of Jamk university. Various other communities, such as subreddits and independent forums were requested to conduct research, however no response was received. In the cases where no response or permission was given, no action was taken and that community was exempt from receiving the survey.

Each response was read and evaluated individually, then all responses to each question were compared in bulk. Notes were taken on which opinions had overlap and if there were any connection to the experience of the developer depending on their role in a team, years being involved in game development and what type of experiences the respondent had within the game industry.

Additionally, the survey was meant to serve as an introduction to the topic for those that may be unfamiliar, to raise awareness and possible causes and effects of technical debt through the information presented in the survey's questions.

4.3 Data Analysis

The survey data comprised both multiple-choice and open-ended responses, analyzed using quantitative and qualitative methods, respectively.

Multiple-choice responses were presented through descriptive statistics to summarize the frequency of each response category. Descriptive statistics are a way to describe and summarize large samples of data through graphical representations, which display patterns and trends found in it for easier interpretation (Cooksey, 2020). Tables were generated in Google Sheets to visualize these distributions. Additionally, cross-question comparisons made by descriptive cross-tabulations were conducted to identify potential patterns between data obtained from several questions. Due to the small and uneven sample sizes between response groups, exact response counts were reported instead of percentages or summary labels for clarity of results. Observed trends were interpreted in the Discussion section to explore possible correlations and patterns. Tables for multiple-choice responses can be found in Appendix 3.

The data from open-ended responses was evaluated through thematic analysis, which is a research method used to identify and interpret patterns or common themes in a qualitative data set, such as opinions or experiences (McLeod, 2024). Responses were analyzed by following the six-phase framework for thematic analysis:

- **Familiarization** - The data was carefully reviewed for similarities, outliers and patterns to identify the most common perspectives.
- **Coding** - The responses were rephrased for clarity, color-coded for significant features and placed in lists.

- **Generating Initial Themes** - Common features were summarized under specific themes.
- **Reviewing Themes** - Themes were re-evaluated, with some appearing less prominent and thus being adopted into a different label for simplifying.
- **Refining, Defining and Naming Themes** - Accurate and distinct labels were created that encapsulates a bundle of related concepts.
- **Writing Up** - The prevalence of themes was evaluated for the study's results, discussion and conclusion.

Each feature and their recurring themes were assigned a unique color for identification, the full theme-color legend can be found in Appendix 4, alongside the summaries of data analyzed by it.

4.4 Suitability, reliability, and validity of the data

Suitability. The data answers the set objectives of this thesis effectively, but faces multiple limitations. A total of 26 responses were collected, giving a detailed, but small sample size, which largely limits the accuracy of descriptive statistics, especially in percentages (Cooksey, 2020), but allowed for cross-tabulation to examine distributions between certain values gathered. Additionally, the small sample size cannot be representative of the gaming industry on a large scale, as the respondents of the survey had various levels of experience, technical skills, professional expertise and role in development, information gathered varies in validity.

Reliability. The survey was structured by being divided into four sections to improve reliability:

1. **Introduction** - Gathered general information on the individual such as skill and role, additionally inquiring their familiarity with the concept of technical debt and asking to describe it.
2. **Definition** - Provided a broad definition of technical debt which was then repeated on each next section to give clarity and set a standard for what is defined by the term in the context of the survey.
3. **Influence of TD on a game project** - Presented ordinal scale questions, as well as optional open-ended questions that explored individual aspects of technical debt in detail, each structured and worded similarly for consistency. Respondents had the choice to skip this section if they felt they did not have enough experience to respond to given questions. The questions selected for this section were inspired by research done by Borowa et al. (2021).

4. **Opinions** - Contained the remainder of open-ended questions which focused on how the phenomenon was perceived by the respondents.

All survey questions are available in Appendix 1, research and data analysis methods and general target communities are detailed in Research Implementation and Data Analysis of this thesis, allowing for this research to be reproduced reliably to generate new results.

Validity. The research operates on imperfect information due to self-reported data from personal opinions potentially being flawed due to misinterpretation by the respondents (Deakin, 2025). For that purpose, the selection of experience was devised by a label such as “Hobbyist”, “Professional” and others as seen in Figure 7, rather than level of education. Thus, a respondent was given the opportunity to describe their own relationship and personal experience with game development.

The questions given in the survey were possibly difficult to respond to for those that have no technical skill or are not involved in implementation, limiting the accuracy of general awareness of technical debt within the industry, due to potential bias towards knowledgeable individuals. Steps were taken to allow game developers of any skill level or role to participate, but the reduced presence of non-technical respondents shows either disinterest from parties outside of technical implementation, or inability for such individuals to respond to the questions provided. The responses given by programmers or professionals may be more valid than those from students or non-technical developers; however, the data was not separated to accommodate for that discrepancy. Additionally, all questions had examples provided to clarify the goal of the question, which may have influenced responses given to be biased towards the examples stated for each survey question. The summarized data that underwent thematic analysis can also be susceptible to researcher bias, where mistakes, assumptions or misinterpretations could have been made (McLeod, 2024), as such, the data has limited validity at larger scale.

To be of note: not all data gathered was analyzed, the author opted not to review any ordinal scale questions in the survey, because the validity of that data is questionable due to response limitation and possible misinterpretation by respondents. One of the respondents stated the need for the option to respond “I don’t know”, indicating that certain individuals may pick a random number, or one they may feel represents a neutral response, such as 1 or 3, possibly invalidating the charts from collected responses. All response summary charts for the survey can be found in Appendix 2.

5 Discussion

How much do various game developers know about technical debt?

The majority of respondents were aware of the term before taking the survey, with 19 out of 26 claiming to be very or somewhat familiar with technical debt, indicating that game developers are typically knowledgeable on the term in some capacity.

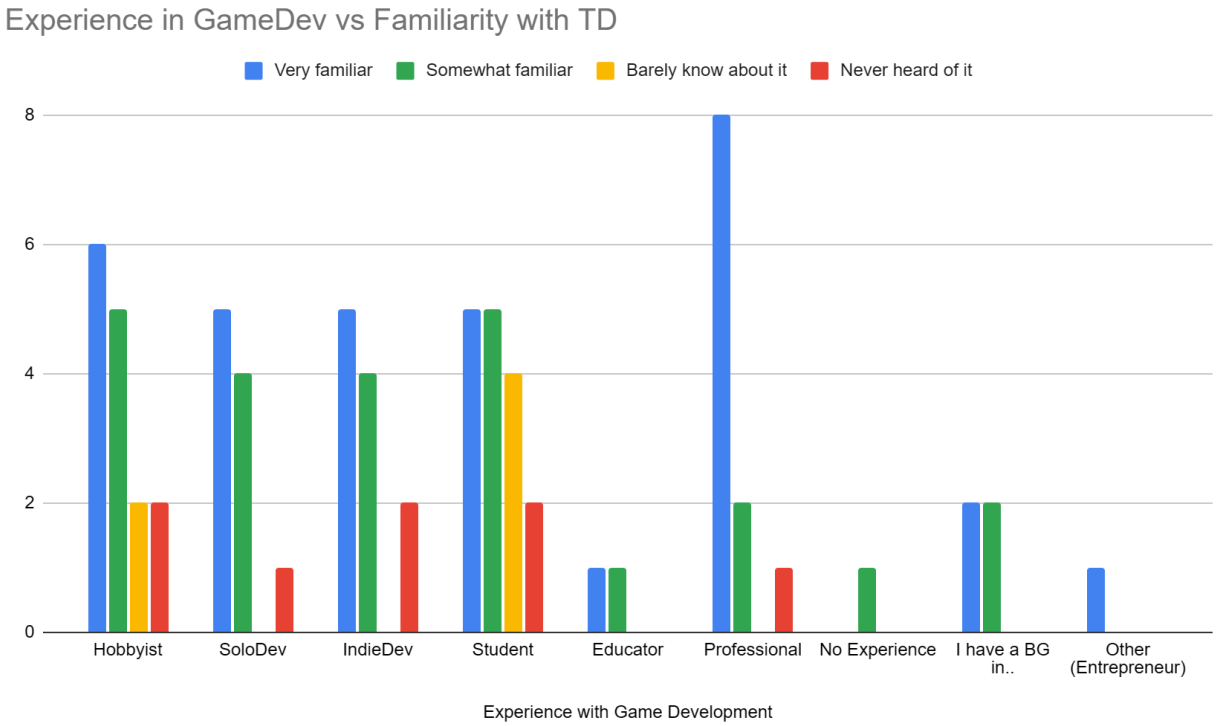
Section 1 of the survey asked the respondents to select their experience and role in game development as seen in Figure 7 and Figure 8 previously. For example, interestingly, one respondent chose “No Experience” while also selecting “Hobbyist” and “IndieDev”, potentially showing trepidation in skill level or interpreting “experience” as “work experience”, while it does not describe exact skill, it may clue in to this developer being very new to the game development sphere. Another respondent chose to describe themself as an “Entrepreneur”, potentially noting particular interest in business-related affairs in the game industry, over other roles. Responses like that may give insight into responses for other questions given by those developers, such as showing what a new developer knows, and what someone focused on the business side of development thinks.

The author finds it interesting and possibly useful to see how developers view their own experience before assessing their personal opinions, as it may give insight on if varying mindsets, personalities and personal education affects how such developers are aware of, approach and understand technical debt. Someone who approaches game development casually as a hobbyist may have more lax expectations and workflow than a professional in the industry for example. However, the information gathered for this purpose is surface-level and should not be regarded with too much attention beyond curiosity and speculation, as such an assessment would require more dedication than was given in this survey.

There is overlap between experience labels and familiarity with TD as seen in Figure 11, showing that developers that describe themselves as professionals are more likely to know what technical debt is already.

Figure 11

Type of experiences in game development VS familiarity with technical debt

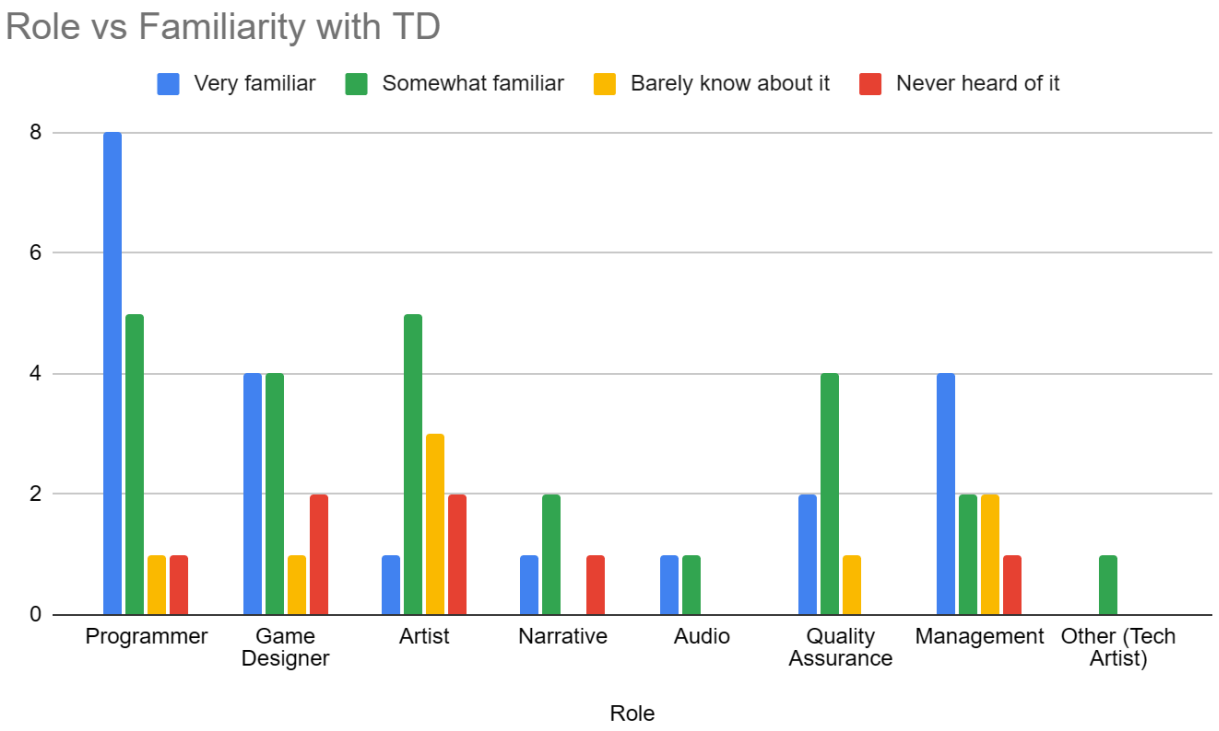


Note. This chart was generated in Google Sheets and compares responses from Q1.1 and Q1.4. The option that is not fully visible says “I have a background(BG) in general software development in addition to the other answer(s)”.

In the roles section in Q1.2, half the people that picked Programmer also selected Professional in Q1.1. As shown in Figure 12 and Figure 14, where 13 out of 15 Programmers reported being Very or Somewhat familiar with TD, while Non-Programmers had mixed results. The majority of people that picked Professional or Programmer claimed to be very familiar with technical debt (TD), with 6 out of 8 Professional Programmers being very familiar with TD. Along with the data from other questions it suggests that the developers that responded to the survey were more likely to be familiar with the topic when describing themselves as Programmers or Professionals, or both.

