## horizontal line



Online Multiplayer Strategy Game

Sarmed Alwan

sa16566

Supervisor: Professor Anthony Vickers

Second Assessor: Carmine Ventre

University of Essex

BSc Computer Science

School of Computer Science and Electronic Engineering

# Acknowledgements

## In-Product Material

* WW2 footage GIF used for background of Main Menu (<http://www.publicdomaintorrents.info/>)
* In-game music commissioned from Richard Ampleford
* Instinct, by BenSound, used for Main Menu music (<https://www.bensound.com/royalty-free-music/track/instinct>)
* Sound effects from <https://freesound.org/>

## Other Acknowledgements

* Supervisor: Professor Anthony Vickers (<https://www.essex.ac.uk/people/vicke53909/anthony-vickers>)
* Second Assessor: Dr Carmine Ventre (<https://www.essex.ac.uk/people/ventr89804/carmine-ventre>)

# 

# Abstract

In Operation Mars, you are plunged into one of the most explosive engagements of WW2, as either the Soviets or the Germans. You can play with a friend either over the internet or on the same computer. You’ll both learn the game within a minute or two, and you’ll soon be trying to outplay each other; using encirclements, feigned retreats, slipping through their lines with tanks, matching the correct unit types, stacking veterancy, and luring them into hasty attacks. The game can be played within 15 minutes, or even less between experienced players, and there is a turn limit of 100, so no game will ever drag on. After the turn limit is reached, the game will measure who’s won, based on which key towns you’ve secured, and how many kills you’ve attained. The German player will have to utilise their veteran units and defensive positions to hold their thin line for as long as possible. The Soviet player will have almost twice as many troops, but is also in the position of having to attack quickly, since they start with much fewer key towns. Operation Mars offers the satisfaction of grand-scale strategy games within a sleek package which is easy to pick up and play, but challenging to master.

# Table of Contents

[**Acknowledgements**](#_ohfii7byfkm) **1**

[In-Product Material](#_f6y4mmv1jrfm) 1

[Other Acknowledgements](#_62b59vbm30ck) 1

[**Abstract**](#_9xlzmljgnvau) **2**

[**Table of Contents**](#_dia22giti9hv) **3**

[**Literature Survey**](#_xaxlqkd9vehx) **4**

[**Technical Documentation**](#_1ke8mctnp5q7) **21**

[Technical Review](#_2ukob7462lxw) 21

[Structure of the Program](#_n0mw8q2bpfvp) 21

[Purpose and Functionality of the Program Classes](#_q55kq57no3r) 22

[Main](#_i1c50ipbs14r) 22

[Unit](#_rqz38kn07b14) 22

[MenuFrame](#_rkojhxr7i83f) 24

[MenuElements](#_k48x1djh945s) 25

[MainFrame](#_uqxxyg3z9wmw) 26

[MainPanel](#_kf69rvgz2qt2) 27

[UnitPanel](#_klgngawqhcq2) 28

[Listener](#_iqqf44ia07v) 33

[Background](#_klgngawqhcq2) 37

[GridPanel](#_klgngawqhcq2) 38

[TerritoriesPanel](#_klgngawqhcq2) 38

[GameState](#_yxo67tmpgpi8) 39

[Selection](#_kv46tq9j7ape) 44

[ClientThread](#_hij4l25n69xk) 44

[Server](#_qfxmlyivsw5z) 45

[ServerThread](#_v67rbeyquwmg) 45

[Bugs and Testing](#_jfav38qoqiq4) 46

[User Manual](#_x01qxwox5rz) 52

[System Requirements](#_p7h077qblk8i) 52

[Minimum Specifications](#_qi1uzkbck21a) 52

[Recommended Specifications](#_fwmyonb1q48q) 52

[Getting Started](#_ldhabwrfmqvw) 52

[Getting Online](#_sduaps1sijch) 52

[How to Play](#_tn93hue7rfba) 53

[**Project Planning**](#_9afhf1ta6taq) **55**

[Overview](#_75ee9h44dfnu) 55

[Maintaining Momentum, Adapting to Change, and Dealing with Risks](#_k64fomhvuvtc) 56

[Achievements and Performance](#_h3wjwka0e5uv) 57

[Reflection](#_eu5qa8se34zf) 58

[**Conclusion**](#_fnix8thfoquw) **59**

[**Bibliography**](#_pl0bndi5fxlt) **60**

# Literature Survey

The games industry has for the last 7 years been undergoing a significant shift in the makeup of its customer base and its locations; between 2012 and 2018, the mobile market has grown from an 18% to a 51% share of the gaming market. However, while the home console market dwindled from 45% to 25% during this time, the PC gaming market lost only 13% of the market share [1]. This alone does show a downturn in the PC gaming market, but there is a much more complex story here to be told. The 5 most played games on mobile right now are Pokémon Go [2], Despicable Me: Minion Rush [3], Candy Crush Saga [4], Fruit Ninja [5], and Jetpack Joyride [6]. There is a clear theme here, and that is that casual games now dominate the market on mobile [7].

There exists in the strategy games industry, and in the games industry as a whole, a dichotomy wherein the ‘casual’ games and gamers are separated from the ‘core’ games and gamers. The casual gamer vs. core gamer is a long-held idea in the industry [8], and dictates much of a developer’s initial ideas about what their game ought to be. Will it be more like Goldeneye, or more like Candy Crush? The line between the two, while relatively easy to see heuristically, is harder to strictly define and to describe. Core vs. casual, however, can be summarised broadly by the core gamer typically spending more time and money on the game, as Ernest Adams of Gamasutra describes [8], but most importantly in their personal investment in thinking deeply about the game when they are playing it and when they aren’t.

The PC gaming market, therefore, has shrunk in its share of the market only if you look at the situation from one perspective, and that is to assume that all games are the same product, i.e. that the casual and core games markets are one. While this is the case to an extent, it is clear from the trends in use cases that this is a simplistic depiction of a deeper situation. The PC gaming market has not by any means shrunk, even despite the slow decline of desktop PC purchases, and PC purchases as a whole [9]. In fact, the core PC gaming market has been booming; from November 2012 to October 2018, the user base on Steam, the largest core gaming digital distribution platform for games on PC [10], has tripled from 6 million to 18.5 million [11], flying in the face of the decline of PCs powerful enough to play most games. It’s a completely different market from that on mobile, too; the top 5 most played games on Steam are currently very much ‘core’ games: DOTA 2, PlayerUnknown’s Battlegrounds, Counter Strike: Global Offensive, Tom Clancy’s Rainbow Six Siege, and Grand Theft Auto V [12]. This means that largely, the catering on PC is typically much more core-focused than the mobile market.

However, there is a hole in the market that isn’t all that often filled; strategy games on PC are typically long-term sit-down-and-play affairs such as Hearts of Iron IV [13], or tense and with a steep learning curve such as StarCraft II [14]. One of the last times that a relatively casual strategy game was introduced to the mainstream PC market was Sid Meier’s Civilization V, and it was incredibly popular. As T.J. Hafer of PC Gamer points out, a survey on Twitter showed us the habits of the dedicated player-base versus the reported numbers on Steam. This survey reported that Civilization VI was the main game of 57% of fans, with Civilization V at 40% [15]. The voluntary nature of this survey demonstrated the opinions of the dedicated fans who are active in the community and answering such surveys. The message here which Hafer took away was that the player-base of Civilization is quite unique, in that “A lot of those still playing older Civs might be someone more like your friend’s dad who doesn’t own anything else on Steam and makes a go of casually conquering the world over morning coffee.” [15] It is evident, then, that there is a quiet yet large niche on the PC games market, which doesn’t get served very often, despite it being large enough to flip the trend when looking at reported surveyed popularity from fans versus actual Steam player numbers. These Steam player numbers are being greatly increased by this large yet quiet and casual section of the market, which is quite aptly described by Hafer.

Seeing that there is a large portion of the market going un-tailored-to for years at a time, it is clear that there is not only space in the market for such products, but a large and important demand for such products. If Civilization V was able nine years ago to take that crown from previous title holders such as SimCity 2000 [16] and Civilization III [17], then there is no reason why new games cannot fill the surprisingly large middle ground between casual, easy to pick up, yet engaging, and just nerdy enough to draw this very specialised audience. As Shearer and Meer suggest [16] [17], these are key ingredients to this formula - that you can “learn most everything you need to succeed simply by paying attention to what's going on,” and that the games make “not even the slightest attempt to be, as my irksome 2001 self might put it, cool.” The latter point is a common theme in these rare games which captivate this elusive sector of the market, with straightforward management or historical themes typically being those of choice, rather than the more modern and cool route which Civilization VI or popular mobile games usually take to appeal to an over-saturated mass market.

A small further experiment was made to establish the extent to which multiplayer is influential on people’s desire to play such a game. Three people who have played all kinds of PC strategy games for a long time on a regular basis [18] [19] [20] were asked two questions:

“Are you typically drawn to strategy games by the prospect of online multiplayer?”

The answers to this question were: Yes, Yes, and Yes

“Would you play more strategy games if it didn’t take so long to do so?”

The answers to this question were: Yes, Maybe, and Yes

This small survey suggests that there was some feeling among strategy game fans that multiplayer is also a significant draw for them. It is also evident from looking at strategy games such as StarCraft II [14] and Age of Empires II [21], which have been kept alive for truly spectacular lifespans by their multiplayer communities; this year, Age of Empires II will turn 20 years old; positively ancient in terms of single-player games. Despite this, both are more alive than ever, with the multiplayer and modding communities buzzing [21], and even with tournaments and live games constantly being broadcast and watched by thousands on online channels such as T90Official [22], which itself has had almost 3 million individual viewers.

The following things can be inferred from all of this information: that the PC games industry is definitely still hungry for casual, easier to pick-up, and slower-paced old-fashioned strategy games such as Civilization V, and that those games which thrive in the industry for the longest time and in the largest volumes tend to be kept alive by multiplayer.

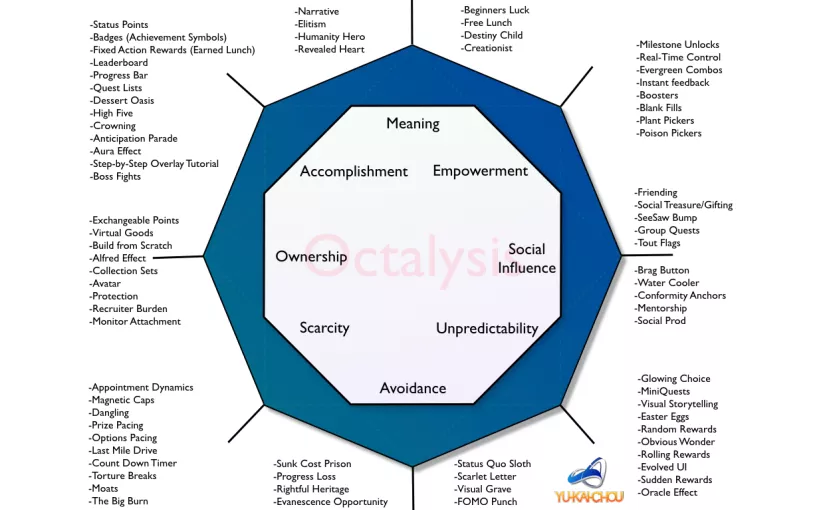
There is another aspect which typically drives the addictive nature of prominent strategy games, and that is uncertainty. Whether that is in the case of something like *StarCraft II* [14], where the uncertainty is largely in the unpredictable behaviour of the opposing player, or whether it’s a case like popular Grand Strategy Game *Europa Universalis IV* [23] where large portions of the game are slightly randomised, which makes the game extremely replayable, even when the player chooses the same countries to play as repeatedly.

As prominent educational YouTube channel Extra Credits puts it, “the only reason to strategise once you’ve got a strategy that’s working is because the environment in which your strategy is working might change on you… that’s what we play strategy games for.” [24] He also elaborates that “Uncertainty comes in lots of forms, and randomness can be one of them; more often, uncertainty comes in the form of hidden information; unexpected events; or even simply playing against a player who might make some unpredictable moves that will force you to reconsider your strategy.” He goes on to use Civilization as an example, wherein a large amount of the uncertainty comes from random number generation by the game in various forms, and yet more uncertainty can be added to the game when one encounters another civilisation - especially if that civilisation is controlled by another human. This uncertainty, which comes in many forms, is a huge primary draw of strategy games, and the strategy game designer who neglects it entirely may find their game lacking something. As Extra Credits puts it, the game becomes a puzzle rather than a strategy game; the difference lying in that once one knows the solution to a puzzle, they can simply repeat the solution to get the result they want. In the case of a game, this would lead to a less engaging experience, even if the player is looking for a casual one.

The unpredictability or uncertainty of a game - a major drive in strategy games - can be hurt by the “natural human tendency to get better at things” [45] For example, in Noughts and Crosses, if your opponent has a very rigid prescribed strategy which they have developed, the game can become boring. This issue doesn’t stem from the fact that the player will know they have poor chances of winning, but mostly because the game becomes predictable, and thus repetitive, and thus boring. This can be “rectified to a degree via good game design.” [45], through things that not even a player who has developed a prescribed strategy can entirely predict. These will typically come in the form of randomisations, either of maps, item drops, combat results, or a variety of things to keep the game unpredictable in the face of the development of formulaic strategies. However, there is a fine line to tread; if the randomisation is over-done, the dedicated players developing such strategies will, when the randomisations of the game don’t go their way, be put off by a justified feeling that they’re being unfairly punished despite using what they know to be a good strategy. This would have the opposite effect from that which is desired, so a sensible game developer will have to carefully consider the particular extent to which their game is made uncertain.

Being a relatively niche product does not necessarily mean that the game would be left in the dust or wouldn’t find its audience. As Koster explains, “the folks with bodily-kinesthetic intelligence will gravitate towards sports, whereas the linguistic folks may end up with crossword puzzles or Scrabble.” [42] He goes on to explain that this “means that not only will a given game be unlikely to appeal to everyone, but that it is probably impossible for it to do so.” [43] Various types of people have different types of intelligence and personalities, and from these, to an extent, stem their preferences. The intelligence types which Koster presents: Linguistic, Logical-mathematical, Bodily-kinesthetic, Spatial, Musical, Interpersonal, and Intrapersonal, are a reflection of a lot of research in the field which comes to similar conclusions - that a game ought to consider which type of person it will appeal to, rather than simply trying to make itself ‘appealing’ in general. Some such research is the well-known Octalysis model formulated by Yu-Kai Chou [44].

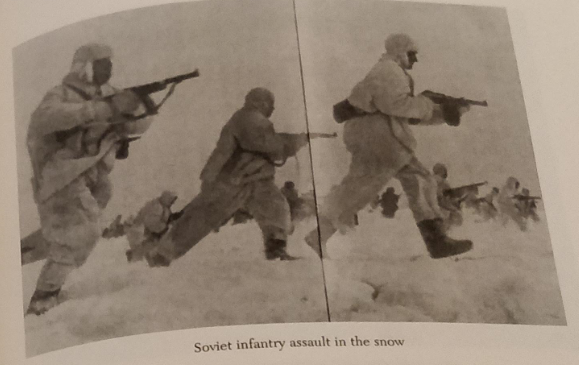
The Octalysis model, rather than presenting personality types, presents things which players may seek from a game. It is a high-level gamification framework; gamification being the focusing on the human experience of a product, rather than simply the functional aspects of it. In the Octalysis model, there are eight axes by which a product can fulfill some aspect of what a human might desire or gain from it. These are: Meaning, Empowerment, Social Influence, Unpredictability, Avoidance, Scarcity, Ownership, and Accomplishment. A product which takes these factors into account “remembers that people in a system have feelings, insecurities, and reasons why they want or do not want to do certain things, and therefore optimizes for their feelings, motivations, and engagement.” This is a big ingredient in the recipe for success. Chou elaborates, demonstrating various examples of successful products which hit these aspects in certain ways which make them engaging to the user. For example, the successful game Diablo III has a fairly balanced Octalysis chart, providing the user with a lot of Avoidance through things such as limited item durability, Unpredictability through things such as randomised item drops and maps, Scarcity through the limited occurrence of good item drops, and Accomplishment through levelling up and obtaining new skills and gear [44]. Facebook has more concentrated drives, centreing around Social Pressure and Ownership. A few well-executed and focused drives to a product - 2-4 of them - can therefore produce just as good results as hitting many drives with a more spread-out result.

 Fig1 *Y. Chou’s Octalysis Diagram as included on his site; an outline of potential aspects of each draw [44]*

It is this thinking which leads to the ideas put behind the creation of Operation Mars, which can enter this relatively vacant yet called-for space in the market, where such games are needed; easy to pick up, with some deeper layers, which can be played for 15 minutes or several hours, and with a good bit of uncertainty, social influence, scarcity, and accomplishment. The scope of such a game would need more content than could be produced by one person in six months, to truly hook players over months and years - Civilization V’s first large expansion pack, Gods and Kings, “enhances the base game immensely, so much so that [one] can't imagine playing Civilization V without it.” [25] Yet, a fun game which fills a much-thirsted-after niche, as the original Civilization V did, could be formed and delivered within the timeframe, and could have its lifespan and word-of-mouth greatly increased by the presence - or necessity - of multiplayer. It could also be made very modular, so that such additions of content could be made easily so long as there is a solid foundation.

The creation of such a product takes some further thought and research; coming up with or designing a product is one thing, but creating it takes far more thought. In the preliminary planning stages of the Initial Report, it is defined that the game should be turn-based, tile-based, and should run in multiplayer. It is also decided there that the setting should be World War 2, and Operation Mars was decided upon as an appropriate focus for the game. It has a lot of asymmetry, meaning that the player could get a fresh new take on the game by switching faction, as they both had very different strengths and weaknesses, and it was historically essentially a stalemate, i.e. neither player won, meaning that the game could be designed and balanced such that either player has an equal chance of winning. A large part of this balance is fairness; an unfair game would definitely put off players. The other main component is that different presented strategies are viable to the player, and aren’t just there to fill out the content and be discarded in terms of actual use [26]. A fair game, as defined by David Sirlin, lead designer of Street Fighter HD Remix, Puzzle Fighter HD Remix, and Kongai, is one wherein “players of equal skill have a roughly equal chance at winning even though they might start the game with different sets of options/moves/characters /resources/etc.” [26] This means that a game can be asymmetrical without being unfair. A game based on Operation Mars would have to be balanced so that despite the Soviet player having almost twice as many troops, the German player would have an approximately equal chance at winning the game.

Real life provides some reasons why this happened; after all, if there weren’t any reasons, then the Germans would have been overwhelmed by the Soviet power. Famously, the weather in Russia caused various problems for the German invaders. However, to an extent, there is some misconception around this, since throughout the war as a whole, it caused just as many problems to the Russians, who were as human as the Germans, and just as vulnerable to the complications of mud and snow. These effects were usually inflicted upon the attacking side at any given time, since it made advancing and moving heavy equipment very slow and difficult, among other potential problems such as hypothermia. Therefore, if the Soviets were on the offensive, as they were in the case of Operation Mars, then these issues would come into effect.

