NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4240 Software Architecture

Group 4 - Requirements

[1]

Jens Martin Norheim Berget
Magnus Vesterøy Bryne
Sverre Nystad
Mattias Tofte
Tim Matras
Tobias Fremming

Chosen COTS: Android SDK, LibGDX and Firebase
Primary quality attribute: Modifiability
Secondary quality attributes: Usability, Performance and Availability

March 03, 2024

---

[1] OpenAI. (2023). *DALL-E* (3. mars versjon) [Diffusjonsmodell]. https://chat.openai.com/

Table of Content

1 Introduction

Our main goal in this project is to learn how to design and use software architecture in a real project whilst also creating a game that is fun to play and develop. We will create the architecture from scratch using the theory we have learned in the course, and implement it using Java.

The requirement phase is a critical phase in our project's lifetime as it defines the requirements that will heavily influence the architecture of the game. These requirements will also work as a blueprint for the game, and will work as a guide to what needs to be done at every step in the development process. These requirements are vital as "[t]he longer the defect stays in the software (...) the more damage it causes" and "tends to have broader effects"[2] and given that the requirements phase is first stage defects has the largest damage potential and also the cheapest to fix.

In this requirement phase we will:

- Define functional requirements
- Define quality requirements
- Highlight COTS components and other technical constraints

We expect the outcome of this phase to be a full list of requirements that we can use for designing a good architecture for our game.

[2] McConnell, 2004, p. 29

1.1 Game description

The name of our game is *Besieged!*. *Besieged!* is a real-time cooperative multiplayer tower defense game based on viking- and norse mythology. Players join shared instances where they have to cooperate in real-time as enemies invade. Together with your friend, you will need to buy and place cards to build towers and defend against waves of foul creatures from norse mythology to save your village. In order to build a tower, a player has to place a “tower-card” or an “element-card” on an empty grid/building-site-block on the map. The other player then has to place another card on the same grid to create a tower. The two cards will combine to create a unique tower with unique characteristics. The tower will then attack any enemies that come within its range. When an enemy is killed each player is rewarded with some money that they can use to buy more cards from the store. Enemies will spawn in waves at a starting position and follow a predetermined path to your village. If they reach your village it will take damage. If the village loses all its health it will burn down and you will lose the game.

Besieged! builds upon ideas from other tower defense games, such as Bloons Tower Defence 4 (BTD4). BTD4 exemplifies some of the core concepts within the tower defense genre, and a snapshot is presented in Figure 1.1.1. The game uses balloons as enemies and monkeys as towers. *Besieged!* attempts to iterate on the core concept of static towers by creating dynamic combinations using multiple cards.

Figure 1.1.1 Snapshot of Bloons Tower Defense 4

2 Functional requirements

Functional requirements define what a system or application is supposed to do. They specify the behaviors, functions, and capabilities that a system must have in order to fulfill the needs and expectations of the users or stakeholders. These requirements are focused on describing the intended interactions between the system and its users, as well as the system's responses to particular inputs or conditions.

Table 2.1: Entity Roles

Role	Description
Player	Abstraction of user interaction with the game; having health, money and cards.
User	User interaction with the game.
Card	A card you can put on the map. Combine different cards to create a tower
Tower	Different kinds of towers use their ranged weapons to throw projectiles at foes in order to defend from the enemy siege.
Enemy minions	Different kinds of creatures rushing towards your defenses, trying to conquer your land.
Wave	A wave of enemies being spawned to attack in a given quantity and time intervals for each wave. There will be multiple waves in a game.
Construction Site	A given place, a tower can be constructed by placing cards. Because of the different locations, the construction sites will differ in tactical value.

Description of priority levels:

- **Low:** Has little to no impact on the gameplay experience. Is not part of the core logic that makes the game playable. Can be thought of as “nice-to-have”-features.
- **Medium:** Has a noticeable impact on the gameplay experience.
- **High:** The game will not function without these features. Can be thought of as “must have”-features.