Figure 12

Role in game development VS familiarity with technical debt

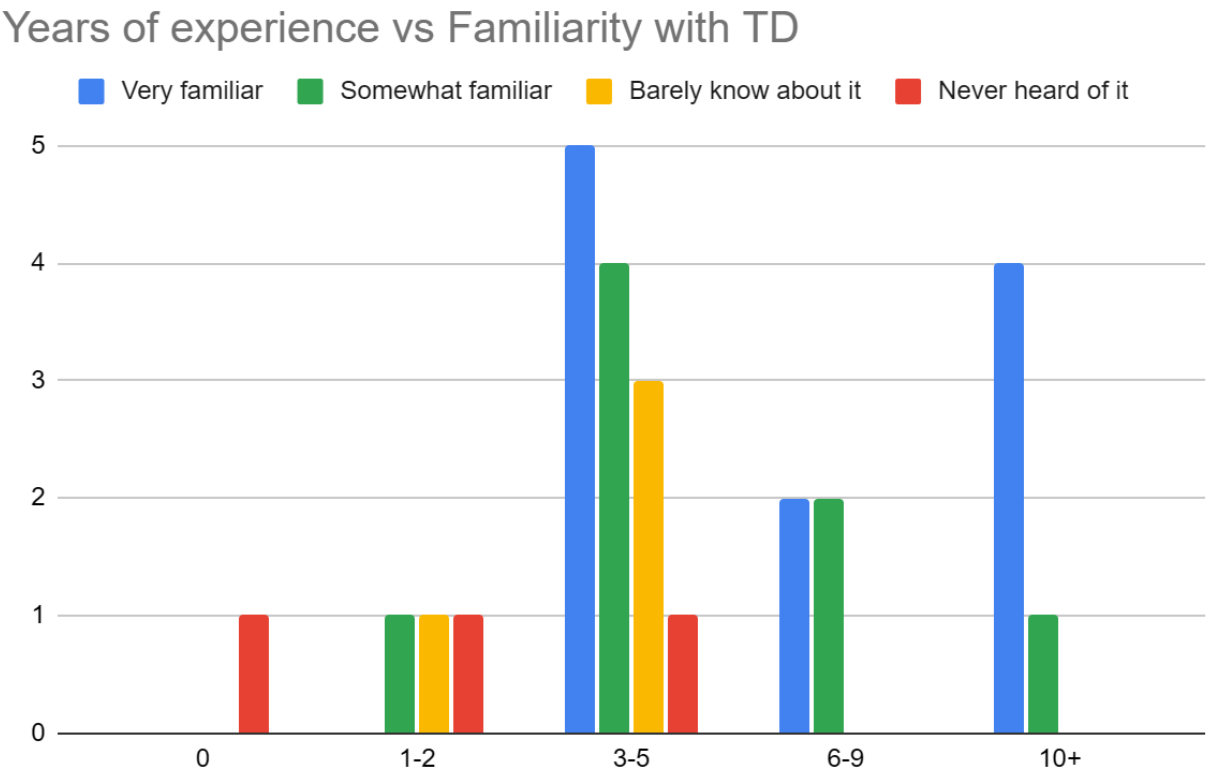


Note. This chart was generated in Google Sheets and compares responses from Q1.2 and Q1.4.

From Figure 13, it can be observed that 22 of those that reported 3-10+ years of experience were more likely to claim to be very or somewhat familiar with technical debt, none of the 9 respondents that chose 6-10+ years of experience claimed to be barely or not at all familiar with the term, and none of the 4 respondents that chose 0-2 years of experience claimed to be very familiar with the term. This shows that developers with 3 or more years of experience are more likely to be aware of technical debt, with a higher amount of years improving the chances of the developer being knowledgeable about the topic.

Figure 13

Years of experience with game development VS familiarity with technical debt

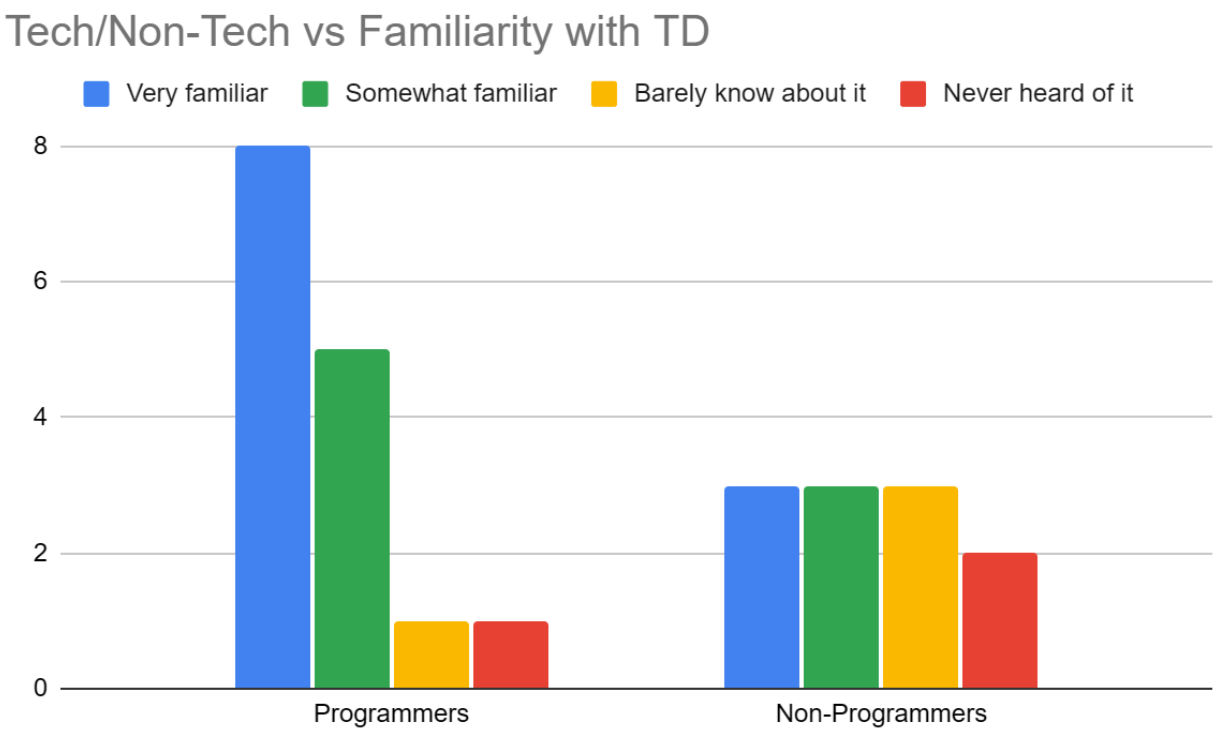


Note. This chart was generated in Google Sheets and compares responses from Q1.3 and Q1.4.

The baseline difference between responses of programmers and non-programmers can be seen in Figure 14, which shows a trend towards tech-savvy individuals being more likely to know technical debt in detail.

Figure 14

Technical and non-technical individuals VS familiarity with technical debt



Note. This chart was generated in Google Sheets and compares responses from Q1.2 and Q1.4.

What is technical debt according to game developers?

The responses indicated that developers considered technical debt (TD) to be **shortcuts** where a lower quality design decision is made instead of a better one for a variety of reasons, such as time pressure, business actions or change of plans. Although some made a point to mention outdated tools and prototypes when defining TD. Generally, TD is characterized as bad design which adds additional work to the project, although one developer argued that bad design was not often considered to be TD. This does not align with previous research by Suryanarayana et al., (2014), Chernishev (2021) and Lasater (2006), and others, who do place bad design as a major cause of technical debt and argue that guidelines for it assist in TD management. Instead, a few developers claimed that no matter how good or bad quality the design is, all code is debt.

The TD definitions given varied a lot, this possibly shows that the term is not particularly well-defined and multiple interpretations of the phenomenon exist. Following that, interestingly, one respondent that has learned of technical debt for the first time through the survey argued that the term is flawed and should not be used for “common occurrences in game production”. This highlights an issue that is evident throughout previous research done on the topic, the term

“technical debt” is described and defined differently between most sources, with one book by Brown (2024) containing chapters for each varying perspective on technical debt, exploring different angles of the term to capture it fully. Again, confirming that the term is vague and can be described in a variety of ways as seen through the mixed responses and the broadness of previous research.

What do developers identify as the cause for technical debt most frequently?

Despite only being mentioned to increase the risk of TD when asked what developers defined as technical debt, every respondent named lack of planning, change of plans, feature creep and general overcomplications of scope as the main cause of TD. Additionally stating that things such as time pressure, inexperience, poor communication or management decisions are what typically leads to these plan changes and complications, worsening their impact on the project. Which, in turn, introduces TD once the designs need to change due to new plans.

Taking all of that into account, that means that the process at which TD occurs in game development according to some game developers typically happens like this:

- **Design** is created to match existing plans and scope of the project
    - The design may be of varying quality, but carries out its purpose
- **Change of plans** occurs due to various reasons such as:
    - Inexperienced leaders introducing new ideas or changing old ones
    - Investors or publishers ask for changes for business reasons
    - The original plan was never well-defined and changes spontaneously
    - The scope was reevaluated, for better or for worse, changing the goal of the project
    - A planned feature is no longer possible due to limitations such as skill or tools
    - Deadlines are coming up and cannot be reached in time
- **Current design** no longer fits the goal of the project and requires changes
    - The original design was not made to support new goals
    - New requirements require more time
- **Additional work** is needed for new requirements due to them being unsupported originally
- **Original deadlines** are no longer possible
    - Time pressure may lead to further changes in plans
    - Upcoming deadlines force developers to make suboptimal decisions as shortcuts
- **Shortcuts** destabilize codebase quality, increasing technical debt

- **Technical debt** present in the system may then worsen the next cycle of changes
- **Legacy systems** gain a lot of TD due to accumulated changes and outdated design

How do game developers manage technical debt?

Inexperienced developers and leaders were reported to be a contributor to TD, because inexperienced people, at first, often work without good planning, organization or design, but over time get better and improve their ability to adapt to changes in plans. This demonstrates how the cause of TD evolves from being *reckless* to *prudent* with experience, both of which are discussed by Fowler (2009) in previous research.

Multiple respondents claimed that creating modular code that is more easily able to adapt to changes in plans is reliable to almost entirely eliminate technical debt. Modularity is created through organization of the code, such as by use of design principles and patterns, which have been discussed multiple times in literature as an effective way to mitigate TD, such as in works by Suryanarayana et. al (2014), Tripathy (2014), Loster (2023) and more.

Unit tests were found to be useful, but not by all developers, some disagreeing with their validity in game development because of fast development, with some stating that unit tests were never used in their projects. This suggests that testing is not well utilized or useful in certain game projects.

Primarily, code reviews, pull requests and self-documented code were said to be enough to manage TD, with documentation being of great use as well, but often being too time-consuming to make properly. Despite that, developers claimed multiple times that good documentation and communication between the team helps projects remain on track and planned well by improving available knowledge on the project. With legacy projects suffering particularly from loss of knowledge over time of development, especially when people in charge of certain systems are no longer on the team. The mention of time pressures being the cause of TD and an appreciation for documentation might suggest that developers may spend less time than desired on proper documentation, testing or communication, which has a slight ripple effect on planning, design and future changes, possibly complicating the process of TD mitigation further.

Nonetheless, modular systems and regular refactoring were considered the most effective way of managing TD. Multiple developers additionally claimed that high quality, modular systems may be

hard to facilitate due to time limitations and the need to keep up with deadlines and business presence. This is another case where time pressures were linked to developers foregoing TD management, indicating that deadlines affect how well a developer is able to adapt to new goals and changes in the project.

Why does managing technical debt in game development matter?

None of the respondents thought TD had no effect on the success of a game, but many suggested that it depends on the type of game and type of TD. Showing that developers are aware of the importance of TD management, although how it affects the development and game is different on a case-by-case basis. They often found it acceptable to take on TD during prototypes or to meet deadlines, which further builds on the evidence that time pressure leads to developers choosing TD to save time and progress on the project.

Overall, developers suggested that all one can do is their best, inexperienced developers will learn better planning and better TD management with years of experience, which is backed up by the expertise of the respondents with higher years of experience under their belt. A few developers confirmed that after learning TD, they developed better plans or adapt to new plans better as a result, with one developer learning to expect projects with a lot of legacy code to be harder to work with due to old TD and lack of knowledge on the systems. Information about TD and its management is valuable and each available response indicates a need for clarity and resources surrounding it.

When developers talked of personal experiences, many discussed rewriting systems because of TD, which took weeks or months to fix. Poor planning and rewrites for terrible designs are shown to clearly complicate and add additional work to the project in multiple cases, highlighting the importance of managing TD in a balanced manner to adjust for deadlines and changes.

5.1 Possibility: Incidental or Unexpected Findings

There has been very little research held on technical debt in the context of game development. Vast majority of material that was found for this thesis stemmed from software development circles, where the term originated. The high amount of sources in software engineering has been expected, but the near total lack of such research in game development was not, at most, it was expected that the research would be fewer, but not absent. A few sources on game development

talked about lack of research as well, such as Borowa et al. (2021) and Agrahari and Chimalakonda (2020) noting its absence.

None of the respondents to the survey had opinions on specific design principles or patterns, even one developer going as far as to state: “design patterns are ridiculous”. This was unexpected due to the amount of attention given to patterns and principles in previous research (e.g., Suryanarayana et al., 2014, and many others), including those focused on game development. Considering that modularity and refactoring is mentioned continuously as a way to manage technical debt, it raises questions on what exactly game developers are doing to create modular systems.

The game development sources in existing knowledge (e.g., Extra Credits, 2019) curiously included non-programming roles in technical debt, such as level designers and artists. Technical debt had been compared to or related to the concept of “development hell”, a situation that happens when development stalls severely and appears neverending. Technical debt has never been mentioned to be anything but programming in research done for general software development, which shows the difference between traditional software and games is bigger than it seems at first glance.

A few 3D modellers responded to optional questions detailing their experiences with technical debt, stating their own points of view that appeared to relate to the experience of programmers. Technical debt as a concept is tied entirely to programmers within software engineering circles, with a few game developers mentioning it being possible to relate it to other roles as well, while that notion was possible, it was unexpected to see it proven in the findings of this thesis.

5.2 Reliability

This research may be reproduced by repeating the survey with a new group of individuals. Online surveys can be easily edited and re-run, improving the research with each iteration (Deakin, 2025). The survey questions are available in Appendix 1. The tables used in the Results section can be generated from survey results by any available tools such as Google Sheets, cross-comparison tables used in Discussion can also be reproduced from data available in individual tables in Appendix 3. Codes used to identify patterns in response summaries can be found in Appendix 4 and can be reproduced by following the six-phase framework for thematic analysis, although the conclusion reached through thematic analysis might vary based on the researcher’s interpretations and may be prone to assumptions or preconceived notions (McLeod, 2024).

Survey questions were chosen with consideration, feedback from teachers and peers assisted with survey design. Additionally, previous research done by Borowa et al. (2021) served as a small inspiration for a number of questions in the survey. However, the ordinal scale questions are unlikely to be reliable due to flaws in response options, better questions can be selected in that area.

This research does not go into deep enough detail that could be used to identify patterns of opinions in specific groups of people within the gaming industry, and cannot serve as a summary of general community opinion due to the low number of responses obtained. This research does not reflect views of the gaming industry as a whole, it portrays the views of a handful of individuals within the industry that only represent a small portion of the community, which allows for a first viewing into the topic, but leaves much to be investigated further.

Lastly, the author is not experienced in conducting research and may have made mistakes that were not mentioned here.

5.3 Ethical Implications

The research was conducted according to rules and guidelines for academic research to produce reliable data. All data was gathered ethically, with explicit consent from community moderators or teachers.

