University of
HUDDERSFIELD

University of Huddersfield Repository

Sharples, Timothy

A Simplified API for the Creation of Bots for Real Time Strategy Games

Original Citation

Sharples, Timothy (2018) A Simplified API for the Creation of Bots for Real Time Strategy Games. Masters thesis, University of Huddersfield.

This version is available at http://eprints.hud.ac.uk/id/eprint/34558/

The University Repository is a digital collection of the research output of the University, available on Open Access. Copyright and Moral Rights for the items on this site are retained by the individual author and/or other copyright owners. Users may access full items free of charge; copies of full text items generally can be reproduced, displayed or performed and given to third parties in any format or medium for personal research or study, educational or not-for-profit purposes without prior permission or charge, provided:

- The authors, title and full bibliographic details is credited in any copy;
- A hyperlink and/or URL is included for the original metadata page; and
- The content is not changed in any way.

For more information, including our policy and submission procedure, please contact the Repository Team at: E.mailbox@hud.ac.uk.

http://eprints.hud.ac.uk/

A Simplified API for the Creation of Bots for Real Time Strategy Games

Timothy Sharples
U1251424
Supervising Tutor: Prof W. Faber
University of Huddersfield

Abstract

Artificial Intelligence research in the past few years has been increasingly focusing on games, with Real Time Strategy games being of particular interest. However, one of the main tools used in the creation of agents in these environments has quite a steep learning curve for entry into development, leading to some potential entry barriers that new AI developers have to overcome to get into the field.

This project aims to investigate ways in which these entry barriers can be lowered. Particular interest is taken in the potential future application of techniques found in Visual Programming Languages. With this view, a library of tools to help new AI developers is created and tested before being given to members of the Bot Development community to provide their technical assessment feedback, and has been given to new developers with no previous experience with the subject area for their evaluation on how effective these tools were in easing the process of starting out with development.

This report describes the research, the planning and testing of this library, and the conclusions drawn from the subsequent evaluation of the tools.

Contents

1 Introduction 4
1.1 StarCraft . . . 4
1.2 BWAPI . . . 6
1.3 Visual Programming . . . 6
1.4 Aim and Objectives . . . 6
1.5 Contributions to Knowledge . . . 7
1.6 Report Breakdown . . . 7

2 Bots in Real Time Strategy Games 9
2.1 RTS Games . . . 9
2.2 BWAPI . . . 10
2.3 Atlantis API . . . 11

3 Visual Programming Languages 13
3.1 Scratch . . . 13
3.2 Unreal Engine Blueprints: Visual Scripting System . . . 19
3.3 Analysis of Scratch compared with Unreal Blueprint Scripting System . . . 20

4 Specifications 22
4.1 Aesthetics . . . 22
4.2 Mechanics . . . 23

5 Design 31
5.1 Production Queue . . . 31
5.2 Enemy Base Tracking . . . 33
5.3 Builder Manager . . . 34
5.4 Squad Management . . . 36

6 Testing Plan 38
6.1 Production Queue . . . 38
6.2 Enemy Base Tracking . . . 42
6.3 Builder Manager . . . 43
6.4 Squad Management . . . 44
6.5 Example Bot using New Library . . . 45

7 Evaluation 47
7.1 Evaluation by members of the Bot Development community . . . 47
7.2 Evaluation by developers with no prior experience . . . 48

8 Conclusion 51
8.1 Future Work . . . 52
8.2 Closing . . . 53

9 References 54

Glossary **56**

Appendices **57**

A: Testing Documentation 57

1 Introduction

Real time strategy games have become, in the last few years, one of the main focuses in the development of artificially intelligent agents. Since Laird and van Lent noted that differing types of computer games contain many of the challenges of creating human level AI systems, the call has been out to use them as the testing ground for new research (Laird & VanLent, 2001). After recent successes with other games using AI techniques, development has been tooled towards finding ways to use these techniques in Real Time Strategy (RTS) games like StarCraft (Cheng, 2016).

In this project we are going to be investigating the tools that are used to conduct the development of artificially intelligent agents, and then observe difficulties presented to new AI developers that use such software. This is in an attempt to ease the presented difficulties through the creation of supplementary systems. The targeted aim of a Visual Programming Language is to make the introduction to programming, and the subsequent learning of development concepts as easy as possible. With this in mind, this project will also investigate ways that other areas of programming have been eased for new-comers through the use of visual programming languages like Scratch and Unreal Engine's Blueprint system in the expectation that this analysis will allow us to gain insight in to ways the new AI developer experience could be improved still further.

1.1 StarCraft

StarCraft is a real time strategy game created by Blizzard Entertainment in 1998. The expansion pack 'Brood War' was released later the same year and added a new campaign along with new units to the game. The game with the expansion is the platform that the API BWAPI is designed for.

Set in the distant future, the game features three distinct and well balanced races:

- The Terran - A human like species with mechanical vehicles and armoured infantry.
- The Protoss - An advanced psionic (mind control abilities such as telepathic, telekinetic, etc.) race focused on higher levels of technologies than the other two races.
- The Zerg Swarm - An insectoid hive-mind species, well suited for swarm-like strategies.

Each of these races have their own strengths and weaknesses based upon their unique units, buildings and play styles. This means that each race has many strategies and tactics that can lead to victory (Blizzard Entertainment, 2017b).

All three races follow the same basic gameplay of using workers to collect the two in-game resources, then using these resources to create buildings, units, and upgrades. These resources require two different methods of collecting, minerals only require a worker to go and mine at one of the mineral patches, which is something they can do with no other prior preparation. To collect Vespene Gas however, a collecting station (exact details

of which vary between races, but basic functionality stays the same) must be built over the gas geyser before a worker can harvest it.

Units are produced using various buildings such as the Terran Barracks, or the Protoss Warp Gate. However some units have prerequisites of other buildings or researches that must be completed before they can be trained. For example, the Terran Marine (a basic infantry unit) can be produced straight from a barracks, but the Terran Medic, requires the Academy to be constructed before they can be trained at the barracks.

Figure 1. StarCraft - Gameplay

Figure 1 shows an example screenshot of a typical moment during a game of StarCraft. The image as a whole depicts an overview of the current players base, in this case the example player is playing as the Terran. The red square highlights the main building of the Terran, the Command Centre and the units that are collecting the resources from the blue mineral patches at the far left of the image. The Yellow square is showing a Vespene Gas refinery where the other resource of the game is harvested from. The green square on the right of the image is showing the Terran Factory with its associated add-on building the machine shop. This is a unit production building and is used to make mechanised units such as tanks. The final highlight, the light blue square in the bottom left is showing the mini-map. The mini-map shows the player an overview of the game world map, where the fog of war has been revealed and any buildings and units that are visible to the player.

The gameplay revolves around being able to send units to engage your enemy around the map and destroy their bases. Particular emphasis can be placed on ’scouting’ which is the act of sending units to explore the game map and discover what the enemy is doing.

As everything outside of a radius around your units and buildings is hidden by a 'fog of war' (which literally conceals the gameplay), it is important to discover what is happening around the map so that you know what to react to, and what strategy to engage to properly counter the enemies.

Winning a game of StarCraft, strictly speaking, is defined as destroying all of your opponent's buildings. However, in practice, it is common for a human player to concede a game once they recognise they are in a losing situation that is impossible for them to come back from enough to attempt to win the game.

1.2 BWAPI

Of all the tools used for researching and developing real time strategy games, among the most popular is the Brood War Application Programming Interface (BWAPI) (Buro & Churchill, 2012). This API gives developers the opportunity to develop agents that operate in the game environment of the popular real time strategy game StarCraft. The problem presented however is the relatively steep learning curve when starting out with development in BWAPI. The current API is so low-level, that the AI developer is presented with all the known difficulties of operating in an RTS game environment with little or no support. Difficulties such as working with uncertainty, working against an opposing agent, and working with non-deterministic outcomes of agent actions. Due to this, many new developers will spend a large portion of their time when starting out trying to deal with the same problems as all other AI developers overcoming difficulties innate in the environment. If that is the case then this time spent dealing with the same previously solved issues is time that isn't being spent on developing novel solutions to the problem area that the AI developer wanted to work on, or being spent in implementing the new learning algorithm that they wanted to test. This leads to more wasted time and a potential stifling of research in this field.

1.3 Visual Programming

For teaching and introducing new developers to programming, visual programming languages are becoming more widespread, due to their focus on programming logic over language specific syntax (Resnick et al., 2009). The techniques developed for these languages and environments that allow users of these languages to quickly and easily understand the meaning of code blocks is something that is unprecedented in other 'normal' written programming languages. The ease of use and readability is something that should be investigated for use in more applications. With the success of the use of Scratch in schools as a high level and visualised introduction to programming, it shows that programming as a whole can be simplified a great deal from the syntax barriers of traditional languages to help get more people interested in the field of software programming.

1.4 Aim and Objectives

With one of the next stages of AI research being to work in RTS games, but one of the leading RTS game API's having such a steep learning curve, this project seeks to lower

this barrier and the combination of the simplifying techniques from visual programming could aid in this respect. Due to the scope of this project, after the initial research into visual programming, only the first steps to achieving this goal have been undertaken. However this will provide a starting point for the continuation of this work in the future. The first step of the simplification process has been to create a library of useful packages and features, as determined by researching what existing AI developers would have found useful when they started out with BWAPI and this step has been realised and assessed.

1.5 Contributions to Knowledge

The project shows the specific ways that the one of the most commonly used development tools in real time strategy game bot development, can be improved to help new AI developers. The improvements coming from research gathered from members of the development community, and then evaluated through being given to developers, new and old, for their feedback.

This project also addresses the ways of how to improve the introductory experience of new comers to the development of AI, or indeed new comers to programming in general. Building upon the work of the developers of existing visual programming languages, and other simplified development tools, this project shows how software can be evaluated and broken down to find what areas should be improved upon to simplify the learning curve of that software for a new user.

1.6 Report Breakdown

Progressing through this report will elaborate on the tasks undertaken throughout this research project.

Section 1 is an introduction to the subject matter and the project overview.

Section 2 details the research into current development of bots in real time strategy games, including an overview of the genre and why BWAPI is used as extensively as it is.

Section 3 is about Visual Programming Languages, the different ones that are popular, and what they are like. This then goes on to analyse the differences between the languages and what parts could be used in making a new language for use with BWAPI.

Section 4 contains the details of the specifications of what the new visual programming language would be like in a broad sense, before narrowing in on the library that will be created as the first step in that production.

Section 5 describes the designs for the packages and classes that will be in that library, including class diagrams and sequence diagrams for each system.

Section 6 details the testing that was done on all the packages of the new library to prove that it followed the specifications and designs laid out in the previous sections. It also contains the details of an example bot and its capabilities after it was created using every function of the new library.

Section 7 has the evaluation of the new library when it was given to both experienced developers and to new developers who hadn't worked with BWAPI before to get their feedback on how much of an improvement it is, if any, over the base API.

Section 8 has the conclusions of the project with what was discovered and how the project may be continued in the future.

2 Bots in Real Time Strategy Games

A "bot" is quite a widely used term in the field of Artificial Intelligence, especially in the field of gaming. In this project, the definition will be taken to be that a bot is an artificially intelligent agent that was created to take the place of a human player within a gamespace by interacting with the game in the ways that a human player would do.

However, almost as varied as that definition is the one for Artificial Intelligence. According to Schwab, this discrepancy in definitions can be attributed to the relative youth of the field, particularly the specialisation for implementation within computer games (Schwab, 2009). The definition that is best suited for use and reference within this project has been laid out by Neil Kirby in the book Introduction to Game AI (Kirby, 2011). Within this book, three stipulations are made for an agent to be considered to have artificial intelligence.

- The agent should have the ability to act.
- The agent's decisions should be intelligent ones.
- The agent must react to ongoing changes within the game state.

For a bot to be able to fulfil the first requirement, it may have inputs to assist in its decision making, but it must have the ability to output in a way that affects the game world it is working within. For the other two criteria, the bot must take the inputs it is receiving and use them to reach its decisions about how to act. It must also be able to take in new information about the game state and analyse it to see if the situation has changed to the extent that it must adapt its decisions and strategy. These latter two processes are the main focus of the development of in-game bots.

2.1 RTS Games

Real Time Strategy games are close to military simulation in concept. Most games will consist of two or more players fighting over control of a two dimensional map with resources and strategic positions scattered throughout. Then with these resources they will produce armies and will command their units in battle in real time engagements (Buro, 2003).

RTS games are set apart from other games that are used in AI research and development by several key game play difficulties. Most of these difficulties stem from the innate real time nature of the game, but some are more specific to strategy games as a genre (Buro & Churchill, 2012).

The four main difficulties or "complexity points" are listed below.

- **Partial Observability** - Players, and by extension bots, can only see part of the map. The rest is hidden from view by what is referred to as the "fog of war". This means that any decisions must be made with incomplete data, either by making guesses or by attempting to extrapolate answers from the available world data.

- **Actions are made in Real Time** - Any time that a player spends thinking and not making actions is time that the game is advancing. This is the largest difference over a game like chess where a player can sit and spend time evaluating game state and calculating optimal moves. In real time, the computational processes must be optimised for quick resolution.
- **Durative Actions** - Nearly all actions within the game take time to complete; be it the movement of a unit or the construction of a building. As such a solid base of temporal reasoning, or at least allowances for tasks to be completed must be built into any bot logic (Ontanon et al., 2013)
- **Non-Deterministic Outcomes** - When starting an action in a Real Time Strategy game, the outcome is not guaranteed. A unit that was sent to explore the game map might be killed or otherwise intercepted before reaching its objective, or a builder might be interrupted before it finishes constructing a building. This uncertainty means that more provisions must be in place to catch unforeseen circumstances presented by interruptions in a bot's actions.

These four main complexity points contribute to the difficulty of creating bots that can succeed in a real time strategy environment. This project aims to look into each of these issues further and investigate the possibilities of making them easier to overcome for bot developers.

2.2 BWAPI

While other RTS engines exist for bot development, such as ORTS or microRTS, one of the most popular ones in the field is the Brood War Application Programming Interface (BWAPI) (Buro & Churchill, 2012). Originally developed in 2009, it was created by reverse engineering the official release of StarCraft: Brood War by Blizzard Entertainment. It works by reading and writing directly into the games memory space to provide interaction and allow commands to be issued by a computer as though they were human player inputs (*BWAPI Homepage*, n.d.).

Due to the nature of this program, it could be seen as a 'hack' of the original game, however the game creators, Blizzard Entertainment, have responded to the use of BWAPI by endorsing development using it, but with a clear stance that no such actions should be taken with their newer releases (*Battle.net End User Liscence Agreement*, 2015; *BWAPI FAQs*, 2014). Blizzard Entertainment have even been known to donate prizes to competitions that are run for the bots developed with BWAPI.

The specific benefits for using BWAPI over other development engines come from multiple angles. Firstly, from a development side, the API is still being actively developed and supported. This is in contrast to one of its competitors, ORTS that hasn't received an update since 2010 (*ORTS Homepage*, n.d.). Having an active team working on the API means that any bugs discovered during development for this project can be raised with the team potentially for fixes, it also means that there are more likely to be other developers

offering advice with the more involved areas of using the API.

The second benefit of using a commercially successful game for AI testing is the game balance. By having a professional game, tested and balanced over years of gameplay, the development environment is going to be more stable for no one strategy is dominant and thus allowing for more innovation by a bot calculating the best possible responses to a given situation.

Another benefit is the innate popularity of the game. If the game is well known, then it is easier to find people to test bots by playing against them as they will already have knowledge of the game mechanics (Magnusson & Balsasubramaniyan, 2012). This same logic can also be applied to developers of bots for instead of having to learn new games they will have pre-existing knowledge to aid them in their bot development.

Finally, and potentially the most crucial reason for choosing BWAPI for this project is that it is the API being used in other research and is popular with other bot developers. This means that while designing the API for development during this project, input can be taken from other programmers' experiences with the base BWAPI to see what aspects of the API, in their opinions, could be most improved upon. These recommendations can then be compiled and examined to evaluate their relative feasibility and to come up with an action plan of development.

2.3 Atlantis API

Of all the existing attempts to simplify the Brood War Application Programming Interface, one of the more comprehensive is Atlantis. Its main goal from the outset was to "Make it much, much easier to create [a] new bot starting from zero" (Poniatowski, 2017).

The author of the Atlantis API lists the improvements that are made compared to the standard version of BWAPI. As the improvements are described as being able to save a potential bot writer the tedium of writing repetitive code that any bot would need, it is important for us to note what these improvements are. If these improvements are accepted simplifications to the process of creating a new bot for current developers, then a more simplified environment should, at least, consider them for its level of abstraction.