Table 2.2: Functional requirements

ID	Requirement	Priority
FR1	The user should be able to interact with the game using touch	High
FR2	A player should be able to create a game lobby and invite friends to join their game	Medium
FR3	If the game lobby is empty the player should not be able to start a multiplayer match.	Medium
FR4	If the player(s) lose all their hitpoint the game over screen should appear.	Medium

FR5	The game should have an in-game menu with the necessary controls to play the game	High
FR6	While the player uses the in-game menu in a singleplayer match, the game should pause for the player.	Medium
FR7	While the player uses the in-game menu in a multiplayer match the game should pause for both players.	Low
FR8	The player should be able to forfeit the game	Low
FR9	The player should be able to adjust the game volume	Low
FR10	The player should be able to place Cards on available tiles and cards.	High
FR11	When a card is placed on another card, the cards combine to create a tower	High
FR12	When a card is placed on a tower, the tower gains an upgrade given by the card.	Medium
FR13	If a card is placed on an uavailable tile, the road or an enemy the card should not be placed and be returned to the player	Medium
FR14	When a card is placed on a legal spot the cost of the card will drain the “wallet” of the player, given the player has enough currency.	Medium
FR15	Enemies shall appear at the entry point during a Wave	Medium
FR16	Enemies shall follow the path towards the Exit Point during a Wave	Medium
FR17	When Enemies arrive at the Exit Point, they shall disappear and the player(s) should lose hitpoints/health	Medium
FR18	When Enemies lose their entire health/hitpoints they should give their bounty to the treasury of the player(s) and disappear.	Medium
FR19	When Enemies are in the range of a Tower it should shoot a projectile at it lowering the health of the Enemy upon hitting it.	Medium
FR20	The player should be able to sell Tower(s) and a portion of the original cost of the tower should be returned to their “wallet”/treasury.	Low
FR21	The active wave will spawn enemies at different times throughout the duration of the wave.	Medium
FR22	Once a tower has found an enemy to shoot, a projectile will be shot towards the enemy in a realistic curve that will be rendered. The enemy will be damaged.	Medium
FR23	After a game has ended, the score should be appended to a public high score.	Medium
FR24	When a player selects a card the placeable areas shall be highlighted.	Medium

FR25	When a player places a card the player needs to confirm the action for the card to be placed.	Medium
FR26	It should be possible to rejoin a game if a player crashes	High
FR27	If a player loses their connection, this player should be thrown out, while the other player can continue playing, and the remaining player receives the other player's resources.	High
FR28	The entity hitpoints should be calibrated so the gameplay is not too easy nor too difficult.	Medium
FR29	There exists a tower for every single combination of cards.	Medium

3 Quality requirements

A quality attribute (QA) is "a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders beyond the basic function of the system"[3]. Put easily: Quality attributes dictate how well the system will work. So while quality attributes are characteristics that determine the overall quality of software, like performance or usability, quality requirements are measurable criteria that the software must meet to achieve these attributes. An example of a quality requirement would be that a new car will be able to drive for at least 10 000 kilometers before needing a service appointment.

In this project we have chosen to focus on the following quality attributes:
"**Modifiability** is about change (...) and the cost and risk of making changes"[4]. High Modifiability means that the system is easy to modify at any time in the development process. This is a major time saver, as "roughly 80 percent of a typical software system's total cost occurs *after* initial development"[5]. This means that the potential cost savings by ensuring high Modifiability are significant over the lifetime of a software system.
"**Usability** is concerned with how easy it is for the user to accomplish a desired task and the kind of user support the system provides"[6]. High Usability means that the system is easy to learn and easy to use, i.e. has an intuitive user interface (UI) and easy-to-understand controls.

[3] Kazman, Bass & Clements, 2022, p. 39
[4] Kazman, Bass & Clements, 2022, p. 117
[5] Kazman, Bass & Clements, 2022, p. 27
[6] Kazman, Bass & Clements, 2022, p. 197

"**Performance** is about time and the software system's ability to meet timing requirements"[7], whether it be in terms of delay/latency, jitter or FPS (Frames Per Second). In gaming, performance is widely regarded as the key factor influencing the in-game experience, since all gamers understand that a laggy game detracts significantly from enjoyment.