The prevalence of time pressures in the responses may raise ethical concerns about sustainable development practices and balanced working conditions that may be worsened by accumulated technical debt. Time pressures and deadlines imply risks related to crunch culture and developer burnout in cases where unexpected changes add additional work on the developer without adjusting the timeline, pressuring them to work faster or sloppier, or both. In a situation where a shortcut is not possible due to the system already being too unstable, pressure to complete a new requirement effectively may cause stress and harm morale, making implications on social and individual sustainability surrounding technical debt. Such effects on the developer's happiness were previously discussed in work by Roberts (2020, p. 6). Similarly, developers often reported additional work time invested into the project as a consequence from technical debt, confirming that the phenomenon affects economic sustainability of the company, which aligns with previous research by Venters et al., (2024). While technical debt is not the root cause of issues concerning

project scope, crunch culture, and burnout, it has the potential to both contribute to these issues and result from them.

6 Conclusion

Starting this thesis, the goal was to understand the nature of technical debt in game development as it is experienced by developers themselves: what they define as technical debt, what experiences caused increase of debt and how it is viewed and managed in the project. That knowledge would then be applied for personal growth and educating any newer developers on the concept before moving onto larger projects and entering the game industry professionally. The knowledge gained in the completion of this thesis has successfully satisfied the research objectives of this study and expanded the author's understanding of technical debt in game development, providing ample information that may assist newer developers. There is still much that can be learned to further expand on the objectives of this thesis, but the findings grant a sufficient introduction into the topic and potential research that can stem from it.

From the individuals that responded to the survey, a small, varied perspective can be observed that answers the research's questions. While the data obtained is limited, it serves as an overview on the topic of technical debt in game development as it is discussed by the people within the industry, opening multiple opportunities for further learning.

The results of this research suggest that the main cause of technical debt in game development is poor planning, typically led by bad leadership decisions, although it can also stem from developer or leadership inexperience in general. Then, due to new requirements, additional changes are now needed in the project, creating more work for developers while the timeline likely remains unchanged. Rapid changes of scope may result in developers being unable to meet previous deadlines and being forced to cut corners and make suboptimal design decisions that undermine the quality of the project. The resulting time pressure may then in turn cause stress for the developers and impact their work, stall project progress and potentially lower quality of the end result.

There is balance to be maintained in the management of technical debt. One of the primary ways to mitigate debt, as described in existing research (e.g., Suryanarayana et al., 2014) and supported

by the findings of this thesis, is through regular refactoring and modular systems – typically grounded in design principles, patterns and OOP practices. However, quality codebases and dedicated refactoring are a time investment that must be measured against the priorities and needs of the project. Games need to be released at some point to provide profit and continue the company's or individual's presence in the game industry, which means that at a certain point technical debt is negligible and is better to be ignored for the sake of progress. The type of technical debt and the game it is present in determines the urgency of its management, making it crucial for developers and management to assign priority proportional to the type and needs of the project. Should leadership be aware of the consequences of rampant technical debt, they may be able to guide their team to smoother development, or, perhaps, be more likely to hear out their technical developers' warnings and recommendations regarding it.

One of the struggles for this research was defining the term "technical debt" correctly for those that may be hearing it for the first time. Awareness and education on the phenomenon is a simple yet effective way to assist new developers and non-technical developers to keep technical debt management in mind during development, making it valuable to define the term clearly, as well explain the presence, purpose and importance of it in a project. Existing literature often defined "technical debt" differently, sometimes referring to the metaphor itself, and other times to the manifestations caused by its accumulation, or both. For example, Fowler (2009) and Martic (2009) disagreed on what exactly counts as technical debt, or whether that distinction matters at all. Not only that, but the term was criticized in the results of this thesis' research as well, only proving that technical debt is a fragmented term.

There are multiple ways to talk about technical debt, it is possible to talk about it from various different perspectives and the people using the term often make their own interpretation and definition of it. The term covers a variety of interconnected topics, some highly specific and others only vaguely related, but each can relate to the phenomenon itself. The language surrounding the term continued to evolve and change depending on who was talking and what they needed to explain, creating a broad and sometimes contradicting definitions of technical debt. But while the term is sporadic, there is a need to describe and define these situations and issues which this phenomenon encapsulates, to better understand the development process and identify issues in a quantifiable and concrete manner. To empathize this point, here are a few ways to define technical debt based on everything covered in this thesis:

- Technical debt is a **metaphor**. Comparing financial debt to explain foundational structure issues in the system.

- Technical debt is a **phenomenon**. The process of a system destabilizing as software quality decreases, making future work harder.

- Technical debt is a **trade-off**. Choices made by developers, whether intentional or accidental, which lead to better or worse readability and maintainability of the system in exchange for short-term benefits.

- Technical debt is **organization**. The failure and success to keep a project easy to work with by following principles, guidelines, or creating tools and catalogues of information for better use.

All these definitions hold truth and value to them, however this makes the conversation about technical debt more difficult when bringing it up to developers that have no experience or understanding of it.

Additionally, what has been discovered through the making of this thesis is that there is an abundance of research and resources made on technical debt within traditional software development, but nearly none has been made with the sole focus on game development from what was possible to find for this study. Game development proves itself to be a distinct sub-area of software development that has vastly different requirements, development practices and production expectations than traditional software. This deviation potentially invalidates information that can be found on technical debt in literature, as it specifically caters to traditional software and may not be as useful for games. Which is a point that could be driven through the results of the research largely lacking discussions on design principles or patterns, where in actuality developers focused primarily on adaptability, clever scope and balance in costs of change for new features. These elements are mentioned in software-centered sources, but not reinforced as much as concepts like design patterns, code smells and the like.

Thus, this may create a need for a better way to define technical debt and related concepts in the context of game development. One of the sources, research done by Agrahari and Chimalakonda (2020), had already considered that need and attempted to create a catalogue of common technical debt manifestations based on themes and their frequency in projects, however their focus lay entirely on the code quality and not the definition of the term and practicality of its education.

Game development sources, such as Engström et al. (2018), Borowa et al. (2021) and Extra Credits (2019), talk about games continuously changing and requiring much more flexibility than traditional software projects, often relying on Agile and similar development practices to maximize the project's versatility and scalability. Following that, professional developers advise to create modular systems that can more easily adapt to upcoming changes in plans, allowing for as much flexibility as possible. By taking into account that previous research, this shows that typically technical debt is taken on deliberately in cases where deadlines need to be met, and accidentally in the event where developers are not experienced enough to efficiently design their project to account for future needs of the everchanging game scope. Games are an art as much as they are software meant to fulfill a purpose and to support the development of such a project requires a certain level of flexibility. This makes it inevitable for developers to face unexpected changes that may result in technical debt, had proper measures to mitigate it not been taken ahead of time because of ignorance, lack of time and resources or simply inexperience to do so. The game development process requires constant change and adaptability, meaning that unpredictable changes will always occur and these changes will lead to time pressures. The presence of deadlines already induces stress on the team and changes that are difficult to fit into the tight production timeline may well result in crunch culture, with developers rushing to complete the project in any way possible under frustrating and stressful conditions.

Because of continual deadline pressures, technical debt may be a potential contribution or complication in crunch culture. And on top of that, previously reported (e.g. Suryanarayana et al.,2014, Fowler, 2009 and Nemchinskiy, 2024) consequences of technical debt include things such as: poor project quality, project development stalling or becoming impossible to continue, additional time and costs being needed to continue development, project cancellation, community disappointment and developer frustration. All this points to technical debt being an inevitable and noteworthy phenomenon to be aware of. Education on the topic would benefit individuals and

companies, especially newer, inexperienced developers, to create projects and businesses that can be developed efficiently, with fewer costs, frustrations, setbacks and would be more likely to deliver a product of higher quality.

6.1 Future

The simplest way this research could have been improved is by targeting specific groups of people for questions, such as specifically programmers or non-programmers. Additionally, taking a note from previous work by Borowa et al. (2021), a comparison research can be made to assess the perspectives of game developers versus traditional software developers. On the other hand, this type of research could be repeated with similar survey questions at a much larger scale and done through quantitative rather than qualitative research applied here. That would allow for recognizing patterns across various types of developers more precisely in terms of frequency and repetition.

Survey questions themselves can be improved, a portion of the data has gone underused due to inability to accurately judge whether the data is valid enough to analyze. Designing the questions in Section 3 of the survey (the ordinal scale questions) to be more precise and making them optional for those that may not be able to answer them may help avoid false default answers.

Providing multiple definitions for technical debt may prove useful to determine which terminology is more frequently used and if there is misunderstanding of the term among developers and researchers.

For other questions, several potential research areas can be suggested:

Is there a difference in how technical debt is handled by developers working in different engines or platforms?

The responses received occasionally named engines the developer had been working in, where in certain cases an update to a widely used engine, such as Unity or Unreal, had caused setbacks due to incompatibility, features no longer being supported or engine bugs. This brings up the question of whether publicly available game engines bring different challenges than custom, in-house game engines in mitigating and managing technical debt.

How should technical debt be defined? Does the term need to change or improve?

Due to the multitudes of descriptions, definitions and explanations given in previous research and the respondents to this thesis' survey, the need for clarity may be required for the term "technical debt" to be useful in describing and notifying developers of the consequences that phenomenon may have on the project. Furthermore, the difference between traditional software and game development proves to be significant enough to diverge the meaning and manifestations of the term, in which game development's "technical debt" may be applied to roles outside of programming and relate more directly to production planning. This begs the questions on where technical debt ends and other concepts begin and should that overlap be encouraged or eliminated. A term's purpose is communication, determining how best to define the term and how it can be used to educate developers on better practices is a worthwhile research area.

Do business-oriented and game as an art-oriented people think of technical debt differently?

Games are an art and a business, and opinions of people that focus on one aspect of game development over the other may show variation in how technical debt is approached by those individuals. The few business-oriented respondents in this research had less concern for technical debt than programmers and other roles, although that difference is largely too difficult to discern in this study due to sample size. Business and art aim to achieve different goals and taking a greater look at how those goals affect technical debt and the quality of the project may prove to be interesting.

Does role and experience affect opinions on how technical debt should be handled?

This research provides a brief look into this question, however the topic can be expanded on more thoroughly. Each role and experience or education level can be reviewed individually and compared against each other, more so than has been done here. Individuals perceive a project differently from their own point of view and may be limited in what can be understood of other people involved in a project. Identifying which roles may contribute to technical debt in a project unintentionally may prove useful for better communication between those individuals and the programmers in-charge of implementation and direct experience with technical debt.

How much does technical debt contribute to crunch culture and developer stress? Should developers slow down?

Data in this research shows that technical debt is often worsened by deadlines caused by changes in scope. Technical debt has the effect of stalling development, making it difficult and frustrating. Existing technical debt may lead to more technical debt due to inability to manage it for various reasons, one of which being strict timelines. Determining whether the presence of existing technical debt plays a role in crunch culture may assist in understanding developer well-being in such scenarios. The topic does not need to revolve around technical debt at this point, but may instead fully focus on the effect of deadlines and changing scope on the project and the team as a whole, while stating technical debt as one of the consequences or causes for it.

How much does technical debt affect game quality and success?

Various sources and responses to this research's survey state that technical debt has an effect on game quality and success, but the extent of its effects highly depends on various factors, such as what the nature of the debt is and what type of game is being made, among other things. What can be gleaned from a research made on such a topic may give valuable insight to developers and assist them in fostering a better, more efficient and balanced production process, as well as ensuring the project's success and avoiding common pitfalls that can ruin the product.

What kind of tools can be created to assist developers with technical debt?

A tool or guidelines that help identify, categorize and plan around technical debt can be of great help for developers. Being able to measure how much time should be invested into technical debt to prove beneficial for the project could improve and ease the process of planning around technical debt and alleviate the time investment into its consideration and documentation. On the topic of documentation, proper guidelines on how technical debt should be documented and how much time should be spent on writing such guidelines can prove useful for efficient development.

References

Agrahari, V., & Chimalakonda, S. (2020). *A catalogue of game-specific software nuggets* [Preprint]. arXiv. https://doi.org/10.48550/arXiv.2006.13129

Besker, T., Martini, A., & Bosch, J. (2018). Technical debt cripples software developer productivity: A longitudinal study on developers' daily software development work. In *2018 IEEE/ACM International Conference on Technical Debt (TechDebt)* (pp. 105–114). IEEE. https://doi.org/10.1145/3194164.3194170

Bethke, E. (2003). Game development and production. Wordware Publishing.

Bhattacherji, A. (2024, January 20). *Design discussion: Code smell*. ITNext. https://itnext.io/design-discussion-code-smell-bb55b19eb1a8

Blaschek, G. (2012). *Object-oriented programming: with prototypes*. Springer Science & Business Media.

Borowa, K., Zalewski, A., & Saczko, A. (2021). Living with technical debt – A perspective from the video game industry. IEEE Software, 38(6), 65–70. https://doi.org/10.1109/MS.2021.3103249

Brown, A. R. (2024). *Taming your dragon: Addressing your technical debt*. Springer. https://doi.org/10.1007/979-8-8688-0264-5

Chernishev, S.. (2021). *Принципы, паттерны и методологии разработки программного обеспечения. Учебное пособие для вузов*. [Principles, Patterns and Methodologies of Software Development]. Litres.

Cooksey R. W. (2020). Descriptive Statistics for Summarising Data. *Illustrating Statistical Procedures: Finding Meaning in Quantitative Data*, 61–139. https://doi.org/10.1007/978-981-15-2537-7_5

Cunningham, W. (2009, February 15). *Debt Metaphor* [Video]. YouTube. https://www.youtube.com/watch?v=pqeJFYwnkjE

Deakin University. (2025, May 16). Surveys & questionnaires. In Qualitative study designs. Retrieved May 23, 2025, from https://deakin.libguides.com/qualitative-study-designs

dyc3. (2020, July 9). *Yandere Simulator complete source code analysis - Code review* [Video]. YouTube. https://www.youtube.com/watch?v=LleJbZ3FOPU

Extra Credits. (2019, April 24). *Technical Debt - Improving the Production Pipeline - Extra Credits* [Video]. YouTube. https://www.youtube.com/watch?v=L5NfbpLmMHI

Engström, H., Marklund, B. B., Backlund, P., & Toftedahl, M. (2018). Game development from a software and creative product perspective: A quantitative literature review approach. *Entertainment Computing, 27*, 10–22. https://doi.org/10.1016/j.entcom.2018.02.008

Farley, D. (2022, September 28). *Types of technical debt and how to manage them* [Video]. YouTube. https://www.youtube.com/watch?v=1MBpK_PxEnU

Fowler, M. (2006, February 9). *Code Smell.* Martin Fowler. https://martinfowler.com/bliki/CodeSmell.html

Fowler, M. (2009, October 14). *Technical Debt Quadrant.* Martin Fowler. https://martinfowler.com/bliki/TechnicalDebtQuadrant.html

Fowler, M. (2019, May 21). Technical debt. *Martin Fowler*. https://martinfowler.com/bliki/TechnicalDebt.html

Griffith, I. D. (2014). *Technical debt management in release planning – A decision support framework* [Master's thesis, Montana State University]. MSU ScholarWorks. https://scholarworks.montana.edu/items/e374006a-0ab6-4992-9341-242da1a4a643

Hamedani, M. (2018, March 30). *Object-oriented programming, simplified* [Video]. YouTube. https://www.youtube.com/watch?v=pTB0EiLXUC8

Jesper, O., Erik, R., Terese, B., Martini, A., & Torkar, R. (2021). Measuring affective states from technical debt. *Empirical Software Engineering, 26*(5) https://doi.org/10.1007/s10664-021-09998-w

Lasater, C. G. (2006). *Design patterns*. Wordware Publishing.