 Fig2

*Winter brought the threat of hypothermia and the difficulty of moving out in the field*

[27] pp.171

 Fig3

*Heavy equipment and even supply vehicles became a hindrance rather than a boon for any attacker*

[28] pp.171

The effect of the weather factor on the Soviet offensive is elaborated upon by Glantz; “The original Stavka directive for implementing Operation Mars on 12 October reached Konev’s Western Front Headquarters on 1 October 1942, but subsequent bad weather made its implementation impossible.” [29] This real-life analogue gave an opportunity to balance the game to match real life to an extent, by always giving the attacker, whether that be the Soviets or the Germans, a small penalty in combat. Despite being applied to both sides, this could be applied to give the Germans a bit more of a balanced chance, since they could be put in a clearly defensive position from the start of the scenario, and their goal could be simply to defend and survive, with the attacker’s greater numbers being nullified to a small extent. However, this could begin to discourage counter-attacking and manoeuvring by the players, which was a prominent part of the battle and which makes for a more fun and dynamic experience, so it was necessary to keep in mind that this effect should be kept limited, and the player shouldn’t be railroaded or forced into one particular strategy - keeping in mind Sirlin’s sound game design advice [26].

There are further ways in which the defender could be compensated for their lack of troops, and again one can look to real life to see how this occurred. As is apparent from available maps of the battle, the front line was held via a network of key towns, such as Rzhev and Belyi. This wasn’t simply because the Germans wanted to capture towns; these ones were of no short-term gain to them, and their populations had largely left. It was because towns have always naturally formed formidable defensive positions, especially in the 20th Century and onward, where “just as now, urban warfare was brutal and unforgiving.” [30]

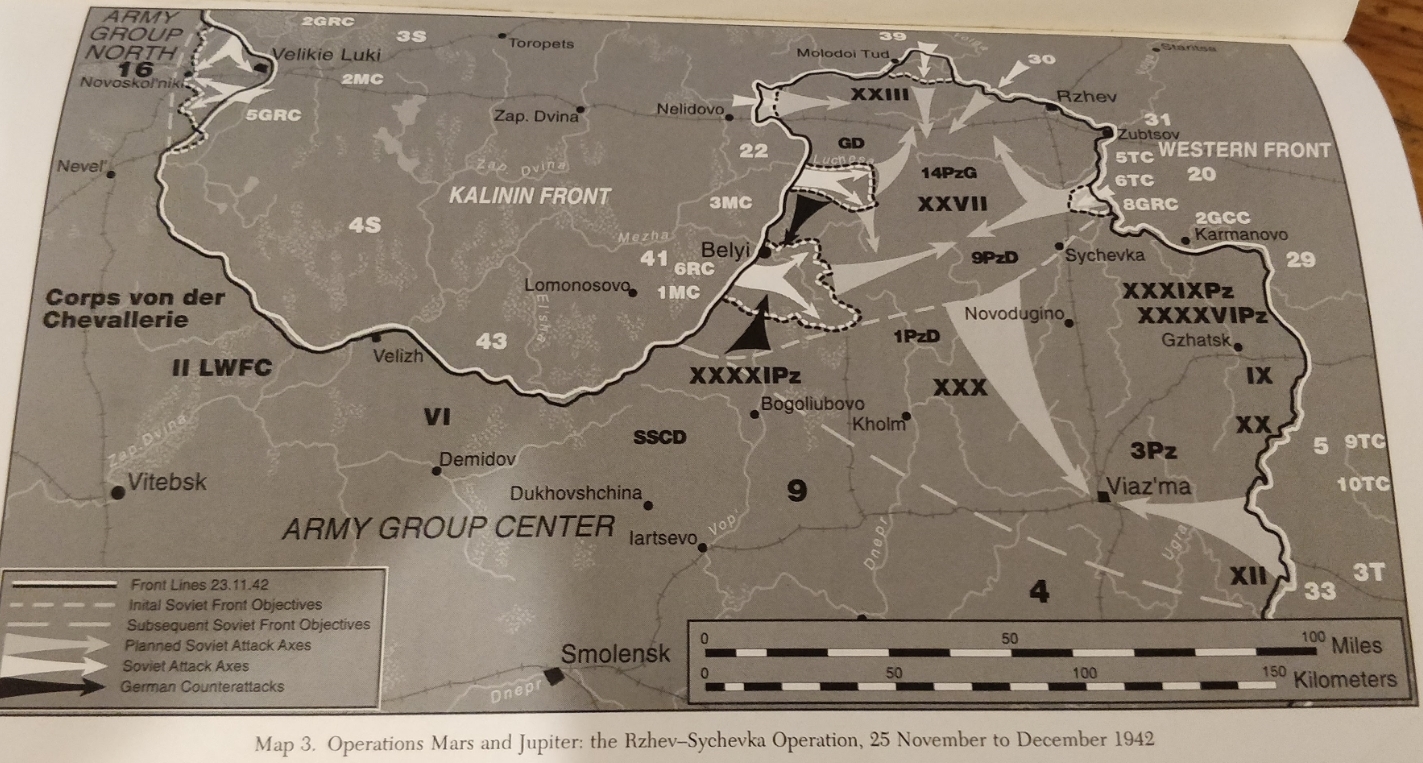


Fig4 *The front line of Operation Mars was lined with key towns and villages* [31] pp.23

Therefore, it is a clear option that to further increase the viability of the German player’s position without adding too much of a deterrent for movement and attacking, another global bonus could be added for defending in towns. This could be made significantly larger than the general penalty for attacking, as it would apply only to key points, and capturing these key points would hand that defensive bonus over to the one who paid the high price to capture it, providing an interesting balance of cost and reward.

Then comes the question of how all of the troops involved in the battle might be portrayed, how they would behave in a technical sense, and how they would affect the gameplay experience. If one looks to the rest of the games industry, there is often a balance which is struck between realism and abstraction; between making a convincing scenario and between making one that is fun to play. One major example of an interesting abstraction of the roles of historical military units in a video game is the Total War franchise. Since its first iteration, Total War: Shogun [32], the popular series has utilised a system wherein each type of unit has a significant bonus versus another type of unit, which forces the players to constantly think about where their units are being positioned relative to the opponent’s [33]. In this case, typically, spear-armed units are good against cavalry, cavalry is good against sword-armed units, and sword-armed units are good against spear-armed units. Although this may not always reflect real life, it allowed the developers to create a nuanced system which is still relatively faithful to the source material. In real life, it is known that spearmen were always a challenge for cavalry [34], and thus the developers used this to create a fun game mechanic; it is relatively “common knowledge that spears and pikes were the bane of cavalry.” [35] An added layer of uncertainty is added to the game; the player constantly has to re-strategise, as Extra Credits puts it [24], and this adds a huge dimension of engagement and thus fun for the player.

Another prominent example of a rock-paper-scissors-like approach in a strategy game is that employed in Company of Heroes and Company of Heroes 2, which is set in the Second World War and thus provides an even better analogue for Operation Mars to take a look at. In this game, the player is expected to utilise certain unit types against certain others; tanks against infantry, infantry against artillery, and artillery typically against armour [36]. This is largely reflective of real life, yet this doesn’t detract from the game, rather adding a dimension of complexity and strategy to it; for example, “anti-tank guns played a crucial role in confronting German equipment on the Eastern Front of the Second World War” [37]. In the case of infantry against armour, “be prepared to meet a grisly death - probably.” [38] It is self-explanatory why an infantry unit could easily dispatch the crew of an anti-tank artillery emplacement, and this is represented well in Company of Heroes. The crews are mostly armed only with pistols and the gun itself isn’t good for dealing with dispersed enemies, and is mostly immobile, so it is easily overrun. The important point here is that this rock-paper-scissors dichotomy is not only an easy and common way to add a lot of strategic depth to a game, but it lends itself to making the theme of the game more convincing. There is good reason that “pretty much every real-time strategy game you can think of is based on rock-paper-scissors, as well as pretty much every Japanese fighting game, as well as many collectible card games.” [39]

It is thus apparent that such a mechanic would not only greatly aid the gameplay of Operation Mars, but was almost an obligation of placing the game in such a theme and setting. The units involved in the battle could all largely be boiled down to one of a few types, and the in-game combat calculation could take into account which unit type was going up against which opposing unit type. In the process of the research of the particular historical units involved in the battle, the historical division, brigade, department, etc. was ‘converted’ into one of the three categories which had been created for Operation Mars: Infantry, Artillery, and Armour. Many of the units researched and used have self-explanatory types, such as the Soviet 243rd Rifle Division [40], which could clearly be classed as infantry, or the German 9th Panzer Division [41], which could easily be classed as an armour/tank unit. There were, however, also going to have to be some major abstractions for the sake of balance in the context of the game. The Soviet 20th Army, for example, was largely formed of infantry, yet in Operation Mars this was overlooked for the sake of in-game balance. In this case, the 20th Army was ‘converted’ to an Armour unit so that the Germans would stand more of a chance on the right side of the board, where many of their units were Artillery units which would be overwhelmed if the Soviet player was given too much infantry in that region.

A common way in which strategy games reinforce the draw of uncertainty which keeps them feeling fresh and engaging is to use something called a fog-of-war system. This refers simply to “an opaque shroud that hides all areas of the map from view.” [54] In the case of Civilization V, which focuses on exploration, there are two types of fog - that in an unexplored area, and that in an unseen area. However, for a game such as Operation Mars, where exploration plays no part in the theme, what can be taken from such a mechanic is rather that not knowing what lies beyond your units’ line of sight can be a major addition of uncertainty and thus strategic depth to the game. It would enable each player to secretly shuffle their units around behind their lines without revealing their hand to the opponent. Such a system is used in StarCraft II, wherein “a player's access to information is limited to the region of map which is currently revealed by his units,” [55] and has been added to the more recent Total War games’ battle system, where “Enemy units are also not visible beyond a maximum range, even if there is line of sight.” [56] This mechanic gives the players a new toolkit to play with, since they can now deceive their opponent in another way, and aren’t forced to always reveal their hand. This is a major part of not only many strategy games, but many real-life card games such as Poker.

In *Civilization V*, the control of terrain by single units - and their manoeuvring around key locations - was made important by the developers, unlike previous instalments of the series where many units could be stacked onto a single tile. This meant that it was important to add the ability to swap adjacent allied units’ positions, so that one could “keep fresh troops engaged with the enemy and rescue your ranged units from contact with melee fighters.” [57] In the case of Operation Mars, there are no ranged units which need to be kept away from melee units, but it could be important to, for example, shuffle a fresh unit in to defend an important town, or swap an Armour unit in to engage the enemy’s Infantry rather than allowing your Artillery to attempt it and face penalties. This mechanic would make the implementation of the rock-paper-scissors mechanic really make sense from a gameplay perspective; if it takes one turn rather than three turns to swap and match your correct unit against theirs, the players will actually regularly attempt to do it, and so the rock-paper-scissors dynamic would actually come into effect and remain a conscious part of the players’ decision-making process, rather than simply an idea which remains too inconvenient to think about using.

The design of Operation Mars would have to take into account that “players are an endless source of new content,” [46] in that placing players head-to-head can reliably produce new experiences, since the game has been developed with limited man-hours, and thus limited content. Schell addresses a similar issue which became a boon for the highly successful baseball game in *Wii Sports*. The designers initially intended for the game to be an accurate and realistic simulation of real-life baseball, with the added bonus of motion controls, yet as development went on, they realised that they lacked the man-hours to implement all of the desired content and mechanics. Subsequently, they decided to focus on making the essential part of that experience - the unique motion-control mechanic - as right as they could. [47] They focused more on the essence of the experience they were trying to create, rather than fulfilling a list of technical aspects that the game should include. Wii Sports has sold over 82 million copies, making it the highest-selling game of all time [48].

An important concept for Operation Mars - and for almost any game - is the Unifying Theme, as canvassed by Schell [49]. The theme is what the game is all about, and knowing what the game’s theme is is a simple but major step towards creating an engaging experience. Schell presents an example in which the theme of a children’s book about what elephants are; the theme is “what are elephants?” The designer should then use every means possible to reinforce this theme. In the case of Operation Mars, the idea of the theme is to create a game which makes you feel like you’re commanding an intense tactical engagement in World War II. The theme needs to be brought together largely by the aesthetics of the game; the unit assets should be drawn and coloured to reinforce the two available factions, the map should be designed to make you feel like you’re in a run-down battlefront of Soviet Russia, and even the main font, Bahnschrift, has been chosen because it is the one which feels the most like the kinds of fonts used in military posters and documents. The theme, however, won’t be truly unified until the music is added in the middle of the game, and the sound effects are added in the late stages of the game.

Other things went into creating the feel of the Unifying Theme, such as the mechanic of encirclement. In WW2 in general and in the real Operation Mars, encircling the opponent’s units was a major objective of both sides; a cut-off unit lacked supplies, and its fighting effectiveness was further reduced by the tactical and morale effects of being surrounded. The Soviets attempted fiercely to create an encirclement known later as the Belyi Pocket [50], where they surrounded the defenders in the town of Belyi rather than conventionally attacking them. The same motivations could be encouraged in the player; they wouldn’t want to take heavy losses attacking the town, since the opponent would get a significant bonus for town defence, so they might attempt to surround the town instead to inflict the encirclement penalty on the defenders, making them much easier targets to finish off. This would reinforce the Unifying Theme to a great extent, making the players themselves inadvertently enter the mind of an Eastern Front commander, and begin thinking in terms of manoeuvre and front lines rather than individual units slogging it out. Another factor involved in these engagements was the usage of tanks as the “spearheads” of an offensive [51]. To reinforce this part of the Unifying Theme, the Armour units in the game should be granted an advantage in manoeuvrability to encourage them to be used as they were in real life; to exploit gaps in the enemy line and get around and ahead, while their infantry held the line in support. This advantage could be in the form of the ability to move diagonally; originally, all units could do this, but after this consideration, it was restricted only to Armour units, to give them this edge in manoeuvre. A further consideration in the reinforcement of the Unifying Theme is the potential idea of implementing the Fog of War system, initially suggested for the project by Professor Anthony Vickers. Being a major part of such a setting, the addition of military intelligence, or lack thereof, would further aid in placing the player deep inside the theme.

The music, commissioned from Richard Ampleford, a music student and regular gamer, will be designed after he has played a prototype of the game and had its goals described to him. The theme music has to portray tension, yet without holding it for too long, and has to be able to be looped and replayed without losing its impact. It should take its stylings from the beloved soundtrack of famous war strategy game Red Alert 2 [52], which tries to mix a feeling of excitement with tension, and has significant ‘Soviet’ theming, with a synthetic and industrial sound to make the player feel like they are interacting with a mechanical and dangerous set of tools. The sound effects, likewise, should be chosen for similar reasons, and can breathe new life into a product in the late stages of development.

With a clear vision of the design of the game, what remains is the largest task: the implementation. In the Initial Report, it was decided that the game would most likely be developed within Java. This eventually stuck as the route of choice, owing to the fact that using Java would allow a variety of relatively low-level programming and thought to go into the project as a demonstration of technical development, rather than using an engine such as Unity [53] which already has much of the work done. Java also has a great deal of libraries such as *Gson* [58] which would allow low-level implementation from scratch but without adding a major layer of complication to the project.

According to Reilly, games which involve complex logic - i.e. logic which would involve more than simply going towards your enemy and hitting them - typically become predictable and exploitable when played by an AI. He goes so far as to say that “playing against human opponents, then, is better than playing against computer-generated ones.” [75]. While this may not be applicable in all cases, particularly now that game AI has advanced since Reilly’s claim, the idea is still true in many games, especially ones which require strategic thinking, patience, and adapting to unforeseen circumstances across a whole board. Java is ideal for such an application, as it provides the programmer with the tools to easily create a client-server system, with built-in support for internet programming. Java also provides very easy ways to distribute the product to users, such as .jar files, which can be run on any computer with Java Runtime Environment installed - which is commonly installed and easily available online. One example of a game which used this feature of Java was Minecraft, which was incredibly popular, and provided all of its client and server programs as .jar files for easy distribution and setup [76].

The first consideration was to look at which libraries would be used, and which tasks they would have to perform. The most popular library for Java GUIs is Java Swing [59], and the most popular for Java 2D graphics is Java 2D [60]. This is because they are both built into the Java Development Kit (JDK8), and provide effective implementations for Graphical User Interfaces and for 2D graphics; developers often find all that they need in these two libraries, and so they stick with them. Swing is reliable and provides “built-in controls such as trees, image buttons, tabbed panes, sliders, toolbars, color choosers, tables, and text areas to display HTTP or rich text format (RTF)” [59], and is lightweight, meaning that it is platform-independent, unlike a framework such as AWT [60]. There are other GUI frameworks on the market for Java, such as SwingX [61], though this is still in development and none of its extra features over Swing seem to be necessary for this project. The current flagship in Java GUI development is JavaFX. It is set to eventually replace Java Swing as the go-to GUI building framework in the JDE, though both will still be included for the foreseeable future [62]. Although Swing does have most of the necessary components for hosting a 2D video game - easily separate-able frames, layer-able panels, Java 2D integration, message dialogs, action-listener implementations, etc., JavaFX might provide some advantages if used alongside it, namely in sound implementation, where Swing falls short. JavaFX provides the MediaPlayer class [63], which opens a world of easy sound implementation.

The Java 2D API expands the old Java AWT API significantly with further 2D graphics rendering capabilities. Its additions reach many packages, such as java.awt.image, java.awt.color, and java.awt.font, bringing further extensions to the 2D rendering capabilities of Java. The central class of the API is Graphics2D, which expands upon the old class Graphics [64]. In the process of creating a game, things like Composites may well need to come into usage, for example for changing the transparency of a component dynamically, or layering components on top of one another to form a new one. The implementation of Graphics2D in Java lends itself well to the creation of a 2D tile-based game, and so these built-in APIs and packages would be a valuable tool in this endeavour [65].

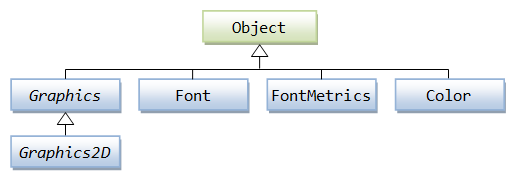


Fig5 *The Graphics2D Class Path as described by Nanyang Technological University* [64]

As demonstrated by Horstmann and Cornell, the Graphics API, and by extension the Graphics2D API, can be useful for rendering even text in particular ways; for example, an easy implementation of rendering text in its own font [66]. The API can even be integrated with more complex GUI features such as JLists, which are built in to the Swing package, to similar effect [67].

The Java AWT API still retains control of many features that are useful in creating a game, and thus despite it being a legacy API, it should remain in the mind of the game programmer. As Brackeen describes, AWT has an event dispatch thread, which allows Java to listen for calls from the operating system whenever, for example, a key is pressed [68]. AWT provides Listener classes such as MouseListener and KeyListener, which enable the user’s inputs to be handled directly as appropriate by the program. For example, one might need the program to print a certain String when the user presses the letter ‘s’ on the keyboard, and AWT’s KeyListener class would allow the programmer to implement this easily. However, the MouseListener class by default does not have some functions which the programmer might want to implement, especially in a game. For instance, they may want to have an event happen whenever the user moves their mouse, such as updating some information displayed to the user, or moving their character or an aiming reticle in accordance with the mouse movements. Within the javax.swing.event package, there is another newer class called MouseInputAdapter [69], which would enable the program to listen for not only the movements of the mouse, but also the movements of the mouse wheel, so a lot of extra functionality would be added which would be useful to a games programmer in such endeavours.