"**Availability** refers to a property of software that it is there and ready to carry out its task when you need it to be" (Kazman, Bass & Clements, 2022, p. 51). If you don't have a game with high Availability, users can simply not play the game, which is obviously not ideal.

Table 3.0: Quality requirements

ID	Attribute	Requirement
U1	Usability	Inexperienced players should be able to view a tutorial for how to play the game.
U2	Usability	The player should be able to sell towers(undo)
U3	Usability	The players should be able to pause/resume the game, given it is single player mode.
U4	Usability	Inexperienced players should be able to easily start a new game within 60 seconds.
U5	Usability	The gameplay should be fun and engaging.
U6	Usability	No towers should feel overpowered or underwhelming, so that they all can be considered usable by players.
P1	Performance	End-to-end delay should be less than 100 ms in singleplayer given the system being in normal operation.
P2	Performance	End-to-end delay should be less than 100 ms in multiplayer given the system being in normal operation and internet connection being stable.
P3	Performance	The game should be able to run at a frame rate that ensures nice movements of entities on the GUI, and smooth frame transitions.
A1	Availability	The database should be available 99% of the time.
A2	Availability	The game should be available to players long after we are done with this subject.
A3	Availability	In case of unplanned outages, automatic recovery procedures should be in place to restore services in 15 minutes.
M1	Modifiability	It should be easy for a developer to add new enemy entities, with little to none refactoring of already existing code.

[7] Kazman, Bass & Clements, 2022, p. 133

M2	Modifiability	It should be easy for a developer to add new cards, with little to none refactoring of already existing code.
M3	Modifiability	It should be easy for a developer to add new maps, with little to none refactoring of already existing code.
M4	Modifiability	It should be easy for a developer to add new towers, with little to none refactoring of already existing code.
M5	Modifiability	The development team should easily be able to change the database if the need arises.
M6	Modifiability	The system should not be too dependent on libGDX, and in case of failure of libGDX, we should be able to switch to other COTS that can provide similar services.
M7	Modifiability	A developer should easily be able to add new states to the game in order to expand it.

Below are some quality attribute scenarios for each of the four quality attributes mentioned above which further specify our quality requirements.

3.1 Modifiability

Table 3.1.1: Add a new Enemy type

ID	M1
Source	Developer
Stimulus	Wishes to add a new Enemy to the system
Artifacts	Code
Environment	Design Time
Response	Changes to the code base and test suit, including internal alpha testing
Response Measure	With no more than 3 hours of work and without introducing any side effects.

Table 3.1.2: Add a new card

ID	M2
Source	Developer
Stimulus	Wishes to add a new card to the system
Artifacts	Code

Environment	Design Time
Response	Changes to the code base and test suit, including internal alpha testing
Response Measure	With no more than 3 hours of work and without introducing any side effects.

Table 3.1.3: Add a new Map

ID	M3
Source	Developer
Stimulus	Wishes to add a new Map to the system
Artifacts	Code
Environment	Design Time
Response	Changes to the code base and test suit, including internal alpha testing
Response Measure	With no more than 3 hours of work and without introducing any side effects.

Table 3.1.3: Change the database.

ID	M5
Source	Developer
Stimulus	The need to change the database arises.
Artifacts	Code
Environment	Design Time
Response	Creates a new database and implements a new data access object.
Response Measure	With no more than a work day (8 hours) of work and without introducing any side effects.

3.2 Usability

Table 3.2.1: New player plays the tutorial

ID	U1

Source	End user
Stimulus	Starts the tutorial
Artifacts	System
Environment	Runtime
Response	User is introduced to controls and mechanics within the game through a short and simple textual tutorial
Response Measure	Within 3 minutes

Table 3.2.2: Add undo for placed towers

ID	U2
Source	End user
Stimulus	Player presses the sell button on tower
Artifacts	System
Environment	Runtime
Response	The tower should be sold and the player should be reimbursed for a portion of the original tower cost.
Response Measure	2 seconds

Table 3.2.3: Pause/resume game

ID	U3
Source	End user
Stimulus	Player opens the in-game menu
Artifacts	System
Environment	Runtime
Response	The game should pause
Response Measure	2 seconds

3.3 Performance

Table 3.3.1: Singleplayer performance

ID	P1
Source	System/game
Stimulus	Player playing the game
Artifacts	System
Environment	Runtime
Response	The game should not feel laggy
Response Measure	End-to-end delay should be less than 40 ms given the system being in normal operation.