Loster, F. (2023, June 6). *Your prototype matters - Avoiding technical debt early on* [Video]. YouTube. https://www.youtube.com/watch?v=VMNPm-35ZaY

Martic, R. C. (2009, September 22). *A Mess is not a Technical Debt.* Uncle Bob Consulting. https://sites.google.com/site/unclebobconsultingllc/a-mess-is-not-a-technical-debt

Martini, A., & Bosch, J. (2015). The danger of architectural technical debt: Contagious debt and vicious circles. *2015 12th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 1–10. https://doi.org/10.1109/WICSA.2015.31

Mason, S. (2025, March 16). *AI and the environment: How can gamers and developers mitigate the impact?* AI Journal. https://aijourn.com/ai-and-the-environment-how-can-gamers-and-developers-mitigate-the-impact/

McLeod, S. (2024, September 30). *Thematic analysis: A step by step guide.* Simply Psychology. https://www.simplypsychology.org/thematic-analysis.html

Nemchinskiy, S. (2021, November 12). *Что такое Технический долг (Technical debt) и к чему приводит его накопление?* [What is technical debt and what does its accumulation lead to?] [Video]. YouTube. https://www.youtube.com/watch?v=I04ANuuvvPg

Nemchinskiy, S. (2024, July 8). Технический долг при разработке ПО [Technical debt in software development]. *Foxminded*. https://foxminded.ua/ru/tekhnicheskii-dolg/

Newman, H. (2020, July 19). *In World of Warcraft: Shadowlands, new game tech and old code are at war*. VentureBeat. https://venturebeat.com/ai/in-world-of-warcraft-shadowlands-new-game-tech-and-old-code-are-at-war/

Pizka, M. (2004). *Straightening spaghetti-code with refactoring?* In *Proceedings of the 11th Working Conference on Reverse Engineering* (pp. 846–852). IEEE. https://www.researchgate.net/publication/221610982_Straightening_Spaghetti-Code_with_Refactoring

Rideout, L. (2017, October 5). *ThursDev: Accruing Technical Debt - Picking your battles, and paying the program piper* [Video]. YouTube. https://www.youtube.com/watch?v=MBfviUm-UHU

Roberts, J. (2020). *Clean C#*. Leanpub. https://leanpub.com/cleancsharp

ShadowZone. (2024, May 24). *"Doomed from the start" – KSP2 development history finally revealed* [Video]. YouTube. https://www.youtube.com/watch?v=NtMA594am4M

shounic. (2024, May 4). *They PATCHED Gordon Freeman from TF2* 😢😭 [Video]. YouTube. https://www.youtube.com/watch?v=UMi847MdESY

Singh, H., & Shareef, S. (n.d.). Qualitative interview: What it is & how to conduct one. *QuestionPro*. https://www.questionpro.com/blog/qualitative-interview/

Suryanarayana, G., Samarthyam, G., & Sharma, T. (2014). *Refactoring for software design smells: Managing technical debt*. Elsevier Science & Technology.

Tripathy, P. (2014). *Software evolution and maintenance : A practitioner's approach*. John Wiley & Sons, Incorporated.

Valve. (2024, April 22). *Team Fortress 2 update released*. Team Fortress 2. https://www.teamfortress.com/post.php?id=220641

Venters, C., Capilla, R., Betz, S., Penzenstadler, B., Crick, T., Crouch, S., Nakagawa, E. Y., Becker, C., & Carrillo, C. (2018). Software sustainability: Research and practice from a software architecture viewpoint. *Journal of Systems and Software, 138*, 174–188. https://doi.org/10.1016/j.jss.2017.12.026

Vidoni, M., Codabux, Z., & Fard, F. H. (2022). *Infinite technical debt*. *Journal of Systems and Software, 190*, 111336. https://doi.org/10.1016/j.jss.2022.111336

Web Dev Simplified. (2021, July 20). *How To Manage Technical Debt.* [Video]. YouTube. https://www.youtube.com/watch?v=nXMYHrERh80

Appendices

Appendix 1. Survey Introduction and Questions

Survey Title: Technical Debt in Game Development.

Introduction: This survey aims to gather opinions and experiences of game developers on technical debt. Your responses are anonymous and will contribute to research for my thesis.

The topic mainly concerns programmers, but non-programmers can have input as well.

Results gathered will be written as analysis in the research section of my thesis, then read and graded by teachers at Jamk University of Applied Sciences in Finland. The thesis will then be published on Theseus presumably at the end of May or later. The information gathered will be deleted once the thesis is completed.

You can answer as little or as much as you want, answering the survey could take 5-20+ minutes depending on how many optional questions you respond to.

Survey questions:

Section 1

- **Q1.1** What is your experience with game development? (Pick all that apply)

1. Hobbyist
2. SoloDev
3. IndieDev
4. Student
5. Educator
6. Professional
7. No experience
8. I have a background in general software development in addition to the other answer(s)
9. Other (Write)

- **Q1.2** What is your role in game development? (Pick all that apply)

- Programmer
- Game Designer
- Artist
- Narrative
- Audio
- Quality Assurance
- Management
- Other (Write)

- **Q1.3** How many years have you been involved in game development? (Pick 1)
    - 0
    - 1-2
    - 3-5
    - 6-9
    - 10+

- **Q1.4** How familiar are you with the term “technical debt”? (Pick 1)
    1. Very familiar
    2. Somewhat familiar
    3. Barely know about it
    4. Never heard of it

- **Q1.5** In your own words, what does technical debt mean to you? (Write response)

Section 2 - Definition

- **Q2.6** Does this definition make sense to you? (Yes/No/Other (Write))

Description: Technical debt is a metaphor used to explain situations where development has become more complicated and disorganized over time due to intentional or unintentional bad design or problem-solving decisions.

The time spent fixing those flaws in the future increases the longer they are ignored. This is compared to going into debt and eventually paying off said debt with interest (using more time or effort) when it would have been easier to do things properly from the start.

Examples:

- quick/shortcut decisions that save time now but will make development harder later on
- bad/spaghetti code that is hard to read or edit
- lack of documentation or testing that makes it harder to understand the system

It typically refers to programming, but in game development it can extend to:

- poorly optimized 3D models
- unorganized scenes
- no design documents/communication with teammates

- **Q2.7** Have you worked on a game project before? (Yes/No)

Description: Any project that took significant effort or time.

If you've only done prototypes or individual mechanics you may not be able to answer the questions you get from selecting "Yes", but feel free to take a look.

Yes (Continue to next section)
No (Go to Section 4)

Section 3 - Influence of TD on a game project

- **Q3.8** How clear and planned are your projects overall? (Scale 1-5)

Description: From start to finish, your general production timeline

- **Q3.9** How frequently do plans in your projects change after development started? (Scale 1-5)

Description: Mechanics getting cut or changed, entire sections of the game being scrapped etc.

- **Q3.10** How much do change of plans influence technical debt in your projects? (Scale 1-5)

Description: For example: The player script and 3D model were made around jumping, but now the jumping mechanic is removed. How much does that mess up your organization and next steps?

- **Q3.11** (Optional) Can you give an example of how change of plans influenced technical debt in your projects? (Write response)

Description: Extra time taken to remove things that aren't needed anymore, restructuring things to fit the new vision etc.

- **Q3.12** How often do you have to work under time pressure?

Description: Deadlines or events pressuring you to complete a task sooner than you'd like

- **Q3.13** How much do time pressure or deadlines influence technical debt in your projects?

Description: For example: You had days left before a feature needed to be added, so you didn't bother optimizing it or checking for errors

- **Q3.14** (Optional) Can you give an example of how time pressure influenced technical debt in your projects?

Description: Script or model that was thrown into the project on a deadline is now a bottleneck that is hard to fix and takes extra time and effort

- **Q3.15** Do you or your team write tests for your projects often?

Description: Either general testing or tests written for scripts (unit testing etc.)

- **Q3.16** How much does lack of testing influence technical debt in your projects?

Description: For example: Something was changed in the project and an unrelated thing broke, but no one knows why or how because there is no way to test it

- **Q3.17** (Optional) Can you give an example of how testing influenced technical debt in your projects?

Description: Extra time taken to figure out why something isn't working etc.

- **Q3.18** Do you or your team write documentation or guidelines for your project often?

Description: How to use scripts, animations, UI elements etc. Standardized scene or folder organization etc.

- **Q3.19** How much does lack of documentation or guidelines influence technical debt in your projects?

Description: For example: Some old scripts or animations are bugging out, but no one knows how they work

- **Q3.20** (Optional) Can you give an example of how documentation influenced technical debt in your projects?

Description: Information on how the project works is lost and doing anything with those parts of the project takes more effort

- **Q3.21** Do you or your team work with old/outdated/legacy systems in your projects?

Description: Examples include:

- outdated/unsupported software

- old scripts that are still used in major systems despite the person who wrote it not being on the team anymore

- reusing scripts/systems/game engines from other projects (think Valve's Source engine being made for Half-Life, but also used in TF2, leading to bizarre bugs)

- **Q3.22** How much does working with an old/outdated/legacy system influence technical debt in your projects?

Description: For example: You decided to reuse an old mechanic from a different project, but now realizing that it creates a lot of bugs that you aren't sure how to fix

- **Q3.23** (Optional) Can you give an example of how old/legacy systems influenced technical debt in your projects?

Description: Old bugs or models errors you use can't be ignored anymore, causing you to either restart or spend more time fixing them

- **Q3.24** How much does lack of experience influence technical debt in your projects?

Description: Someone came up with a solution on their own and a few months later that solution is now making adding anything new that involves it very difficult because it's too messy

- **Q3.25** (Optional) Can you give an example of how inexperience influenced technical debt in your projects?

Description: Think of your first scripts, animations, models etc.

- **Q3.26** How often do you or your team address technical debt in your projects?

Description: Such as refactoring, planning around it, documentations or tests to minimize it etc.

- **Q3.27** How much stress, frustration or burnout does technical debt cause for you or your team?

- **Q3.28** How often do you see a project failing because of technical debt?

Description: Failing to profit, finish development etc.

Section 4 - Opinions

- **Q4.29** In your opinion, does technical debt affect the success of a game?

Description:

- Yes, every time
- Depends on the game
- Not really
- Other (Write)

- **Q4.30** In your opinion, when is it okay to take on technical debt in game development?

Description: Such as copypasting existing code to add a new mechanic quicker or not bothering with tests or clever solutions because it would take extra time to do it cleaner

  - Always
  - During prototyping
  - To meet a deadline
  - To ensure a game is released on time
  - Never, it should always be minimized
  - Other (Write)

- **Q4.31** (Optional) Do you think game developers should invest more time in preventing and managing technical debt, even if it means delaying the game's release? Why or why not?

- **Q4.32** (Optional) Has learning about technical debt changed how you approach development, design or planning? If so, how?

- **Q4.33** (Optional) In your opinion, what causes technical debt? (Write response)

Description: Deadlines, lack of planning, feature creep, inexperience etc.

- **Q4.34** (Optional) How do you or your team manage technical debt? (Write response)

Description: Detailed planning, coding guidelines, design patterns, regular refactoring, documentation, code reviews etc.

- **Q4.35** (Optional) What do you wish you knew about technical debt? (Write response)

- **Q4.36** (Optional) Tell a story, what are your experiences with technical debt?) (Write response)

Description: Personal experience or about anything you've heard about. Games being delayed or cancelled, communities upset at unfinished or unstable games, old/legacy systems causing issues etc.