It is also important to note that this framework was built with the specific goal of removing the minutia of managing a bot's micro actions throughout the game and instead allowing developers to create bots without their focus becoming distracted from trying to calculate the optimal overarching strategy to beat the opponent. This takes everything except for creating a build order out of the developer's hands and while this is useful for research and development into enemy action interpretation, this level of abstraction is too high for the goal of simplifying bot development. This is due to in essence the bot already having a created framework that is now just waiting for commands. The goal of this project is to find ways of simplifying creation and thus the lowering of entry barriers, without taking too much of the development work out of the AI developers' hands.

The improvements that have been made by the Atlantis API however are relevant springboards for examining what areas should be simplified. Some of its key features that are listed in its documentation are:

- takes care of workers during mining and construction
- assigns workers to optimal mineral fields
- scouts to find enemy bases and detects the build order used
- micro manages unit combat
- automatic base expansion when reaching enough minerals
- automatically begin creating buildings and units needed to counter opponent's strategy choices

Some of these key features exceed the level of autonomy that is intended for the new API this project is looking to create. For example, if we look at how the feature "scouts" to find enemy bases and detects the build order used' works, we see that this framework will automatically choose one of the bot's units, start scouting the map systematically for enemy bases, then will analyse what it finds to interpret the enemy's build order. While all of these actions are things the average bot playing StarCraft will want to do to do well, this project aims to leave the management of it as much as possible in the AI developer's hands while still simplifying the process. So in this example, instead of automatically choosing a unit for the scouting, the developer will be given the option of designating a unit as a scout and then setting up how they want the scouting to be conducted and starting it when they want. Also, the feedback from the scouting will be presented to the AI developer so that they can program the bot to interpret the information in whatever way they so desire instead of the interpretation and output actions being pre-determined by the Atlantis API.

With these differences in design choices our aim is to create a development tool that lets the AI developer feel that they are more in control of how the bot will think and react. This leads to a more conducive environment for bot creation with the ability to see why the bot has reacted in a given way without all the behaviour being pre-programmed and hidden "under the hood".

3 Visual Programming Languages

A visual programming language (VPL) is a way of programming that, rather than being based in written code like a common language such as C++ or Java, is based in visually represented blocks or other similar on-screen structures. Similar in view and use to a flowchart, VPLs allow for easier representation of code structure and program flow to those who aren't necessarily as versed in traditional programming techniques and syntax. As an aside, it is worth mentioning that the Microsoft Visual language suite such as Visual Basic and so on, just to be confusing, are traditional languages and not true visual programming languages.

VPLs started to become used when it was noted that professors, when teaching introduction to programming courses, spent more time instructing on the minutia of language syntax rather than the core principles of logic and algorithmic thinking (Shackelford & LeBlanc, 1997). A study conducted at the United States Air Force Academy during an introduction to programming course showed that in general, students preferred to work using visual representation when given the choice in a final exam rather than other traditional options like MATLAB (Carlisle et al., 2005). As such, VPLs are being used to make programming easier in many areas. The ones that will be examined further in this paper, will be ones pertaining to development for games and similar media. They have been demonstrated to be popular with new-comers to programming, with the users of Scratch, a VPL and environment designed for younger developers, uploading more than 1,500 new projects in 2009 with the majority of its users between the ages of 8 and 16 (Resnick et al., 2009). With this level of usage, it seems prudent to examine the success of these languages to see why they are used, and which successes of them can be attempted to be replicated.

3.1 Scratch

The main focus of reference for drawing inspiration in creating a Visual Programming Environment comes from Scratch. Scratch is a visual programming language and environment designed from the ground up to make entry into programming and other media development easier. With both an online-based and a stand-alone desktop application, Scratch is easily accessible to many users. It is used to create multi-media projects, using imported media and scripts put together in the engine.

3.1.1 Core Concepts. The Scratch development environment was created with three core design principles to stay true to its intended purpose (Resnick et al., 2009). The creators aimed to make the language, compared to others, "more tinkerable", "more meaningful", and "more sociable". So it is worth looking more closely at how each of these has been realised.

The developers of Scratch define tinkerable to be the ability to be able to switch things around easily, just to see what would happen, without being hampered by syntax errors. To make the development process more tinkerable, the Scratch environment is designed in a way that makes it easy to make quick changes to a program or process, even during runtime. This allows a developer to tweak and change what they want to reach

their eventual desired outcome. However, Scratch programming blocks are also created in such a way that they only fit together in ways that make syntactic sense. This is by design to allow developers that use Scratch to be able to try out different combinations easily to see what happens, without the worry of the end result not running at all. As a result of this tinkering new plans could form and the project will naturally evolve as new ideas come to light. To enable the timely feedback of tinkering changes, Scratch runs live, meaning that there is no compilation time and that code can be changed on the fly as the program is running.

The next design point was to make the work feel more meaningful to the user. People learn best when working on a project that they are invested in, or is meaningful to them (Resnick et al., 2009). One way to make people immersed in their projects that Scratch focuses on is personal customization. It is, by design, very easy to import photos and music clips or even voice recordings. While restricted to a 2D environment, it facilitates a great deal of personalised assets in the projects. It has been noted that other introductions to programming are achieved by creating programs that the new developer has no connection to such as creating a prime number generator or testing basic maths skills. Instead, the basis of Scratch is to help users create projects that interest them while still presenting opportunities for the learning of key programming principles (Maloney, Resnick, Rusk, Silverman, & Eastmond, 2010).

While the final design aspect of being more sociable is, on the surface, not as relevant as the others to the development in this project, it is pertinent to investigate this aspect as it was something identified as being important enough to be an aim expressed in the creation of the Scratch development experience.

Since its launch, the online community surrounding Scratch has gone through many revolutions. In just over two years after its beginning in 2007, over half a million projects were uploaded and shared on the site (Resnick et al., 2009). The large library of projects can serve as inspiration for users or can serve as a knowledge base full of examples on how to accomplish different goals. One change in the online community came with the inclusion of a way for the creator of a project to be cited if somebody used part of their project in their own. This was a move by the creators of Scratch to try and help people to feel proud rather than annoyed if someone used their project.

This online community has also led to the rise of collaborative projects between a diverse set of users. Some of these groups have even gone on to call themselves 'micro-companies' and have ongoing projects recognised in the community. The collaboration-based development allows for more investment from a developer for a given project while providing more experience and learning angles to draw upon.

It is a reasonable conclusion to draw from the popularity of the online community that having a shared environment for projects with terms of reference that all users understand, can lead to more collaborative work. In turn this can lead to each developer making more progress in their work due to the support network the community provides.

From this it could be reasoned that perhaps development in other areas should look into having a more solid base for each user to draw upon the same points of reference in their work. This would then potentially enable stronger collaboration which in turn enables development to be pushed further along.

3.1.2 Design. One of the design focuses of Scratch, making programming more accessible, was aimed to be accomplished by making the scripting seem more open (Maloney et al., 2010). For this, three major design decisions were made.

- Code execution would be visible
- There would be no error messages
- Variables would be visible

These design points are aimed towards making the programming side of creating a project as easy as possible. As the aim of this project is to investigate simplifying bot creation, these design decisions offer interesting insight.

While a project in the Scratch engine is running, the programming block or blocks that are currently being executed are highlighted in the scripting area. This offers similar visibility of code execution to stepping through code after a breakpoint in more traditional development environments, but without the pausing of execution. This shows the developer the order a sequence of code blocks are being run in, and also whether perhaps some of them are not being run at all. These simple on-screen clues can prove invaluable when troubleshooting code that is not running as intended.

"When people play with LEGO® bricks, they do not encounter error messages" (Maloney et al., 2010). This statement about how working with Scratch should feel is reflected in the development of its engine. The drive to eradicate syntax errors lead to the creation of the programming blocks that will only snap together in ways that would make programmatical sense. But in order to work towards having code that avoids run-time errors, each block must attempt to do something that makes sense when presented with an invalid input. For example, the 'set size' function for an on-screen sprite will restrict itself from drawing something larger than the screen, or drawing something that would be invisible for being too small. With this philosophy, syntax and runtime errors are reduced but of course it still remains possible that there could be logical errors in the script that the developer created. However if a script at least does something, even if not what the creator intended, then it can provide some insight as to what went wrong in execution and what needs to be done to fix the problem.

In most common programming languages, without setting up separate watch windows for the purpose (if the tool supports such things) then a simple piece of information such as the value of a variable is hidden to the user of the development tool. This can lead to frustration during development and testing as developers are often left in the dark as to what value their operation is using, and so why certain procedures are reacting in unexpected ways. When using Scratch, whenever a variable is used it is displayed in the

engine so that the developer can easier trace the interactions the program has with that variable. This aim of this representation is to bring the concept of variables into the light and make their uses more obvious. An extension of this concept is employed whenever a list or array of variables is used, such that all the elements of the list are displayed on-screen thus allowing the developer to see how they are changing.

These three design criteria that drove the development of Scratch specifically to make programming easier offer strong direction on how to simplify the process of creating bots. Each of these design decisions point towards giving the user of the development engine an 'under the hood' look at the software execution, but without overwhelming them with too much information at once. This process of showing what is being executed while not showing unnecessary technical details could be useful in the design of a visual development environment for creating bots, as using this philosophy could show the creator of the bot what processes and 'thoughts' the bot is having in the current situation, in a more visually readable way.

3.1.3 Programming Environment. Scratch is built with the objective of being an object-based language. To be called a true object-orientated language, it must support inheritance (Wegner, 1987), and so with Scratch not fulfilling this criteria it can only be classified as object based and not object orientated. what this means in practice is that in Scratch, game assets, such as sprites, are the programming objects as well. That is, they contain their own variables and their own scripts and are completely independent from each other.

Scripts are made up of 4 different categories of programming blocks that represent different programming statements, thus providing a wide range of functionality and levels of complexity. These 4 programming blocks, detailed below, are trigger blocks, control structures, function blocks and command blocks.

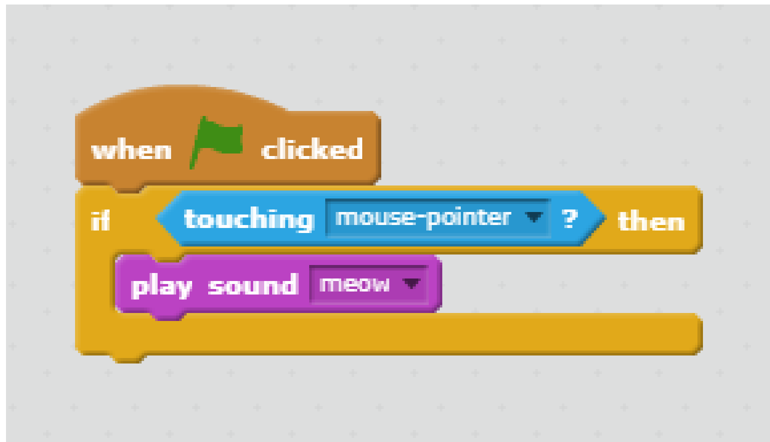
Trigger Block - This is shown curved at the top to visually show that it will not snap to the bottom of any other blocks. It is used to start a sequence once the specified event is triggered.

Control Structure - These represent commands that contain other nested commands in text-based languages, such as an 'IF' statement or a loop. In Scratch these are visually represented by a "catching block" that is shaped like a capital 'C' and the nested commands are placed inside the control structure.

Function Block - Function blocks represent conditions and parameters. They are placed in other blocks that have open parameter slots. They can represent a Boolean assertion for an 'IF' statement or a variable for checking if a condition is met for breaking a loop.

Command Block - These represent the base statements of a text based language. Tasks like moving objects, playing sounds or outputting a message to the screen are all done with basic command blocks.

Figure 2. Scratch Programming Blocks



Screen-shot taken in Scratch Editor January 2018
https://scratch.mit.edu/projects/editor/?tip_bar=getStarted

As already mentioned (section 3.1.2), Scratch programming blocks are designed to only fit together in ways that mean something. The shape of the blocks prevents them snapping together in ways that make no syntactical sense. Command blocks will fit together in sequence and so do control structures, but it would be impossible to place a function block or a trigger block in the same places. Figure 2 shows a Trigger Block followed by a Control Structure which checks a Function Block and contains a Command Block. This way of visually constricting where blocks can be placed in script construction is all designed to make it more intuitive for developers who are not familiar with underlying programming syntax when introduced to Scratch.

Another large simplification of the Scratch programming language is the amount of available variable types. In Scratch, data is either a Boolean, a number or a string and so these are the only data types that are available for use in expressions, or for use as variables. As Scratch automatically converts between numbers and strings as required, the only two visual shapes of variable blocks represent a Boolean or a number/string. This way of utilising variables means that developers don't have to worry about declarations of variable types at initialization or explicit conversions during runtime.

The final point to make about Scratch's programming environment is the ease at which it allows concurrency. In traditional programming languages, concurrency or multi-threading is regarded to be a complex technique and is rarely used by beginners. However in Scratch it is surprisingly simple to implement. If a given trigger fulfils the start prerequisites of multiple trigger blocks, then all of the code stacks will start to be executed simultaneously. While it doesn't contain the explicit concurrency protection locks that most languages have access to, the Scratch engine protects its execution by constraining where thread switches can occur. Once a control structure has been started, such as an 'IF' statement, the thread will not switch until the end of that control structure is reached.

If this happens to be too limiting, then an exception can be made by placing a 'Wait' command block in the stack. This implementation of multithreading allows developers that potentially might not understand the complexities of thread synchronisation issues to work on a code stack independent of other code stacks without worrying about unexpected outcomes from execution sequencing issues.

3.1.4 Development Points. Several of the main points of development of Scratch, and the evaluated successes of the engine, are relevant to the planning of this project. The ease of use of the visual language demonstrates key ways in which the entry barrier to programming can be lowered and so while this project is not focusing on programming as a whole, rather the focus is a specific area of development, some of these concepts should be taken on board in the planning stages.

An undertaking focusing on simplifying working with BWAPI may not be able to be slimmed down to as few code blocks as Scratch uses, but the concept of having debugging information visible by default is a promising one. Having the code visibly showing the program's execution could aid in development by allowing the AI developer to track the flow of the bot's logic and see if all of the written program is being utilised in the correct circumstances (Resnick et al., 2009). This simple trait of a visual programming environment could lead to logic errors, which are innate to programming new software, being addressed in a much more timely manner.

As has already been mentioned, simplifying the entirety of the Brood War API into a small number of command blocks may not be feasible without extensive further study into the optimal level of abstraction without cutting down programming options. However a certain level of abstraction over low level game mechanics that are not specifically pertinent to bot strategy but are still needed for gameplay should still be addressed. As seen in Scratch, when a user wants to have a sprite object move across the screen, they do not have to concern themselves with exact graphical interface instructions, instead they tell the sprite to move a certain number of pixels and the engine interpolates the rest. This same concept could be applied to working in BWAPI. Instead of having the bot developer worry about building creation, or resource pool management when these specifics are not key to strategy development, perhaps a better approach is to have the simplified API deal with the specifics so that the bot creator can deal with coding how the bot will decide what to build and what strategy to use.

The final point of analysis on Scratch is the success of its online community. It was heralded as one of the development engine's key successes as a beginner's programming environment (Resnick et al., 2009). While the development community for BWAPI also has online collaboration for asking for help from other AI developers, it is not such a central piece of the working experience as it is with Scratch's 'mix and match' tinkering style of development. A newly created engine for bot development could have online sharing and collaboration as a core, integrated part of the engine. This would allow AI developers to take a look at how others had achieved complex tasks or how they had overcome certain obstacles, which would mean more time can be spent on advancing new ideas rather than

reinventing the wheel.

3.2 Unreal Engine Blueprints: Visual Scripting System

Although not as much focus has been put on the Unreal Engine for inspiration for this project compared to Scratch, the relevance of the software to the area means it should still be examined. Unreal Engine is a professional game development engine initially showcased in 1998 as a first person shooter game engine. Since then it has grown and has been expanded until the most recent version, Unreal Engine 4, is now used for creating all types of games and other media.

Made "free to all" in 2015, Unreal Engine is now accessible by anybody no matter their skill level and fees are only taken if a product created using this engine starts earning enough money (Sirani, 2015). This policy has lowered the entry barriers for new aspiring game developers who are just starting out as it allows them to create their own projects without professional sponsorship to purchase the required software.

The Blueprints Visual Scripting System built into the Unreal Engine is a gameplay development scripting system designed to bring abilities previously restricted to professional programmers to all users. In addition to this, the Blueprints system is integrated with the C++ framework used in Unreal so that gameplay programmers can create baseline systems that can then be extended by other users of the Blueprint system.