Swing contains some features which could be usefully adapted for usage by a games programmer, such as the JLabel class. Being the most basic subclass of JComponent, it would allow the programmer to render a wide variety of things onto an easily manageable Swing component. This can include Strings, Integers, and images in the form of Icons [70]. The ImageIcon class can be utilised to render images onto an Icon for use on a JLabel [71]. These ImageIcons can even be a useful way to display an animated image such as a GIF. However, they have some limitations, namely in that the icon cannot necessarily fill the component, and this isn’t addressed, because the component’s size is determined by the icon, rather than the other way around. The StretchIcon class, developed by D. Burke [72], would allow the programmer to have an image, animated or not, stretch to accommodate the JLabel being used, rather than having to tailor-make the assets with the appropriate size for the JLabel. This allows for more adaptable code, wherein any image can easily replace an existing one without being tailor-made. This would also allow resizing of the image within the program, for example to adapt to a resized window.

The last-updated state of a game will need to be stored separately from its individual classes and components, so that the various factors affecting the game can always be independently verified from a central source. This is important for consistency, especially in a multiplayer context, where it is important that a client knows what to expect from itself, let alone what it expects from the server. Barnes, in his Tic Tac Toe game example, uses a class called *TTTGame* to get and set some of the important variables affecting the game state at a given time, and even for checking whether or not someone has won [73]. Such a class can be useful for such general utilities as this, as well as storing and updating important variables. The class could also be used for initialising the game state, as Barnes demonstrates [73], where the game board could be set up at the start of the game by a method within the class, to keep such a task in a central and sensible position rather than writing it directly into the game-start code.

Brackeen uses an interesting technique in one of his games. For the running of the client-server communications, which enable the two clients to communicate with each other, he uses two separate threads for each client; one for receiving messages from the server and implementing changes based on their contents, and another thread for transmitting changes to the server to be processed and sent along to the other client(s) [74]. Such a modular implementation allows each client to always have a thread available to do what it needs to do, rather than waiting for the main execution thread to become free so that it can receive the next communication from the server. This is more of an issue for real-time games rather than turn-based ones, but more importantly, even for turn-based games, this approach would allow the game to be expanded to an indefinite number of clients on the server, so even with an finite number of players - 2 or 3 - there could be a lot of spectators on the server watching the match unfold.

A game, or any program for that matter, must also be able to handle Exceptions which occur at runtime. While “developers might never see them in testing, users most certainly would,” [77] and thus the developer must take great care in providing every ‘risky’ part of the program with clean error handling. This might be as simple as adding an exception catch which produces a message dialog whenever another user disconnects from the server, or perhaps when the server sends an erroneous message, indicating that something has gone wrong. In this case, the user must be cleanly informed that something has gone wrong and they should, for example, restart the game. Otherwise, the user experience could be seriously hampered, and the entire illusion built by the game designer, and the immersion that comes with it, might be shattered, or they may even become frustrated with the game’s opacity and discard it. This is not, however, an appropriate solution for bugs - only for exceptions that are entirely beyond the control of the programmer, which occur at runtime.

For a multiplayer Java game to work, one is likely going to have to encounter Streams. Output streams are producers; they create bytes and send them either to a file, a data structure, or to a network connection [78]. Input streams are consumers, meaning that they receive this data in its byte form. The Output stream works sequentially, in that bytes are read and processed in the order in which they enter the stream. However, for speed and efficiency, Reilly recommends using a Buffered stream, meaning that a large chunk of bytes will be read together, rather than one at a time, the first time the read method is invoked. Subsequent read commands can then read from the saved buffer as needed rather than having to re-read the whole data sequence in order [79]. The importance of clean code in such a project cannot be overstated. The amount of different classes and methods working together in a multiplayer game means that a developer can’t afford to allow their code to simply work, and must take steps to make it human-readable, manageable, and efficient. A delay of even 1 second between clicks and their responses could begin to grate on the player and put them off the game, and so such things should be minimised as much as possible.

R. Martin puts forward several famous and important methods for making one’s code clean and human-readable. Not only does this allow for the code to be understood, but importantly allows for the code to be easily modified by other programmers or even by the original developer(s), a common and necessary occurrence in the industry. Famously, the popular classic *Red Dead Redemption* was never able to be ported to PC from the Xbox 360 system because the code was such a mess that it was financially and technically unviable to make the attempt [80]. One of Martin’s first-discussed methods in his famous book Clean Code is the naming of parts of program code; whether this mean primitives, objects, classes, methods, packages, etc. He provides an example in which a list had been named ‘*theList*’. This provides no indication of intention, and thus makes the code more obscure than it needs to be. The code isn’t complicated, nor is the arrangement of the lines poorly done, but this generic variable name contributes to making the code less understandable [81].

Programmers should also avoid “leaving false cues that obscure the meaning of code.” [82] For example, a variable having “list” in its name while in fact being a Map or a String rather than a List. Such a problem would invariably make program code much less legible to anyone trying to understand or modify it past its original development, and would make the original development process in itself harder. Programmers are human, and so even during the original development cycle things might become difficult if it is unclear what a variable written yesterday was supposed to be and what it was supposed to do. The names of variables should also be kept searchable, i.e. not ‘x’, for the ease of testing and expanding the program. As well as the names of variables, the fewer instance variables a class has, the better, and as many of the methods of that class as possible should manipulate each instance variable [83]. However, maximal cohesion is not necessarily desirable, as it can become difficult to break functions up into small pieces if they constantly have to have variables passed between them [84]. Therefore, although we would like cohesion to be high, there is a balance to be struck.

Program classes should also be as small as possible; breaking up code can make it easier to understand and read. However, as software developer P. Torok points out, there is another balance to be struck here - “At a certain point, making your methods / classes smaller starts to decrease readability, rather than increasing it.” [85] The size of classes will naturally grow over time, and it should be a constant process to break them up as much as possible, without going too far and making the code harder to read and write than it was to begin with.

In the case of a multiplayer game with multiple program threads, the issue of concurrency should be understood well by the developer before the project is undertaken. It is a common misconception that concurrency always increases the performance of the program. This is not always the case; concurrency only increases performance when there is a significant waiting list for the processor’s attention. In some cases, however, concurrency can allow program processes to run without interrupting each other; processes which may need to run constantly and indefinitely until the program’s end, such as a client thread listening for information from a server, or vice versa [86].

Code formatting is key to a project’s success. It is “about communication, and communication is the professional developer’s first order of business.” The vertical size of files, according to Martin, should be kept to within 200-500 lines, even within very large projects. This is desirable because, again, it increases the readability of the code [87]. Furthermore, having good vertical spacing between different concepts can vastly improve the readability of code; having all of the code mashed together can make it a vastly bigger task to try to decipher it. Simply having an empty line between two pairs of lines can make it far clearer that they are conceptually different [88]. One should, along the same lines, avoid ‘train wrecks,’ wherein many methods are linked together in a long and messy chain, and are generally considered “sloppy style” [89]. In taking all of these techniques together, the developer should be able to greatly increase the efficiency of the project, and ensure the continued life and expandability of their game.

## 

# Technical Documentation

## Technical Review

### Structure of the Program

The program is built with the following structure. The white arrows represent one part leading to or creating another, the black arrows represent one part being a part of another, and the grey arrows represent communication between parts.

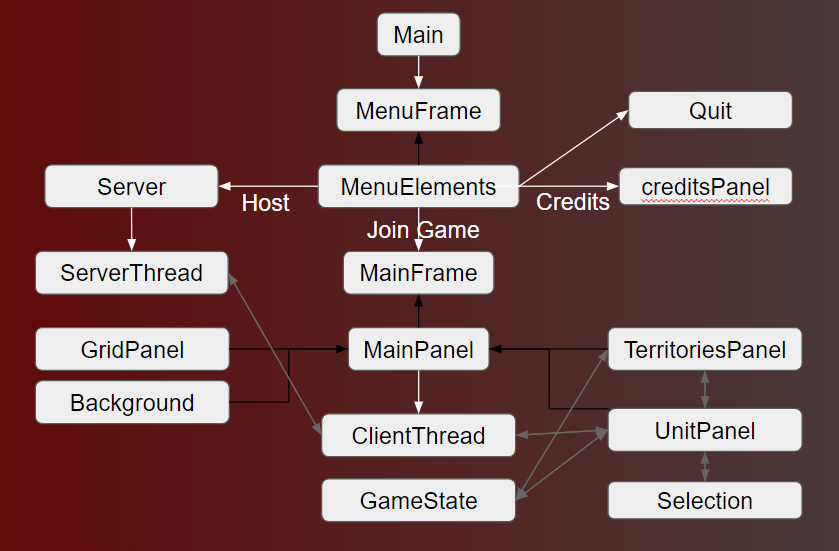


Fig1 *Diagram describing the program’s technical layout*

The class which the .jar file first leads to is the Main class, which only contains the ‘main’ method. This method simply sets the last moved player as 2, so that Player 1 will always go first, and then opens the main menu, in the form of MenuFrame.

* MenuFrame is a JFrame made up of MenuElements, which is a panel containing the menu’s elements. MenuFrame creates MenuElements, which has 4 buttons on it; Join Game, Host Game, Credits, and Quit
* Clicking Credits will create creditsPanel, and open a JOptionPane message dialog, which will contain the panel. The message dialog can be closed by clicking ‘OK’
* On MenuElements, clicking the Quit button will close and terminate the program
* Clicking Host Game will start a server. The server will simply wait to receive connections. When a connection is received, the server will run a new ServerThread to listen to and communicate with that given client
* If the Join Game button is clicked on MenuElements, the MainFrame will be created
* MainFrame will create a MainPanel and add it to itself
* The MainPanel will create 4 other panels and add them to itself. These 4 panels will make up the game display, and are layered on top of each other with OverlayLayout
* Background is a panel which simply loads and displays the background map
* GridPanel is a panel which creates and displays the game’s grid lines
* TerritoriesPanel is a panel which creates and displays the two players’ territories as coloured squares
* UnitPanel is the panel which displays the players’ units and handles their movement
* These four panels are created by, and come together to form, MainPanel
* MainPanel also initiates and runs ClientThread, a thread which will listen to and communicate with the server on behalf of the client
* ServerThread and UnitPanel communicate back and forth with ClientThread
* The Selection class simply contains the currentSelected method. The class is invoked by the game logic, mainly within UnitPanel.
* GameState is essentially a utility class; it initialises and stores the game state, calculates combat, and some other small tasks.
* ServerThread, TerritoriesPanel, and UnitPanel communicate back and forth with GameState, such as to request combat calculations or the initial state of the game board.
* UnitPanel and TerritoriesPanel communicate with each other to update the territory board.
* The Unit class is not part of the ‘structure’ per se, as it simply defines the Unit object and allows it to be modified.

### Purpose and Functionality of the Program Classes

#### Main

The Main class is the initialising class of the whole program, and is the one that is run when the .jar file is run. The class’s only method - *main* - sets GameState’s stored lastMovedPlayer variable to 2 so that the first player will always be 1. The method then creates the frame for the main menu, and gives it a title:

**new** MenuFrame(**"Operation Mars"**);

#### Unit

The Unit class defines the Unit object, which is the most important object in the program, and represents one of the players’ units which they control. The class implements the Serializable interface, meaning that it can be serialised, which is necessary for the transmission of unit information over the Java socket system used in this program, because the system uses Gson, a Java library which can serialise objects into their Json representation. The class declaration begins with the class variables, which make up the definition of a unit: *icon*, *iconFileName*, *name*, *xLocation*, *yLocation*, *health*, *vet*, *type*, *faction*, and *selected*. The variable *icon* is a BufferedImage, and directly stores the image icon associated with the Unit object, such as the following:



Fig2 *Artillery Unit Icon*

This doesn’t include the veterancy counter or the health bar, which are rendered in-game from values rather than loaded in or stored. This variable is *transient*, meaning that it is left behind when serialisation is performed, because otherwise an error would occur, because BufferedImage objects can’t be serialised. The variable *iconFileName* stores the name of the image file of the icon. This not only allows the initial icon to be loaded from the file location, but also allows the icon to be easily re-loaded from the same location once the Unit object has been serialised and then de-serialised and the original *icon* has been left behind. The *icon* simply loads the image from the media folder using the Unit object’s *iconFileName* and the ImageIO class.

* *xLocation* and *yLocation* are simply ints (integers) which store the unit’s current X and Y coordinates on the game board. These will range from 0-9, as there are 10 squares on the board and the index starts at 0.
* The variable *type* is a String which stores the type of the unit. This string’s values can be either “inf”, “art”, or “arm”. This type is given to the unit at the start of the game and never changes.
* *health* is an integer which starts at 100, and stores the unit’s current health. This is the only one which is set by default in the constructor rather than being passed into it, because new units always start with 100 (full) health.
* *name* is a String which stores the unit’s historical name, such as “11th Cavalry Corps”. This is only for historical flavour, and has no impact on the functioning of the game.
* *faction* is an int which stores the unit’s faction, either 1 or 2, meaning either Soviets or Germans. This determines which player the unit belongs to.
* *selected* is a Boolean (true or false) signalling whether or not the given unit is currently selected. If it is, it will get a white highlight rendered around it to show this. By default, *selected* is of course false.
* *vet* stores the ‘veterancy’ of the unit (how experienced it is) as an int value. This affects the unit’s combat abilities; higher veterancy will give them an advantage in combat.

The primary constructor for the unit takes the *iconFileName*, *xLocation*, *yLocation*, *type*, *faction*, *vet*, and *selected* arguments. They are received when the game is first started, from the GameState class. It sets them in the corresponding values in the Unit object, and sets *health* to 100 as mentioned above.

The copy constructor is below the primary constructor and can be identified by the fact that it takes only a Unit object as its argument, rather than a series of features of the Unit. The copy constructor simply takes a unit object and copies it into a new one by copying all of its attributes one by one into the new object. The icon is not directly copied, but rather inferred from the *iconFileName* variable of the object being copied.

Following the copy constructor are a series of ‘getter’ and ‘setter’ methods, which simply return and modify the attributes of the Unit object. Two examples are below:

**int** getHealth(){**return health**;}

If [unit’s name].getHealth is invoked, the int *health* of that unit will be returned.

**void** setHealth(**int** newHealth){ **this**.**health** = newHealth; }

If [unit’s name].setHealth is invoked, the int *health* of that unit will be set to whatever is passed into the invocation as an argument. The method is void because it doesn’t return anything.

The setIcon() method stands out in these ‘getter’ and ‘setter’ methods as slightly more complex. It contains a try/catch block to catch an IOException if the image file is not found from the file name. It reloads a unit’s *icon* based on its stored file name when it has just been de-serialised.

#### MenuFrame

The MenuFrame class defines the MenuFrame object, which is extended from the JFrame class. This means that it is a subclass of the JFrame class, and extends it to create a new object. It receives the String argument *title* from the Main method. The ‘super’ call at the start of the constructor calls the super class’s constructor to set the title of the window (frame) to whatever was received from Main, in this case simply “Operation Mars”. The default close operation is set to terminate the whole program. This may be inconvenient if the user wishes to close the main menu window after launching the program, but it means that there won’t be the case of the program not actually terminating if the user doesn’t open the game frame and close the program from there. The MenuFrame then creates a MenuElements object, and uses BorderLayout to add the new object to the centre of itself.

#### MenuElements

The MenuElements object class extends from the JPanel class and implements the ActionListener interface. The ActionListener interface allows it to contain interactive buttons. The panel MenuElements constitutes all of the elements of the main menu, and is added to the frame MenuFrame, which holds and displays it. The file *song.mp3* is loaded into a MediaPlayer at the start of the class, so that it can be played on the main menu. The void method getPreferredSize() tells Java that the menu should start at a certain size. In this case, that size is 1104x832, which is the original size of the background video. The JFXPanel is declared outside and then inside the constructor, because without the seemingly redundant declaration outside, it doesn’t work. The cycle count of the MediaPlayer is set to “INDEFINITE,” meaning that the song will loop indefinitely when it finishes. Its volume is set to 0.3, meaning 30% of the original.

The music is then played, and the background file is loaded as a StretchIcon (meaning that it will stretch to fill the frame, so the user can resize the menu and the GIF background will respond). The icon of the panel is then set to the StretchIcon, meaning that the StretchIcon will appear in the background of the panel and be directly part of it. This is contained within a try/catch block so that a missing file won’t throw an unhandled IOException.

**try** {

StretchIcon image = **new** StretchIcon(**this**.getClass().getResource(**"/media/"**+fileName));

**this**.setIcon(image);

**this**.setVisible(**true**);

} **catch** (Exception e) {

System.***out***.println(**"Background file not found"**);

}

The panel’s layout is set to GridBagLayout, which allows its elements to be laid out with coordinates like a grid. This allowed for the buttons and text to be laid out neatly in the desired way. A GridBagConstraints object is declared to allow for defining the layout and location of each element of the menu. The anchor attribute of the GridBagConstraints object defines where on the panel the elements will be placed from, and the insets attribute defines the padding around each element. The North argument means that the elements should be centred horizontally. The title and buttons are then created, and are laid out one by one onto the panel using the GridBagConstraints, with coordinates for each one. For example:

gbc.**gridx** = 0;

gbc.**gridy** = 4;

**this**.add(hostGame, gbc);

This places the hostGame button at position 0,4 relative to the starting position, which is in the top centre (so it will be 4 spaces down from the top centre). Some of the insets are also modified between each element.

The buttons all have String arguments which define their actions when clicked. For example, the Host Game button has the action “host”:

JButton hostGame = **new** JButton(**"Host Game"**);

hostGame.addActionListener(**this**);

hostGame.setActionCommand(**"host"**);

When the button is clicked, it will call the action listener implemented by the containing class (i.e. *this*) and the action listener will see that the action is “host”, and act accordingly:

} **else if** (**"host"**.equals(e.getActionCommand())) {

**if**(!GameState.*getServerHosted*()) {

**new** Server().start();

GameState.*setServerHosted*(**true**);

JOptionPane.*showMessageDialog*(**null**, **"You are now hosting a game. To join your own game, click Join Game and press Enter"**, **"Game Hosted"**, JOptionPane.***INFORMATION\_MESSAGE***);

} **else**{

JOptionPane.*showMessageDialog*(**null**, **"Game already hosted. To join your own game, click Join Game and press Enter"**, **"Game Already Hosted"**, JOptionPane.***INFORMATION\_MESSAGE***);

}

As can be seen, there is also an implementation to avoid the hosting of two servers on one client, which would cause an error. In this case, a message dialog comes up to warn the user. The same is done in the case of the user trying to launch the game twice from the same client.

The openCredits() method is called when the Credits button is clicked, and creates a message dialog with a custom panel embedded onto it. This panel is also written using the GridBagLayout. The lines of text are all defined at the top, and then laid out one by one in the same fashion that the main menu itself is laid out. The background is then changed to white and the panel is displayed on a MessageBox which comes up when the Credits button is clicked.

JOptionPane.*showMessageDialog*(**this**,creditsPanel,**"Credits"**, JOptionPane.***PLAIN\_MESSAGE***);

#### MainFrame

The MainFrame class is like the MenuFrame class in that it extends from the JFrame class. Its main jobs are to play the game’s music and to hold the main panel which houses the game itself. It initialises the MediaPlayer in the same way as the MenuFrame, and plays the game music at volume 0.2 (20%). It creates MainPanel and sets MainPanel’s layout to OverlayLayout, which will allow it to layer its constituent panels on top of each other. This frame’s default close action will also close the program as a whole, so if the user clicks the X button on the window then the program will terminate. The MainPanel is added to the centre of MainFrame using BorderLayout.