- **Q4.37** (Optional) Open response: If you have anything else to add, feel free! Also, if you want to talk more in-depth about the topic, I can contact you through Discord. You can leave your username here! Thanks for taking the time to respond! (Write response)

Appendix 2. Survey Charts

Figure A1

Section 1, Question 3

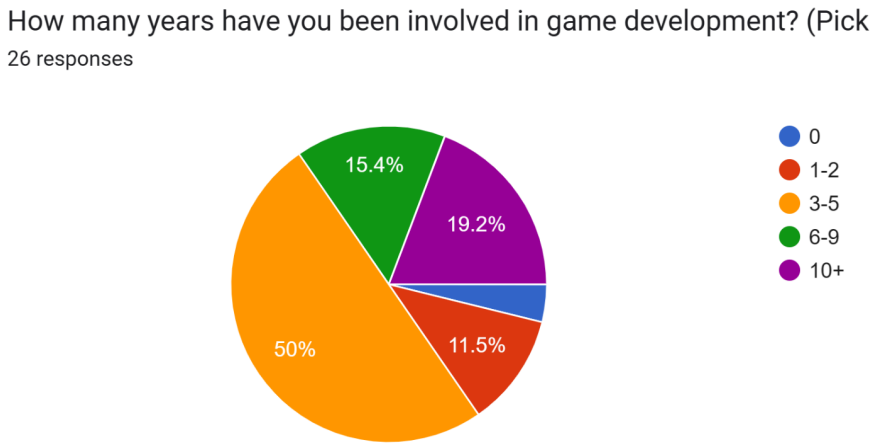


Figure A2

Section 1, Question 4

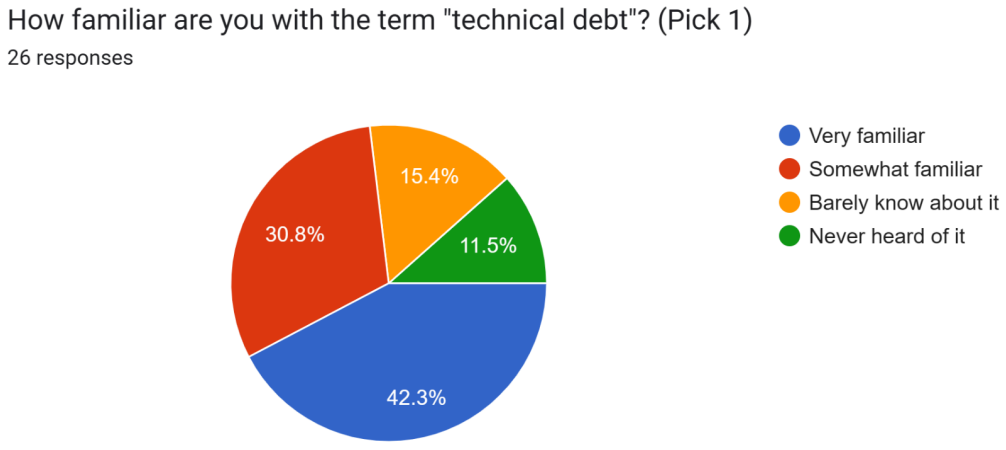
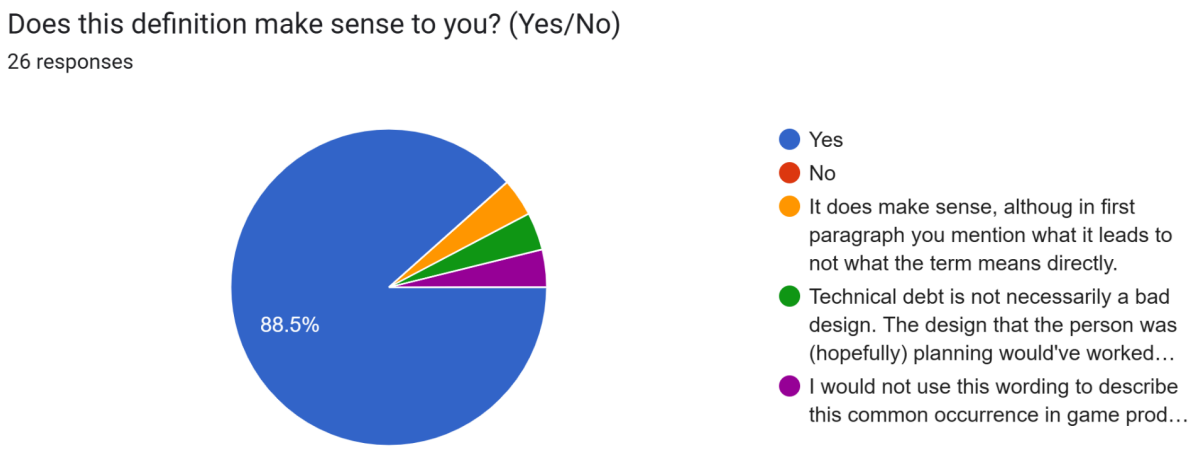


Figure A3

Section 2, Question 6



Note. The custom responses in full are as follows:

- "It does make sense, although in the first paragraph you mention what it leads to, not what the term means directly."
- "Technical debt is not necessarily a bad design. The design that the person was (hopefully) planning would've worked out for what the initial goal was. If initially you wanted to build chess, but suddenly you decided to make a first person shooter, it doesn't matter how good your design was, you can't really re-use it for an FPS. I think that lack of documentation is not a technical debt either. In fact, you have nothing to change in case it's not there. From my perspective, it's like saying that the lack of my completed MMO RPG is a technical debt."
- "I would not use this wording to describe this common occurrence in game production, as it is misleading in my opinion. Though having a badly organized term describing bad organization is fitting."