The different types of Blueprints include:

Blueprint Classes - An in-game asset that allows functionality to be added to existing gameplay classes.

Data only blueprint - An instance of a blueprint class with no changed functionality from its parent, but allowed to have different variable values.

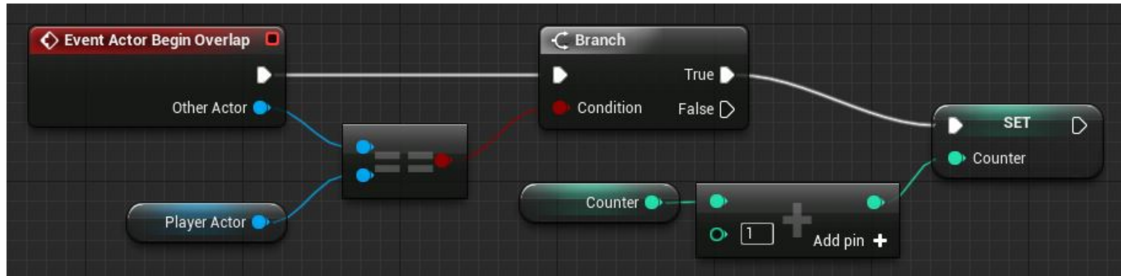
Level Blueprint - The global event graph for a game level. These are used to trigger events when certain actors interact in the level and are unique to each game scene.

Unreal Engine also supports Blueprint macro libraries which allow for custom nodes to be created that can be added to blueprints. Similar to writing a function in other standard languages these are used for time saving or for simplifying processes for other developers.

Blueprints are constructed using nodes representing actions and triggers similar to Scratch. As a contrast to Scratch however, Blueprints are not designed specifically for developers in a lower age group and as such can be vastly more complex in nature. As shown in figure 3 instead of having blocks that 'snap' together, blueprint components are connected via drawn lines that represent program flow. The white lines shown in the figure represent the flow of the program and functions as a direction to the compiler thus signifying that this block is executed once the previous one has been completed. The other colours of lines show either the flow of data, where information is pulled from to complete

the main line of the program or what conditions must be met before it proceeds. These different construction lines allow for the separation of data sourcing and program flow in the creation of a Blueprint.

Figure 3. Blueprints Example



Event Actor Begin Overlap (n.d.). Retrieved September, 2017, from ***https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/Events/index.html***

A user of Unreal Engine will be presented with an open empty workspace of with which they can add a node to start this part of the Blueprint, similar to the way a trigger block works in Scratch. From there they can draw a line from the output of this block and at the end of the line they will be presented with a context menu of what node they would like to add now. This flow of work allows developers to block out their software and determine what functions need to be in line, before sourcing all the checks and data that need to accompany the flow of execution.

3.3 Analysis of Scratch compared with Unreal Blueprint Scripting System

When analysing the successes of the two main inspiration points for this project, it is important to analyse the differences between the systems and to understand why certain design decisions were made.

The first contrasting point to analyse between them is the starting position in their designs for the user experience. The Blueprint Visual Scripting System is designed to be workable as a professional media creation tool. This is a distinct difference from the philosophy for Scratch which was made to be a creative learning environment for newcomers to programming. This difference is reflected in the complexity of the visual nodes that each software uses. Blueprints have an ever expanding library of different nodes that offer complex functionality options at quick access if you know what you are searching for. Whereas Scratch, as discussed earlier, was designed specifically to have a low number of easily understandable blocks so as not to overwhelm its users with too much choice. These differences show that the intended user of the visual development environment can make a crucial difference in development design. It is important therefore to establish the intended user base early on in design so that the planning stage can take into account the number and complexity of the nodes to be used in the language.

The next major difference between the two development engines is their relative scope and versatility. Scratch is rather limited in the scope of the projects you can create with it because it is mainly meant for smaller media projects. It is however very versatile in what type of media can be made within that scope and has been used to make everything from small games to instructional video tutorials. The main restriction it has is that it is limited to a 2D environment. But this restriction was purposefully made to ensure that the software is easier to use as developers do not have to worry about translations of objects in 3D space. Unreal Engine however is much larger in scale and allows for nearly any project to be created. It can support both 2D and 3D projects but as mentioned earlier, this versatility comes at the cost of complexity. The obvious differences between the two engines again show that the intended scope of what developers that use the environment should be able to create should be a key design point for the new system. If the system only needs to support one thing, then this could well be accomplished with the use of a simple system that does not need to have excessive functionality that can cloud the use of its one intended function. In the case of this project, a simple system would only incorporate a visual programming environment specifically designed for BWAPI. Adding any other functionality with the aim to make it more open-ended so that it works with more AI engines, would merely subtract from its capabilities and usability with BWAPI. Using the simplified coding block style shown in Scratch will allow for easier development with easy to understand visual elements, while the environment design aspects from Unreal Engine will cope better with the complexities of working with the environment that BWAPI creates.

4 Specifications

Having investigated the ways that other tools have been developed to ease the learning experience of new comers to programming, the next step is to look into how these lessons can be implemented for BWAPI.

4.1 Aesthetics

While traditionally not an important part of developing an API, the aesthetics of a new Visual Programming Language is important to be planned out. This is due to, as implied by the name, the majority of the user's interaction with the system being visually based. The first point to look at would be the development window itself. Scratch and Unreal Engine have different ways of approaching this. As Figure 4 shows, Scratch has a large portion of the screen dedicated to the live game view, with the rest of the window taken up by a large scripting area, the tool selection menus and the asset selection bar. The design is focused on the importance of attempting to provide immediate feedback to the developer on the effect of their most recent changes and how these changes affect the game world they are making. As a counter alternative to that model, Unreal Engine has almost all of the window dedicated to its scripting area while the developer is editing the Blueprints. The developer has to switch out of this view to see the game world, but while editing Blueprints the focus is very much on the scripting environment. To access the scripting blocks, a user must click in the scripting area and use the context menus to search for the required command structure or block.

Figure 4. Scratch Programming Environment

Screen-shot of Example Project taken in Scratch Editor January 2018. Example project supplied by Scratch Team
https://scratch.mit.edu/projects/editor/?tip_bar=getStarted

If a visual programming environment based on BWAPI is made, then it is exceedingly unlikely without serious extra development and work with the underlying system, that a live game view window would be practical. This is due to the nature of the API being instructions that are injected into a running game of StarCraft, and not it not being executable in any sense without setting the whole system running. Due to this, the visual aspect of the environment is going to be a balance between the two extremes presented by Scratch and Unreal Engine. There would need to be a large portion of screen dedicated to the scripting area for development with no game view like Unreal Engine, but, as with Scratch, the code blocks would be easily visible in menus for adding into the scripts. The reasoning behind this decision is to help reduce the amount of learning that a new user has to do on remembering programming blocks and structures. If all the programming blocks are visible and are separated in meaningful menus by category, then the developer only has to remember what type of block they need and what it looks like to have a chance of finding it, rather than having to remember each block by name to search for it.

With Scratch, great care has been taken to ensure that the different categories of programming blocks have unique shapes in the menus and scripting areas. As previously described, Control Structures are open middled block that represent the fact that they enclose other blocks, and the Command Block is a basic shape that can attach to the bottom of other Command Blocks or can go in, before, or after a Control Structure. The shapes of the blocks also help with the process of how to construct a script as they fit together only in ways that make syntactical sense. With Unreal Engine, the blocks themselves do not fit together. Instead they are attached together with lines that direct the flow of the program. The blocks are also all the same basic shape, just with more input or output points dependent on the type of block. For the proposed new Visual Programming Environment, the approach taken by Scratch would be the better option in theory. The simplicity offered by having the blocks snap together in ways visually representing the programming concepts they replace is invaluable along with the possibility of having the language structured in such a way that means instructions cannot be put together incorrectly.

A potential drawback to this inter-fitting block approach is that issues may arise over the visual interpretation of the intended flow of data. With Unreal Engine, the data required as an input for a block is shown as being connected by a line from another source where as the only way of passing data in Scratch is with variables, which are inserted as Function Blocks into other commands. With a Scratch based system, it is possible that the system would be too complex with a variable required for every piece of data that is to be passed from block to block, and that another solution would be needed. One possible solution would be to use something similar to the solution Unreal Engine has, with lines from block outputs to other block inputs.

4.2 Mechanics

The visual description of the blocks, while important of course, is not as crucial as establishing both what the blocks are going to do, and how the user is going to interact

with them. The intention of this new development engine is to have the AI developer be able to create a bot that will work in the game "StarCraft: BroodWar", without the developer spending large amounts of time studying existing API's and techniques. The command blocks in the new engine should therefore be self explanatory, and self contained. What this means in practice is that the function of any given block should be clear from its name without needing to go extremely in depth in its description. Being self contained means that there shouldn't be any block that is impossible to use without the presence of another block. This would mean that a AI developer could browse through the menus of command blocks and find one that sounds like what they need without concerning themselves with thinking about having prerequisite blocks already in the script or project.

Having defined the basic requirements for a programming block the next step is to determine the level of abstraction the different programming blocks will take over the existing API which will then determine the kinds of operations that these programming blocks will execute.

Currently when using BWAPI, there is only one level of built-in assistance for an AI writer over and above what a player of the game would have access to themselves. This assistance is a small utility that does not come as standard in the API but is available on the website for basic instructions in using it. This function contains a way to find the closest area where a building can be placed that a builder can get to. This function will be used as part of this project as it isn't an aim to recreate this function and it isn't possible to create buildings without using some version of the function. Other than that there are no development functions that a player of the game doesn't have access to. This means that everything must be done explicitly in programming by every new AI developer that uses the API. Consequently, there is no way of tracking any unit being created, without building such a system and there is no way of keeping track of what enemy buildings you have observed, or any other such capabilities. Each of these systems would be a challenge in itself for the prospective bot creator but some of these tasks could be simplified into multiple blocks so that a new AI developer could put a working bot together with some level of customization with relative ease. With this project the aim is to give these developers a library of tools to create their bots how they want, using their own ideas, but without them getting immediately bogged down in some of the more intricate parts of the API. Another important part of the mechanics of this project would be the option for a AI developer to create their own blocks that could add to or replace existing blocks or be just added to the library. This functionality would cater for whenever the developer wanted something that currently isn't created or if they wanted one of the current blocks to work slightly differently. The aim is to ensure that the new tools always enable more free bot creation and not restrict it.

The starting point when drawing up a list of the possible main features this library of tools should contain, was to approach part of the current development community which already uses the current API with a survey. This survey listed a selection of different things they would have liked to be easier or done for them when they first started out using the API. Below is the list of options presented to them, with an explanation of what each meant.

Production Queue
Buildings and units can be added to the queue with a priority level. The queue will then sort by priority and manage saving up resources for the item at the top of the list. Once resources are available it will either assign a builder to construct a building or when all the required buildings are complete, it will start training the unit.

Base Manager
Stores a list of buildings that should be in each base. If the given base doesn't have that building then notify the bot that it needs certain buildings, so the bot can decide how to fulfil the base needs.

Builder Manager
When a construction job is ordered, a builder is assigned to it. If this builder is killed than another free builder will be assigned and sent to work on it. When the builder is finished constructing it will be sent back to mining minerals.

Basic Map Exploration
Units assigned the task of map exploration will visit all possible base spawning positions as a basic search path.

Squad Management
A basic implementation that holds groups of units. This allows for collective objectives and the squad will request replacement units if any are killed.

Basic Combat Micro
A simple implementation of micro for units. Possibly including activating abilities and falling back when the average health of the squad drops too low.

Enemy Base Tracking
Keeps track of enemy base locations and their respective make up of buildings for later use. This can be useful for analysis or for targeting of attacks.

Enemy Unit Tracking
Storing of enemy unit makeup to allow for analysis and creation of counter strategies.

After these options in the survey were open questions allowing for more general responses about what people would want improving. The responses to this survey are outlined in table 1.

Something of note from the responses shown in this table, is that every suggestion was marked as detrimental at least once. Upon examination of individual responses to the survey, it was found that one participant had marked each suggestion as negative while remarking that augmenting the API defeats the purpose of writing an AI. While not discounting this person's views, it is of note that these negative responses all came from one

Table 1
Questionnaire Responses

Suggested Feature	Responses			
	Detrimental (1)	Not Useful (2)	Useful (3)	Don't Know (0)
Production Queue	1	-	7	1
Base Manager	2	2	3	2
Builder Manager	1	-	6	2
Basic Map Exploration	1	-	6	2
Squad Management	1	-	6	2
Basic Combat Micro	1	-	6	2
Enemy Base Tracking	1	-	7	1
Enemy Unit Tracking	1	-	5	3

person while as a whole the survey and its ideas were well received. Also of note is the relatively small number of responses received, so while this data is from current AI developers that use the API, there is potential for misrepresentation due to a small sample size. Table 2 shows each of these features listed by order of their average response score, with the higher number representing a better response. If they had the same average score, then they have been ranked by my personal preference of the features I would have liked to exist when I first started using BWAPI.

Table 2
Sorted Average Response

Suggested Feature	Average Response
Production Queue	2.75
Enemy Base Tracking	2.75
Builder Manager	2.71
Squad Management	2.71
Basic Map Exploration	2.71
Basic Combat Micro	2.71
Enemy Unit Tracking	2.67
Base Manager	2.14

It is clear that a lot of the responses had similar results as to what people would want in a new API. Due to time related restrictions, only four of the results will be

augmentations that are focused on, and as such it will be the top four shown in the above table. Each of these features will now be further examined and their specifications expanded.

Production Queue
Survey brief outline:

1. Buildings and units can be added to the queue with a priority level.
2. The queue will be sorted by priority level.
3. The function will make allowances for saving up the required resources for the item at the top of the list.
4. When resources are available, a builder will be assigned to construct a building, or when all the required buildings are constructed then the unit will be started.

Since writing the survey, it has become apparent that this specification needs amending to include the point

5. If a building is destroyed mid construction or a unit or research is cancelled by the producing building being destroyed, then the respective building will be changed back to a status of "Unstarted" in the construction queue.

The first specification will require a list of objects that stores different construction and production requests. These requests will come in one of three different styles. A building or unit, a research, or an upgrade. Each of these styles will require separate objects on the list because they are all worked on and completed in different ways. The completion of each would be tracked by different triggers in game. A building or unit being finished, a research being completed or an upgrade being finished all trigger different in game events and as such need to be monitored for differently. These 'production orders' as they will be called, must also store the priority level of the order, the time that they were requested, and the current status of the order along with the details of what it is to produce. Priority level will be recorded with a higher number meaning a higher priority. This is to allow for ease of adding a new item with a higher priority than all current items in the queue without changing all existing priority levels to accommodate. The queue must then be sorted automatically by priority level, and then by time requested every time that the list changes. This will enable important buildings to be 'fast tracked' through production if required by the bot.

Now we have a sorted list of the three different types of production, we can address saving up resources to produce the top item on the list. For each production order, there will be an associated cost embedded in the details about what the order is for. This cost could be in the form of minerals and/or vespene gas, but by the nature of it being at the top of the priority list, it is the most important thing to create at that moment. As such if the bot is not able to produce what is required yet, then it should save up its resources until it can. This is achieved simply by not allowing the bot to spend its resources on anything but the top, "Unstarted" item on its construction queue.

Along with the cost of each production order, there will be a list of prerequisites to its production. This list is provided by default through the base API. If it is attempting to construct something before all the prerequisites are in place then nothing in game will happen. To address this, there will be an option when adding a production order to the queue, enabled by default, that will add any missing prerequisites also to the queue before the item that was initially to be added. When interrogating the list to see what item is the current one to save up for, all items without the prerequisites met, will be ignored. This is to prevent the bot being stuck, unable to produce anything as it is not allowed to spend resources on something that is not the top item of the list, but unable to produce the top item of the list as it is waiting for one of its prerequisites to be made, which has been placed further down the priority list.

Once the correct item has been put to the top of the queue, and then enough resources have been saved, the production order can now be started. For a building, this requires acquiring an available builder and telling it to begin construction. For units and upgrades, it requires telling the appropriate building to begin working on it. Once this is done, the task is started and should be marked as such in the production orders status. Items will only be removed from the queue once completed, but they should be marked as "Begun" because the item is still at the top of the queue and as such nothing else will start while it is still there. If this is done and "Begun" items are ignored when scanning the list, then the bot can move onto the next thing to start working on.