LayoutManager overlay = **new** OverlayLayout(main);

main.setLayout(overlay);

setLayout(**new** BorderLayout());

getContentPane().add(main, BorderLayout.***CENTER***);

#### MainPanel

MainPanel is an object class which extends JPanel. It contains all of the other panels which make up the game: Background, GridPanel, TerritoriesPanel, and UnitPanel, as shown in Figure 1 (p21). When an instance of it is created, it receives as arguments the containing frame and the IP which the user entered, so that the IP can be passed to the client thread to connect to the server. The frame is received as an argument so that the MainPanel can change the frame’s title at initialisation when the game starts, to include the player’s faction and whose turn it currently is (always the Soviet player’s turn initially). The frame is then passed to UnitPanel, which further changes the frame title as appropriate as the game goes on, such as when turns change. This is performed here rather than in the frame constructor because it must be performed after the player number has been received from the server by the ClientThread.

**if** (GameState.*getFaction*() == 1) {

frame.setTitle(**"Operation Mars | Faction: Soviets | Your Turn"**);

} **else if** (GameState.*getFaction*() == 2) {

frame.setTitle(**"Operation Mars | Faction: Germans | Opponent's Turn"**);

}

MainPanel’s main duties are to create the 4 panels of the game, and then to add them to itself. It also creates and runs the ClientThread, which communicates with the server on the client’s behalf. The panels are added in the following order:

**this**.add(BorderLayout.***CENTER***, unitPanel);

**this**.add(BorderLayout.***CENTER***, territoriesPanel);

**this**.add(BorderLayout.***CENTER***, gridPanel);

**this**.add(BorderLayout.***CENTER***, background);

Because MainPanel (this) is in OverlayLayout (as set previously in MainFrame) this order of the adding of the panels is important, and means that unitPanel will visually be in front, with territoriesPanel behind, then gridPanel, then background. The units sit on top of the territory and are displayed first and foremost. The background panel, on its creation, is passed the file name of the map image file, in this case *operationmarsmap.png*. This tells the background panel where to look for the image file which will display the background of the game, i.e. the map with the towns etc.

#### UnitPanel

UnitPanel contains most of the code for the rendering of, movement, and interaction between the units in the game. It also handles most of the sound effects, as these also occur via the units. The class extends JPanel, and within it is defined the class Listener which extends MouseInputAdapter and handles the user’s mouse clicks and movements and their effects. UnitPanel has a lot of class variables, since it has a variety of methods which need constant shared access to most of them. Variables *w* and *h* are ints and define the width and height of the game grid, and are assigned the number 10 in the panel’s constructor. Variable *faction* tells the panel which faction the player belongs to; 1 or 2. *totalVPs* and *totalEnemyVPs* are used to store the last calculated figure for each player’s victory points.

The floats *mainGraphicsAlpha* and *combatAnimationAlpha* are used to store the alpha values for the composites of the graphics, such as the ‘your turn’ message and the combat explosion image. These control the transparencies of the graphics, so they can fade in or out. The Unit objects *hurt1* and *hurt2* store copies of the units which the opponent just made fight, and the objects *combatant1* and *combatant2* store copies of the units which the player just made fight. These variables allow for appropriate animations and sounds etc to be used. The int *size* dictates the pixel size of each tile, and if this is altered then the size of the game could be changed, though a proportionally larger background map would need to be used, or the same image could be stretched

The ArrayList *gameGrid* is two-dimensional (ArrayList of ArrayLists) and each ArrayList stores 10 Unit objects, resulting in a 2D 10x10 grid which stores the units of the game in their current state.

The Boolean variables *won* and *lost* simply record whether or not the client has won the game, and whether or not they have lost. If both values are false, which is the default, then the game has not yet ended.

The Unit object selected stores a copy of the currently selected unit; this is so that the selected object can be operated upon without the original object on the game grid being modified.

Color transparent holds a colour object which has no RGB values and an alpha value of 0 (the last argument), meaning that it is transparent. This is used to make the background of the unit panel transparent so that the lower layers can show through behind it.

Color **transparent** = **new** Color(0,0,0,0);

*territories* stores a 2D array of ints indicating the current layout of the two players’ territories, so that they can be checked and modified, and sent to the territories panel. territoriesPanel stores a reference to the *territoriesPanel* created in *mainPanel’s* constructor. This allows unitPanel to interact with it. In much the same way, *frame* and *mainPanel* store references to the main frame and the main panel so that *unitPanel* can interact with them.

The JFXPanel fxPanel needs to be created for the media players to work, though it isn’t used itself.

There are a series of Media objects which hold the references to the sound effects for the MediaPlayers to use. They are declared as such.

Media **turnSound** = **new** Media(getClass().getResource(**"/media/yourturn.mp3"**).toString());

They contain string arguments referring to the file paths of their corresponding sound effect, which are all stored in the media folder, like the graphical assets.

The MediaPlayers are then declared to utilise these Media objects like so:

MediaPlayer **turnSoundPlayer** = **new** MediaPlayer(**turnSound**);

The UnitPanel class’ constructor first sets the UI background to black and the foreground to white, mainly affecting the colour of the tooltips. It then makes itself focusable and grabs the program’s focus, so that it can be interacted with by the user. It assigns all of the received arguments to its own variables declared above, like so:

**this**.**frame** = frame;

The w and h variables are then set to 10, representing the width and height of the game grid. The last moved player is set to 2, so that Player 1 always goes first, and then the volume of all of the sound effects’ media players is set to 0.7 (70%), to be louder than the music.

The paintComponent method tells the UnitPanel how to render itself.

First, it sets the panel’s opacity to false so that it is transparent to the panels beneath it. Three Graphics2D objects are created; mainGraphics, combatGraphics, and endGameGraphics. The mainGraphics object is used to draw the ‘YOUR TURN’ message and the opponent’s combat animations. The combatGraphics object is used to draw the client’s combat animations, and the endGameGraphics object is used to display the ‘YOU WON’ and ‘YOU LOST’ messages to the client.

The board is updated from the one stored in GameState, and an embedded For loop begins to cycle over every tile in the game and render it.

If a unit exists (isn’t null), has more than 0 health, and is nearby, then it should be rendered:

**if** (*gameGrid*.get(i).get(j) != **null** && *gameGrid*.get(i).get(j).getHealth() > 0 && GameState.*isNear*(*gameGrid*.get(i).get(j), *gameGrid*)) {

If that unit is selected, then a white rounded rectangle should first be drawn under it, whose edges will appear around the unit’s edges.

**if** (*gameGrid*.get(i).get(j).getSelection()) {

g.setColor(Color.***WHITE***);

g.fillRoundRect((*gameGrid*.get(i).get(j).getxLocation() \* *size*) + 8, (*gameGrid*.get(i).get(j).getyLocation() \* *size*) + 8, 44, 44, 10, 10);

}

The rectangle is drawn at the unit’s location multiplied by the pixel size of each grid square, plus 8 pixels so that it isn’t right at the edge of the grid square, because we want the rectangle to be in the middle with some space around it for the user to be able to see the map behind it. 44 is the pixel width and height; again, smaller than the grid square. 10 represents the arc width and arc height of the curved corners of the rectangle.

The rendering loop then draws each unit icon in its corresponding place, in much the same way that the rounded rectangles are rendered above. The mainGraphics object then draws a green line over the top of the unit’s icon to represent its health; the arguments provided were obtained by trial and error to try to position the line so that when health is 100, the line symmetrically spans from the top left of the unit icon to the top right, with space on the sides.

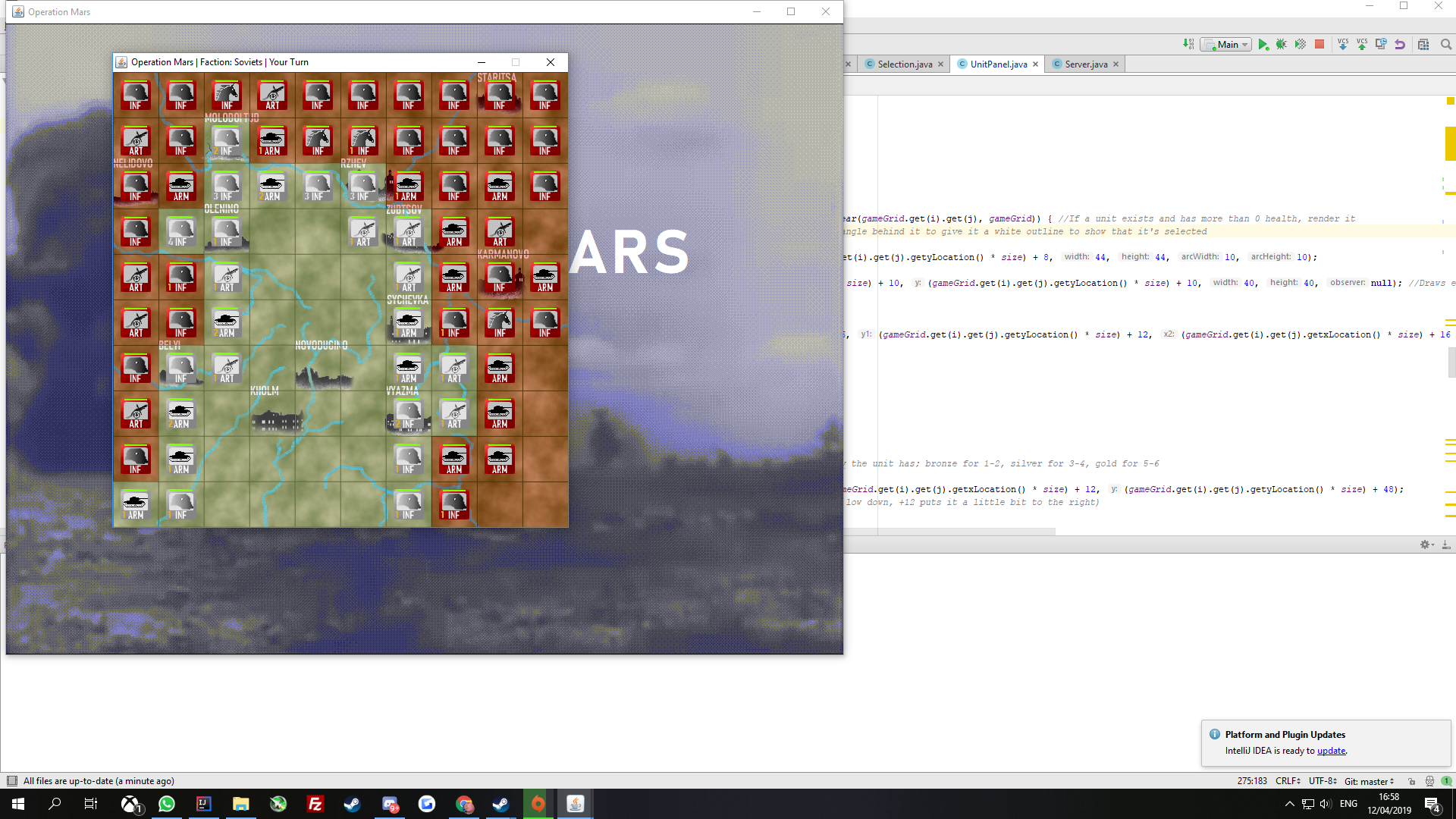


Fig3 *Unit with health bar and veterancy*

The loop then checks what the unit’s current veterancy is, and writes it in the bottom-left of the unit icon. If the veterancy is 1 or 2, it is written in bronze, if it’s 3 or 4 it’s written in silver, and in gold if it’s 5 or 6. This provides another visual aid for users to quickly see the value of a unit to them.

**if** (*gameGrid*.get(i).get(j).getVet()>0) {

**if** (*gameGrid*.get(i).get(j).getVet()<=2) {

mainGraphics.setColor(**new** Color(214,175,54)); *//Bronze*

} **else if** (*gameGrid*.get(i).get(j).getVet()<=4) {

mainGraphics.setColor(**new** Color(212,212,212)); *//Silver*

} **else if** (*gameGrid*.get(i).get(j).getVet()>4) {

mainGraphics.setColor(**new** Color(255, 223, 0)); *//Gold*

} *//Set the colour of the veterancy display to correspond with the amount of veterancy the unit has; bronze for 1-2, silver for 3-4, gold for 5-6*

mainGraphics.setFont(**new** Font(**"BahnSchrift"**, Font.***BOLD***, 12));

mainGraphics.drawString(Integer.*toString*(*gameGrid*.get(i).get(j).getVet()), (*gameGrid*.get(i).get(j).getxLocation() \* *size*) + 12, (*gameGrid*.get(i).get(j).getyLocation() \* *size*) + 48);

*//Write the unit's veterancy over the bottom corner of its icon (+48 to Y puts it low down, +12 puts it a little bit to the right)*

}

Once the unit rendering loop has completed, the method checks whether or not it’s the user’s turn, and whether or not the game has ended. If it is the user’s turn and the game hasn’t ended, the ‘YOUR TURN’ message rendering is executed, as well as the rendering of the opponent’s move.

**if**(GameState.*getLastMovedPlayer*()!=GameState.*getFaction*() && !(*won*||*lost*)) { *//If the player wasn't the last to move and he has neither won nor lost*

AlphaComposite alCom = AlphaComposite.*getInstance*(

AlphaComposite.***SRC\_OVER***, **mainGraphicsAlpha**);

mainGraphics.setPaint(Color.***WHITE***);

mainGraphics.setComposite(alCom);

mainGraphics.setFont(**new** Font(**"BahnSchrift"**, Font.***BOLD***, 50));

An AlphaComposite object is created to utilise the alpha value to give the *mainGraphics* object a transparency, so that the combat graphic and the ‘YOUR TURN’ message can fade out. The alpha is reduced by a small amount, and then the painting is done again. This continues until the alpha reaches 0.

**mainGraphicsAlpha** -= 0.02f;

**if** (**mainGraphicsAlpha** <= 0.0f) {

**mainGraphicsAlpha** = 0.0f;

} **else** {

repaint();

}

If it isn’t the client’s turn and the game hasn’t ended, the frame’s title is updated to remind them of this. The second rendering section, which uses *combatGraphics*, simply renders the explosion effect for when the client makes two units fight, rather than when the enemy does so. It is executed in the same way as the previous method; a BufferedImage of the explosion is drawn, and then is slowly faded by reducing its alpha value and repainting. The location of the image is also chosen in the same way, with the coordinates being halfway between the two fighting units - taking one unit’s coordinates and adding half of the distance to the other unit, and adding an offset of 10 since the units also don’t start at the edges of the grid squares. This puts the explosion halfway between the two units.

BufferedImage explosion = ImageIO.*read*(getClass().getResource(**"/media/explosion.png"**));

combatGraphics.drawImage(explosion, ((((**combatant1**.getxLocation()\**size*)+(**combatant2**.getxLocation()\**size*)))/2)+10, ((((**combatant1**.getyLocation() \* *size*)+(**combatant2**.getyLocation()\**size*)))/2)+10, 40, 40, **null**);

Finally, if the player has won or lost, i.e. if the game is over, then either the ‘YOU WIN’ or ‘YOU LOSE’ message is rendered by *endGameGraphics*.

**if** (*won* || *lost*){

This doesn’t need to use an alpha value as it doesn’t need to fade at this point, since the game is over.

The deselect() method simply cycles through the 2D ArrayList of units in standard fashion - with an embedded *for* loop, and then sets each Unit object’s stored selection value to false:

gameGrid.get(i).get(j).setSelection(**false**);

setCombatants() is used to set which two units the client just made fight, so that the combat animation can be rendered between them:

**combatant1** = **new** Unit(attacker);

**combatant2** = **new** Unit(defender);

updateGrid() is used to update the grid of units when the opponent makes a move to match what the server says is the new state of the game grid.

Firstly, the method increments the turn counter and then checks if the opponent won the game during their turn. If this was the case, then the server will send an 11 if the opponent is player 1, and a 12 if the opponent is player 2. Therefore, if the opponent’s player number is greater than 2, then they must have won:

GameState.*setTurnCount*(GameState.*getTurnCount*()+1);

**if** (GameState.*getLastMovedPlayer*()>2){

*lost* = **true**;

repaint();

}

Next, an embedded *for* loop runs through the board. If any two opposing units from the newly received board have less health than their counterparts on the client’s board, then they are marked as ‘hurt’, meaning they’ve just fought and need to have a combat animation displayed between them:

**if** (newGrid.get(i).get(j).getHealth() < *gameGrid*.get(i).get(j).getHealth()) {

**try** {

**if** (**hurt1** == **null**) {

**hurt1** = **new** Unit(newGrid.get(i).get(j));

} **else** {

**hurt2** = **new** Unit(newGrid.get(i).get(j));

}

} **catch** (IOException e) {

e.printStackTrace();

}

}

Then, within the same embedded loop, each unit has its icon set via its built-in setIcon method, and then it is copied from the received board to the client’s own board.

newGrid.get(i).get(j).setIcon(); *//Sets the units' icons based on their icon file locations, since the icons themselves aren't sent over by the server*

}

*gameGrid*.get(i).set(j, newGrid.get(i).get(j));

Then the ‘your turn’ sound effect is reset to the start and played to coincide with the start of the client’s turn:

**turnSoundPlayer**.seek(**new** Duration(0));

**turnSoundPlayer**.play();

The method *updateTerritories* also works by cycling through a 10x10 embedded *for* loop; it goes to each tile in the *territories* 2D array, and updates its value based on which unit is in that corresponding location on the unit grid (*gameGrid*). 1 is subtracted because the territories grid works with 0s and 1s rather than 1s and 2s, but it works in the same way; 0 is player 1’s territory, 1 is player 2’s territory.

**if**(*gameGrid*.get(i).get(j) != **null**) {

**territories**[i][j] = *gameGrid*.get(i).get(j).getFaction() - 1;

}

The new *territories* grid is then saved to the one stored by *territoriesPanel* so that it can be rendered.

The *playCombatSound* method checks with 3 *if* statements which unit type the attacking unit is; “inf”, “art”, or “arm”, and plays the corresponding sound effect. For example:

**if** (attacker.getType().equals(**"inf"**)) { *//If the attacker was infantry, play the infantry combat sound effect*

**infSoundPlayer**.seek(**new** Duration(0));

**infSoundPlayer**.play();

}

*playMovementSound* works in exactly the same way, but plays the corresponding movement sound effect, like so:

**if** (selected.getType().equals(**"inf"**)) {

**infMoveSoundPlayer**.seek(**new** Duration(0)); *//If the moved unit was infantry, play the infantry movement sound effect*

**infMoveSoundPlayer**.play();

}

The getPreferredSize() method tells Java which size the unit panel should be; the preferred size is defined as the pixel size of a tile multiplied by the amount of tiles in the game:

**return new** Dimension(**w** \* *size*, **h** \* *size*);

##### Listener

Class *Listener* is defined within class UnitPanel, and is used to keep track of the user’s mouse movements and clicks within the game. The class extends MouseInputAdapter, and takes the unit panel as an argument. It overrides the methods of MouseInputAdapter: mouseClicked, mousePressed, mouseReleased, mouseEntered, mouseExited, mouseDragged, and mouseMoved. The only ones of these which contain code are mouseClicked and mouseMoved; the remaining ones do nothing. Method mouseClicked handles the user’s clicks and their effects, being the selection and control of units on the board, and mouseMoved simply handles the tooltips which appear when the user hovers over a unit.