Figure A4

Section 3, Question 8

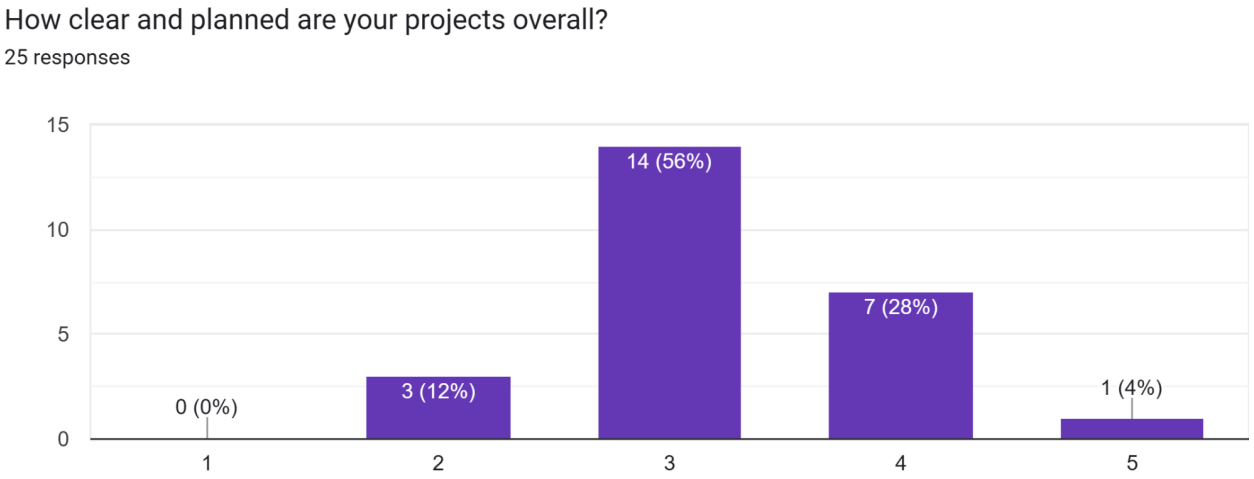


Figure A5

Section 3, Question 9

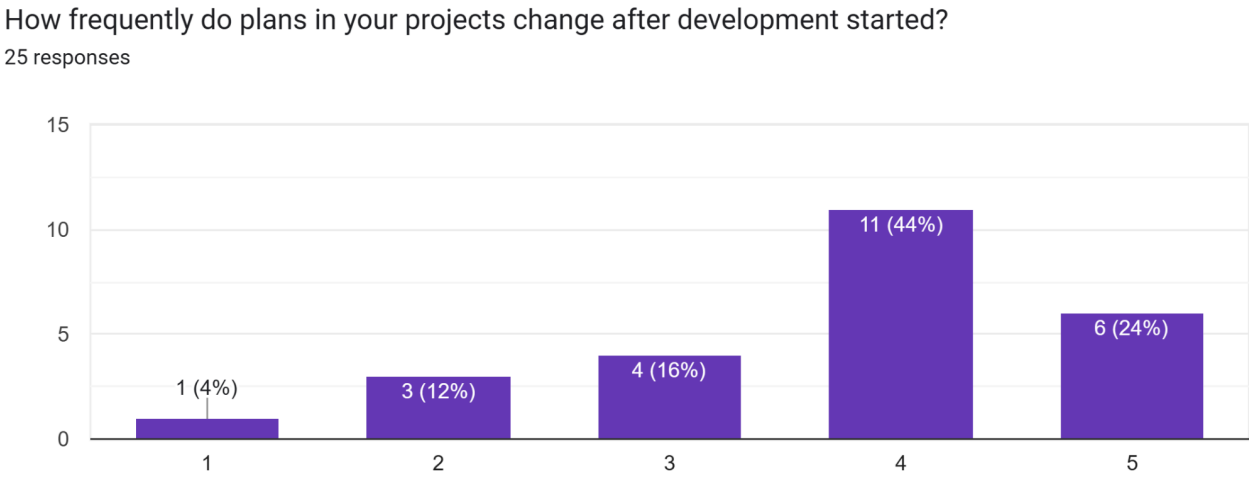


Figure A6

Section 3, Question 10

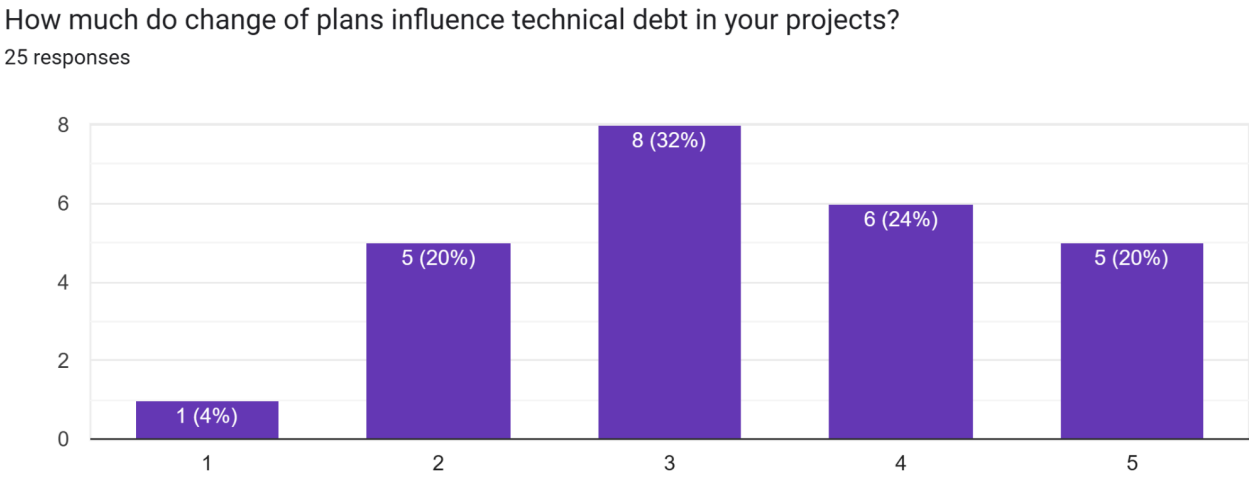


Figure A7

Section 3, Question 12

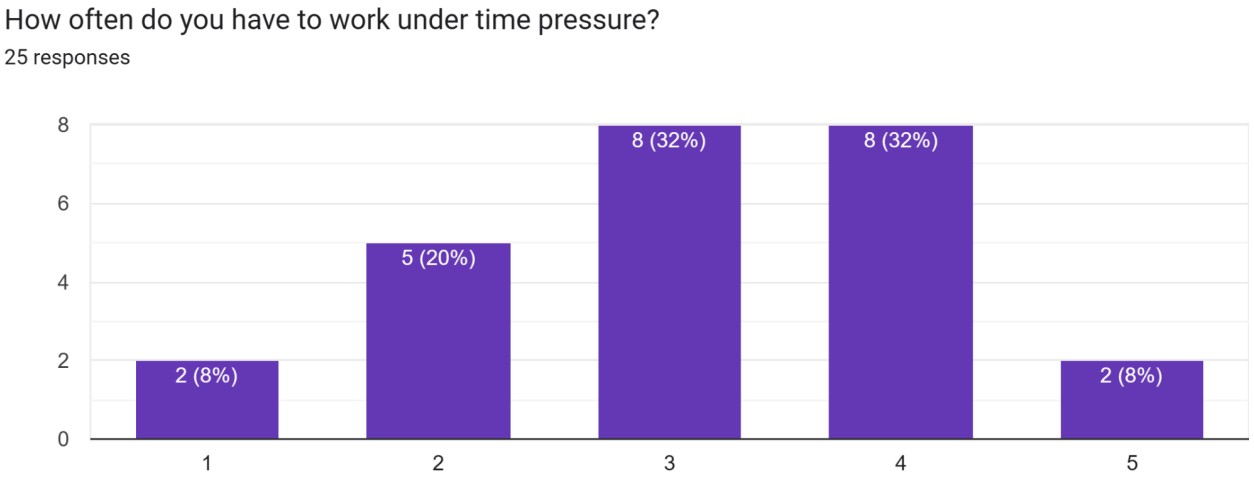


Figure A8

Section 3, Question 13

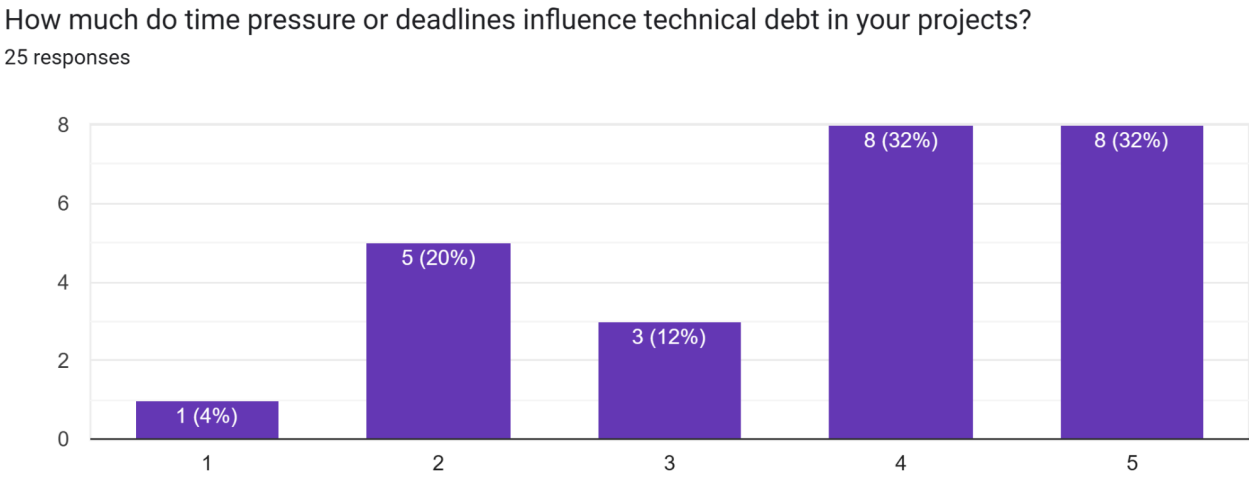


Figure A9

Section 3, Question 15

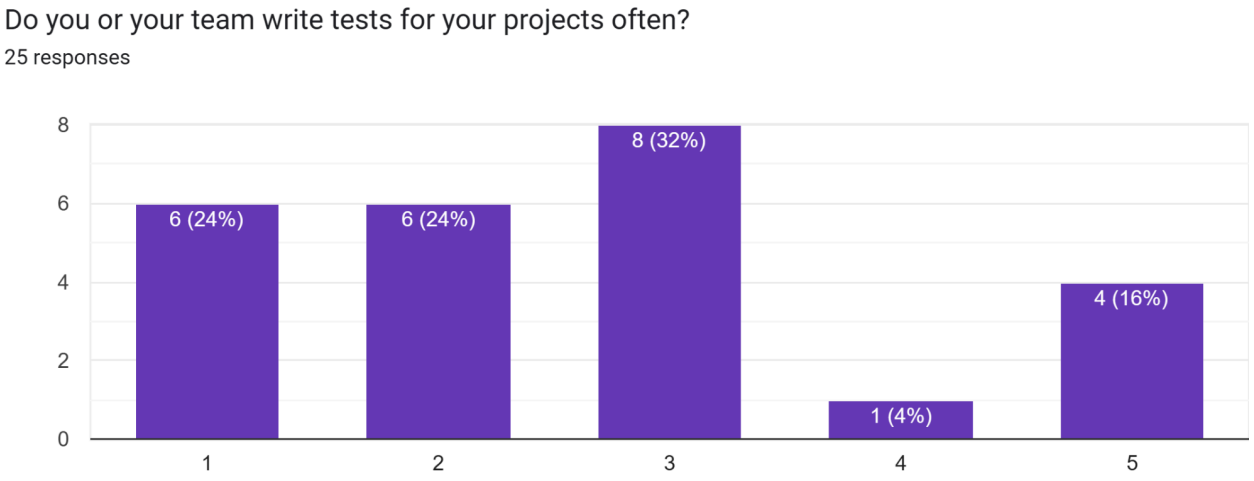


Figure A10

Section 3, Question 16

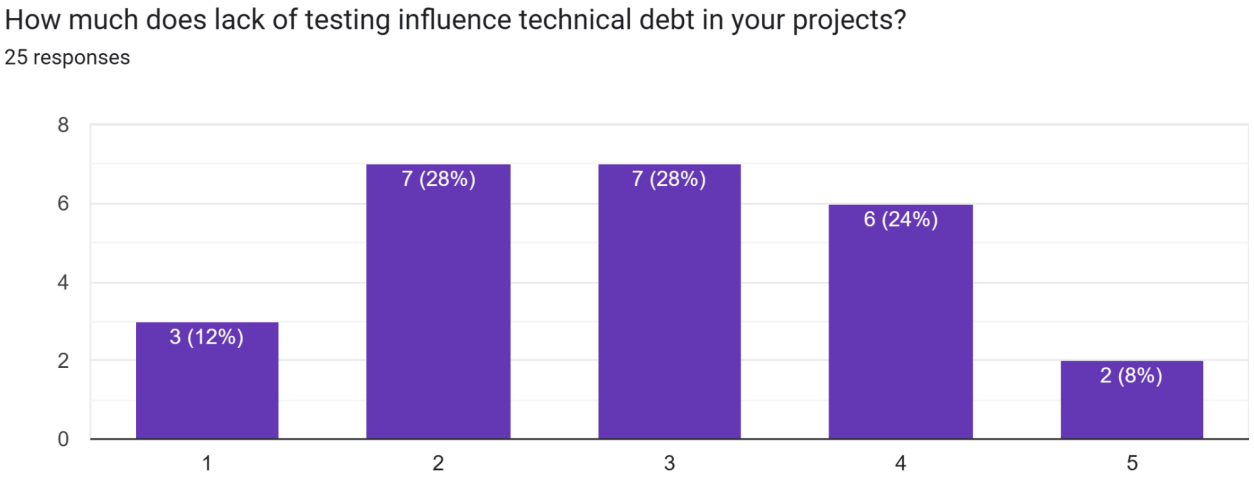


Figure A11

Section 3, Question 18

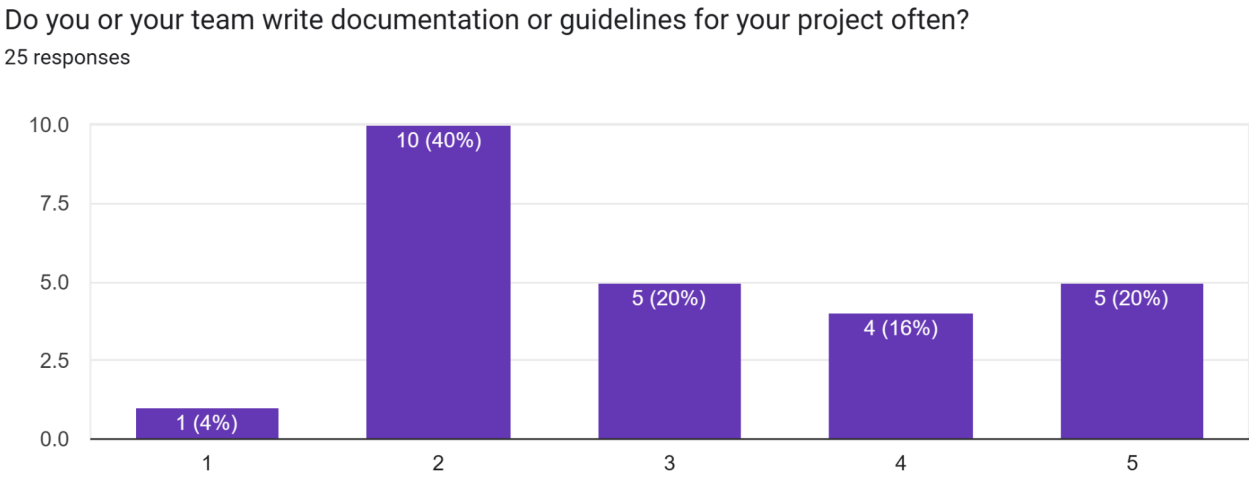


Figure A12

Section 3, Question 19

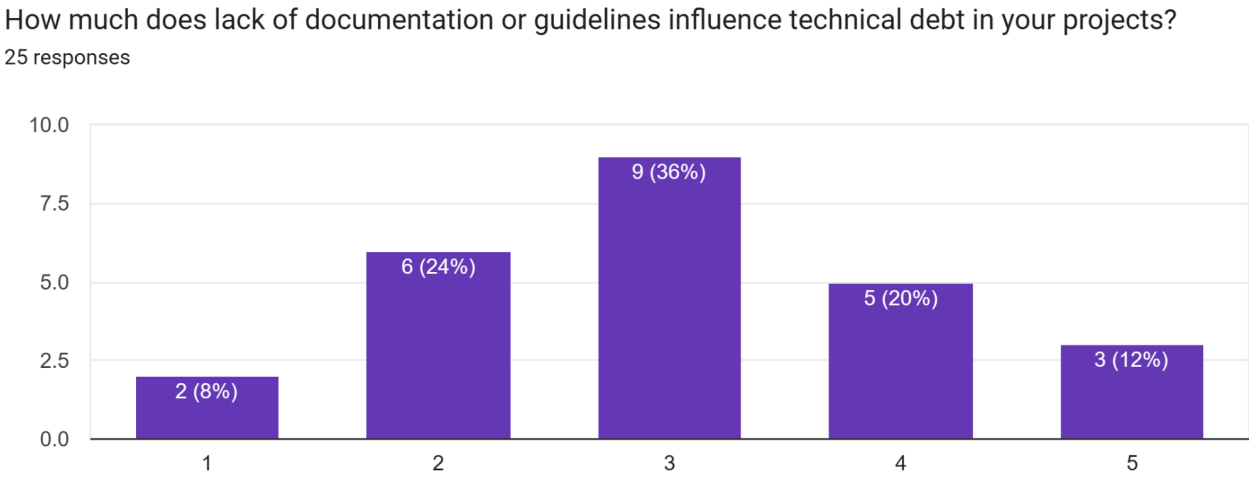


Figure A13

Section 3, Question 21

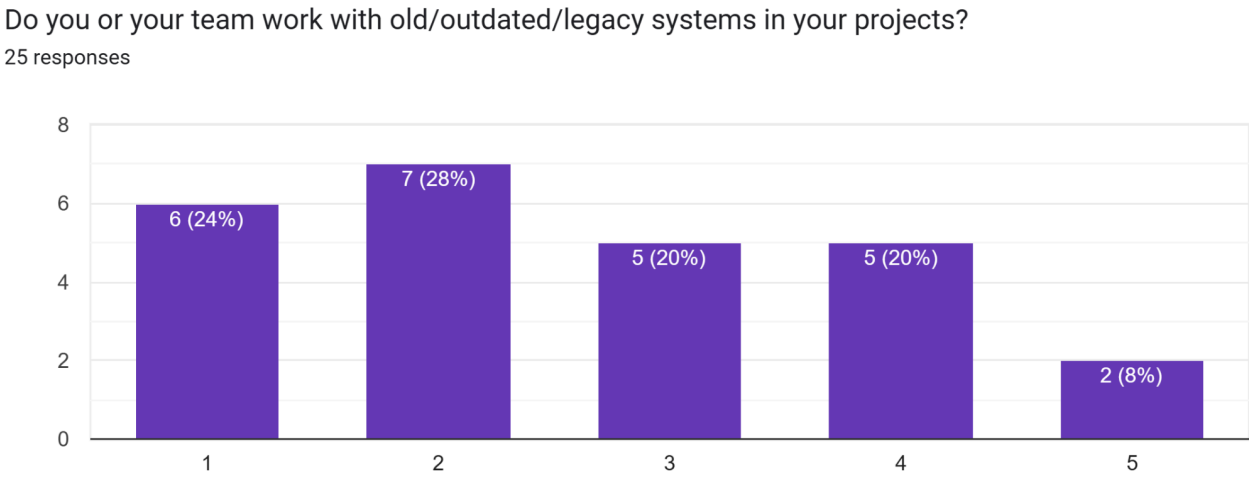


Figure A14

Section 3, Question 22

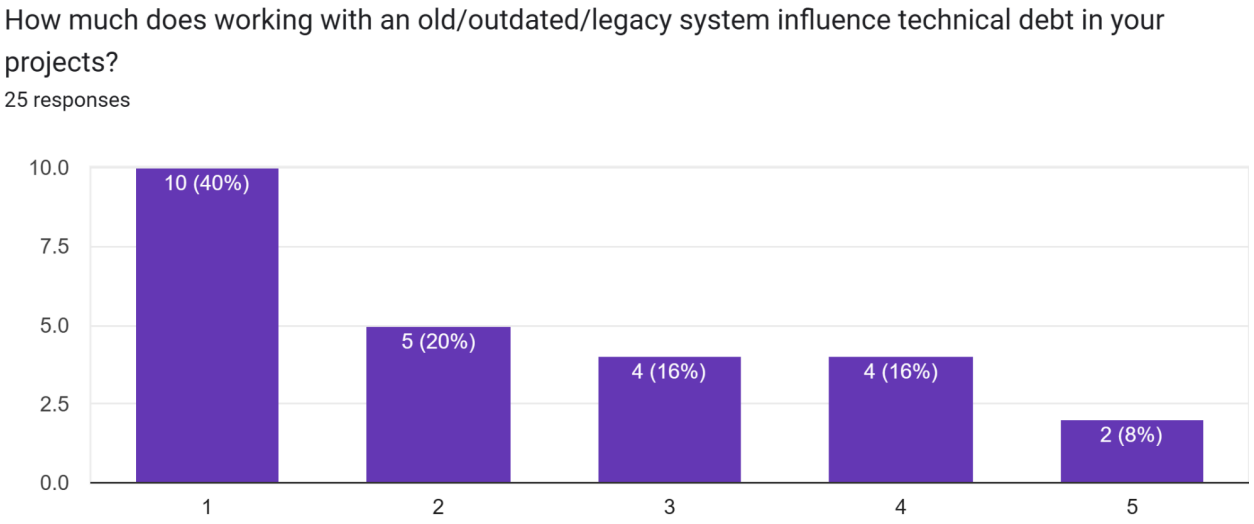


Figure A15

Section 3, Question 24

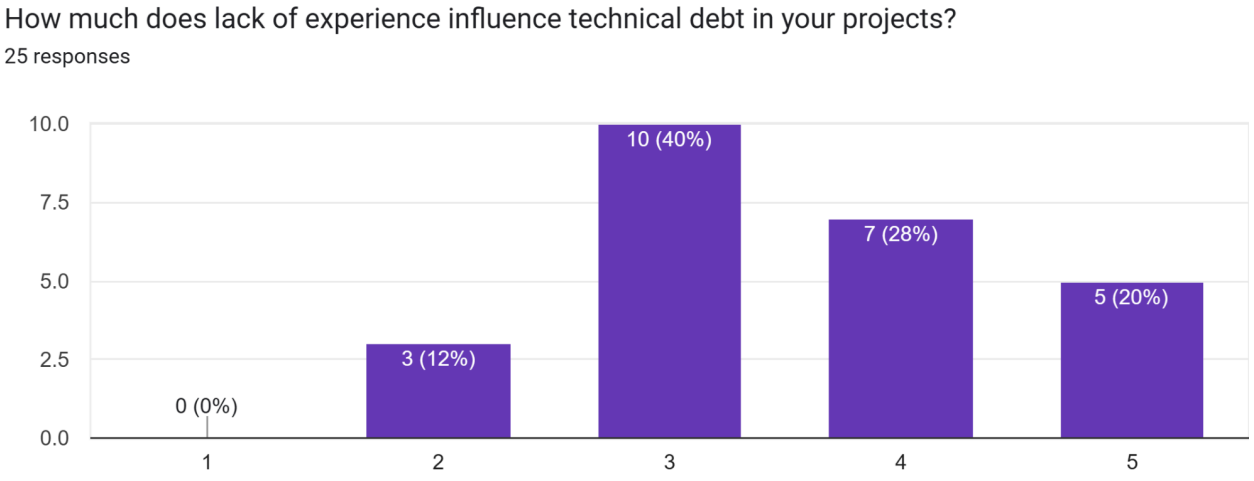


Figure A16

Section 3, Question 26

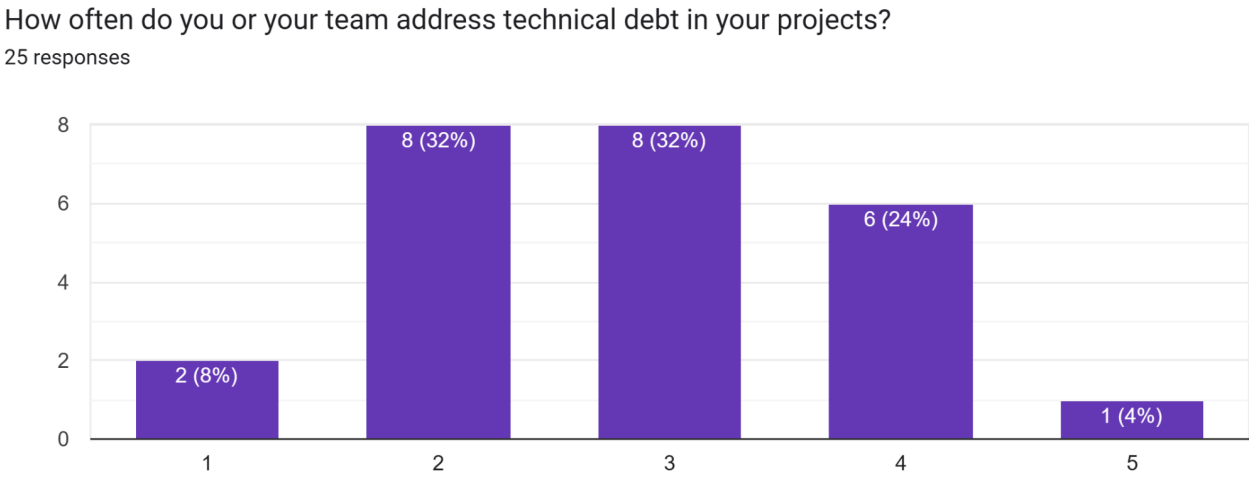


Figure A17

Section 3, Question 27

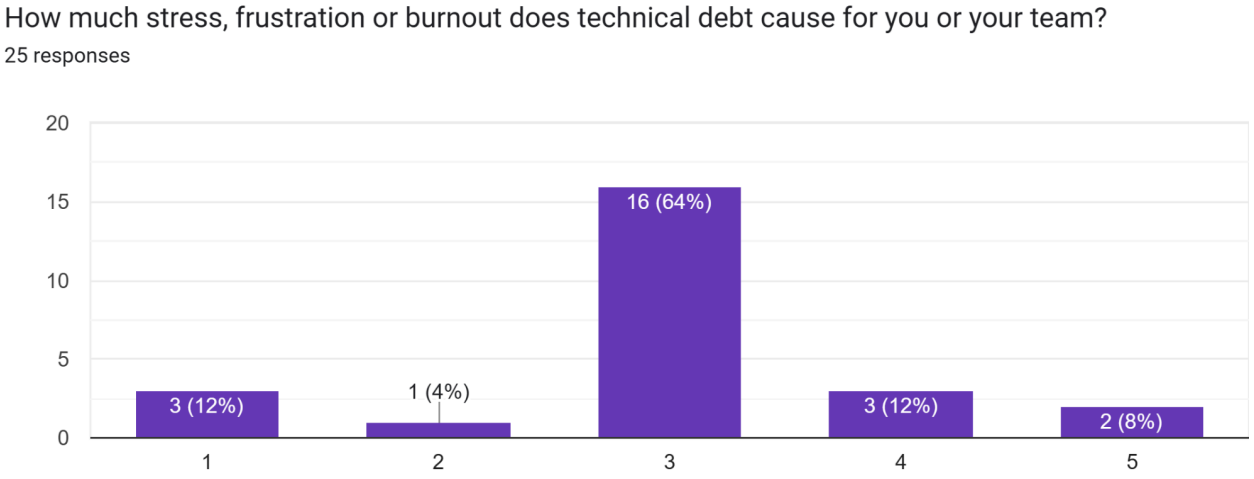


Figure A18

Section 3, Question 28

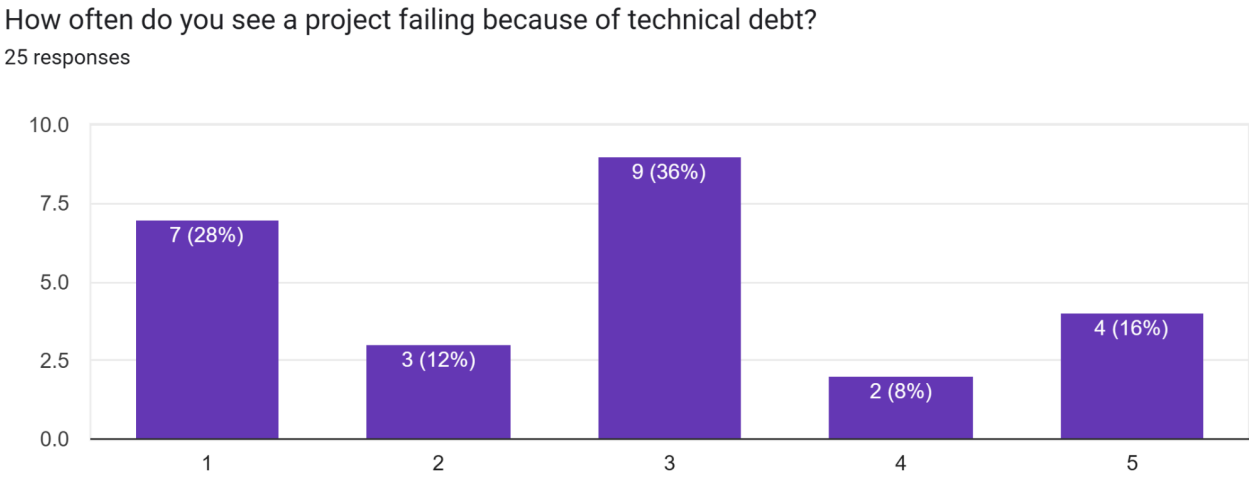
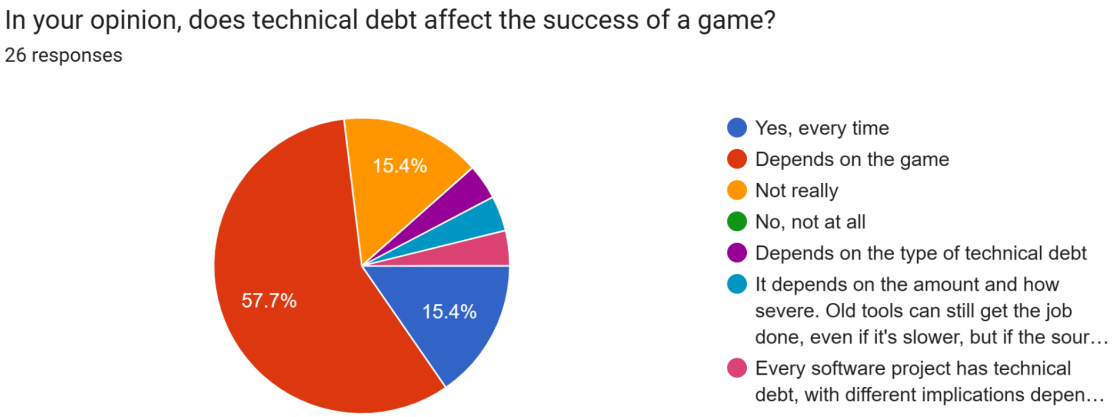


Figure A19

Section 4, Question 29

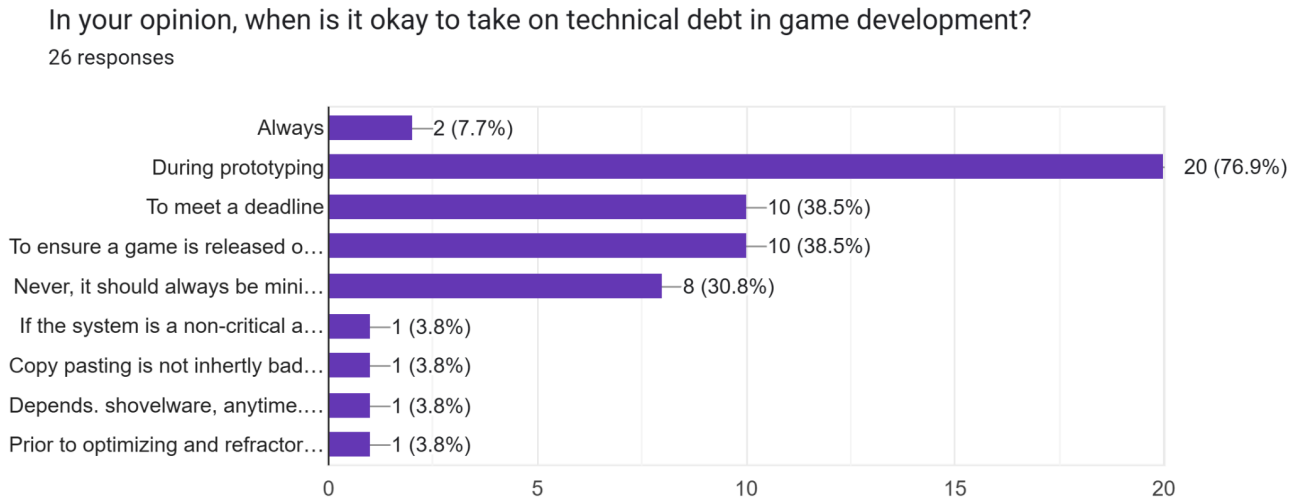


Note. The custom responses in full are as follows:

- “Depends on the type of technical debt.”
- “It depends on the amount and how severe. Old tools can still get the job done, even if it's slower, but if the source code has something inherently wrong due to a quick fix solution, it affects the whole game.”
- “Every software project has technical debt, with different implications depending on the implementation. A Saas based game may end up broken due to it, but a single release may be fine.”

Figure A20

Section 4, Question 30



Note. The custom responses in full are as follows:

- “If the system is non-critical and rarely used it doesn't need to be written the most performant way or to be the most scalable.”
- “Copypasting is not inherently a bad thing if you know that the code is good already.”
- “Depends. Shovelware, anytime. Any more serious game, only during prototyping.”
- “Prior to optimizing and refactoring.”

Appendix 3. Survey Tables

Table A1

Experience vs Familiarity

Experience vs Familiarity

Amount	Experience	Very familiar	Somewhat familiar	Barely know about it	Never heard of it
15	Hobbyist	6	5	2	2
10	SoloDev	5	4	0	1
11	IndieDev	5	4	0	2
16	Student	5	5	4	2
2	Educator	1	1	0	0
11	Professional	8	2	0	1
1	No Experience	0	1	0	0
4	I have a BG in..	2	2	0	0
1	Other (Entrepreneur)	1	0	0	0

Table A2

Experience vs Familiarity in percentages

XP vs Familiarity Percenta...

Amount	Experience	Very familiar	Somewhat familiar	Barely know about it	Never heard of it
15	Hobbyist	40.00%	33.33%	13.33%	13.33%
10	SoloDev	50.00%	40.00%	0.00%	10.00%
11	IndieDev	45.45%	36.36%	0.00%	18.18%
16	Student	31.25%	31.25%	25.00%	12.50%
2	Educator	50.00%	50.00%	0.00%	0.00%
11	Professional	72.73%	18.18%	0.00%	9.09%
1	No Experience	0.00%	100.00%	0.00%	0.00%
4	I have a BG in..	50.00%	50.00%	0.00%	0.00%
1	Other (Entrepreneur)	100.00%	0.00%	0.00%	0.00%

Table A3

Years vs Familiarity

Years vs Familiarity

Amount	Years of XP	Very familiar	Somewhat familiar	Barely know about it	Never heard of it
1	0	0	0	0	1
3	1-2	0	1	1	1
13	3-5	5	4	3	1
4	6-9	2	2	0	0
5	10+	4	1	0	0

Table A4

Role vs Familiarity

Role vs Familiarity

Amount	Role	Very familiar	Somewhat familiar	Barely know about it	Never heard of it
15	Programmer	8	5	1	1
11	Game Designer	4	4	1	2
11	Artist	1	5	3	2
4	Narrative	1	2	0	1
2	Audio	1	1	0	0