Finally, if a building is destroyed mid construction, it is possible to detect that through a trigger on building destruction. If this happens then a check can be run to see if that building was one of our ones that was being constructed. If it was, then the production order in the construction queue needs to be amended to say not started so that the queue manager will re sort it into what should be built. It can also be checked to see if a building destroyed was producing a unit or researching something for the queue. If so, that can also be changed back to an "Unstarted" state.

Once a building/unit/research is complete, the production queue will be scanned to find the appropriate production order and it will marked as complete and removed from the queue.

Enemy Base Tracking
Survey brief outline:

- Keeps track of enemy base locations and the buildings in their make up.
- Allow access for possible interpretation or attacking targets.

This is, in principle, a simple augmentation to implement. The base API has a trigger for when a building or unit is discovered. On this trigger, a function can be called to check the discovered buildings unique ID against a list of known enemy buildings, and if it's not on there then append it to the list. The list must be checked to ensure than no multiple entries because a trigger happens whenever the building appears from the fog of war and as such could be discovered multiple times as the fog comes and goes. This list will then be

made available for interrogation for finding out if the enemy has a certain type of building, or other needed details of the enemy buildings. The list will also be updated by removing items from it whenever a building is destroyed, or is discovered to be gone.

Builder Manager
Survey brief outline:

1. When a construction job is ordered a builder is assigned to it.
2. If this builder is killed then another will be assigned to it.
3. When the construction is complete, the worker will be returned to mineral mining.

It is important to note that this part of the library would have to include radical differences to allow for the other two races of StarCraft that will not be included in this project build. This build will focus on the Terran race, where a builder unit will work on a building until completion. The other two races are different in that the Zerg's builders actually morph into their buildings and as such are expended whereas a Protoss builder only needs to be there to start a building, then the building will construct itself while the builder is free to do other things.

To have a builder be assigned to a construction job, there must be a construction job object that will contain reference to that worker along with the other details of where and what will be constructed. This object will be created when a building is deemed ready for construction by the Construction Queue, or can be created manually, then will be put into a list of ongoing jobs. A builder that is currently free, or mining minerals will then be chosen for the job and a reference added, while the builder is given the command to begin construction according to the details passed in.

Assigning a new builder if ever the current one is killed is another task simple in concept once the surrounding framework is there. When a unit is killed, a trigger is raised in the base API, allowing for the catching of this event and checking to see what the unit was. If the unit was one of our builders, then a check can be run against all current construction jobs to see if it was the builder assigned to one of them. If it was, then a new worker can be secured in the same way that the original one was by pulling it from mining or from idle.

Buildings being completed is another trigger that can be monitored in the base API. When receiving this trigger, a check can be run to ascertain if the building was one of ours and if so, which construction job it was. When a construction job is found to have been completed, the assigned builder can be returned to mining, and the job object can deleted from the list.

Squad Management
Survey brief outline:

1. A simple implementation that holds a collection of units

2. Allows for collective objectives
3. Management to request replacement units if any one is killed.

With the current API, controlling units involves giving each unit individual orders. Consequently if you wanted a group of five marines to move across the map, you would have to select each one in turn and individually set its movements target. Having squad logic allows for a squad of units to be requested and created as a batch, then once each unit is created it can be added to a squad object containing a list of all the units in it along with other details. A target or destination can then be given to the squad object and the squad manager would then relay the movement target to each unit in the squad.

The squad object will hold a list defined by the AI developer of units that is the desired make up of the squad, a list of current units it has, and the current objective of the squad. On request, the function will return the difference between the two lists so as to see what is missing from this squad. The manager could automatically place requests to the construction manager if enabled, but otherwise will only report what it needs.

The squad objective or target will be able to be provided with additional parameters pertaining as to whether the squad will move straight there or will engage enemy units on the way. The squad management function will then either use the inbuilt function of attack move or the normal move for each of the units in the squad depending on which was selected in the parameters.

5 Design

For each of the four main augmentations listed in section 4, below are more technical designs and diagrams of how they will be implemented and how the different parts of the library will work together.

5.1 Production Queue

Figure 5 shows the class diagram of the production queue. With the relations between how the different types of production order will inherit, and how they will interact with the main class.

Figure 5. Production Queue - Class Diagram

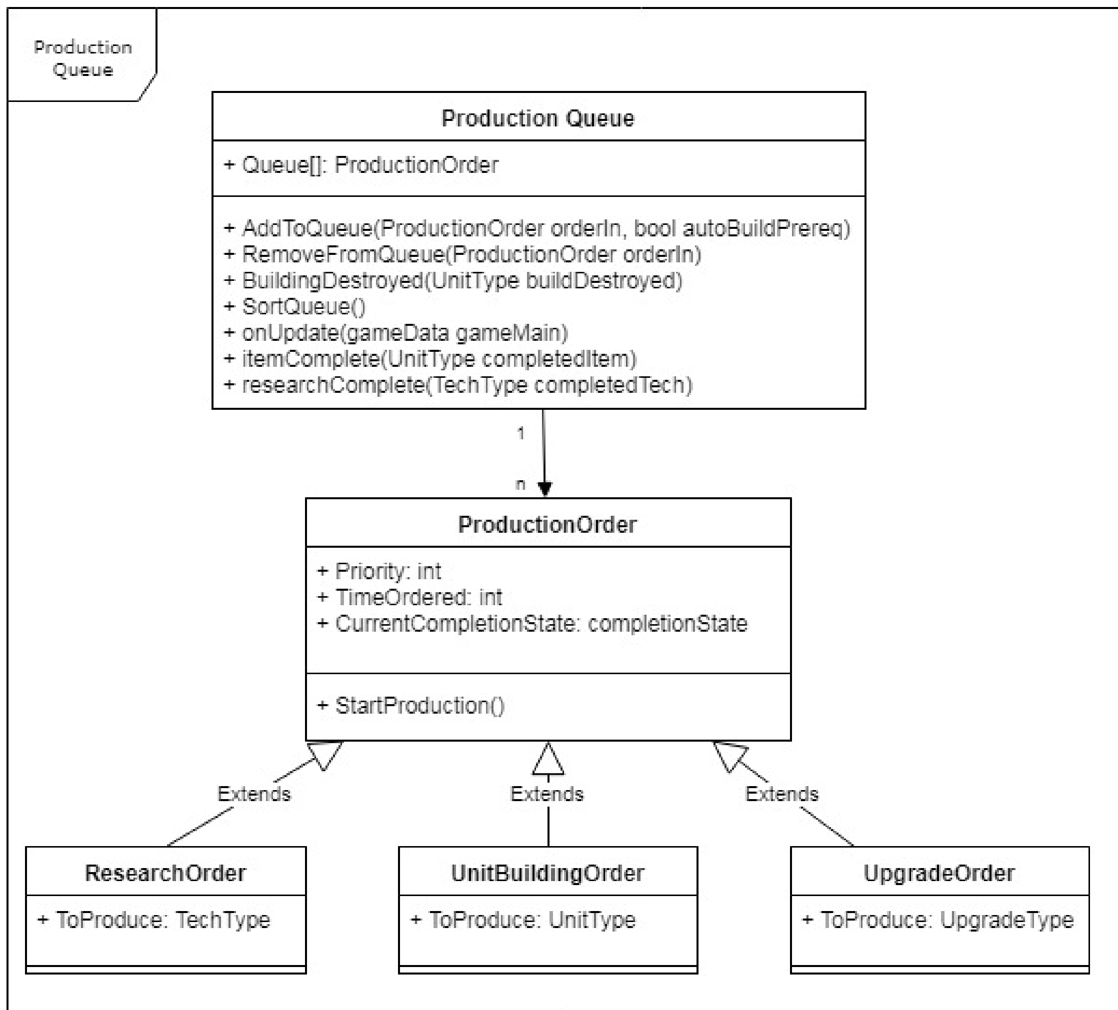
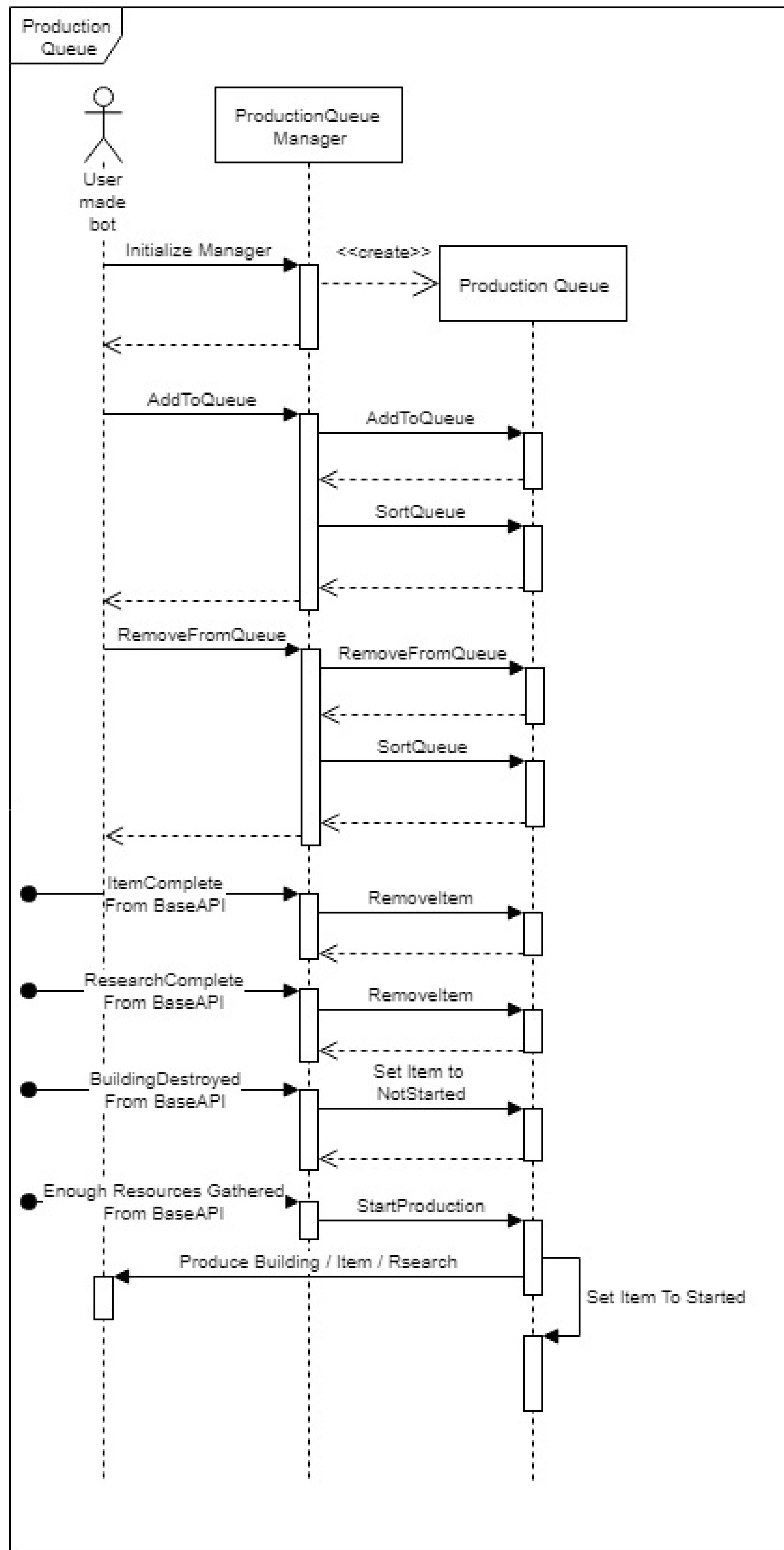


Figure 6 shows the sequence diagram explaining how the Production Queue Manager will interface and be interacted with by the AI developer's bot, how triggers that come from the base API are handled, and how the Manager will interface with the actual Production Queue.

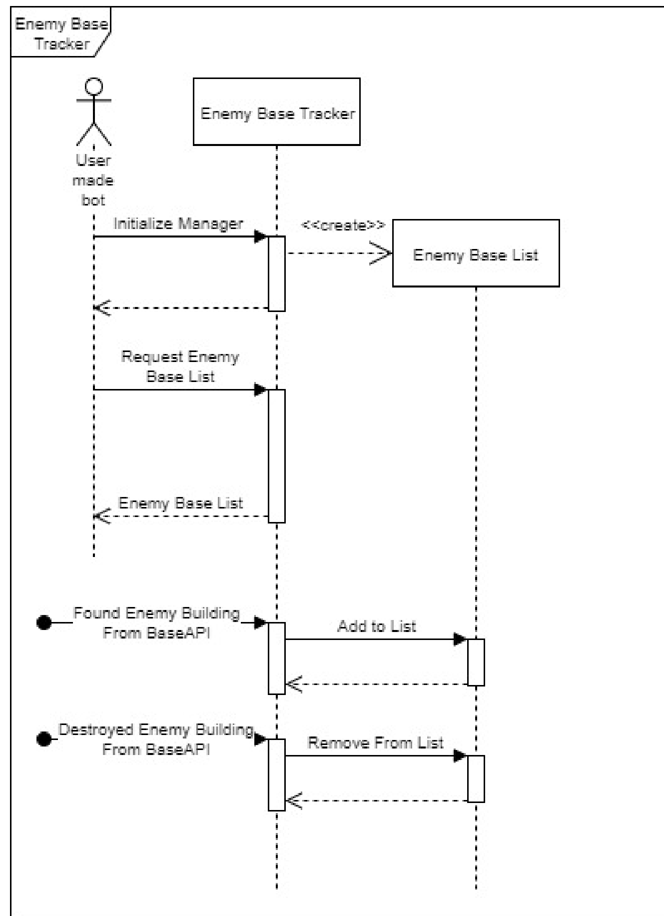
Figure 6. Production Queue - Sequence Diagram



5.2 Enemy Base Tracking

Figure 7 shows the sequence diagram of the Enemy Base Tracking block. There wasn't a need for a class diagram for this section as it would be contained in one class, but the sequence diagram shows how it interacts with the AI developer's Bot, and the base API.

Figure 7. Enemy Base Tracking - Sequence Diagram



5.3 Builder Manager

Figure 8 shows the class diagram outlining the Manager, the Construction Job and the shell for the builder class.