Table 3.3.2: Multiplayer performance

ID	P2
Source	System/game
Stimulus	Players playing the game together
Artifacts	System
Environment	Runtime
Response	There should be low latency between players
Response Measure	End-to-end delay should be less than 40 ms given the system being in normal operation and internet connection being stable.

3.4 Availability

Table 3.4.1: Database availability

ID	A1
Source	End user
Stimulus	Plays the game
Artifacts	System
Environment	Runtime
Response	Database is available
Response	99% uptime

Measure	

4 COTS components and technical constraints

Commercially off-the-shelf (COTS) software products are invaluable to many applications, these components are any installable software product or service either paid for or licensed under open source. Using COTS components is an alternative to building software from scratch, possibly greatly decreasing development time, however there exists tradeoffs with using COTS as the application becomes dependent on external software not under direct control. That can change or cease to exist. Therefore it is important that the architecture stays “[i]independent of frameworks" and “[i]independent of database”[8] and limit the coupling from these COTS. This section will document all COTS components that *Besieged!* will use.

4.1 Android SDK

Besieged! target platform is Android, and is under development, utilizing the Android SDK, a software development kit for the Android Software ecosystem. This includes development tools and libraries tailored for creating applications on the Android operating system. Since we are using the Android SDK to develop our game for Android devices, we have limited hardware resources available in terms of computing, as our application will be run on a phone or tablet. This significantly impacts our architecture, as we must make the game be able to run smoothly even with limited computing resources. Using an emulator for android however, could in our case be a great asset to determine if the demand is greater than the processing power limited by the necessary hardware. This will ensure the app’s performance by testing throughout the development.

4.2 LibGDX

LibGDX is a powerful cross-platform game development framework for Java game development. It comes with out-of-the-box components like an audio system, a graphics system supporting both 2D and 3D games, input handling supporting both gesture detection and powerful abstractions for touchscreen, math and physics systems, and more. All of the listed systems will be very useful for *Besieged!*, as we will not need to implement our own systems from scratch, which would have been very time-consuming. It is also important to note that the LibGDX COTS is under the Apache 2.0

[8] Martin, 2018, p. 202

license so if one of the systems does not exactly tailor our specific needs, we could fork it and modify the system to our needs, but this would make the project incompatible for receiving support and future improvements to the LibGDX framework and would also require a lot of work. Even though it gives us a lot of functionality, it also requires tight coupling between our application and the LibGDX framework. To mitigate some of this coupling, we should create architectural boundaries to the systems by using interfaces to the audio, graphics, and input systems. LibGDX also constrains the application in terms of what programming languages we can use in our tech stack as well as the versions used. Only languages that run on the JVM (Java Virtual Machine) like Scala, Kotlin, Java, Clojur and Jython. Although LibGDX can be used together with all of these languages, not all platforms support them. Out of all these languages, LibGDX supports Java and Kotlin the most, but it does not yet support Java 21 or newer versions.

4.3 Firebase

The application is going to need a way to share data across devices both for the online and the multiplayer aspect. The data needs a way to be saved persistently to make it available for all users. Firebase can support this by leveraging Firebase's real time database, this technology makes it possible to share data across many devices all at once at incredibly fast speeds. This service is free for our intended use as Firebase has a free tier that allows its users to use its functionality without any monetary cost. Firebase gives a lot of functionality but it also comes with some restraints. Firebase requires the users of the application to have an internet connection to be able to use its persistence and multiplayer functionality, this makes it so that *Besieged!* online and multiplayer aspects are directly tied to Firebase's uptime, and as consequence Firebase's availability will affect the games availability. Firebase's Service Level Agreement says that the uptime will be "at least 99.95%"[9].