When the mouse is clicked, the mouseClicked method is triggered, with MouseEvent *e*, which allows the mouse event to be further analysed, such as for coordinates. The method checks if the player has won or lost; if neither are true, this means the game is still going, and so it proceeds to process the user’s click. The method then checks if the user was the last player to move; if they were, then they can proceed, but otherwise the click isn’t processed because they just had their turn. There is only a console print command here, for debugging purposes. Having a message come up would be annoying to the user. If the click is processed, the grid coordinates of the user’s click are obtained by taking the pixel coordinates from the MouseEvent e, and dividing them by the pixel size of the grid squares:

**int** x = (e.getX() / *size*);

**int** y = (e.getY() / *size*);

Variables are also created to hold the previous coordinates of a unit, and the *gameGrid* and *faction* variables are updated from their storage in GameState, as indicated in Figure 1 (p21):

**int** oldX;

**int** oldY;

*gameGrid* = GameState.*getBoard*();

**faction** = GameState.*getFaction*();

In the below loop, a 2D (embedded) *for* loop is used, so that each tile in the game grid can be checked one by one. The For loops run until the size of the grid (10) and are iterated by 1 each time. The *if* statement within checks if the user has clicked a unit (a non-null location) and the clicked unit is marked as selected within its object, it is copied into the *selected* variable to be operated on. Once the selected unit is copied, the embedded loop is broken using the SelectionLoop label. The loop is broken early because only one unit needs to be selected. Breaking early saves on extra CPU time.

SelectionLoop:

**for** (**int** i = 0; i < GameState.***width***; i++) {

**for** (**int** j = 0; j < GameState.***height***; j++) {

**if** (*gameGrid*.get(i).get(j) != **null** && *gameGrid*.get(i).get(j).getSelection()) {

**try** {

**selected** = **new** Unit(*gameGrid*.get(i).get(j));

} **catch** (IOException e1) {

e1.printStackTrace();

}

**break** SelectionLoop;

}

}

}

Following this is an if statement which sets the Unit in the game grid’s selection as true:

**if** (*gameGrid*.get(x).get(y) != **null**) {

Unit selectedUnit = **null**;

**if** (*currentSelected*(*gameGrid*) != **null**) {

selectedUnit = *currentSelected*(*gameGrid*);

}

**if** (*currentSelected*(*gameGrid*) == **null** && *gameGrid*.get(x).get(y).getFaction() == **faction**) {

*gameGrid*.get(x).get(y).setSelection(**true**);

}

}

The selectedUnit object simply exists so that the reference can be handed to a persistent instance variable so that unit re-selection can happen smoothly without having to first deselect. If the first if statement inside the outer one is removed, the game still works, but it takes 2 clicks to switch unit.

Following this, an if statement checks if the user has clicked somewhere while a unit is selected:

**if** (**selected** != **null**){

*//If a unit is selected and you've clicked somewhere*

**try** {

The code that follows is the code for movement and combat. The processing only takes place if the move is legal, however, which is checked by a method in GameState, as indicated in Figure 1 (p21):

**if** (GameState.*isLegal*(**selected**, x, y)) { *//If the move is legal (within 1 square, allowing diagonal movement for armour units)*

Following this, 3 scenarios are checked for: the user clicks an adjacent empty square, the user clicks an adjacent enemy, or the user clicks an adjacent ally.

In the case that the user clicks on an adjacent empty tile, the currently selected unit’s location is saved, then it is copied into the new location, and then the object at the old ArrayList location is deleted so that the unit is only in its new place and not in its old place. The *territories* 2D array is then updated from the one stored by the *territoriesPanel*, the unit’s new location is updated to belong to his faction territorially, and then the newly adjusted territories array is saved again to the territoriesPanel. The method for playing the unit’s movement sound effect is then called, and the if statement ends.

The second case is that the user clicks on an adjacent enemy unit. This is checked by the following if statement:

} **else if** (*gameGrid*.get(x).get(y).getFaction() != **faction**) { *//If the selected unit has been clicked onto an adjacent enemy unit*

If this happens, then the game handles combat and its effects. First, the territories array is updated from the one stored by territoriesPanel. Then, the combat simulation method is called, which is in the GameState class. The method modifies the two fighting units which are passed to it as arguments. Method setCombatants is then called with the two fighting units as arguments, so that the rendering code can know that two units have just fought, and which ones. This is followed by a call to repaint() so that the panel is redrawn and the combat animation can be shown. There are then two checks to the units’ health. If a unit has less than 1 health, i.e. it is dead, then it is removed from the board and its opposing unit is rewarded with 1 veterancy point, provided that its opposing unit still exists and has less than the maximum of 6 veterancy. If a unit dies, the opposing player is also granted 1 victory point here, regardless of whether or not their own unit also died. The combat sound effect method is then called, with the attacking unit as the argument, so that the appropriate corresponding sound effect can be played.

The third case is that an adjacent friendly unit is clicked:

} **else** { *//If a friendly adjacent unit is clicked, a swap will take place*

Like with the movement code, the selected unit’s old location is temporarily stored. The unit which was clicked (not the selected one) is also copied into a temporary Unit object using the Unit class’ copy constructor. The clicked Unit’s location is set to the selected Unit, and then the selected unit’s location is set to the clicked Unit. Territories don’t need to be updated because no territories would ever be changing hands in the event of a swap move.

The deselect() method is then called to make sure that after the click, whether it’s a move, an attack or a swap, the unit is deselected to prepare for the next turn. The alpha of the combat animation is then reset to 1.0 (100%) so that it appears, and then the turn counter is iterated by 1. Each of the two users’ victory points are calculated as the total of the user’s victory points from unit kills and their victory points from controlled territory. If the turn count has reached 100, these two totals are compared, and the player with the higher total wins. The alpha for the main graphics (the Your Turn message and the combat animation) is then also reset to 1.0, in preparation for the next turn. Once these processes have been completed, the move is submitted to the server via the ClientThread, and the last player is set to the client’s faction because they just moved. The selected unit copy is reset to null, and the new board is saved to GameState.

The mouseMoved method handles the user’s mouse movements to provide them with tooltips when they hover over units. This is a useful way to give information such as unit names and specific health numbers without having to over-fill the GUI itself. When the user moves the mouse, the mouse’s grid coordinates are calculated in the same way as they are when the user clicks; by dividing the pixel coordinates by the pixel size of the grid squares. Then, if the user is hovering over a non-null square (one with a unit in it) and that unit is ‘nearby’ (within sight range of their own units), a corresponding tooltip will appear. The font is set to Microsoft’s Bahnschrift in bold with size 12, and the tooltip’s text is set to display the unit’s name, health, and veterancy. This is done by passing HTML code as the string argument to the setToolTipText() method, because otherwise the tooltip would not allow line breaks to be manually added. Using HTML here allows for line breaks to be added where aesthetically necessary, so that the tooltip can be compact and clear in its presentation of information, rather than spanning one long line.

component.setFont(**new** Font(**"BahnSchrift"**, Font.***BOLD***, 12));

String fontFamily = component.getFont().getFamily();

component.setToolTipText(**"<html><body style=\"font-family:"** + fontFamily + **"\"<b>"** + *gameGrid*.get(x).get(y).getName() + **"<br>"** + **"Health: "** + *gameGrid*.get(x).get(y).getHealth() + **"<br>"** + **"Veterancy: "** + *gameGrid*.get(x).get(y).getVet() + **"</b></html>"**);

The font of the text is set by the fontFamily part of the argument. If the user is hovering over an empty area or the unit isn’t nearby, the tooltip text is set to null, which means the tooltip won’t appear.

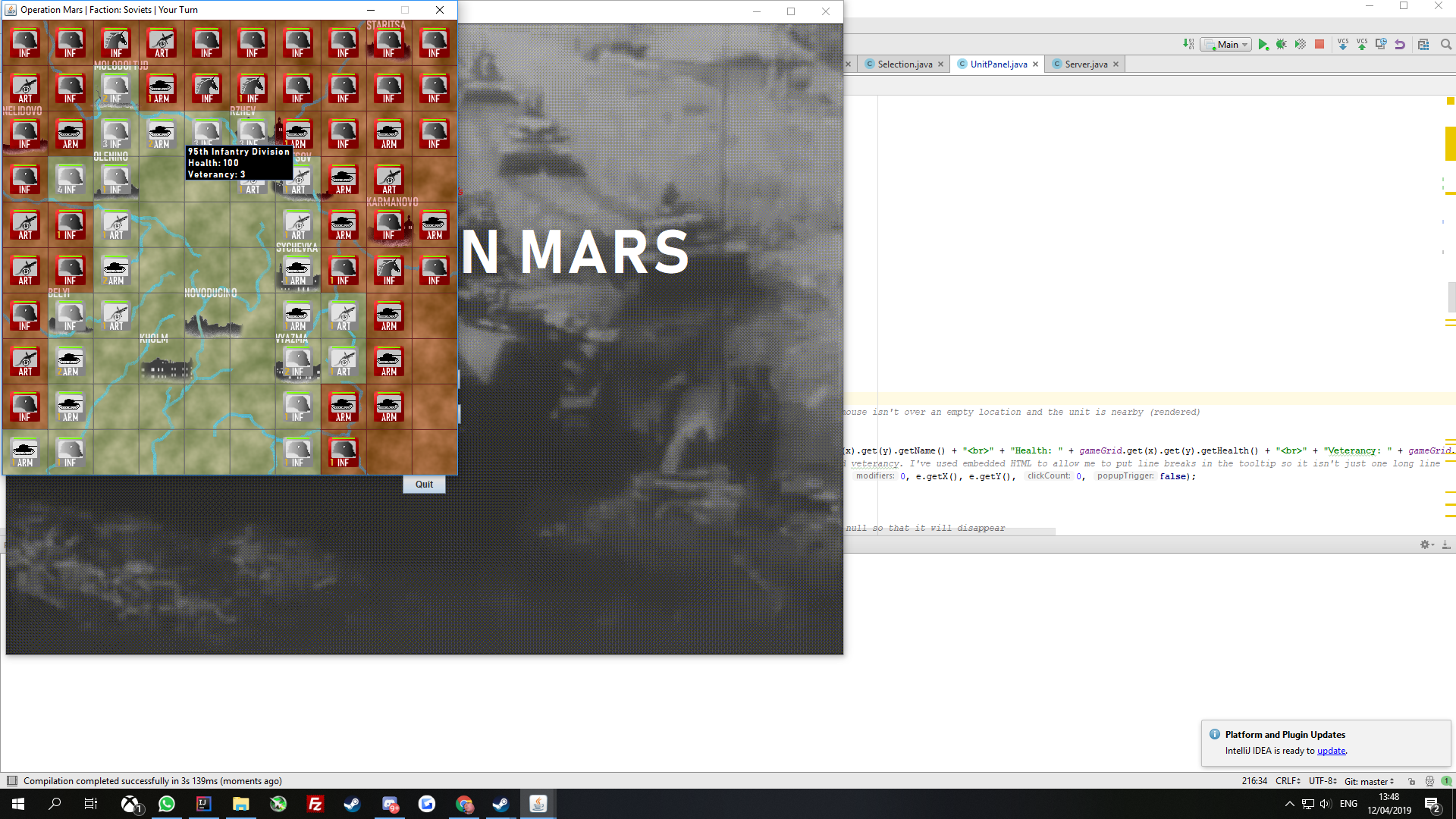


Fig4 *Infantry Unit with mouse hovered over*

#### Background

The Background class is one of the 4 panel classes whose instances make up the MainPanel instance. Like the others, it extends the JPanel class. The class has 5 variables; *w*, *h*, *size*, *image*, and *fileName*. The ints *w* and *h* tell the panel which size the game grid’s width and height are in terms of tiles; in this case they are both 10. *size* is the pixel size of each grid square, which is 60. The BufferedImage *image* will hold the png image which is used as the background of the game, i.e. the map. The String *fileName* tells the Background object the file path of the png file.

The class only has two methods; the first is paintComponent, which loads the png file into *image* and draws *image* from the top-left corner of the panel (0,0).

image = ImageIO.*read*(getClass().getResource(**"/media/"**+**this**.**fileName**));

g.drawImage(*image*, 0, 0, **null**);

The other method in the class is getPreferredSize(), which works the same as in the other panels; it uses the square size and the number of squares to determine the size of the panel. This size should come out the same as the other panels in mainPanel. The height is simply *h*\**size*, and the width is *w*\**size*.

#### GridPanel

GridPanel is another of the 4 panels making up mainPanel, and is approximately as simple as Background, and also extends the JPanel class. The class has 3 variables; *w*, *h*, and *size*. The ints *w* and *h* tell the panel which size the game grid’s width and height are in terms of tiles; in this case they are both 10. *size* is the pixel size of each grid square, which is 60. The paintComponent method performs an embedded *for* loop to cover each of the 100 grid squares, and for each coordinate it draws a rectangle at that grid square’s coordinates, using size as the height and width of the rectangles. This simply adds a line grid to the game to denote the locations of the squares.

The other method in the class is getPreferredSize(), which works the same as in the other panels; it uses the square size and the number of squares to determine the size of the panel. This size should come out the same as the other panels in mainPanel. The height is simply *h*\**size*, and the width is *w*\**size*.

#### TerritoriesPanel

TerritoriesPanel is the last of the 4 panels making up mainPanel, and is about as simple as the others, and also extends the JPanel class. The class has 5 variables; *colors*, *territories*, *w*, *h*, and *size*. The ints *w* and *h* tell the panel which size the game grid’s width and height are in terms of tiles; in this case they are both 10. *size* is the pixel size of each grid square, which is 60. The array colors holds two colours, which are essentially red and grey with an alpha value to make them transparent. The alpha value is set to 80 and can range up to 255, which would be completely opaque, or down to 0, which would mean totally transparent. In this case we want the territory colours to be apparent without blocking the layers beneath such as the map (background). The class’ paintComponent method does an embedded *for* loop through the 10x10 territories array, and draws a corresponding rectangle in the appropriate location, with the colour determined by whether that spot in the 2D *territories* array is a 0 or a 1; 0 referring to the index location of the red colour in the *colors* array, and 1 referring to the grey colour, i.e. Soviets are 0 and Germans are 1.

**for** (**int** i = 0; i < **w**; i++)

{

**for** (**int** j = 0; j < **h**; j++)

{

g.setColor(*colors*[**territories**[i][j]]);

g.fill3DRect(i \* *size*, j \* *size*, *size*, *size*, **true**);

}

}

The class also has a ‘get’ method and a ‘set’ method for the territories grid, called *getTerritories* and *updateTerritories* respectively. *getTerritories* simply returns the territories array, while *updateTerritories* uses the Arrays.copyOf() method to copy the new *territories* array - called *owners* - into *territories*.

**public void** updateTerritories(**int**[][] owners){

**territories** = Arrays.*copyOf*(owners, owners.**length**);

}

**public int**[][] getTerritories(){

**return territories**;

}

#### GameState

GameState is a class which manages the state of the game, including storing important values like the game board, and also centrally handling combat. Its class variables are *width*, *height*, *size*, *gameBoard*, *faction*, *victoryPoints*, *enemyVictoryPoints*, *turnCount*, *territoryVictoryPoints*, *enemyTerritoryVictoryPoints*, *serverHosted*, *serverJoined*, and *someoneHasMoved*. The ints width, height and size determine the grid width and height, and individual square pixel size, respectively. gameBoard is a 2D ArrayList which stores the board of Unit objects. The int faction stores which faction the player belongs to; 1 or 2, meaning Soviets or Germans. The four ints *victoryPoints*, *enemyVictoryPoints*, *territoryVictoryPoints*, and *enemyTerritoryVictoryPoints* hold the current values for how many points each player has from kills (*victoryPoints* and *enemyVictoryPoints*) and how many points each player has from held territories (*territoryVictoryPoints* and *enemyTerritoryVictoryPoints*). The Boolean values *serverHosted*, *serverJoined*, and *someoneHasMoved* allow the client to know what’s happened so far in the game; if the user has hosted or joined a server, they can’t host or join again. If someone has already moved, the server won’t accept any new connections to avoid conflicting information.

The getNewBoard() method creates the initial board of units to be used by all of the clients. It first uses an embedded for loop to fill an ArrayList with 10 ArrayLists, each with a capacity of 10 Unit objects. This forms a 2D 10x10 grid. Each of the 10 ArrayLists is then filled with 10 null values, which fills the whole grid, providing a blank slate for the units to be placed in their historical locations, so that the null places don’t have to be explicitly set one-by-one to null.

*gameBoard* = **new** ArrayList<ArrayList<Unit>>(10);

**for** (**int** i = 0; i < GameState.***width***; i++)

{

*gameBoard*.add(**new** ArrayList<Unit>(10));

**for** (**int** j = 0; j < GameState.***height***; j++)

{

*gameBoard*.get(i).add(**null**);

}

}

The method then runs a *Try* block which contains a one-by-one definition of each unit as it should be in its historical location, with its own name, type, veterancy value, etc. There are 47 units defined and placed on the board for the Soviet player, and 26 for the German player. Below is an example of one of the unit placements.

Unit s1stMC = **new** Unit(**"1st Mechanised Corps"**, **"sovietinfcounter"**, 0, 8, **"inf"**, 1, 0, **false**);

*gameBoard*.get(0).set(8, s1stMC);

First, the unit is given an object name, and then is given its historical name to be displayed to the user, which image file it should use as its icon, which coordinates it’s at, which type of unit it is, which faction it belongs to (1 for Soviets, 2 for Germans), how much veterancy it has, and whether or not it is selected (this is false for all units at the start). Then, the unit is placed in its corresponding historical location in *gameBoard*. The icon image couldn’t just be inferred from the unit type, because some infantry units have a different icon - a horse’s head - for historical flavour, because they were cavalry divisions and are named as such. This also allows for future expandability, with different units possibly having unique or more varied symbols. Once the Try block has completed, the completed initialised *gameBoard* with all of the units in their historical locationsis returned.

Following this method are simple *get* and *set* methods for 12 different variables. They all function in the same way, so here is one example:

**static void** setFaction(**int** newFaction){ *faction* = newFaction;}

**static int** getFaction(){**return** *faction*;}

The set method receives an argument and assigns its value to the corresponding variable. The get method just returns the corresponding variable.

The getTerritoryVictoryPoints and getEnemyTerritoryVictoryPoints methods are more complex than the other get methods, and involve a decent amount of calculation. The method getTerritoryVictoryPoints calculates how many victory points the player is currently gaining from towns that they hold, and getEnemyTerritoryVictoryPoints calculates the same but for the opponent. getTerritoryVictoryPoints sets the total to 0 to begin with, and then for each specific location where each town resides, if the player controls that location, 5 is added to the total. In the case of two locations - Rzhev and Vyazma, at positions (5,2) and (6,7), 10 is added rather than 5, because these are key towns and controlling them is more important than controlling peripheral villages.

**if** (territories[2][1] == GameState.*getFaction*()-1) {*territoryVictoryPoints* += 5;}

This check is performed for each town, and then the total is returned. getEnemyTerritoryVictoryPoints works in the same way, but calculates the points for the opponent instead. These could be merged into one method with *else* statements, but being separate methods allows for them to be called separately without one of them having to take slightly longer to get through the *if* statements to get to the *else* statements. It also makes for easier returning of the values, and makes the code more obvious in function.

The void combat method simulates combat between two units, and applies the resulting changes to their health. The two Unit objects are passed to the method, along with the territories board, which allows for encirclements to be taken into account. The int value standardDeviation, set to 8, determines how predictable the combat results will be; a lower value means more predictably-spread results. The combat calculation uses a Gaussian curve to determine each unit’s damage received, and 8 is the value of 1 standard deviation away from that curve’s mean average value. The typeBonus value determines how much of an impact having an appropriately matched unit - such as artillery versus their armour - will have on the enemy’s damage received. This is added to the mean damage, so that the average damage the armour would take in that case would be 35 higher. The townDefenceBonus value determines the usefulness of defending in a town. It is set to 15, so the attacker would have 15 added to their final damage number, and the defender would have 15 subtracted from theirs. The vetBonus value is also added/subtracted from the units’ damage at the end, and determines how much extra damage a unit takes and how much less it receives per level of veterancy it has. This is set to 5, so if a unit has 2 veterancy, it will deal 10 more damage and take 10 less. The encirclementPenalty represents the cost of being surrounded by enemy territory (diagonal territory not taken into account). The encircled unit will take this much more damage, and the other unit will have half of this value subtracted from its damage taken. It is set to 35, so an encircled unit who’d otherwise taken 50 damage would take 85 damage, and the opposing unit, if it had otherwise taken 50 damage, would take 34 damage (50-16).