Figure 8. Builder Management - Class Diagram

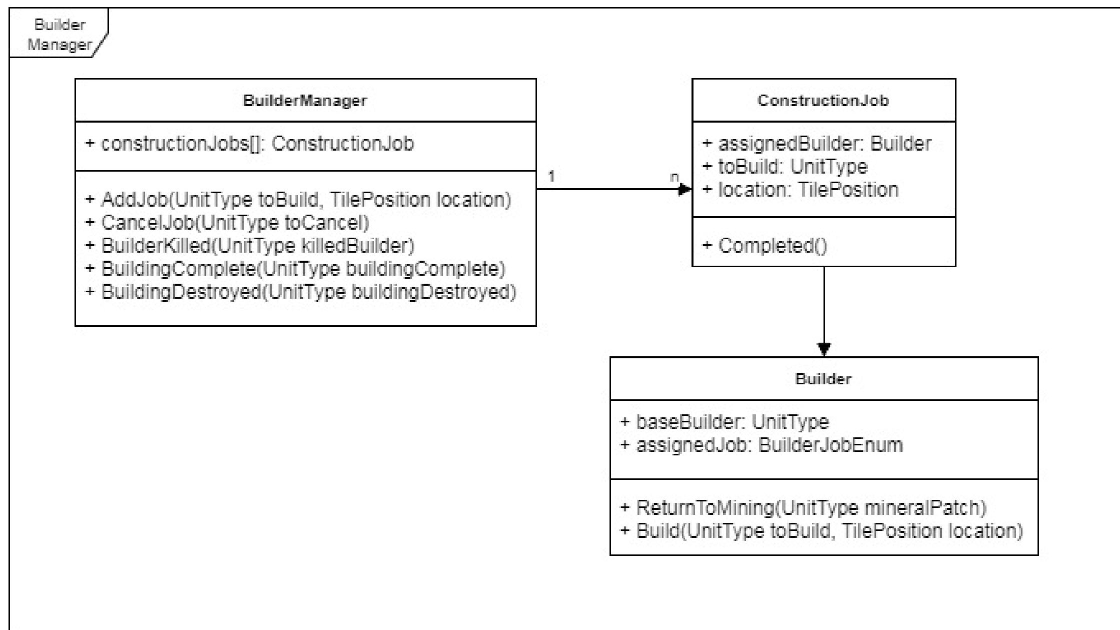
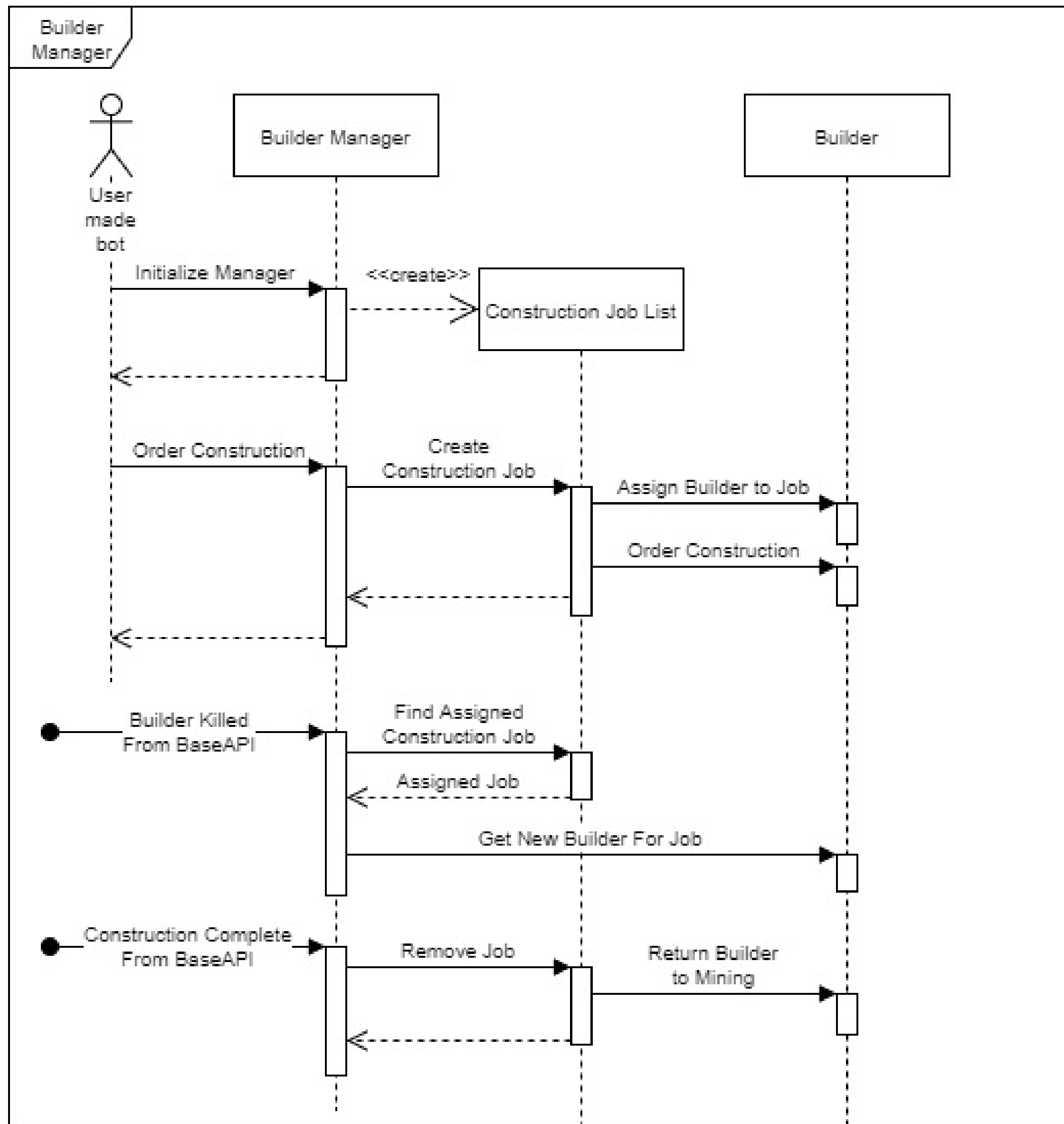


Figure 9 shows the way that each task is dealt with including the triggers from the base API.

Figure 9. Builder Management - Sequence Diagram



5.4 Squad Management

Figure 10 shows the two different classes of the squad manager and what each would contain.

Figure 10. Squad Manager - Class Diagram

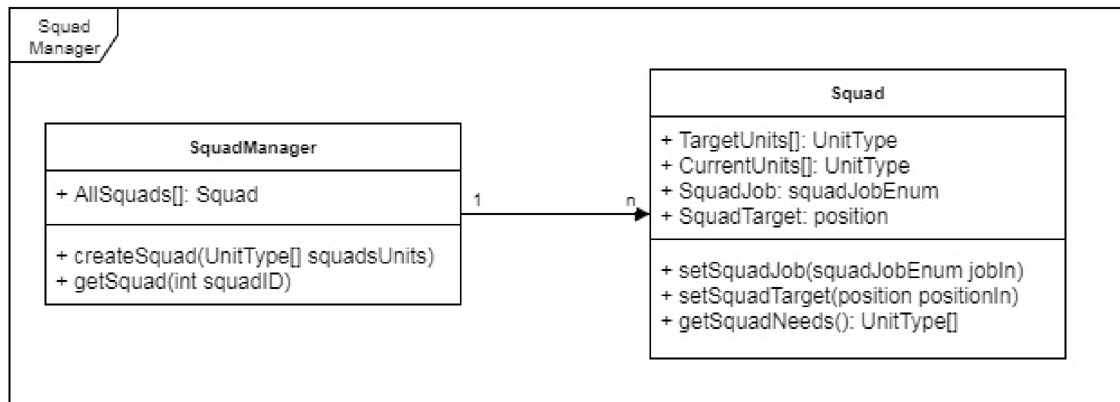
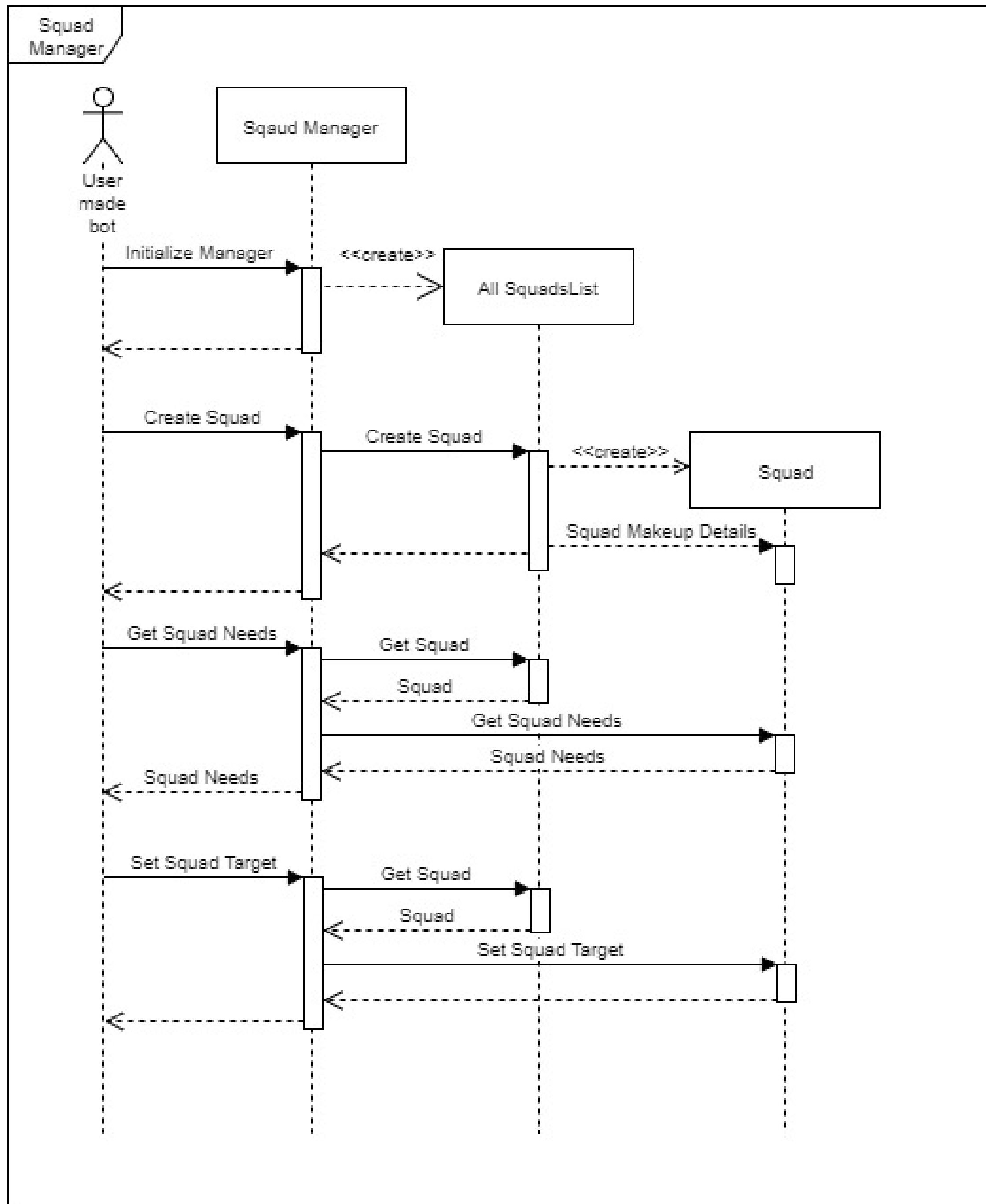


Figure 11 shows the way that each task is dealt with including the triggers from the base API.

Figure 11. Squad Manager - Sequence Diagram



6 Testing Plan

In normal software development, testing is conducted in two major stages. These stages are known widely as white and black box testing (Khan & Khan, 2012).

White Box - With a working knowledge of the code base, an in-depth check of logic and code sections is run, checking inputs and outputs to the different code methods and classes.

Black Box - No code base knowledge is required for just the output of the entire program is analysed and compared to known correct results.

Often, testing is done as a suite of individual tests with results for each of them checked against the expected results from that test. With Java systems, a common way of achieving the white box testing is to use a suit of JUnit tests (Do, Rothermel, & Kinneer, 2004). These tests are made to be run after each large change to the code base to ensure that new changes don't cause faults in previously tested code.

When developing with BWAPI, the use of normal testing procedures is not as easy as when developing in other environments or circumstances. This is due to the innate requirements that BWAPI needs to run. Any time that BWAPI is to be run, it needs to be run simultaneously with the base game of StarCraft. Because of this, no small scale localised system tests can be run en masse unless a way of running the game engine for interrogation was developed. Also of note is that for a lot of bot development to be tested, the bot needs to have been run from its beginnings and set up to have the correct frame of reference and responses triggered for any given test to be valid. For example, there is no point in running a test on a bot's ability to build end game units, when the bot has not got through to the end game and does not have the in-game capabilities of doing so.

For testing the developed new library of functions made for this project that are for use alongside BWAPI, a variation of standard white box testing is used. This variation involves writing a series of function tests for the different planned packages of the library. Most of these tests are in the form of different configurations of a test bot built using the new library packages. If certain parts of the library can only be tested under specific circumstances, then a custom game is created containing the testing bot and the human tester as a player, to be able to force the game to go into the given circumstances.

Below, each specification point is listed, together with an explanation of what test was run and how that proves the capabilities of the specified functionality.

6.1 Production Queue

6.1.1 Buildings and Units can be added to the queue with a priority level. To test the first and most crucial point of the production queue, the testing bot, created to interact with the new library, is instructed to call the methods to add different types of items to the queue. These multiple calls are made to request the different types of order

that can be made, and also represent a variety of priority levels on the orders. The bot requests the production of the following items, in this order.

- Stim Packs Research : Priority 1 : Research Order
- One Terran Marine : Priority 2 : Unit/Building Order
- One Terran Barracks : Priority 3 : Unit/Building Order
- Terran Infantry Armour Level 1 Upgrade : Priority 1 : Upgrade Order

For the test, one frame of game time elapses between each order so that the secondary sorting metric of order-time is shown to be relevant.

This test shows the three different types of order that is created by what production the AI developer requests. As shown in figure 5, the three different order types are Unit/Building Order, Research Order, and Upgrade Order. Each of these are covered by the requests made by the testing bot, with the Unit/Building order shown twice to demonstrate both types of production.

On running this test, the production queue receives each request and interprets it by creating a new order of the appropriate type, then adding the new order to the production queue. This is demonstrated by the on-screen display option for the construction queue. With this option enabled, each item in the queue is displayed along with its details of what is ordered, its priority, time ordered, and its status. This allows a tester to see the items in the queue, what order they are in, and what status each order is in. When the test is run, the queue correctly shows on screen, four separate items in the queue, each with the details and associated priority levels.

6.1.2 The Queue will be sorted by priority level. Using the same test set-up as the previous point with the four different production orders, the sorting of the queue can also be tested. The production queue is sorted by priority and order-time every time that an item is added, edited, or removed. This means that by adding four orders to the queue, the list automatically sorts each time.

The test bot is run with the order requests in the sequence shown by the test above, and this produced the following output of list of items on screen once all were added.

- Barracks : Priority 3 : Time 3
- Marine : Priority 2 : Time 2
- Stim Pack Research : Priority 1 : Time 1
- Terran Infantry Armour Upgrade : Priority 1 : Time 4

This list shows that the Barracks order, which was issued after the Marine order is sorted above it due to the higher priority attached to the Barracks. It also shows that the Stim Pack Research is sorted above the Armour Upgrade, because even though they have the

same priority level, the Stim Pack was ordered before the Armour Upgrade and as such should be higher in the list.

6.1.3 Automatic prerequisite items requests. This function is designed to check to see if all prerequisites for the requested item are met, and if not, add them to the queue. This function is enabled by default on each add to queue call, but can be manually disabled if so desired. The test for this function incorporates several iterations. Each iteration tests different possible scenarios of when the function could be called.

- Requirement not met and not ordered: Order a marine when there is no barracks complete, and no barracks in the production queue.
- Requirement not met, but under construction: Order a marine when there is no complete barracks, but one is in the production queue.
- Requirement Met: Order a marine when there is a complete barracks.

Then as a final test, the bot orders a unit that has at least one prerequisite that also has its own prerequisites. This tests the self-recursive nature of the function that is designed to make sure that all requirements are met, not just the direct ones.

For each of the initial tests, the bot adds a marine to the production queue under the differing circumstances. The first test is to ensure that the function adds a Barracks to the production queue because a Barracks is the prerequisite for a Marine. The Barracks is added once the function has ascertained that there is not one already complete or one already in the queue. The other two initial tests have no additional items being added to the queue as the barracks was in the queue already, or was complete. The final test has the bot order a BattleCruiser. This is considered a late game unit due to its long list of requirements, and some of those requirements themselves having prerequisites that need to be met before they can be constructed. The test results show that when a BattleCruiser is ordered, the function adds the following extra buildings to the queue:

- Barracks
- Factory
- Starport
- Science Facility
- Physics Lab
- Control Tower

Of these added orders, only the Starport, the Physics Lab, and the Control Tower are directly listed in the API as requirements for a BattleCruiser. As these three orders for buildings are added, then their own prerequisites of the Barracks, Factory, and Science Facility are also automatically added by the function.

These four tests show that the function recursively adds prerequisite orders to the queue when a new order is made, as long as they are not already in the queue or have been already created by the bot.

6.1.4 When resources are available for an order, a builder will be assigned to its construction, or the order will be started at the appropriate building. During development, this part of the library was changed slightly, so that instead of the order being handled automatically when the resources and other requirements are met, the production queue instead passes the order details back to the AI developer for them to execute themselves. This change was made to keep the library package for the production queue more self contained and to give more control of how things are executed back to the AI developer. After all, this project is about providing aides to the AI developer, and not doing everything for them.

To test this part of the library, the test bot is set up to start the mining of both minerals and gas, and then to request a selection of different items. Then the tester needs to observe the current amount of resources the bot has, and the unit and building availability of the bot to manually check when the first item on the production queue would be possible for production. Then check to see if the production queue has returned that production order marked ready for execution.

A test of setting the bot mining and making the same orders set out earlier was run, with the following being the queue once all items are added and the queue is sorted.

- A Barracks
- A Marine
- Stim Pack Research
- Armour Upgrade

The expected outcome would be that nothing would be returned by the queue until 150 minerals have been gathered by the bot. This is the cost of the barracks, which is the first item to be executed as it is the top item in the queue, but also due to the prerequisites for the other items not being met. When 150 minerals are gathered, the production queue returns the production order detailing the barracks to the AI developer, marking this as being ready to execute. Then, to continue the test, the barracks is constructed manually by the tester, which is automatically tracked by the production queue and then removed from the queue once completed. After the barracks is completed, and another 100 minerals had been gathered, the order for the marine was returned by the queue, as now all of its prerequisites have been met and enough resources have been gathered.