4.4 Interfaces

"An interface is a boundary across which elements meet and interact, communicate and coordinate."[10] Interfaces play a crucial role in the context of *Besieged!* by abstracting the complexity of interactions between our game and the COTS components such as Android SDK, LibGDX, and Firebase. The integration of these interfaces is imperative for ensuring that the game's architecture not only maintains modularity but also exhibits flexibility in response to alterations in external components or the game's evolving requirements. By establishing well-defined interfaces, the functionality offered by the COTS components can be effectively abstracted. This abstraction fosters a degree of autonomy essential for robust software architecture.

---

[9]Firebase, 2020
[10]Kazman, Bass & Clements, 2022, p. 217

5 Changes

Table 6.1: Changelog

Date	Change History	Comments
March 03, 2024	First released version	None
April 20, 2024	Removed “time” from LibGDX chapter	We did not use LibGDX’s time/clock in our game directly.
April 20, 2024	Changes to functional and quality requirements in light of feedback	Some quality requirements were written like functional requirements.
April 20, 2024	Add new attribute scenario for M5: change the database	This was added because we felt it was an important quality requirement.
April 20, 2024	Remove all references to element card or tower card, and use the word card instead.	We decided to have a tower for every combination of cards, and therefore the need to segregate cards into different types vanished, as this was to prohibit the need of drawing too many different towers.
April 20, 2024	change functional requirement FR10 from: “The player should be able to place Cards on available tiles, Towers or any other Card” to “The player should be able to place Cards on available tiles and cards”	We changed this, because this functional requirement overlapped with FR12. These two functional requirements have different priorities and therefore we don’t want them to overlap.
April 20, 2024	Add new functional requirement FR29: There exists a tower for every single combination of cards.	User tests showed us that it was not intuitive what cards can be placed together. We fixed this by adding a functional requirement stating that every card should be able to be combined, resulting in a unique tower.
April 20, 2024	Change the phrase “construction site” with “available tile”	This is to better reflect the fact that a card can be placed at available tiles, and not specific tiles called construction sites.
April 20, 2024	Change priority on FR23 from	During implementation of the

	High to Medium	game, we found that this wasn't essential for the game, especially since there is another multiplayer aspect to it. Therefore we changed the priority.
April 21, 2024	Reformulated game description	The game description needed to be clearer about how multiplayer works in our game
April 21, 2024	Readjusted line spacing	Line spacing was 1,15 in some parts, switched to 1,5 for improved readability and document concistency

6 References

Rick Kazman, Len Bass, Paul Clements. *Software Architecture in Practice* - Fourth Edition. Addison-Wesley, 2020.

Firebase, (2020, April 9) *Service Level Agreement for Hosting and Realtime Database*. Firebase. https://firebase.google.com/terms/service-level-agreement

Robert C. Martin. (2019). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Person

Steve McConnell, (2004). Code Complete: A Practical Handbook of Software Construction (2. Edition). Microsoft Press, USA.

7 Individual Contribution

Table 8.1: Individual contribution

Name	**Nature of work**	**Hours**
Jens Martin Norheim Berget	• List of tokens • Token interaction table • Functional and quality requirements • Quality attribute definition and introduction • Description of priority levels for FR's	9 hours

Magnus Vesterøy Bryne	● Functional requirements and quality requirements ● Came up with project name and theme ● Read through a couple of times ● Made changes in light of feedback ● Fixed line spacing	9 hours
Mattias Tofte	● Functional requirements ● Interfaces ● Temporary logo	6 hours
Sverre Nystad	● Set up the projects GitHub (6 hours) ● Writing various places in requirements document (12 hours) ○ Functional requirements ○ COTS writing Android SDK, LibGDX, Firebase ○ Quality scenarios ○ References	18 hours
Tim Matras	● Writing on requirements document ○ Introduction (1h) ● Reading through the document and noting where we need to make changes after feedback (0.5h)	1.5h hours
Tobias Fremming	● Define tokens (1 hours) ● Create token interaction table (2 hours) ● Proof of concept/ sketching cards and towers(2 hours) ● Define some functional requirements (1 hours) ● Wiriting on requirement document (3 hours) ● Trying to figure out an art style for design(1 hours) ● Define some quality attributes, and create a quality attribute table (2 hours)	12 hours