The unit type penalty is determined by a series of if statements describing each of the 6 possible match-ups where a bonus would apply. An example is below.

**if** (attacker.getType().equals(**"inf"**) && defender.getType().equals(**"arm"**)){attackerMean+=typeBonus;} *//Infantry performs worse against armour*

As is apparent, when attacking infantry faces defending armour, the type bonus is added to the mean of the attacking infantry’s damage taken. Once this new mean is calculated, the random damage values are calculated for both units.

attackerLosses = (**int**)(random.nextGaussian()\*standardDeviation+attackerMean);

defenderLosses = (**int**)(random.nextGaussian()\*standardDeviation+defenderMean);

The veterancy bonuses are then applied as such:

attackerLosses -= attacker.getVet()\*vetBonus;

defenderLosses += attacker.getVet()\*vetBonus;

attackerLosses += defender.getVet()\*vetBonus;

defenderLosses -= defender.getVet()\*vetBonus;

Each unit’s veterancy bonus is added to their opponent’s damage, and subtracted from their own. The town defence bonus is then applied, wherein the defender’s location is checked against the locations of the game’s 12 towns:

**if** (((defender.getxLocation() == 2)&&(defender.getyLocation()==1))

|| ((defender.getxLocation() == 0)&&(defender.getyLocation()==2))

…

If there is a match, the town defence bonus is subtracted from the defender’s damage taken, and added to that of the attacker.

defenderLosses-=townDefenceBonus;

attackerLosses+=townDefenceBonus;

Four *if* statements then check whether the attacker is encircled, and then four others check the same for the defender. This is done by adding 1 to or subtracting 1 from each coordinate to check all 4 adjacent territories. If any of them is friendly, the unit isn’t encircled, but if none of them are, then the encircled value remains true. The encirclement penalty is then added to the damage taken of the unit which is encircled, and half of it is subtracted from the damage taken of the other unit.

*//If attacker is encircled, make them take more damage and deal less*

**if** (attackerEncircled){

attackerLosses+=encirclementPenalty;

defenderLosses-=encirclementPenalty/2;

}

The units are also made to take a minimum of 7 damage each turn, so that the game can’t be broken by extremely unlucky or lucky rolls, or a single high-veterancy unit having a snowball effect and wiping out 15 units.

**if** (attackerLosses<7){attackerLosses = 7;}

**if** (defenderLosses<7){defenderLosses = 7;}

Finally, the final changes to the units’ health are applied.

attacker.setHealth(attacker.getHealth()-attackerLosses);

defender.setHealth(defender.getHealth()-defenderLosses);

The getNewTerritories() method is similar to the getNewBoard() method in that it generates a new data structure for the game at the very start to set up the historical situation as it should be. In this case, it is a 2D array of ints rather than a 2D ArrayList of Units. As discussed earlier, a 0 in the territories array represents Soviet player control, and a 1 represents German player control. Since the Soviet player will hold most of the territory and the German player will only hold the central area, an embedded for loop first fills the 2D array with 0 values, and then a series of for loops fill certain lines of tiles with 1s to represent the German territory among the Soviet territory. For instance, the following loop fills the bottom (10th, i.e. index 9) row with 1s (German control) from index 0 (1st tile) to index 6 (7th tile). The remaining 3 tiles are left unaltered as 0s (Soviet control). This is repeated as necessary for all of the German territory.

**for** (**int** i = 0; i<7; i++){

owners[i][9] = 1;

}



Fig5 *Bottom row of the game board, to show territory placement*

The updateBoard() void method updates the stored board from a new board by looping through each tile in the 2D ArrayList via an embedded *for* loop and copying the Unit object over if there is one:

**if**(*gameBoard*.get(i).get(j) != **null** && updatedBoard.get(i).get(j) != **null**) {

**try** {

*gameBoard*.get(i).set(j, **new** Unit(updatedBoard.get(i).get(j)));

} **catch** (IOException e1) {

e1.printStackTrace();

}

}

The isLegal() method takes a Unit object and the coordinates it wishes to move to, and determines whether or not that move is legal. Legality means that the coordinates aren’t the same as the Unit’s current coordinates, and the unit is moving to an adjacent location. Armour units are allowed to move diagonally, so for them a simple if statement checks that the unit isn’t moving more than 1 tile. For non-armour units, there is an extra check added, to see that the x and y coordinates aren’t both changing, which would mean a diagonal move; only one coordinate can change for a cardinal (non-diagonal) move. Below is the legality check for non-armour units. The check for armour units is the same, but without the last condition that only one coordinate is changing.

} **else if** (!selected.getType().equals(**"arm"**)

&& (!(Math.*abs*(selected.getxLocation() - x) > 1)

&& !(Math.*abs*(selected.getyLocation() - y) > 1))

&& !((Math.*abs*(selected.getxLocation() - x) > 0) && (Math.*abs*(selected.getyLocation() - y) > 0))){

**return true**;

}

The Boolean method isNear() takes a Unit and the *unitBoard* ArrayList, and checks if that given unit is adjacent to one of your (the player’s) units, including diagonal adjacency. Firstly, if the unit is one of yours, then *true* is automatically returned because your own units are inherently always near to your units. Then, all 8 adjacent locations are checked to see if there’s one of your units in one of them. If there is, then the unit is considered nearby because one of your units is there to see it, and *true* will be returned so that the unit is rendered. The if statements have conditions to make sure that none of the checks go outside of the board’s bounds and throw a NullPointerException. Below is an example of one of the 8 checks.

|| (i<9&&j>0&&(unitBoard.get(i+1).get(j-1)!=**null**)&&unitBoard.get(i+1).get(j-1).getFaction()==*faction*)

If none of these 8 checks are passed, the unit is considered not nearby, so *false* is returned.

#### Selection

The Selection class has only one method, and that method takes the *gameBoard* 2D ArrayList of Units from unitPanel (as indicated in Figure 1, p21) and returns the Unit object which is selected. It performs an embedded *for* loop to cycle through the ArrayList, and then performs an if statement to check that a Unit exists there and is selected. If it is, *true* is returned. If the whole loop completes without true being returned, *null* is returned.

#### ClientThread

The ClientThread class extends the class Thread and is used to communicate back and forth with the server on behalf of the client. It is initialised by the MainPanel, as shown in Figure 1 (p21). In its constructor, it first takes the three parameters passed to it - the territories panel, the unit panel, and the server’s IP, and assigns them to instance variables. Then, an int portNumber is assigned the value 8888 so that this port will be used for the connection. A high-numbered port was chosen so that it wouldn’t conflict with anything else. A new Gson object is then created for later serialisation and deserialisation in the communication with the server. A Socket object is created using the provided IP address and the port number, which creates a connection to the server at that IP’s location and on that socket. A BufferedReader and PrintWriter are created to read from the socket connection and write to it. A 2D ArrayList has a fresh board copied to it from GameState.getNewBoard(), and then this is saved as the current game board. The player’s assigned faction is received from the server and read via the BufferedReader, and saved. The Boolean serverJoined in GameState is also flagged as true to record that the client has already joined a server, so they can’t join again. Finally, the last moved player is set to 2 so that player 1 will always go first.

The ClientThread’s run() method is where the thread’s socket communication takes place from the server to the client. The method runs a *while* loop wherein it listens for lines of communication from the server. These are the int value of the player who just moved, and a Json string of the 2D game grid (ArrayList), serialised via Gson. The int is first listened for, and then it is saved as the last moved player. Then the board is listened for, and is deserialised via Gson, and then saved as the updated board. Then, if the client isn’t the one who just moved, his local grid and territories are prompted to refresh, and the other things that go with the start of one’s turn will be triggered from there.

The makeMove() method is where the client’s socket communication takes place from the client to the server. The method receives the client’s new board, and sends to the server the client’s player number, and the board, serialised via Gson:

**public static void** makeMove(ArrayList<ArrayList<Unit>> unitBoard){

*out*.println(GameState.*getFaction*());

Gson gson = **new** Gson();

*out*.println(gson.toJson(unitBoard));

}

#### Server

The Server class extends the Thread class and is used to accept and open socket connections with clients who try to join the server. The thread is run when the user clicks the Host Game button on the main menu. The thread creates a socket at port 8888, then waits indefinitely, and when a client connects to the server, it accepts the socket connection and adds it to an ArrayList of connections, so it can know how many players are connected. It then starts a new ServerThread to communicate with that client, and sends it the new client’s Socket object, and the ArrayList of all of the Socket objects.

#### ServerThread

The ServerThread class extends the Thread class and is used to communicate over the socket to the client on behalf of the server. Its constructor simply receives the parameters passed to it by Server, as indicated in Figure 1 (p21), and assigns them to instance variables. Like ClientThread, the run() method uses for communication a PrintWriter, an OutputStream, and a BufferedReader. Firstly, the thread prints the size of the ArrayList of socket connections to the output stream, so that the user joining can be told their player number based on the order in which they joined. Then, the last moved player is set to 20, so that the thread can know when it’s the first turn of the game. Then, the communication loop begins, and loops indefinitely. Within here, the thread uses the BufferedReader to receive communication from the client. Once it receives communication from the client, it can save the fact that someone has moved, so that it won’t accept any more socket connections after that. Then an if statement checks if the following communication should be relayed to the client. The communication is sent to the client in the event that either they just sent it themselves to the server, or it is the first turn, or either of the players has won the game, in which case both players need to be informed.

**if** (factionWhoMoved == lastMovedPlayer || lastMovedPlayer == 20 || factionWhoMoved == 11 || factionWhoMoved == 12) {

Within this statement, the server thread then cycles through every connection one by one and informs every other client on the server of the changes which have happened; these being the new board, and the player who updated the board, i.e. who made the move. This is done on the condition that the client isn’t the one who just moved, via the line: if (index != factionWhoMoved). After this is completed, the player who just moved is saved as the last moved player so that the process can repeat with the next received communication.

**for** (Socket socket : **sockets**) { *//For all of the connections*

index++;

**if** (index != factionWhoMoved) {

OutputStream outputStr = socket.getOutputStream();

PrintWriter printWriter = **new** PrintWriter(outputStr, **true**);

printWriter.println(factionWhoMoved); *//Tell the client who moved*

printWriter.println(playerBoard); *//Tell the client the new state of the board*

}

}

lastMovedPlayer = factionWhoMoved;

### Bugs and Testing

The primary method for testing in the games industry is called QA, also known as Quality Assurance. This primarily involves playing the game and trying to force bugs to happen so that they can be found and fixed - more than playing the game, this involves “breaking the game”, and seeing how and why something went wrong so that it can be fixed [90]. In the case of games and interactive media in general, they can rarely be tested by simply providing example values and getting the resultant outcomes. This is possible, but the example value must be an action on the part of the player, rather than a particular variable being provided to a particular method; the bug must be possible and recreatable in the context of playing the game.

The fact that the game must be played and that bugs must be heuristically produced doesn’t, however, mean that the testing can’t be formalised. The testing for Operation Mars typically set out with more of a goal than simply to play the game and to wait for bugs to appear. As J. Schreier puts it, games testing isn’t just a “saunter through level 5,” but rather “14 straight hours running into different walls to see if they’re solid.” This is the methodology from which much of the testing for Operation Mars came. Below is the methodology for testing for and fixing many of the program’s bugs. Some of these are from very early in development, such as simply getting the unit panel’s mouse listener to work at all.

|  |  |  |  |
| --- | --- | --- | --- |
| Test | Expected Outcome | Actual Outcome | Solution |
| Click on two units | The units’ coordinates are printed to the console | Nothing printed to the console | Fixed bug by creating new MainPanel object class rather than simply creating it within the MainFrame class. This allowed the focus to be allocated correctly. This also allows the classes to follow Clean Code principles by only doing one task each. [Fig6] |
| Call the updateBoard() method with the unit board as the argument | Board is successfully copied to the one stored in GameState | NullPointerException thrown by updateBoard() | Added an *if* statement to check that the ArrayList slot it’s over isn’t *null*, so it doesn’t try to copy *null* values |
| Use Gson call to try to clone Unit object | Unit object is cloned | Serialisation throws Exception | Stopped using Gson to clone, instead just created a copy constructor within the Unit class. This is also much faster than Gson cloning. |
| Click unit, then click another square | Unit moves to new square and isn’t in the old one | The unit appears in the new tile but also lingers in the old one | Switched to ArrayLists rather than Arrays to allow removal of the old Unit and insertion of a null value in its place. Also had to re-configure the rendering of the board so that it redrew after each unit move. |
| Un-comment copyBoard() method and attempt to click a unit from one square to another | Board should be loaded from the one saved in GameState | ArrayIndexOutOfBounds exception is thrown in the copyBoard method. A println() reveals that the size of the array is only 9 rather than 10 | Issue solved by adding null value to same index after removing value from that index, since removal alone reduces the size of the Arraylist by 1. |
| Select a unit and then click another to make them fight, then hover over them and see their health. Temporarily make combat method simply remove a fixed 35 health for one unit, and 40 health for the other. Hover over the units and check the Health | Tooltips should have the new health values of 65 and 60 | Tooltips show 100 health for both units | Had to first fix another bug wherein the units’ changed health wasn’t persisting past the Listener |
| Select a unit and then click another to make them fight, then hover over them and see their health. Temporarily make combat method simply remove a fixed 35 health for one unit, and 40 health for the other. Have a println statement print the two units’ health values | The printed health values are 65 and 60 | The printed health values are both 100 rather than 65 and 60 | Copy constructor in the Unit object was setting the health to 100 rather than to the health of the unit being copied, so each time the board updated everyone's health was being reset. Copy constructor altered so that it copies the new health rather than setting it to 100 |
| Make a defending unit die by attacking it until it runs out of health | Defending unit disappears | The icon disappears but the game considers the unit still present; it’s still in the ArrayList and the tooltip still appears, showing 0 health. A println also shows that it is still there, just not being rendered. | Removed 'defender' object and just replaced it with a reference to the location on which the attacker clicked; this means that the defender's location won't have an extra 'ghost' object hanging around after it's died. |
| Make an attacking unit die by running it into several different enemy units until it runs out of health | Attacking unit disappears | The icon disappears but the game considers the unit still present; it’s still in the ArrayList and the tooltip still appears, showing 0 health. A println shows that it is still there, just not being rendered. | Fixed bug by referring to the ArrayList slot of the selected unit's location when deciding to remove the unit rather than referring to the copied selected unit itself |
| Call the makeMove() method with a game board and attempt to transfer the game board over the network using Gson serialisation | Game board is serialised, transferred, and de-serialised | Gson implementation reports that multiple fields called *maxX* have been declared, which throws and exception and prevents selection and movement | Solved by making the *icon* for the Unit object transient, since a BufferedImage isn't serialisable so it was causing issues when Gson attempted to serialise the Unit object |
| Try to play several turns of the game | Turns proceed one after the other, swapping between the two players | Turn-taking only works for the first two turns | Consistent turn tracking added to the clients so that they pass the turn to the other user when they make a move |
| Launch game and wait a few minutes for the music track to completely finish | Music track loops to the start when it finishes | Music track stops to silence when it finishes | Set the music player’s cycle count to “INDEFINITE” so that it loops indefinitely |
| Select a unit and click an adjacent friendly unit to swap them | Units swap as intended | Erroneous territorial changes (there should be no territorial changes), and the units’ icons swap but their names don’t | Replaced the *remove* and *add* method calls to the ArrayList with *set* method calls to the ArrayList |
| Set up a combat scenario wherein both units die at the same time | Both units disappear | NullPointerException thrown | Swapped deletion and updating of veterancy so that the program isn’t trying to add veterancy to a non-existent Unit, and this fixed the bug, but introduced a new one |
| Set up a combat scenario wherein both units die at the same time | Both units disappear | An invisible version of the defending unit with negative health remains when both attacker and defender die at the same time in combat | Added an *if* statement to check that the unit being modified is not null |
| Click unit and then very quickly click another non-adjacent friendly unit | Selection switches to the new unit, no Exceptions thrown | Selection switches to the new unit, no Exceptions thrown | Not needed |
| Un-comment ‘YOUR TURN’ message code and run the game between two clients | ‘YOUR TURN’ message appears when it becomes a player’s turn | The message and the explosion effect don't appear until it's a player's second turn | Fixed by making a separate alpha value for the combat explosion effect so that it doesn't interfere with the Your Turn message. Also had to manually set the last moved player to player 2 on the first turn so that both clients would know that player 1 must go first. |
| Change game turn limit to 15 rather than 100 so game can be quickly completed, manually count players’ points, and complete a match | Game correctly assigns winner to Player 2 | Game correctly assigns winner to Player 2 | Not needed |
| Change game turn limit to 15 rather than 100 so game can be quickly completed, and complete a match | Game ends and ‘YOU WIN’ and ‘YOU LOSE’ messages are displayed appropriately | ‘YOU WIN’ message shows for winner but ‘YOU LOSE’ message fails to appear on the loser’s client | Fixed by removing fading effect for ‘YOU WIN’ and ‘YOU LOSE’ messages - not necessary, since the final message should stay up anyway |
| Select unit and try to move it to various non-adjacent locations | Nothing happens | Nothing happens | Not needed |
| Select a few non-armour units and try to move them diagonally adjacently | Nothing happens | Nothing happens | Not needed |
| Set up a combat scenario wherein both units die at the same time | Both units’ ‘line-of-sight’ is removed from their respective clients’ displays | Both units’ ‘line-of-sight’ is removed from their respective clients’ displays | Not needed |