This test shows that when an order has all of its requirements met, the production queue sends that order back to the AI developer to be executed as and when they deem fit. It also shows that the production queue tracks orders that have been started, and removes them from the queue upon completion.

6.1.5 If a building is destroyed mid construction, or a unit in production, a research, or an upgrade is cancelled by the producing building being destroyed, then the corresponding order will be changed back to "Unstarted" status in the queue. To test the functionality of this specification, a custom game is set up between the bot and the tester. This allows the tester to destroy buildings as they are being constructed, or as they are producing other orders. The test is carried out by having the tester send a group of units to destroy the bot's barracks while it is producing a marine. The order for the marine, before the destruction of the barracks, has the status of "Started" to signify being in production. After the destruction of the Barracks, the order is changed back to "Commissioned" which indicates that is has been ordered and is in the queue, but has not yet been acted on.

This test demonstrates that the destruction of a building is recorded by the production queue and then interpreted to see if it was relevant to current productions. If it was, then the appropriate records are updated so that, when ready, the orders can be given back to the AI developer again for them to be re-executed when possible.

6.2 Enemy Base Tracking

6.2.1 Keeps track of enemy base locations and their respective make up of buildings. During development it was discovered that while creating a manager that will track enemy base buildings, it was the same principal to have the manager track enemy units at the same time. The tracker will store each building and unit discovered, along with the last known location of each of them when they disappear into the "fog of war".

To test this system, another custom game is set up between the test bot and the tester. This one has the test bot send a unit to where the testers' base is, to be able to discover their buildings and units. The exploring unit is then to retreat far enough to where the tester's units are again hidden by the fog of war. Doing this allows the enemy base tracker to see the enemy buildings and units, and add them to the records. This is shown on screen by a drawn green circle around the location of visible enemy units and buildings. This circle then changes to red when the unit or building is now hidden by the fog of war, to signify that it is only displaying the last known location and not live data.

This test shows that the base tracker can recognise an enemy building or unit and add it to the list. It also shows that if a building or unit is found, it can check to see if it is a new unit or if it was one that was previously found but hidden. This is useful to make sure that the same unit does not end up on the list multiple times by being discovered in multiple places.

6.2.2 Allow access for possible interpretation or for attacking targets. Using the same test setup as the previous test, with the bot facing against the tester and sending out an exploratory unit, this test consists of having the bot query the Enemy Base Tracker to find out what the Tracker has learned about the enemy during the scouting.

After exploring the enemy base, the bot is interrogated to see if it has a record of the enemy's main command building, the enemy's builders, and also how it responds when queried about a building that the enemy did not have at the time of the exploration. These interrogations show that the records hold both units and buildings, and return a false when asked about buildings that the bot did not observe an enemy having during its scouting.

6.3 Builder Manager

6.3.1 When a construction job is ordered, a builder is assigned to it. For this test the bot is set up to wait until it has the resources and builder available to produce a building, and then to order a construction job for that building by using the Builder Manager. This method does not wait for any conditions nor does it test that any conditions are met but immediately allocated a builder to the task. For this reason, the method also requires the intended location of the building to be constructed.

The results of the test show the job being created and added to the job list, by the closest builder being acquired from among the free ones or the ones mining minerals, and the worker being sent to construct the requested building in the requested location. This test demonstrates that the manager is capable of getting the closest 'free' builder and instructing it to build the requested building.

6.3.2 If this builder is killed then another will be assigned to it. Similar to testing the implementation of when a building was destroyed mid production for the production queue, this specification is tested by having the bot in a game against the tester. The bot is set up to start building a sequence of buildings while the tester sends units to kill a builder while the buildings are being constructed. The tester withdraws their units to observe if the bot sends a new builder to continue the construction work.

Running this test shows that the bot, when using the Builder Manager, is able to recover from a builder being killed mid construction, by sending in a new builder as a replacement.

6.3.3 When the construction is complete, the worker will be returned to mineral mining. To accomplish task, the library forces all idle workers to go mine minerals, and so to test this functionality a few different scenarios of why a builder has become idle need to be created.

- The builder has just been trained and is idle outside the command centre it was made at.
- The builder has just finished constructing something and is now idle and is adjacent to the completed building.
- The builder was moved to somewhere and is now idle upon reaching the destination.

For each of these tests, the bot's workers are manually put into these situations either by moving them, or by performing the appropriate production actions. Each test shows the

worker being moved to mining at the closest mineral site after it was discovered to be idle by the builder manager.

6.4 Squad Management

6.4.1 A simple implementation that holds a collection of units. The testing of this function specification is less straight forward than some of the others, as the method doesn't do anything but only holds information. So to prove the ability of creating a squad and that the Squad comprises of the units it has been assigned, a test is run that creates a squad, then requests the units required to complete it, and once these units have been built it assigns them to the squad. Finally the bot then interrogates the squad object to ascertain its status of completion. When a squad is created, the list of desired units is passed in to the function. The function records the passed in list of desired units and responds upon request with a list showing the disparity between the desired units and the currently assigned units.

For this test, a simple squad composed of four Marines and two Medics is commissioned. The bot queries the function to ascertain what items are missing from the squad and instructs the required units to be created. Upon completion of each unit, the squad manager is notified and the unit is then assigned to any squad that needs a unit of that type. When all units are trained, the squad is queried to see what its status is, and it responds that it is complete. This will only happen if the comparison of the desired composition list and the list of currently assigned units shows no discrepancies.

This test shows that the squad manager can create a squad object that will hold the details of what its desired composition is, the details of its presently assigned units, and can respond to status queries.

6.4.2 Allows for collective objectives. To test this requirement, a squad is created in the same way as the previous test. Then, upon completion, the squad is given a location on the map to move to. After the squad completes its movement objective, the squad is given a target to collectively attack.

These simple tests show the ability to use the two different squad command modes, movement and attack, to make the whole group of units move to an objective with one command, instead of the AI developer having to set up a way to give commands to each unit individually.

6.4.3 Management to request replacement units if any one is killed. For this test, again a testing environment is set up with the testing bot playing against the tester. Note that, the squad manager does not automatically place orders for replacement units, it instead updates its response to the query about whether it needs any more units to be complete. This maintains the independence of the package in the library and allows it to remain self-contained. Consequently, the testing bot is set up to create new units if the Squad Manager replied to a query that it needed a replacement unit after one had just

been destroyed.

To test this, a squad is made in the same way as the previous two tests, then the tester sends units to destroy one of the units in the created squad. After this happens, a request comes through for a replacement unit of the same type as the one that was destroyed. When the replacement unit is completed, it is automatically assigned to the squad.

This test shows that the lists of assigned units in the squad is updated when a unit is destroyed, and that when0 this happens the Squad requests a replacement unit next time it is queried.

6.5 Example Bot using New Library

As the final internal test before moving to external evaluation, a full test bot that incorporates all the new library packages is made. This bot is not designed with a strategy that will win a full game, but rather it is developed to demonstrate a basic game progression plan that showcases each capability of the library.

This test bot is made to show the simplicity that the new library can bring to a AI developer's bot coding, with each package simplifying different areas. By completing the relatively simple game plan that the example bot has, each of the four packages created for the library are showcased. Below is an outline of the bot's game plan, along with an explanation of which library packages it incorporates.

On game start, the initial idle workers are set to mining by the Builder Manager while also being added to the Builder Manager's list of workers. Then, whenever there is not a worker in the production queue, and the current number of workers is less than the target number of workers, the example bot commissions another worker in the production queue.

The test bot is set up to read what is returned from the Production Queue, to see what is ready for being worked on. The test bot will then either tell the appropriate building to begin production, or it will instruct the Builder Manager to start construction of a new building using a function that returns a possible build location. [1]

As more workers are produced, the supply used amount will increase and as such, the bot is set up to automatically add a supply depot to the production queue whenever the supply available count gets too low. Once a usable count, in this case 10, of workers are completed, a squad of four Marines are commissioned at the Squad Manager. Then, by using the result of querying the Squad object about what it requires to be complete, the four marines are passed to the Production Queue. By adding a Marine, the Barracks is automatically added by the queue as it is a requirement for producing a Marine. A Refinery and a Bunker are also added to the queue. By adding all this to the queue in one go, and leaving it to the Production Queue to manage when the bot actually produces

[1] Example BuildTile Function provided by http://sscaitournament.com/index.php?action=tutorial

things, it is demonstrating the capabilities of the Production Queue function and shows the bot does not get stuck trying to produce units or buildings that are currently impossible to create because their prerequisites are not yet in place.

Once the Squad Manager registers that the squad is complete, and the Bunker is completed, the first squad is given the collective command to load into the Bunker. This then demonstrates the Squad Manager's collective commanding ability. A new squad is then commissioned from the Squad Manager and added to the Production Queue. This squad consists of five Marines and two Medics. Due to the request coming through to the Production Queue for a Medic, an Academy will automatically be added to the queue as an Academy is an additional prerequisite of the Medic. Another Barracks is also commissioned by the bot to supplement the production of units for the squad.

When this second squad is reported as complete, it is sent exploring to find out what the enemy is doing and what buildings they have. When any of these are found, it is reported to the Enemy Base Tracker and added to its internal list. While this squad is exploring, another squad is created for base defence along with both the Stim Pack research being commissioned and the Infantry Weapons Level 1 upgrade. As the research is conducted at the Academy, which is a building that is already complete, then no new building is required, but the upgrade requires the Engineering Bay to be built.

Through all of this basic game set up, each of the base library interactions are demonstrated by the example bot. The final bot is now more robust against the common pitfalls of designing automated agents in a non-deterministic game world, than a typical beginner's bot would otherwise be.

7 Evaluation

For the evaluation of of whether this project met the goals set out in development, two different main strategies were employed.

- An online mass request of feedback on a dedicated social media platform
- An individual in-person evaluation

In theory these two methods should allow for a widespread group of overview accounts detailing responses of different types of people, along with an in depth analysis of different points of the library with comparison to the base version of BWAPI. These would also give the best overview of the reception this library received by both experienced AI developers and new developers, while still being feasible within a reasonable time scale to complete.

This method was chosen in the hope that insight would be given into how the library impacted new users to the API, while at the same time receiving feedback from developers who had knowledge of what it was like starting out without it. Through this it is hoped that it can be seen that when a AI developer starts out development using the new library, they can create a functioning bot faster and easier, than would have been possible previously without the library.

For both testing scenarios, the online and the in-person, a brief instruction booklet containing documentation of what the classes and methods in the library did, was compiled from the code base. This, along with the library and example bot was then given to the testers. The full documentation can be found in Appendix A.

7.1 Evaluation by members of the Bot Development community

For the online evaluation, the project was uploaded to a source control repository[2], along with the documentation and brief instructions about how to use the library and what evaluation was required. A link to this repository and a request for evaluators was then posted to a social media page dedicated to people who work with BWAPI.

After one week, there had been only one returned evaluation from the social media post. In that response, the main message was clarifying that the project is a library and not an augmentation to BWAPI as the documentation suggested. This was due to the fact that the responder was not clear that the classes are not being intended to be something that would be installed with BWAPI by default, but are to be something that would be added later if the AI developer wanted them. Other than that, the reply just went on to detail other projects that had similar goals, including the Atlantis API mentioned in section 2.3.

[2]Repository can be found at: https://github.com/TJSharples/MastersBWAPILibrary

7.2 Evaluation by developers with no prior experience

With the lack of meaningful feedback in the evaluation from the online method, the results from the in-person evaluation become crucial. The aim was to evaluate the differences in experience between developing a bot using the newly developed library and using only the original BWAPI. To do this, a developer, or developers, who had the required knowledge of Java development, and StarCraft were required.

The first potential tester that was approached did not have the game knowledge of StarCraft to know what a bot would need to do, although they did have the knowledge of Java to be able to do the programming. Because of this, a second tester was approached who did have sufficient game knowledge, but who was not versed in Java development. Fortunately, the testers agreed to work together throughout the experiment.

The setup of the experiment was to provide the testers the documentation provided for the online test and a copy of the new library project downloaded from the repository. The testers would then be given as much time as they wanted to make a functioning bot. Throughout this time, assistance and guidance was provided as requested if it was information that could have been found through a thorough reading of the example bot, the new library documentation, or the original API documentation.

After the testers had created a bot to their satisfaction, they would then be given a clean install of BWAPI exactly as a newcomer to the subject area would have. They then had the objective of recreating the functionality of what they made in the previous stage within the same time limit. After the second stage of the experiment they would then be asked for their comments. Throughout both stages, observations were made about what they struggled with, what came easily to them, and what could be improved upon to make starting easier. Below are the details of these observations, followed by the testers' comments and what these imply for the furtherance of development in this project.

At the start, both testers spent some time looking through the example bot and the code documentation. After some clarification of where they were supposed to work within the project, the programmer began to tinker around with the library functions, briefly looking through what they were all called and what they took in and passed back. During this time, the second tester with the game knowledge began describing what they needed to do to begin competing in a game of StarCraft. It became evident quite quickly that more guidance was required as to how to start out, with both testers struggling to immediately grasp which sections of the library were created to be used for what they were trying to accomplish. Due to this knowledge being available by reading through the documentation thoroughly, a few pointers were given as to where to look for the information, to expedite the process rather than having the experiment be stuck at the start for longer than needed. With this knowledge, they returned to their first task of assigning workers to mine minerals. After they had a bot that could do this, they set about making it create supply depots when the bot ran low on supply, then following that with aiming to create some units to go and explore the map. Most of the explanation on how to do this using the new library

came from the example bot provided, with the main differences being in programming style and exact workings. After creating a bot that mined enough resources to produce a barracks, and enough marines to then send a squad out scouting, the testers called the first stage of the experiment complete as they were happy with what they had created. From starting out to final testing, this first stage of the experiment had taken around two hours.

From these observations it became clear that perhaps a quick start guide or some equivalent would be prudent to go along with the documentation as a better explanation on how to use the library, and how to get started with using BWAPI in general. However, once the testers understood the function of each package within the library, the development progressed a lot more rapidly than personal previous experiences of starting out in using BWAPI. From the results of this experiment, it appeared that the first impressions of the new library were that it did indeed make starting out in developing bots for BWAPI easier.

For the second stage of the experiments, the testers were provided with the basic example bot from BWAPI and a link to the online documentation for the API. Straight away the difference in the difficulty of the task was clear as the testers kept attempting to use functionality that was not included in the base version and instead was part of the new libraries. The task of assigning the workers to mine minerals straight away was more challenging than they expected, then trying to acquire a worker to build a supply depot took up most of the remaining time in the experiment. By the end when the two hours were up, the bot created with the base API could collect minerals, create workers and create a supply depot. But each of these actions were mostly fixed scripts with fixed timings and catches, and not dynamic to changing situations in the world space. Comparing the bots of the two parts of the experiment distinctly showed that more had been accomplished in the time allocated while using the new library. This still held up when considering the beginning of the first half of that experiment was spent familiarising themselves with the workspace and the base API, whereas that knowledge was used in the second half, potentially saving time there but yet still not accomplishing as much.

At the end of the two stage evaluation experiment, the testers were asked to make comments on their experiences using the API with and without the new library. Below are their comments, mostly made from the point of view of the programmer in the testing pair, with some elaborations for clarification.

Using the newly developed library

- The environment was a bit overwhelming at first
- The library classes are documented, but there was no "how to start" directions.
- Once we figured out the base ideas, we found it easy to get the bot to do something useful.
- The library seemed to shield me from a lot of the complexities around managing orders in the system.

- The queue of orders [Production Queue] was intuitive and worked straight away.
- Making more complex AI will be easier with the library as I don't have to juggle orders, priorities and unit dependencies myself. This makes the AI logic more expressive as it isn't hidden amongst Java/BWAPI boilerplate.

Using BWAPI as downloaded from their website with the base example project

- Just using BWAPI made it very hard to issue multiple commands. For example, how to build workers and build a supply depot at the same time was not intuitive.
- Having to manually manage and check on resources adds a lot of mental overhead that is not helping to make a "smart" AI, but is necessary to make a bot work at all in the system.
- After a while I realised that I would need to remake a lot of the constructs that were made in the new libraries I used previously, just to get something working that was extendible and maintainable.

Several points are raised here, but most are in line with what was noted during the observations of the experiment. As observed, the testers say they struggled figuring out how to get started, but once they got past that initial hurdle they got on with development quite well. They seemed to like the way the library classes worked and the point that was raised about the library seeming to shield them from the intricacies of working with BWAPI showed that once they had experienced working without the aid of the library then they realised the weight of what the classes were doing for them.