7	Quality Assurance	2	4	1	0
9	Management	4	2	2	1
1	Other (Tech Artist)	0	1	0	0

Table A5

Programmers and Non-programmers vs Familiarity

	Amount	11	8	4	3
Amount	Tech/Non-tech	Very familiar	Somewhat familiar	Barely know about it	Never heard of it
15	Programmers	8	5	1	1
11	Non-Programmers	3	3	3	2
	Programmers	53.33%	33.33%	6.67%	6.67%
	Non-Programmers	27.27%	27.27%	27.27%	18.18%

Appendix 4. Summaries of Open-ended Responses

The following section contains color-coded summaries of long-form responses, organized by recurring themes. Colors were assigned based on topic categories identified during thematic analysis.

Figure A1

Color legend used to categorize themes in qualitative response

- Time pressure or deadline
  - Time-related limitations as an obstacle
- Short-term solution or shortcut
  - Any mention of a worser solution being used instead of a robust one
- Code quality or design
  - Related to codebases, refactoring and optimization, or quality of a task
- Outdated tools or legacy code
  - Libraries, folders, software, frameworks, age of a project
- Planning
  - Quality of plans, changes, scope, schedules and development leadership
- Choice or action
  - Deliberate choice of an individual that is not a leader or manager
- Amount of work or complications
  - Additional time or resources spent as consequences of actions or events
- Business or management
  - Decisions of management or publishers based on business interests or pressures
- Prototypes, demos or concepts
  - Early stages of a project
- Inexperience
  - Related to skill level of developers, but not leaders
- Documentation, testing or knowledge
  - Anything used to share information within the team, including in-person communication

Figure A2

Summarized responses and themes for Q1.5 - What is TD to you?

- Unresolved issue, no time to fix, bandage fixes
- Balance between easy short-term vs hard long-term solutions
- Choice to make short-term solutions for speed. Also outdated libraries
- Time pressure, short-term solution over quality
- Bad code that slows development, adding new features is too hard. Also outdated tools
- Manifestations that need to be refactored due to optimization or bugs
- Amount of work for easy short-term vs hard long-term solutions
- In implementation: Cost of change, correlated with regression and time to implement
- Action of using short-term solution for deadline over quality
- Suboptimal code due to changing plans/designs, shortcuts for speed
- Shortcuts for speed
- Stacking features, poor design, rigid system, not planned features
- Bad or no planning. Building on prototype
- Shortcuts to save time for deadlines, hard to fix after
- Bad planning = more work in the future
- Shortcuts that you hope to fix later, usually for demos or trials
- Code that causes problems later because hard to work with or expand on
- Accumulation of old code, methodologies, bugs ignored for business reasons
- Messy code slowing development in the long run
- Poor maintenance of code. Usually cus team inexperience, budget or deadlines
- "I'll update this later"

Figure A3

Summarized responses and themes for Q3.11 - Change of Plans

- A prototype turned into a game released on Steam, decisions made early in the project were obviously bad in the long run, but the assumption was that there would be no long run. When development continued, all those decisions forced the developer to implement further short-cut decisions like copy-pasting parts of the code and editing all of them when something needed to change.
- Decoupling is an effective way to adapt to any change of plans more easily.