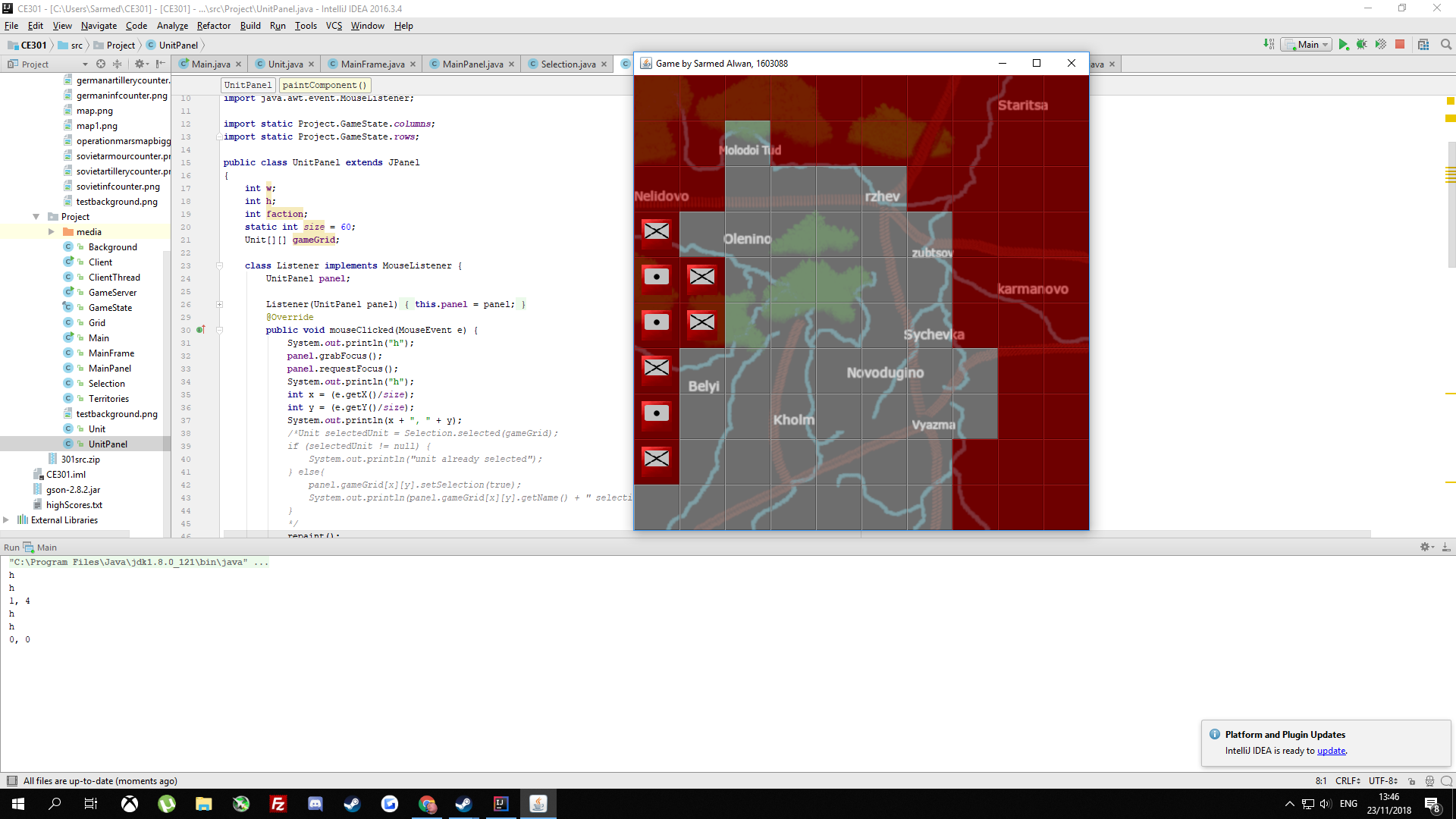


Fig6 *Units’ grid locations successfully printed to the console when the units are clicked*

## 

## 

## 

## 

## 

## 

## 

## 

## 

## User Manual

### System Requirements

#### Minimum Specifications

CPU: Intel® Core™2 Duo Processor E8400 or AMD® Athlon X2 6000+ Processor or Equivalent

RAM: 2GB

GPU: Intel® HD Graphics 2000 or Equivalent

Operating System: Microsoft® Windows® 7

Storage: 200MB

Java SE Runtime Environment 8

#### Recommended Specifications

CPU: Intel® Core™ i5-2400 or AMD® FX-4300 or Equivalent

RAM: 4GB

GPU: Intel® HD Graphics 2000 or Equivalent

Operating System: Microsoft® Windows® 10

Storage: 200MB

Java SE Runtime Environment 8

### Getting Started

#### Getting Online

If both players wish to play on the same machine, you do not need anything other than the game itself. If you wish to play Operation Mars over a remote internet connection, the recommended way is to download Virtual Private Network software such as LogMeIn Hamachi, and host your own Virtual Private Network. LogMeIn Hamachi can be downloaded for free at the following location:

<https://www.vpn.net/>

LogMeIn Hamachi is the property of LogMeIn Inc. and is not affiliated with Operation Mars. You may use another similar program if you know how to use it for the same purpose.

Create an account and begin the application. Click Network -> Create a New Network, and then type in the Network Name and Password which you would like to use. Do not use sensitive information here, as you will be sharing this information with whoever you wish to join your game server.

Your friend should click Network -> Join an Existing Network

They should then type in the network name and password which you chose, and click Join. Once the connection has been established, the IP to be used for the game should be visible (may be labelled as IPv4).

The game is distributed as a .jar file called Operation Mars.jar. To play the game, run the file by double-clicking it or right-clicking it and clicking Open. One of you should click Host Game, and then close the message which comes up. If you wish to both play on the same machine, you can open the file twice to open two instances of the game, and as usual only one of them should host. Once one of the players has hosted, the other should click Join Game, and enter the IPv4 address given by Hamachi, and then click OK. If you are both playing on the same machine, you don’t need to enter anything for the IP address, and can simply press Enter or OK to join the local game. Do not close the Main Menu window, or the game will close. If you do not wish to see it, simply minimise it.

The player who wishes to play as the Soviets should join the hosted game first, and the player who wishes to play as the Germans should join second. The Soviet player will play first. Once both players are in, the game can begin. More people can join the game, and they will be treated as spectators.

#### How to Play

The game ends after 100 turns. The player with the highest score at the end of these 100 turns wins. The Soviet player’s units are represented in red, and the German player’s units are represented in grey. Below is a figure of an example unit counter.

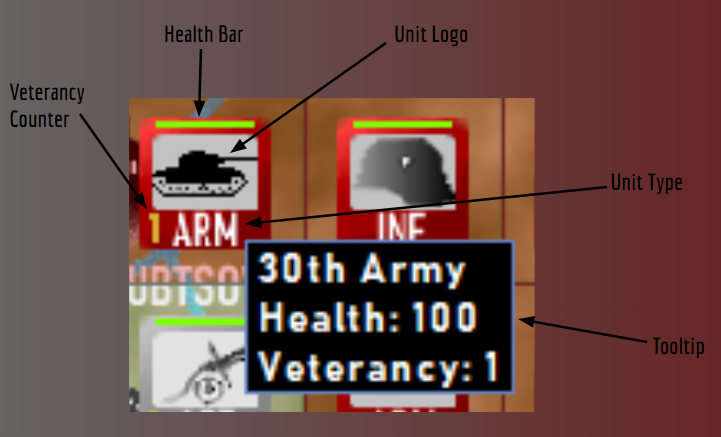


Fig1 *Diagram describing composition of a unit*

You can only see units which are yours, or are next to yours. The bar at the top of the window will remind you which faction you control and whether or not it’s your turn.



Fig2 *Title bar of the window*

To select a unit, when it is your turn, simply left-click on it. A white highlight will appear around it to show which unit you have selected. To switch which unit you have selected, simply click another of your units, unless it is adjacent, since that would be a swap command. To move your unit, click an adjacent empty tile. For Armour units (ARM), this can be diagonally adjacent. If the tile you move into has an allied unit, the two units will swap places. If the tile belongs to the enemy but is empty, it will swap to your control (represented by the red (Soviet) or grey (German) colouring of the tiles) If the tile you move into has an enemy unit, combat will be initiated. Armour units have a combat advantage against Infantry, Infantry have a combat advantage against Artillery, and Artillery has a combat advantage against Armour. Use this system to your advantage. For example, attack their infantry with your artillery. Defending units have a slight advantage in combat, but it is only slight. Units defending in towns have a large advantage. Units which are encircled (surrounded by adjacent enemy territory, not counting diagonal territories) will have a large combat advantage. You should aim when possible to encircle your enemy’s units to reduce your casualties, especially if they are defending a town. If you simply throw your units into the enemy to die, the enemy’s Veterancy will rise, which gives them further combat bonuses and makes them even more difficult to dislodge. If your unit kills an enemy unit, your unit’s veterancy will rise by 1.



Fig3 *Selected armour unit*



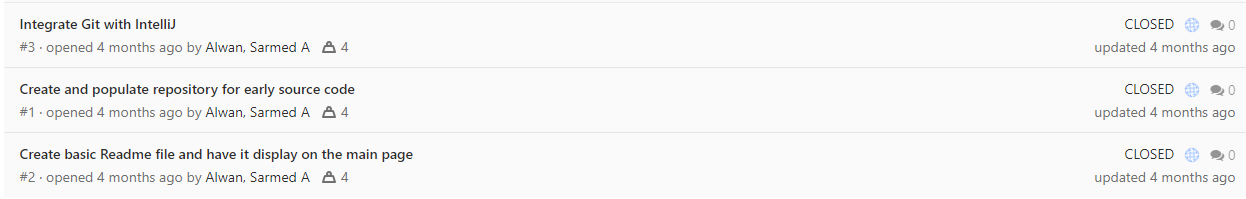
Fig4 *A town on the game map*

To defeat your opponent, you must wrest control of the towns on the map, represented on the map with their corresponding names, as shown in Figure 4. All towns on the map are worth 5 Victory Points each to the player who holds them, except for Rzhev and Vyazma, which are worth 10 Victory Points each. The German player starts with more of the towns, and thus more points, but they are outnumbered almost 2:1. This means that the Soviet player must attack the German player and try to gain enough control of the map before 100 turns pass and the operation is considered a failure. The German player must utilise their defensive position to try to defend long enough to win.The German player should still consider counter-attacks where necessary, as they can be advantageous, especially with the correct unit types matched up. Killing an enemy unit also gives you 1 Victory Point. When 100 turns have passed, the player with the higher total score will be declared the winner.

# Project Planning

## Overview

The planning and management for this project were primarily done through the online Jira software in-browser. This allowed smooth management of the project in an Agile approach, with adaptability and clear division of project issues built in to the software to be utilised. Initially, some limited representation of issues via the CSEE GitLab repository was also done:

Fig1 *GitLab issues list*

However, it was decided that this was redundant because the same issues were already being covered on Jira. The main tool for planning was the Kanban board in the Jira software, which enabled a clear flow of an issue from its creation in the To Do column, to In Progress, to Done, as shown in Figure 2.

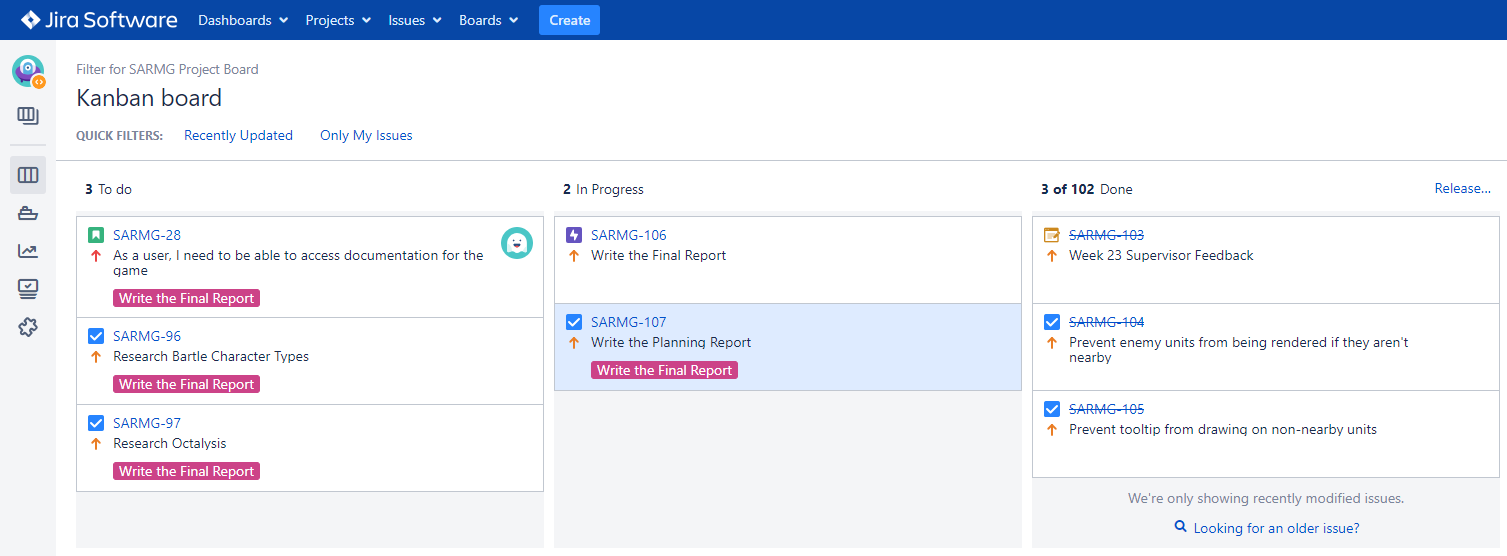


Fig2  *Jira Kanban Board*

It would have been possible to add more columns, but this is the setup that was arrived at. It would have been further redundant to make too many categories for issues to exist in, and the planning and management would have been unnecessarily complicated by it. This way, it was clear at which stage in its life an issue was, and the movement of issues was more consistent and easier to keep track of.

## Maintaining Momentum, Adapting to Change, and Dealing with Risks

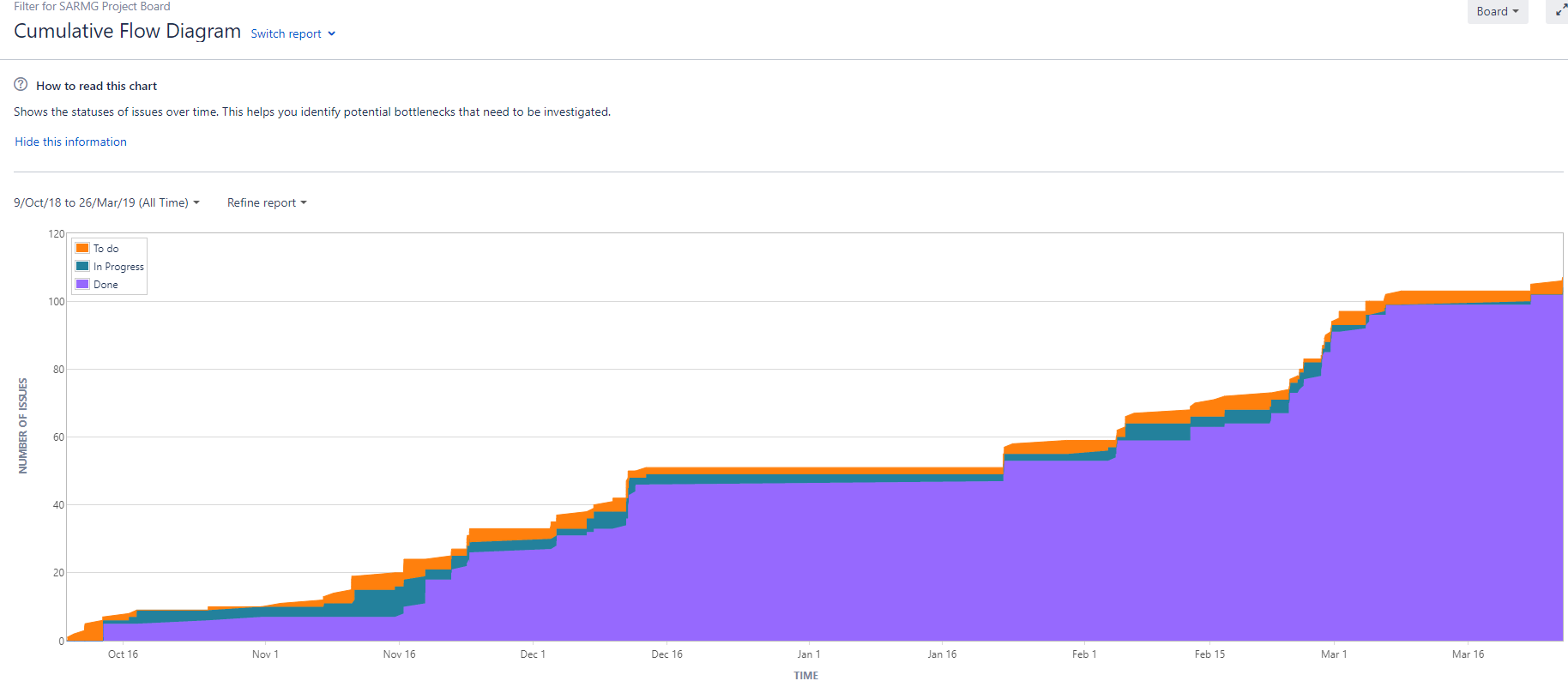


Fig3 *Jira Cumulative Flow Diagram of Issues*

Figure 3 above represents the motion of the project throughout the first and second halves of development, and towards completion. To Do tasks are represented in orange, In Progress in blue, and Done in purple. In total, slightly over 100 issues made up the project. The momentum of the project slowly picked up during the beginning stages, where it is apparent from the graph that it took time to get accustomed to the Jira system. Initially, many issues resided in the In Progress list, and the To Do list sat relatively unpopulated. This was a less accurate way to portray the motion of the project, as typically only a couple of issues are actually in progress. It also lent less clarity to the goals of the project, as the To Do list is best used to give a look at the near horizon. As can be seen slightly later down the line, the To Do list was better utilised to this end, but the In Progress list was still over-inflated. This isn’t ideal, though since it was still relatively near to the start of the project it is understandable that many smaller tasks are being done simultaneously to get the project moving.

Early on in the project, it was decided that the game should be a competitive game rather than a cooperative one, and so there was a change in the focus of the project, and tasks had to be appropriately adapted. This was done on the fly, without progress being significantly hampered.

There was a steady growth of completed tasks throughout November and December as the project found its feet, with more balanced and consistent progress leading up to the Minimum Viable Product and the first oral examination. The presence of the deadline kept the momentum high in this period, and so did introducing structured versioning as shown in Figure 4:

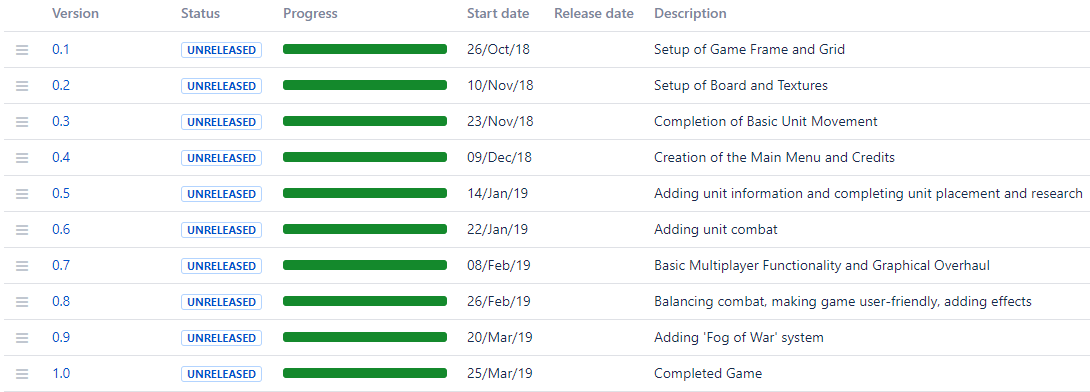


Fig4  *Jira Versioning List*

As can be seen, all versions were kept as ‘unreleased’ despite meeting full progress. This was simply a practical decision so that the versions could always be seen together and be altered if needed. As is also apparent, there were about 1-2 versions per month. Generally, each version was completed before the next was started, to keep coherence.