The most important points however come in the explanation on their experiences with the base API without the libraries. The penultimate point was that a lot of what they were struggling with managing, and as such spending a lot of development time on, didn't seem like it was actually helping to actually make good AI. In fact they remarked that it was just necessary overhead to working in the environment. And the final point goes on to say that the testers thought that they would have to recreate a lot of the functionality of the library themselves to get back to being able to develop at the level they were doing so previously.

With the main aim of this project being to lower entry barriers into the field of developing AI bots for the real time strategy game environment, the feedback that after using the newly developed libraries and then trying without, the testing programmer felt that they would have to recreate the functionality the libraries provided, definitely marks a success. Improvements to these libraries of course can be made, and with these libraries only being the first step in creating the visual programming environment that would be more ideal for making entry into the area easier, perhaps even for non-programmers, these developments are far from the final step. However the evaluations at this stage proved positive that the library does make starting out on the road of AI bot development a lot easier than having just the base BWAPI.

8 Conclusion

Throughout this project, we have investigated how to lower the entry barriers that are preventing people from easily getting into the development of artificially intelligent agents for real time strategy games. The real time strategy game chosen for detailed analysis was StarCraft as it represents a professionally developed game with good game balance and with no defined best strategy for game mission success which provides a better environment for game AI research.

Initially we looked into existing tools in the area of developing agents in these design spaces. We saw the problems traditionally presented by developing in these non-deterministic world environments and what action can be taken to help account for this. The StarCraft BWAPI was examined to discern the reasons for its place in the development world as one of the most used tools for creating these agents. We saw that there were many reasons that BWAPI is used over other possible development environments not least due to the popularity of the StarCraft game that it is an API for but also that it provides an easier learning curve for understanding the game mechanics.

We took a look at existing projects designed at making the process of starting out with BWAPI easier, such as the Atlantis API, and we established how the differences in design philosophy in projects can be crucial. With Atlantis, the project seems to be designed to do as much as possible for the AI developers, leaving them to only define either the building order for the game strategy, or a way for the bot to create a build order. For this project however, we wanted a more central approach that would not do almost everything for the developer, but would instead just provide them with the tools that would make it easier for them to develop an AI for themselves.

We then proceeded to examine Visual Programming Languages and what they brought to the field of games programming. With Scratch's design criteria of making coding more accessible by opening up the inner workings to the view of the developer in a visually understandable way, the steps required to make learning the concepts of programming easier began to appear. We also looked at Unreal Engine's Blueprint system and saw that visual Programming can be used for much more complex systems and not just the 2D small projects that are creatable in Scratch. Thus the benefits and potential methods of creating a Visual Programming Language for BWAPI were established.

After investigating Visual Programming Languages in depth, we began to outline what would be an ideal visual programming environment as a front end to BWAPI. However, due to the practical constraints of this single project, we scaled back the scope to just the first steps on the way to the creation of that environment. We took input on what AI developers that had used BWAPI would have liked to have been included in the API when they started out, and began designing how those suggestions would work and how they would go together in a library. This was then followed through by implementing these designs in a new library that could be used alongside BWAPI.

Once this library was tested, and was proved to follow the designs laid out, it was made available for evaluation both online and by local in-person testers. Although the online comments from existing users of BWAPI did not provide useful feedback, the in-person evaluation yielded much more expressive results. The testers showed that development using the new library was much more successful than without it, with the testers even going so far as to say that they would probably need to develop significant parts of the library for themselves if they were to try to use BWAPI without it.

In conclusion, this project has demonstrated that there are ways of developing supporting systems that will reduce the learning curve associated with working with BWAPI. Even though this project only covers the first steps, there is positive evidence that continuing development in this line will yield the result intended of easing the learning curve and lowering the entry barriers for working in bot development for real-time strategy games.

8.1 Future Work

Starting with the library developed as part of this project, (for which the code base, the example bot and a copy of the documentation can be found at https://github.com/TJSharples/MastersBWAPILibrary), there are two main directions that could be taken as this research project is moved forward in the future. The first and most direct extension of this project would be to continue with the next steps towards developing this library of functions into a full visual programming environment. From the start made here with four packages in a library that address different problem areas that AI developers raised, this could first be extended to include the other areas that were listed in the research in section 4.2.

Once these have been added to the existing new library packages, this will still only be a library of useful support classes for a new developer. To truly move forward in developing a visual programming language, research should be done into developing fully self-contained library methods that can be run in any sequence that would be used as visual nodes in the language. The ways that other languages take the on-screen visual code structures created by the developer and change them into something executable should also be investigated. If a visual programming environment is to be created then it should be established at the start what code limitations will have to be imposed during development to ensure that the interface remains easy to learn and use.

The alternative direction that the project could be taken in would be to look into the new API that was released by Blizzard Entertainment during the course of this project. The new API is for working with StarCraft 2, the sequel game to the StarCraft: BroodWar, the game that BWAPI was developed for (Blizzard Entertainment, 2017a). Investigating AI bot development using the new API would offer similar advantages as the reasons for working with BWAPI. Having a popular game as the environment offers a broader understanding of the objectives for the AI to be developed while at the same time offering a similarly balanced game to develop for. Using the new API instead of BWAPI increases the complexity of the bot development but with the benefit of making the projects more relevant to new AI

developers with a newer and more sophisticated game. To continue this project using the new API would involve going back to researching what AI developers would have liked in the new API to make it easier, and then looking into how to implement the suggestions. All this would need to be done before then continuing with the research into how the library could be developed into a Visual Programming Language.

8.2 Closing

The aim of this project was to investigate ways that a new API could be created to ease starting out with AI bot creation for real-time strategy games. During the project, we developed a library using suggestions from the community surrounding this development area. From there we tested it by getting some developers who had never used BWAPI before to try the new library and then to try using BWAPI without it to ascertain which they found easier and why. Their descriptions and remarks after this experiment showed that the new library is an improvement over the base BWAPI but still has room for improvement if it is to be used by complete novices in this area. This project however has demonstrated that continuing in this research will positively impact the development experience of any new developer coming into the field of AI for gaming and as such will both encourage and enhance the future development of AI in general.

9 References

Battle.net End User Liscence Agreement. (2015). Retrieved from http://us.blizzard.com/en-us/company/legal/eula.html

Blizzard Entertainment, B. (2017a). *The StarCraft II API Has Arrived.* Retrieved from http://us.battle.net/sc2/en/blog/20944009/the-starcraft-ii-api-has-arrived-8-9-2017

Blizzard Entertainment, B. (2017b). *StarCraft: Remastered.* Retrieved from https://starcraft.com/en-us/

Buro, M. (2003). Real-time strategy games: A new AI research challenge. In *In proceedings of the 18th international joint conference on artificial intelligence* (pp. 1534–1535). International Joint Conferences on Artificial Intelligence. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.96.6742 doi: 10.1.1.96.6742

Buro, M., & Churchill, D. (2012). Real-Time Strategy Game Competitions. *AI Magazine 33.3*, 106–108. Retrieved from http://search.proquest.com/docview/1368560768?accountid=11526

BWAPI FAQs. (2014). Retrieved from https://github.com/bwapi/bwapi/wiki/FAQ

BWAPI Homepage. (n.d.). Retrieved from http://bwapi.github.io/

Carlisle, M. C., Wilson, T. A., Humphries, J. W., Hadfield, S. M., Carlisle, M. C., Wilson, T. A., . . . Hadfield, S. M. (2005, 2). RAPTOR. *ACM SIGCSE Bulletin*, *37*(1), 176. Retrieved from http://portal.acm.org/citation.cfm?doid=1047124.1047411 doi: 10.1145/1047124.1047411

Cheng, J. (2016, 4). *Computers That Crush Humans at Games Might Have Met Their Match: 'StarCraft'.* Retrieved from http://www.wsj.com/articles/computers-that-crush-humans-at-games-might-have-met-their-match-starcraft-1461344309?mod=WSJ_TechWSJD_topRight

Do, H., Rothermel, G., & Kinneer, A. (2004). Empirical Studies of Test Case Prioritization in a JUnit Testing Environment. In *15th international symposium on software reliability engineering* (pp. 113–124). IEEE. Retrieved from http://ieeexplore.ieee.org/document/1383111/ doi: 10.1109/ISSRE.2004.18

Khan, M. E., & Khan, F. (2012). A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. *IJACSA) International Journal of Advanced Computer Science and Applications*, *3*(6). Retrieved from www.ijacsa.thesai.org

Kirby, N. (2011). *Introduction to Game AI.* Cengage Learning. Retrieved from http://common.books24x7.com.libaccess.hud.ac.uk/toc.aspx?bookid=34468.

Laird, J., & VanLent, M. (2001, 6). *Human-Level AI's Killer Application: Interactive Computer Games* (Vol. 22) (No. 2). Retrieved from http://www.aaai.org/ojs/index.php/aimagazine/article/view/1558 doi: 10.1609/aimag.v22i2.1558

Magnusson, M., & Balsasubramaniyan, S. (2012). *A Communicating and Controllable Teammate Bot for RTS Games.* Retrieved from http://urn.kb.se/resolve?urn=urn%3Anbn%3Ase%3Abth-4360

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010, 11). The Scratch Programming Language and Environment. *ACM Transactions on Computing Education*, *10*(4), 1–15. Retrieved from http://portal.acm.org/citation.cfm?doid=1868358.1868363 doi: 10.1145/1868358.1868363

Ontanon, S., Synnaeve, G., Uriarte, A., Richoux, F., Churchill, D., & Preuss, M. (2013, 12). A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft. *IEEE Transactions on Computational Intelligence and AI in Games*, *5*(4), 293–311. Retrieved from http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6637024 doi: 10.1109/TCIAIG.2013.2286295

ORTS Homepage. (n.d.). Retrieved from https://skatgame.net/mburo/orts/

Poniatowski, R. (2017). *Atlantis GitHub Documentation.* Retrieved from https://github.com/

Ravaelles/Atlantis

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., ... Kafai, Y. (2009, 11). Scratch: programming for all. *Communications of the ACM*, 60–67. Retrieved from http://dl.acm.org/citation.cfm?id=1592779 doi: 10.1145/1592761.1592779

Schwab, B. (2009). *AI Game Engine Programming.* Cengage Learning. Retrieved from http://library.books24x7.com.libaccess.hud.ac.uk/toc.aspx?bookid=33925

Shackelford, R., & LeBlanc, R. (1997). Introducing computer science fundamentals before programming. In *Proceedings frontiers in education 1997 27th annual conference. teaching and learning in an era of change* (Vol. 1, pp. 285–289). Stipes Publishing. Retrieved from http://ieeexplore.ieee.org/document/644858/ doi: 10.1109/FIE.1997.644858

Sirani, J. (2015). *UE4 Free for All.* Retrieved from http://uk.ign.com/articles/2015/03/02/unreal-engine-4-is-free-for-everyone

Wegner, P. (1987). Dimensions of Object-based Language Design. In *Conference proceedings on object-oriented programming systems, languages and applications* (pp. 168–182). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/38765.38823 doi: 10.1145/38765.38823

Glossary

academy In game building required to train certain infantry units.

barracks In game building that produces infantry units.

BWAPI Brood War Application Programming Interface. The main software used for development of bots for StarCraft.

fog of war Common game mechanic in real time strategy games. A shroud that obscures the map unless it has been explored and then partially revealed unless currently in vision of the players unit..

marine In game basic infantry unit. Can move and attack both air and ground units. Created from the barracks plural.

medic In game basic infantry unit. Can move and heal other infantry. Created from the barracks, requires the Academy plural.

minerals An in game resource that is mined by builders from deposits. Used for building construction, unit production and upgrade purchasing.

Protoss One of the three game races. Ancient aliens with very advanced technology.

refinery In game building placed on Vespene Gas outlets that allows workers to collect the gas.

RTS Real Time Strategy. Genre of games focusing around managing game events in real time.

SCV In game unit that functions as the Terran's builder. Constructs all buildings and mines resources plural.

supply In game peak of how many units a player can control at once.

supply depot An in game building that increases the maximum number of units that a player can control at once.

Terran One of the three game races. Human based with advanced technology.

vespene gas An in game resource that is collected from refineries by builders. Used for building construction, unit production and upgrade purchasing.

Zerg One of the three game races. Biology based swarm.

Appendix
Testing Documentation

Contents

1 Production Queue 61
1.1 Constructor . . . 61
1.2 onGameStart . . . 61
1.3 Update . . . 61
1.4 AddToQueue . . . 62
1.5 RemoveFromQueue . . . 62
1.6 unitStructureStarted . . . 62
1.7 unitStructureDestroyed . . . 63
1.8 checkIfHaveInProduction . . . 63
1.9 checkHowManyInProduction . . . 63

2 Production Order 64
2.1 getPriority . . . 64
2.2 getOrderTime . . . 64
2.3 getStatus . . . 64
2.4 setStatus . . . 64
2.5 toString . . . 65
2.6 canAfford . . . 65
2.7 isToProduceFree . . . 65
2.8 checkHasStarted . . . 65
2.9 checkHasFinished . . . 65
2.10 getToProduce . . . 66
2.11 UnitBuildingOrder Specific - setStartedUnit . . . 66
2.12 UnitBuildingOrder Specific - getStartedUnit . . . 66
2.13 UnitBuildingOrder Specific - enoughSupply . . . 66
2.14 UnitBuildingOrder Specific - checkIfStillTraining . . . 66

3 Enemy Base Tracker Manager 67
3.1 Constructor . . . 67
3.2 onFrame . . . 67
3.3 createTrackersForAllEnemies . . . 67
3.4 getEnemyUnitBuildingList . . . 67
3.5 checkIfEnemyHas . . . 68
3.6 unitFound . . . 68
3.7 unitDestroyed . . . 68

4 MemoryUnitBuilding 69
4.1 getUnit . . . 69
4.2 getLastKnownUnitType . . . 69
4.3 getLastKnownPosition . . . 69
4.4 isVisible . . . 69

5 Builder Manager **70**
5.1 Constructor . . . 70
5.2 getBuilders . . . 70
5.3 addJob . . . 70
5.4 cancelJob . . . 71
5.5 onFrame . . . 71
5.6 unitBuildingStarted . . . 71
5.7 unitBuildingComplete . . . 71
5.8 unitBuildingKilled . . . 71
5.9 getSpareBuilderCloseTo . . . 72

6 Squad Manager **73**
6.1 Constructor . . . 73
6.2 onFrame . . . 73
6.3 createSquad . . . 73
6.4 disbandSquad . . . 73
6.5 getAllSquads . . . 74
6.6 getSquad . . . 74
6.7 onUnitDestroy . . . 74
6.8 setRemoveDeadSquads . . . 74
6.9 isUnitAssignedToSquad . . . 74
6.10 getAllSquadNeeds . . . 74

7 Squad **75**
7.1 getID . . . 75
7.2 getTargetSquadMakeup . . . 75
7.3 getAssignedUnits . . . 75
7.4 getTargetPosition . . . 75
7.5 getTargetUnit . . . 75
7.6 getRightClickTarget . . . 75
7.7 getWasComplete . . . 76
7.8 getIsDead . . . 76
7.9 getSquadNeeds . . . 76
7.10 setSquadMoveTarget . . . 76
7.11 setAttackTarget . . . 76
7.12 setSquadRightClick . . . 76
7.13 addUnitToSquad . . . 77
7.14 removeUnitFromSquad . . . 77

8 Example Bot **78**
8.1 onStart . . . 78
8.2 onFrame . . . 78
8.3 gameProgressionUpdate . . . 78
8.4 onUnitCreate . . . 79
8.5 onUnitMorph . . . 79

8.6 onUnitComplete 79
8.7 onUnitDestroy 79
8.8 onUnitDiscover 79
8.9 getBuildTile 79

1 Production Queue

This holds all production orders that are requested of it. Orders will be sorted by priority first, then by time ordered. On each frame it will go through list of orders to find which are ready to be executed and will pass back a ready order.

Orders are added with the use of the AddToQueue methods, then when they are ready for execution will be returned in the form of one of the three types of ProductionOrder: UnitBuildingOrder, UpgradeOrder, or ResearchOrder.

REQUIRED USER BOT CALLS:

- onGameStart: Call onGameStart
- onFrame: Call update
- onUnitCreate: Call unitStructureStarted
- onUnitMorph: Call unitStructureStarted
- onUnitDestroy: Call unitDestroyed

1.1 Constructor

```
public ProductionQueue(boolean debugMessagesOn, boolean debugOnScreen)
```

- @param debugMessagesOn True if you want the console output debug messages detailing the workings of the production queue. False if you don't.
- @param debugOnScreen True if you want the in game on screen listing of the items in the production queue. False if you don't.