- Technical debt is not hard to avoid if people are aware of it. Most of it is caused by time pressure leading to misunderstandings or lack of planning.
- Unity updated a large framework that caused substantial code structure changes, a project built on this frame took a month to convert. The event is rare, but significant.
- A project took 3 months to switch from Unreal Engine's physics movement to a custom one, which also required remaking the rest of the system.
- Legacy tools needed to be found and updated because the original creators of the tools were no longer in the team and were insufficient for current project requirements.
- A 3D model causing multiple issues when implemented. "Spending two extra hours to make a 3D model properly would save ~30 hours elsewhere".
- "Technical debt may have crippled some core mechanics, but instead gave us **invaluable insight into how the same systems can be made much easier, faster and more robust**." (Learned Experience - Accidental/Prudent TD)
- A project without a design document continuously changed due to feature creep or change in direction during 6 months of attempting to start a company.
- A shooter game designed to be a free-for-all was later changed to a team deathmatch gamemode, causing a lot of rewriting for the system that was not designed to support it.
- A decision was made to add streaming support to the game that was built on an engine that had no architecture for it, requiring all systems to be revisited to ensure their safety and performance.
- In time-pressure situations a shortcut is required to reach the goals in time, a change of plans worsens the issue.
- The developer wanted to improve a weapon system but was unable to do so because it would require rewriting the system entirely, **leading to them leaving the system as is.** (Contagious TD)
- Changes in plans increase risk of technical debt.

Figure A4

Summarized responses and themes for Q3.14 - Time Pressure

- TD occurs from early design decisions. Time pressures cause additional shortcuts because it's outside of the scope of the project, not "it takes more effort".
- Quick changes lead to parts being scrapped for a shortcut solution. Scope needed to be adjusted to fit the abilities of the team.

- Mechanics that need refactoring or rewriting are instead solved with shortcuts that work well enough.
- Unoptimized features written in OOP get cut because there is no time to rewrite it.
- When under time pressure the developer gives attention to keep code decoupled, however poorly optimized, bad code is made because it still functions as intended.
- Time pressures change timelines, timelines change how features are developed.
- No time constraints, if there were, half of the features would be cut.
- Projectiles mechanic was implemented poorly due to time constraints that likely caused performance issues later.
- Three sprints of work disrupted due to new requirements for the next build on short notice.
- TD was ignored because the **game was live** and had limited time to update systems.
- Two weeks to take a prototype to demo required a lot of shortcuts.
- Script added into the project because it solved a problem temporarily.
- Poorly scalable code which made it harder to expand the project.

Figure A5

Summarized responses and themes for Q3.17 - Testing

- Unit tests and test tools are a great time saver. Automated tests trivialized testing of thousands of game levels after changes, saving testers time.
- Unit tests are not common in gamedev, because bugs are found through gameplay. Logs and debugging tools provide enough to identify problems.
- Untested products lead to unplanned schedule or feature changes and how implementation is handled.
- Bugs were found in systems that are undocumented nor tested. No one on the team knew how the system worked, or assumed it was intentional.
- Testing is unreliable and time consuming in most cases, iteration and internal testing proved more efficient. Prints and data logs locate most simple bugs.
- Depends on the tests. Executive decisions made before test results come in invalidate the testing unless there are significant results.
- Meaningful TD does not occur at testability. The real problem isn't "is this code doing what it intended," it's "is the intent of this code the right thing at all"
- Testing can become TD itself if not maintained. Source code management and pull request reviews ensure this does not happen.

- Boss instructed that there was no time for writing tests so none were made, everything tested internally.
- A character model was tested for animation, but the weight painting was done incorrectly, adding TD.
- The lead programmer refused to acknowledge an issue that would become worse later on.

Figure A6

Summarized responses and themes for Q3.20 - Documentation

- Settling and vetoing decisions is important. Documentation keeps track of conversations and decisions, so that features are not misunderstood and the developer knows what to do short-term and long-term.
- Writing documentation when you are the solo developer of your role is unmotivating because others will not understand it. Documentation is time consuming and tedious and can take a third of development time.
- Code should be self-documented. Scripts with poor clarity that require outside assistance are likely to have bad naming.
- It is possible that better design decisions could be made with better documentation in cases where a gameplay feature needs to be implemented quickly for evaluation.
- Time constraints make documentation worse. Talking to people involved solves the issue but costs time when there is uncertainty on current plans.
- The developer typically writes documentation or self-documented code, but the current project has neither, making it difficult to understand systems.
- Lead programmer does not provide documentation and instead spends hours explaining the system when no one understands it.
- A key person leaving the team can increase TD due to knowledge drain from the project.
- Documentation helps developers plan for the future, but is not TD. TD is simply bad code.
- No documentation in a project made everybody confused and unaware of the state of the project.
- Instead of learning a better system, the developer used a worser one that increased CPU time unnecessarily.
- Unity Engine project folder is severely unorganized.

Figure A7

Summarized responses and themes for Q3.23 - Legacy Systems

- Unity store assets are often not modular, outdated or poorly made.
- Legacy systems are **often untouched** due to bad code breaking new features easily. Legacy code is TD that has not been refactored properly.
- Modularity eliminates almost all issues with legacy systems.
- Updates to plugins, legacy code written by people that are no longer in the project and dependencies on outdated software all contribute to TD.
- Not a lot of outdated systems, but issues could arise from such.

Figure A8

Summarized responses and themes for Q3.25 - Inexperience

- It is impossible to avoid TD from inexperience. The developer improved over the years, but expects to find new issues in the future and continue improving.
- Inexperienced developers write mechanics to work as is, without further planning. While experienced developers are able to **predict and prepare** for such cases.
- The CEO of the company created half of the codebase using GenAI, thousands of lines of code that are incomprehensible and serve as the base for the project.
- Poor organization resulted in strange, entangled behaviours due to failing to follow conventions. Later, neglect to enforce conventions made it harder to teach the team to follow them.
- Inexperienced 3D modeller had to manually edit animation key frames because the level sequencer was not saved.
- Inexperienced developers often make something that is already present in the project, or has poor scalability and usability.
- Organization does not always depend on experience, but it helps.
- A developer rewrote 80% of the codebase in 3 out of 4 large projects after joining them in order to fix inexperienced developers' work.
- Most juniors are not productive enough to contribute enough code to be a problem.

Figure A9

Summarized responses and themes for Q4.31 - Investing Time to Manage TD

- Games should be structured as logically as possible. Shortcuts are acceptable when a project is nearing release or deadline. Issues can be fixed after release, and if they cannot, then it indicates issues that needed to be addressed earlier in development.
- It is case-based, but in general the more TD is prevented, the better. Shortcuts are acceptable in non-critical situations, unless all problems require shortcuts, in which case it shows an issue with design.
- TD affects release and value of the project too much to be ignored. Investors might avoid projects with severely low quality codebases.
- The industry moves too fast to be preoccupied with TD, it only matters in projects that have direct consequences for your business.
- Depends on the type of TD. Anything minor can be ignored, while issues that affect the user experience need to be addressed before release.
- There is no need to dedicate time on TD when it is not necessary, such as when the game will be released soon and never touched again.
- It is not possible to prevent TD, because all code is debt, which can be good or bad. Improving architecture should be a continual investment.
- It depends on the return of investment for the project at the time. TD should be managed at some point, and not left for a later time.
- “Naive answer: Yes, pragmatic answer: No. A perfect world would have perfect code. But this is not a perfect world.”
- It is always a business decision, not the developers. If project managers do not prioritize it, it is not prioritized.
- Game developers should be aware of their TD, the type of debt and what it means in production.
- Corporate should not be forcing developers to implement bad solutions for the sake of a deadline.
- TD should always be managed when it affects game performance, in other cases it can be accepted.
- TD should be minimized, but publishers likely have to make that decision, rather than the developers.
- TD should be minimized, because it is important to release a finished product rather than a buggy one.
- TD should be minimized, because otherwise it sacrifices game quality.

- TD should be minimized, because it leads to bugs and worsens the user experience.
- “We already have enough buggy games”
- “Good planning takes time”

Figure A10

Summarized responses and themes for Q4.32 - Learning about TD

- Learning about TD came with experience. Each next project a better approach can be taken and faults become evident in past projects. The developer notes making similar setups that worked before when applicable.
- TD was already visible and considered, but time constraints and unexpected difficulties ruin even the most meticulous planning. Work with what is possible and avoid the same mistakes again.
- The developer learned to take a different approach to legacy and “green field” engineering, expecting more TD in older projects.
- The developer learned to do more planning and develops features that can be traced when issues occur.
- TD is taken into account to manage milestones better to be planned and addressed in a controlled manner.
- A designer ensures to write detailed documentation and talk to developers, as well as anticipate future needs and write flexible code.
- The developer learned to watch out for common pitfalls and write code as future proof as possible.
- The term is badly worded but has not changed the views to minimize it.
- “It further strengthened the fact why delivering a sloppy product doesn't work.”
- Learning about TD affected the developer “slightly”, they are “still lazy”.

Figure A11

Summarized responses and themes for Q4.33 - Causes of TD

- Deadlines, lack of planning, feature creep, inexperience (All examples stated)
- Deadlines, lack of planning, feature creep, inexperience (All examples stated)
- TD happens because everyone operates on imperfect information, the cost of getting perfect information may be too high to warrant. Changes in design, new constraints, poor

understanding of architecture and human error can all lead to TD. TD is a feature of development and is neither good nor bad, only needs to be managed.

- Overcomplication is much more dangerous than TD. TD arises not from time pressure, but an unrealistic scope. Complex, future-proof design may never be used or become detrimental, despite the time dedicated to it originally. Proper design documentation, planning and pragmatism can eliminate almost all of TD.
- Lack of leadership or planning. In indie development, people treat it as both a job and a hobby, which is unrealistic to work as a business, sometimes it is necessary to choose.
- Deadlines, lack of planning, feature creep, inexperience are all good examples, but small occurrences. Lack of leadership is a big factor. The developer has only seen major TD from legacy code in projects with 100+ programmers, where vision is unclear and individuals focus on their own parts only.
- Deadlines, lack of planning, feature creep are good examples. Inexperience is rarely the cause.
- Deadlines, lack of planning, feature creep, inexperience (All examples stated). Lack of communication and bad management.
- "Early prototyping, MVPs not being rewritten properly before being moved to full production stage."
- "Poor planning, deadlines, corporate interference, inexperience - mostly short term."
- "Inexperience, lack of planning, deadlines and the lifetime of the project."
- Lack of planning and changes in the team.
- Deadlines, lack of planning, feature creep, inexperience (All examples stated).
- "Inexperienced and out of touch management and planning."
- "Lack of planning, inexperienced developers or management."
- "Laziness and time constraints."

Figure A12

Summarized responses and themes for Q4.34 - Managing TD

- Games are an art, the end result of which may be not what was expected and may be disliked by some, design or planning cannot prevent that. Games must be flexible and modular. When starting a project, it is hard to predict the process and outcome, but correct frameworks can support modular features. Code reviews and conversing with others help ensure implementation fits within the existing code. Leads and seniors managing code

reviews well helps juniors learn quicker and better, ensuring no shortcuts make it into the project.

- Good documentation and planning and avoiding feature creep. Planning several different scopes to ensure it is possible to achieve an end goal.
- All examples given are good (Detailed planning, coding guidelines, design patterns, regular refactoring, documentation, code reviews), aside from design patterns. TD requires allocating a fixed time budget to address it properly. Managing TD needs to be balanced against fixing bugs and implementing features.
- Good planning, creating patterns and templates, continuous iterative refactoring and mindset to simplify systems.
- Code reviews, regular refactoring, documentation of TD.
- Planning, common sense, a lot of testing, conversing about effects of changes and shortcuts.
- Planning and modularity for easier refactoring.
- Semi-regular refactoring, code reviews, planning and flexibility with deadlines when possible.
- The developer does not have a deadline. Modularity, creating templates and testing are all used to manage TD.
- Several sprints are dedicated to TD per year.
- Coding guidelines and documentations.
- Only writing code that can be understood.

Figure A13

Summarized responses and themes for Q4.35 - Wish to know about TD

- Measurements to calculate how much addressing certain TD will improve game performance or development.
- The existence of TD should be discussed more often.
- Sometimes TD has to be accepted and the project needs to move on.
- “How detailed technical documentation should be.”
- “A better name for the term.”

Figure A14

Summarized responses and themes for Q4.36 - Additional Words & Personal Stories

- “There are some companies out there that buy code assets from the marketplace assuming that they are like plugins, and that anything that you put in the game is already interconnected and perfectly functioning. The only thing is that the programmer tries to tangle things up creating a quick, yet bad, communication between the two. Sometimes people try to make something be too extendable. They might not even not meet their requirements, but meet others. They spend a lot of time on that, to then only realise that they have to refactor that. It's not always the case, but some engineers like to over-engineer things, especially when it comes to some people working with Lyra.”
- “A long time ago, when I joined a new company, one of the designers was full-time on a tutorial that was already done, it just required constant upkeep, because it was a giant node graph of every single thing that had to happen in order. Every node then needed to reference each specific target by its entire path in the hierarchy as a hand-typed string, not a reference. So, if a button had to be highlighted, it would be a single node with some the path like /UI/HUD/Menus/PlayerInventory/Container/Toolbar/Buttons/SwapButton, and every single time anyone changed the name or order of anything, the entire tutorial would break, it was madness.”
- “Not strictly game development related, but programming. Developing a web based system with an old framework. The old framework was ok at the time when the project started, but as time went on it started to suffer from performance, compatibility and other issues. Some newer servers wouldn't even support that old framework anymore. At some point the team decided to move the project to a newer framework and at that point the rewrite took the team of 5 programmers months to complete, without any new features being deliver to the customers”
- “Very classic one that most people probably saw happening real time. The initial gameplay trailer for Cyberpunk77 was entirely scrapped and only made to function for showcase under very specific conditions. It was never prepared for the final game. Big shame, besides the car chases that people got very caught up on, the narrative seemed a lot more promising than it turned out to be in the final version.”
- “I worked in an internship project where the project was reinvented 4 or more times and taken back to square one with each new feature the management had dreamt the previous night. Nothing was documented or written down as decisions. Each worker basically had their own view on what was the goal.”

- “A project that I am working on right now just planned time for addressing technical debt because our live game started having performance issues which are directly affecting our business.”
- “Well, the story would be that most of the games use 30-year-old outdated solutions, with no means of improvement leading to lack of innovation in games.”