## Achievements and Performance

The project met all of its objectives as stated in the initial report, which were:

* Create a turn-based multiplayer strategy game, playable by two people on two clients
* Set in the real-life scenario of Operation Mars, i.e. the Battle of Rzhev
* Three unit types with rock-paper-scissors-style system
* One player has significantly fewer forces than the other, and must defend adequately to win
* The game runs on a grid, and works on a client-server model which handles players and moves
* Random element to the combat to add more replayability and unpredictability

The project also introduced extra features to improve the quality of the game beyond the original specification, such as:

* The ability for another person to spectate the game
* User-friendly menu system
* Fog-of-war system wherein players can only see nearby units
* Sound effects
* Specially-commissioned music

These extra features lend to reinforcing the sensation of the setting and to reinforce the unifying themes, which experts say is an important way to engage the player in a game [91]. The overall performance in this project was as good as had been hoped, with all planned objectives being met and even more being added and completed. There was a lull in work output in between the two main periods of progress, during the winter holiday, and this could have been shortened to get the basic project completed sooner, which would have allowed for even further extra features. However, all objectives were still delivered on time, and to a good level of quality. Initially, the Jira system seemed like a burden, and it was used only as a necessity. However, as the project progressed, it became an important backbone for the workflow, and it was not only kept updated for record-keeping, but also referred back to for guidance and a planning aid. This can be seen by the more steady and consistent usage of Jira updates in the graph above, as opposed to early on in the project when updates were all performed in batches to avoid using the system as much.

## Reflection

Overall, the performance of this project has been very good, as all that was hoped for from the outset has been fulfilled, and some more. The approach to planning and working proved suitable; it was straightforward enough that it could be adapted to, and it was adaptable enough that it could fit a changing and personal style of working. For example, it was decided that in general it is to do 2-3 related tasks in tandem rather than one at a time, and the Agile approach integrated with Jira allowed for and encouraged this style. This would involve something like doing the combat calculation task alongside fixing a bug with the call to the combat method, as they are often related and it helps to combine them as needed while keeping them separate. It it also beneficial to constantly-available working prototype such as the methodology of Scrum or Extreme Programming [92] put forward, and this Agile approach via Kanban planning allowed for this while granting clarity of objectives and flexibility, something which a Waterfall approach, for example, wouldn’t offer [93].

A good deal of skills in programming were gained, such as a much more confident grasp of Object-Oriented Programming - for example a first-time usage of copy constructors, or of using a thread’s constructor separately from its running method so that its execution is timed correctly. In terms of planning, the usage of Jira was a major source of proficiency growth. Prior to this project, with little experience in truly utilising planning and related software, projects plodded along at an inconsistent pace, and the lack of recording changes and progress had hindered past projects: bugs weren’t recorded and described to be learned from, versions weren’t cleanly separated into themes to help with clean programming, and issues weren’t typically brought up in a manageable fashion and an appropriate order of importance. The previously unencountered value of grouping issues into Stories, Sub-tasks and Epics also helped, since it was made clear which work needed to go into solving a particular problem. Having a space to cleanly brainstorm and have a clear idea of what needs to be done slowly became a natural part of the work process, and the hindrances in past projects, due to lack of such planning, gradually fell away, and the project was able to progress predictably and take a form that was more than satisfactory.

# Conclusion

At the beginning of this project, the goal was to produce a turn-based multiplayer game. This was expanded in the Initial Report to include that it should be a tile-based strategy game, and should be playable online by two players. It was also further decided that it should be set during Operation Mars, and should be asymmetrical, with a rock-paper-scissors-style system for combat between the units. The execution of each of these requirements has not only been a success, but further additions have been made to the product, such as a fog-of-war system, spectation, sound effects and music, and an encirclement system. What has been produced is a highly expandable product, where further factors can easily be added to the game without reconstruction of the code’s structure, and which is modular enough that an asset swap and a re-definition of the starting conditions could produce an entirely different-feeling game within the same framework. For instance, within this framework, one could produce a game set around Ancient Greek combat, or even produce a game without a war theme, simply by replacing the unit icons, the map, and the sound effects. A sports game, for instance, such as Rugby, could be produced within only a matter of hours, with a quick asset swap and a quick rewriting of the ‘new board’ definition within the code. New maps and scenarios could even be added within the existing context of WW2, simply by changing the background map and the starting positions of the units and territories.

Possible improvements to the existing game could include making a separate panel on the side to display the information which is currently shown in the title bar of the game window, or, on that note, the ability to resize the game as desired or play it in full-screen. Further quality-of-life improvements could also be made, such as an in-game chat system so that clients such as Discord aren’t necessary, or the saving of previously used IP addresses so that they don’t have to be entered each time. Adding further unit types such as Cavalry was considered during development, but it was decided that this would compromise the original vision of the rock-paper-scissors dynamic. Further animations could also be added for unit movement, wherein the unit slides from one tile to the other, for example, which could add a further feeling of quality to the game, and could definitely be worth the extra programming time.

The work undertaken in this project, according to the Jira records, has been approximately 672 hours, though a more reasonable estimate compensating for inaccuracies in the logging is about 500 hours, averaging about 17 hours per week over a 6-7 month period. During this time, all original objectives have been met, and every objective has been completed. The product could be made unrecognisably better within only 50 more hours, though that is the nature of the speed of progress of a project. With only bonuses to bring to the table of ideas and objectives, the project to produce Operation Mars has been an overall success and good experience, and brings a product with which can be done a lot in the future, and from whose creation has been learned even more.

# Bibliography

[1] Mobvista (2018, May 7). *Why Mobile Gaming Is Now Bigger than Console and PC Gaming Combined, and Still Growing and Always Changing* [Online]. Available: <https://www.mobvista.com/en/blog/mobile-gaming-now-bigger-console-pc-gaming-combined-still-growing-always-changing/>

[2] The Verge (2019, Feb. 28). *Pokémon Go spurred an amazing era that continues with Sword and Shield* [Online]. Available:

<https://www.theverge.com/2019/2/28/18243332/pokemon-go-sword-shield-franchise-history-niantic-nintendo-switch>

[3] iTunes App Store (2019, Mar. 29). *Minion Rush* [Online]. Available:

<https://itunes.apple.com/jp/app/minion-rush/id596402997?l=en&mt=8>

[4] N. Alderman (2014, Jun. 23). *Why Candy Crush Saga likes to play on your sweet tooth* [Online]. Available:

<https://www.theguardian.com/technology/2014/jun/23/candy-crush-saga-freemium-games>

[5] D. Passmore, CourierMail (2013, Aug. 21). *Fruit Ninja, Jetpack Joyride creators Halfbrick Studios top Queensland's Entertainment Rich List* [Online]. Available:

<https://www.couriermail.com.au/news/queensland/fruit-ninja-jetpack-joyride-creators-halfbrick-studios-top-queensland8217s-entertainment-rich-list/news-story/81baaac485b8a004fa8ffeac019f7da8>

[6] iTunes App Store (2019, Mar. 29). *Jetpack Joyride* [Online]. Available:

<https://itunes.apple.com/us/app/jetpack-joyride/id457446957>

[7] C. Daro (2019, Feb. 6). *Will Casual Games Dominate Mobile Gaming in 2019?* [Online]. Available:

<https://www.gamespace.com/all-articles/news/casual-games-mobile-gaming-2019/>

[8] E. Adams (2000, Aug. 1). *Casual Versus Core* [Online]. Available:

<http://www.gamasutra.com/view/feature/131529/casual_versus_core.php>

[9] R. Moore-Colyer (2018, Mar. 1). *Desktop PC sales slumped in 2017 as the death rattle sounds once again* [Online] Available:

<https://www.theinquirer.net/inquirer/news/3027646/desktop-pc-sales-slumped-in-2017-as-death-rattle-sounds-once-again>

[10] C. Edwards (2013, Nov. 4). *Valve Lines Up Console Partners in Challenge to Microsoft, Sony* [Online] Available:

<https://www.bloomberg.com/news/articles/2013-11-04/valve-lines-up-console-partners-in-challenge-to-microsoft-sony>

[11] Statista (2019, Mar. 29). *Number of peak concurrent Steam users from November 2012 to October 2018 (in millions)* [Online] Available:

<https://www.statista.com/statistics/308330/number-stream-users/>

[12] SteamCharts (2019, Mar. 29). *Top Games by Current Players* [Online] Available:

<https://steamcharts.com/top>

[13] R. Zacny (2016, Jun. 9). *Hearts of Iron 4 Review* [Online] Available:

<https://www.pcgamer.com/uk/hearts-of-iron-4-review/>

[14] T. Edwards (2010, Aug. 11). *StarCraft 2 Review* [Online] Available:

<https://www.pcgamer.com/uk/starcraft-2-review/>

[15] T.J. Hafer (2018, Feb. 7). *Why is Civilization 5 still more popular than Civilization 6?* [Online] Available: <https://www.pcgamer.com/uk/why-is-civilization-5-still-more-popular-than-civilization-6/>

[16] S. Shearer (2013, Sept. 19). *Good Old Reviews: SimCity 2000* [Online] Available: <http://www.escapistmagazine.com/news/view/127892-Good-Old-Reviews-SimCity-2000#&gid=gallery_1778&pid=1>

[17] A. Meer (2016, Nov. 14). Have You Played… Civilization III? [Online] Available:

<https://www.rockpapershotgun.com/2016/11/14/have-you-played-civilization-iii/>

[18] T. Przybylski (2019, Jan. 21).. Available: [97tomasz.przybylski@gmail.com](mailto:97tomasz.przybylski@gmail.com)

[19] J. Beverina (2019, Jan. 22). Available: [jmbeverina@protonmail.ch](mailto:jmbeverina@protonmail.ch)

[20] S. Huijbrechts (2019. Jan. 22). Available: [seppehuijbrechts@hotmail.com](mailto:seppehuijbrechts@hotmail.com)

[21] R. Moss (2015, Dec. 28). *How a Mod Team Helped Age of Empires 2 Thrive* [Online] Available: <https://www.rockpapershotgun.com/2015/12/28/age-of-empires-2-forgotten-empires/>

[22] Twitch (2019, Mar. 30). *T90Official* [Online] Available: <https://www.twitch.tv/t90official>

[23] R. Kaiser (2013, Aug. 15). *Europa Universalis IV Review* [Online] Available: <https://uk.ign.com/articles/2013/08/15/europa-universalis-iv-review>

[24] Extra Credits (2017, Jan. 18). *Strategic Uncertainty - Keeping Strategy Games Fresh - Extra Credits* [Online] Available:

<https://www.youtube.com/watch?v=PJKTDz1zYzs>

[25] A. Gallegos (2012, Jun. 19). *Civilization V: Gods and Kings Review* [Online] Available: <https://uk.ign.com/articles/2012/06/19/civilization-v-gods-and-kings-review>

[26] D. Sirlin (2019, Mar. 30). *Balancing Multiplayer Games, Part 1: Definitions* [Online] Available: <http://www.sirlin.net/articles/balancing-multiplayer-games-part-1-definitions>

[27] D. M. Glantz, *Zhukov’s Greatest Defeat*, 1999, *Soviet Infantry Assault in the Snow*, pp. 171

[28] D. M. Glantz, *Zhukov’s Greatest Defeat*, 1999, *Artillery Being Hauled Forward*, pp. 171

[29] D. M. Glantz, *Zhukov’s Greatest Defeat*, 1999, pp. 25

[30] J. Clark (2015, Aug. 11). *A Look at Urban Warfare on World War II’s Western Front* [Online] Available: <https://taskandpurpose.com/a-look-at-urban-warfare-on-world-war-iis-western-front>

[31] D. M. Glantz, *Zhukov’s Greatest Defeat*, 1999, pp. 23

[32] R. Fahey (2000, Jun. 27). *Shogun: Total War - Stunning 3D Strategy Game Reviewed* [Online] Available: <https://www.eurogamer.net/articles/shogun>

[33] Reddit User FrancisFrey (2017, Jun. 9). *Rock paper scissors gameplay?* [Online] Available: <https://www.reddit.com/r/totalwar/comments/6g9d4j/rock_paper_scissors_gameplay/>

[34] RomanArmyTalk (2011, May 14). *How effective were spears against cavalry?* [Online] Available: <https://www.romanarmytalk.com/rat/archive/index.php/thread-18838.html>

[35] Reddit User DubstepLies (2013, Dec. 27). *What are some ways cavalry dealt with spearmen?* [Online] Available: <https://www.reddit.com/r/AskHistorians/comments/1ts731/what_are_some_ways_cavalry_dealt_with_spearmen/>

[36] Steam User SilentForce3 (2013, Jun. 21). *Rock Paper Scissors & COH.* [Online] Available: <https://steamcommunity.com/app/231430/discussions/0/864971287562863754/>

[37] R. Budnik (2018, Sep. 26). *Panzer Hunters – Soviet Anti-tank Guns of WWII with Photos* [Online] Available: <https://www.warhistoryonline.com/instant-articles/soviet-anti-tank-guns.html>

[38] D. Kim (2014, Aug. 19). *What can infantry do against tanks without air or artillery support?* [Online] Available: <https://www.quora.com/What-can-infantry-do-against-tanks-without-air-or-artillery-support-Is-infantry-equipped-to-deal-with-tanks-and-if-they-are-what-weapons-and-tactics-are-at-their-disposal>

[39] Reddit User VitruvianSquid1 (2017, Jun. 9). *Rock paper scissors gameplay?* [Online] Available: <https://www.reddit.com/r/totalwar/comments/6g9d4j/rock_paper_scissors_gameplay/diojlz1/>

[40] W. S. Dunn, *Stalin's Keys to Victory*, 2007, pp. 75

[41] Lexikon Der Wehrmacht (2019, Jan. 20). *9. Panzer-Division* [Online] Available: <http://www.lexikon-der-wehrmacht.de/Gliederungen/Panzerdivisionen/Gliederung.htm>

[42] R. Koster, *A Theory of Fun for Game Design*, 2005, pp. 100

[43] R. Koster, *A Theory of Fun for Game Design*, 2005, pp. 102

[44] Y. Chou (2019, Mar. 31). *Octalysis - the complete Gamification framework* [Online] Available: <https://yukaichou.com/gamification-examples/octalysis-complete-gamification-framework/>

[45] R. Koster, *A Theory of Fun for Game Design*, 2005, pp. 114

[46] R. Koster, *A Theory of Fun for Game Design*, 2005, pp. 128

[47] J. Schell, The Art of Game Design - A Book of Lenses: Second Edition, 2014, pp. 22

[48] Statista (2018, Mar.). *Top selling Nintendo Wii game titles worldwide as of March 2018 (in million units)* [Online] Available: <https://www.statista.com/statistics/248204/top-selling-nintendo-wii-titles-worldwide/>

[49] J. Schell, *The Art of Game Design - A Book of Lenses: Second Edition*, 2014, pp. 59

[50] D. M. Glantz, *Zhukov’s Greatest Defeat*, 1999, pp. 191

[51] D. M. Glantz, *Zhukov’s Greatest Defeat*, 1999, pp. 6

[52] F. Klepacki (2000, Oct. 23). *Command and Conquer: Red Alert 2 - Soundtrack* [Online] Available:

<https://www.youtube.com/watch?v=fqlb0p11RcE>

[53] Unity (2019). [Online] Available: <https://unity.com/>

[54] Civilization Wiki (2019). *Fog of war* [Online] Available: <https://civilization.fandom.com/wiki/Fog_of_war>

[55] Liquipedia - StarCraft II (2019). *Fog of War* [Online] Available: <https://liquipedia.net/starcraft2/Fog_of_War>

[56] Gamepedia Total War Warhammer Wiki (2016, May 28). *Visibility* [Online] Available: <https://totalwarwarhammer.gamepedia.com/Visibility>

[57] IGN (2010, Mar. 8). *Civilization V Preview* [Online] Available: <https://uk.ign.com/articles/2010/03/08/civilization-v-preview-2>

[58] GitHub (2018, May 22). *Gson* [Online] Available: <https://github.com/google/gson>

[59] Techopedia (2019, Apr. 1). *Java Swing* [Online] Available: <https://www.techopedia.com/definition/26102/java-swing>

[60] Oracle Java Documentation (2019, Apr. 1). *Trail: 2D Graphics* [Online] Available: <https://docs.oracle.com/javase/tutorial/2d/>

[61] GitHub (2013, Feb. 27). *SVN export of SwingLabs Swingx* (read-only) [Online] Available: <https://github.com/ebourg/swingx>

[62] Oracle (2019, Apr. 1). *JavaFX Frequently Asked Questions* [Online] Available: <https://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html#6>

[63] Oracle (2014). *Class MediaPlayer* [Online] Available: <https://docs.oracle.com/javafx/2/api/javafx/scene/media/MediaPlayer.html>

[64] Nanyang Technological University (2012, Apr.). *Java Game Programming - 2D Graphics, Java2D and Images* [Online] Available: <http://www3.ntu.edu.sg/home/ehchua/programming/java/j8b_game_2dgraphics.html>

[65] Oracle (2017). *Compositing Graphics* [Online] Available:

[66] C.S.Horstmann & G.Cornell, *Core Java Volume II - Advanced Features*, 2008, pp. 365

[67] C.S.Horstmann & G.Cornell, *Core Java Volume II - Advanced Features*, 2008, pp. 367

[68] D. Brackeen, *Developing Games in Java*, 2003, pp. 94

[69] Oracle (2018). Class MouseInputAdapter [Online] Available: <https://docs.oracle.com/javase/8/docs/api/index.html?javax/swing/event/MouseInputAdapter.html>

[70] D. J. Barnes, *Object-Oriented Programming with Java - An Introduction*, 2000, pp. 653

[71] D. J. Barnes, *Object-Oriented Programming with Java - An Introduction*, 2000, pp. 658

[72] D. Burke (2012, Mar. 31). Stretch Icon [Online] Available: <https://tips4java.wordpress.com/2012/03/31/stretch-icon/>

[73] D. J. Barnes, *Object-Oriented Programming with Java - An Introduction*, 2000, pp. 249

[74] D. Brackeen, *Developing Games in Java*, 2003, pp. 287

[75] D. Reilly & M. Reilly, *Java Network Programming and Distributed Computing*, 2002, pp. 41

[76] MCVersions (2019, Apr. 1). *The easiest way to find Minecraft versions!* [Online] Available: <https://mcversions.net/>

[77] D. Reilly & M. Reilly, *Java Network Programming and Distributed Computing*, 2002, pp. 45

[78] D. Reilly & M. Reilly, *Java Network Programming and Distributed Computing*, 2002, pp. 75

[79] D. Reilly & M. Reilly, *Java Network Programming and Distributed Computing*, 2002, pp. 80

[80] E. Makuch (2016, Apr. 13). *Red Dead Redemption "A Recurring Nightmare" During Development, Sam Houser Says in 2009 Email* [Online] Available: <https://www.gamespot.com/articles/red-dead-redemption-a-recurring-nightmare-during-d/1100-6438770/>

[81] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 2008, pp. 18

[82] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 2008, pp. 19

[83] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 2008, pp. 140

[84] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 2008, pp. 141

[85] P. Torok (2011, Dec. 15) *Keep my methods as small as possible?* [Online] Available: <https://softwareengineering.stackexchange.com/questions/125357/keep-my-classes-and-methods-as-small-as-possible>

[86] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 2008, pp. 179

[87] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 2008, pp. 77

[88] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 2008, pp. 79

[89] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 2008, pp. 98

[90] J. Schreier (2017, Jan. 18) *Quality Assured: What It’s Really Like To Test Games For A Living* [Online] Available: <https://kotaku.com/quality-assured-what-it-s-really-like-to-play-games-fo-1720053842>

[91] J. Schell, *The Art of Game Design: A Book of Lenses, Second Edition* , 2014, pp. 59.

[92] Altexsoft (2018, Feb. 23). *Extreme Programming: Values, Principles, and Practices* [Online]. Available: <https://www.altexsoft.com/blog/business/extreme-programming-values-principles-and-practices/>

[93] Andrew Powell-Morse (2016, Dec. 8). *Waterfall Model: What Is It and When Should You Use It?* [Online]. Available:

<https://airbrake.io/blog/sdlc/waterfall-model>