1.2 onGameStart

```
public void onGameStart()
```

Basic setup of the ProductionQueue that needs to be run at game start. This would be in the constructor but due to the way that BWAPI initialises, some things should be done on game start, after the StarCraft engine has setup.

1.3 Update

```
public ProductionOrder Update()
```

Method must be called in the bot's onFrame method. Checks are run to see if an order is ready. Any that are ready for execution will be returned to the user to execute how they want.

- @return ProductionOrder that is ready for execution. NULL if none are ready.

1.4 AddToQueue

```
public void AddToQueue(UnitType toBuild, int priority,
boolean autoBuildPrereq)

public void AddToQueue(TechType toBuild, int priority,
boolean autoBuildPrereq)

public void AddToQueue(UpgradeType toBuild, int priority,
boolean autoBuildPrereq)
```

Three overloaded methods that add an item to the production queue. Depending on the type of the toBuild object passed in will depend on what type of production order is created for the list. The parameter autoBuildPrereq determines whether or not any non-completed prerequisites will automatically be added to the queue. In the versions of the method call that do not have this option, it is automatically taken to be true.

- @param toBuild The Unit, Building, Research or Upgrade wanted
- @param priority The Priority level of this order. The Higher the number, the higher the priority.
- @param autoBuildPrereq True will automatically build all prerequisite structures not currently owned. False will only add this order to the queue.

1.5 RemoveFromQueue

```
public void RemoveFromQueue(UnitType toRemove, boolean removeAll)

public void RemoveFromQueue(TechType toRemove)

public void RemoveFromQueue(UpgradeType toRemove)
```

Three overloaded methods that remove an item or items from the production queue. Due to it not being possible to have multiple instances of the same tech or upgrade in the queue, only the unit or building removal method has the option to remove all.

- @param toRemove The Unit or Building to be cancelled
- @param removeAll True will remove all of that type from the queue. False will remove only the first one found starting with the lowest priority.

1.6 unitStructureStarted

```
public void unitStructureStarted(Unit startedUnit)
```

This method must be called by the users bot on both the onUnitCreate trigger from the base API and the onUnitMorph trigger. This informs the queue that a unit or building has been started and the order will store the reference to that unit or building to be able to check if it has completed.

- @param startedUnit The started unit

1.7 unitStructureDestroyed

```
public void unitStructureDestroyed(Unit destroyedBuilding)
```

This method must be called by the bot on the onUnitDestroy trigger from the base API. This informs the queue that a building was destroyed. It will then go through and find out if that was one of the ones that was either under construction for an order, or if it was producing something for an order. If it was then it will set the associated order to be commissioned again.

- @param destroyedBuilding The destroyed unit

1.8 checkIfHaveInProduction

```
public boolean checkIfHaveInProduction(UnitType toCheck)
```

```
public boolean checkIfHaveInProduction(TechType toCheck)
```

```
public boolean checkIfHaveInProduction(UpgradeType toCheck, int level)
```

Three overloaded methods that runs a check to see if the player currently has one of this item in the production queue.

- @param toCheck The item to check
- @return True if one is in the queue. False if it isn't.

1.9 checkHowManyInProduction

```
public int checkHowManyInProduction(UnitType toCheck)
```

Runs a check of the production queue to see how many of a certain unit or building are currently in the queue

- @param toCheck The unit type to check how many are there
- @return The integer value of how many of the desired type are in the queue

2 Production Order

The abstract base class for the three types of production order. Used with the production queue, the user will need to be able to take one of these orders and interpret it to find out what the bot should create after it has been given to them by the update method of the production manager.

2.1 getPriority

```
public int getPriority()
```

Gets the priority of the production order

- @return Int Priority : Higher the number the higher the priority.

2.2 getOrderTime

```
public int getOrderTime()
```

Gets the time that the order was made

- @return Int orderTime : The number of frames that had passed when the order was made.

2.3 getStatus

```
public OrderStatus getStatus()
```

Gets the status of the order

- @return OrderStatus enum CurrentStatus : The current status of the order

- Order Status
  - commissioned - on the queue but not yet acted upon. Not enough resources or not a priority
  - ordered - Enough Resources have been gathered, there is a unit that is free to produce the order and the order has been passed back to the bot to begin production.
  - started - work on the order has begun.
  - aborted - when a job has been cancelled but has not been cleared off the queue yet.

2.4 setStatus

```
public void setStatus(OrderStatus statusIn)
```

Sets the status of the order

- @param statusIn The status to set the order to

2.5 toString

```
public abstract String toString();
```

Gets the string representation of the order, different for unit/building, research or upgrade but the general form is:

Create: Unit/Building - , Priority - , Time - , Status -

- @return The string representation of the order

2.6 canAfford

```
public abstract boolean canAfford();
```

Run a check to see if the current player can afford to purchase what the order is going to create

- @return Boolean : true means the player can afford it, false means they can't

2.7 isToProduceFree

```
public abstract boolean isToProduceFree();
```

Run a check to see whether there is a building that can produce this item and if it is idle

- @return True if there is an idle building of the type needed. False if there isn't.

2.8 checkHasStarted

```
public abstract boolean checkHasStarted();
```

Run a check to see if order has been started. Only to be run if the order is 'ordered'. Does nothing for UnitBuildingOrders. If it is started then the status in the order is set to started.

- @return Boolean. True if the order has started. False if it hasn't

2.9 checkHasFinished

```
public abstract boolean checkHasFinished();
```

Run a check to see if the order has finished. Only to be run if the order is 'started'. If it is finished then the status if the order is set to 'finished'.

- @return Boolean. True if the order is finished. False if it isn't.

2.10 getToProduce

```
public UnitType getToProduce()
```

```
public UpgradeType getToProduce()
```

```
public TechType getToProduce()
```

Different for each specific type of Production order, and only accessible by parsing the Production Order to the correct subtype, this will return the item to be produced by the order.

- @return UnitType. The type of unit or building to be produced.
- @return UpgradeType. The upgrade to be produced.
- @return TechType. The research to be produced.

2.11 UnitBuildingOrder Specific - setStartedUnit

```
public void setStartedUnit(Unit unitIn)
```

Automatically called by Production Queue unitStructureStarted method. Set the unit that is being produced in the order. This is later used to check if it is finished.

- @param unitIn The unit that has just started for this order.

2.12 UnitBuildingOrder Specific - getStartedUnit

```
public Unit getStartedUnit()
```

Get the unit that is being produced or built.

- @return The unit that is being produced or built.

2.13 UnitBuildingOrder Specific - enoughSupply

```
public boolean enoughSupply()
```

Checks to see if the bot currently has enough supply for the unit or building to be produced.

- @return True if there is enough supply. False if not.

2.14 UnitBuildingOrder Specific - checkIfStillTraining

```
public boolean enoughSupply()
```

Check to see if the unit that was being built still exists. This will be false if the building creating it was destroyed or the unit was cancelled manually.

- @return True if the unit is still being built. False if it isn't

3 Enemy Base Tracker Manager

Manager that holds all of the different Enemy Base Trackers. Then allows for accessing each tracker's records while keeping them up to date. Users should not need to interact with individual EnemyBaseTracker objects.

REQUIRED USER BOT CALLS:

- onGameStart - Call Constructor. Then call createTrackersForAllEnemies
- onFrame - Call onFrame
- onUnitFound - Call foundUnit
- onUnitDestroy - Call unitDestroyed

3.1 Constructor

```
public EnemyBaseTrackerManager(boolean visualsOn)
```

Setup of the EnemyBaseTracker Manager. Needs to be run before any other method.

- @param visualsOn True if the on screen visual representations of unit locations are wanted

3.2 onFrame

```
public void onFrame()
```

Must be run in the bot's onFrame method. Runs the Update method for each EnemyTracker. Allowing each record to stay up to date with the most recent movement information and unit status.

3.3 createTrackersForAllEnemies

```
public void createTrackersForAllEnemies()
```

Creates new Trackers for each enemy in the game. Must be run in the game start method.

3.4 getEnemyUnitBuildingList

```
public ArrayList<MemoryUnitBuilding> getEnemyUnitBuildingList(Player forEnemy
```

Gets all MemoryUnitBuilding records from the tracker for the requested memory.

- @param forEnemy The enemy you want the records for.
- @return ArrayList of MemoryUnitBuildings: All current records for that enemy.

3.5 checkIfEnemyHas

```
public boolean checkIfEnemyHas(UnitType type, Player forEnemy)
```

Checks to see if a given enemy has a particular unit or building

- @param type The unit or building to check for
- @param forEnemy The enemy to check for
- @return True if that enemy has been discovered to have that unit or building. False if they don't, or it hasn't yet been discovered that they do.

3.6 unitFound

```
public void unitFound(Unit foundUnit)
```

Must be run in the bot's onUnitFound method. Tells the manager that a unit has been found. Manager will discern which player it is from and if pertinent will add it to the records.

- @param foundUnit The unit found.

3.7 unitDestroyed

```
public void unitDestroyed(Unit destroyedUnit)
```

Must be run in the bot's onUnitDestroy method. Tells the manager than a unit has been destroyed. Manager will discern which player it is from and if pertinent will amend the records.

- @param destroyedUnit The unit destroyed.

4 MemoryUnitBuilding

Storage Class for details about an enemy unit, used with the EnemyBasedTracker Manager. When a unit is no longer visible, its details can't be accessed so a few pertinent details are stored along with the unit class. Not creatable by the user but the user will be given a list of these objects when requesting all details on an enemy from the EnemyBase-Tracker Manager.

4.1 getUnit

```
public Unit getUnit()
```

Gets the unit that this Memory contains. Interrogation of the Unit class will return null if tried on a non-visible unit.

- @return the contained memory unit

4.2 getLastKnownUnitType

```
public UnitType getLastKnownUnitType()
```

Gets the last known unit type of this unit. Changes will happen most often with Zerg morphing units, but also interrogation of the Unit class will return null if tried on a non-visible unit.

- @return The last known Unit type of the unit.

4.3 getLastKnownPosition

```
public TilePosition getLastKnownPosition()
```

Gets the last seen location of the unit when it disappeared into the fog of war.

- @return Tile position of last known location.

4.4 isVisible

```
public boolean isVisible()
```

Returns whether the unit is currently visible

- @return True if the unit is visible and the details are now accessible. False if not.

5 Builder Manager

Manages the constructing of buildings and ensures that idle workers keep mining minerals. You can pass in jobs of what to build and where to build it and the manager will get a builder for it and set it to creating that building, and the manager will look after replacing builders if they are destroyed. Idle builders will be sent mining, this includes when the builder is first produced, if the building it was creating was completed or if it was moved and has now finished its movement.

REQUIRED USER BOT CALLS:

- onGameStart: call Constructor
- onFrame: call onFrame
- onUnitCreate: call unitBuildingStarted
- onUnitMorph: call unitBuildingStarted
- onUnitComplete: call unitBuildingComplete
- onUnitDestroy: call unitBuildingKilled

5.1 Constructor

```
public BuilderManager(boolean debugMessagesOn)
```

Constructor. Initializes the two Arrays of Jobs and Builders.

- @param debugMessagesOn True if you want the console output debug messages detailing the workings of the builder manager. False if you don't.

5.2 getBuilders

```
public ArrayList<Unit> getBuilders()
```

Gets the list of all builders

- @return ArrayList of all builder units

5.3 addJob

```
public void addJob(UnitType toBuild, TilePosition position)
```

Tells the manager to get a builder and start constructing the required building at the location. Assumption is made that the player has the resources to do so when this method is called.

- @param toBuild The building to construct
- @param position The location to build it at

5.4 cancelJob

```
public void cancelJob(UnitType toCancel, TilePosition position)
```

Cancels a previously requested job.

- @param toCancel The type of building to cancel
- @param position The location where it was to be built

5.5 onFrame

```
public void onFrame()
```

Must be called in the bot's onFrame method. Catches any idle workers and sends them mining to their closest mineral source.

5.6 unitBuildingStarted

```
public void unitBuildingStarted(Unit started)
```

Must be called in the bot's onUnitCreate and onUnitMorph methods. Call to let the manager know that a unit has been started. The manager will then ascertain if it is relevant to the builder manager or not. If it is relevant then the construction job will be marked as started.

- @param started the unit that has been started

5.7 unitBuildingComplete

```
public void unitBuildingComplete(Unit completed)
```

Must be called during the bot's onUnitComplete method. Call to let the Manager know that a unit has been completed. The manager will ascertain if it is relevant. Depending on whether it is a building or a relevant unit that has been completed, the manager will either add the builder to the builder list, or will mark the job as complete and remove it from the list.

- @param completed The completed unit

5.8 unitBuildingKilled

```
public void unitBuildingKilled(Unit killedUnitBuilding)
```

Must be called during the bot's onUnitDestroy method. Call to inform the manager that a unit or building has been destroyed. The manager will then ascertain if it is relevant to it. If it was a unit relevant to the manager then the manager will retrieve a new builder but if it was a relevant building then the job will be cancelled.

- @param killedUnitBuilding The unit destroyed.

5.9 getSpareBuilderCloseTo

```
public Unit getSpareBuilderCloseTo(Position closeTo)
```

Retrieves the closest builder that is only mining minerals

- param closeTo The position you are getting the closest builder to
- @return the builder

6 Squad Manager

Allows for the creation of squads with unit type lists for what they should consist of. The squad will then keep a list of what units are assigned to it and can compare what it has to what it is supposed to have. The squad allows for groups of units to be referred to as a group to give them targets or movement directions.

REQUIRED USER BOT CALLS:

- onGameStart : constructor
- onFrame : onFrame
- onUnitDestroy : onUnitDestroy

6.1 Constructor

```
public SquadManager(boolean debugMessagesOn)
```

- @param debugMessagesOn True if you want the console output debug messages detailing the workings of the Squad Manager. False if you don't.

6.2 onFrame

```
public void onFrame()
```

Must be called in the bot's onFrame method. Each frame checks to see if any squads are now dead and will remove them from the list if the flag RemoveDeadSquads is true.

6.3 createSquad

```
public int createSquad(HashMap<UnitType, Integer> squadMakeup)
```

Create a new squad with the desired unit make up

- @param squadMakeup The Map of units that are to be created. HashMap of Unit Types and the required integer count of each of them
- @return the unique squad ID

6.4 disbandSquad

```
public void disbandSquad(int squadID)
```

Disbands the squad by removing it from the list of squads. Units that were in the squad will no longer return true to when querying if the unit is in a squad

- @param squadID The unique squad ID to be disbanded

6.5 getAllSquads

```
public ArrayList<Squad> getAllSquads()
```

Gets all the squads

6.6 getSquad

```
public Squad getSquad(int squadID)
```

Gets the squad with the requested ID

- @param squadID the ID of the squad requested
- @return The requested Squad if it exists. Null if it doesn't

6.7 onUnitDestroy

```
public void onUnitDestroy(Unit destroyedUnit)
```

Must be called in the bot's onUnitDestroy method. Lets the manager know a unit has been destroyed. This will let each squad check to see if it was one of their units that was destroyed.

- @param destroyedUnit The destroyed Unit

6.8 setRemoveDeadSquads

```
public void setRemoveDeadSquads(boolean setToo)
```

Set to true by default. This will remove squads from the squad list if all units in it are dead and at some point it was a complete Squad. If false then the squad will remain in the list, just with no units assigned.

- @param setToo Set the parameter to. True if you want to remove the dead squads, false if you want to leave them.

6.9 isUnitAssignedToSquad

```
public boolean isUnitAssignedToSquad(Unit unitToCheck)
```

Runs a check in each squad to see if the requested unit is attached to that squad

- @param unitToCheck The unit to check if it belongs to a squad
- @return True if the unit is in a squad. False if it isn't

6.10 getAllSquadNeeds

```
public HashMap<UnitType, Integer> getAllSquadNeeds()
```

Gets the needs of all squads compiled into one hashmap

- @return Hashmap of UnitType : Integer of all the squad needs

7 Squad

Holds a list of desired squad make up and a list of assigned units. Allows for mass giving of orders to all units in the squad. Used with the Squad Manager, should not be instantiated by the user.

7.1 getID

```
public int getID()
```

Get the Unique ID of the squad

- @return Unique ID number of the squad

7.2 getTargetSquadMakeup

```
public HashMap<UnitType, Integer> getTargetSquadMakeup()
```

Gets the target squad make up

- @return HashMap of Target Squad Make up

7.3 getAssignedUnits

```
public ArrayList<Unit> getAssignedUnits()
```

Gets all the units assigned to the squad

- @return ArrayList of assigned units

7.4 getTargetPosition

```
public Position getTargetPosition()
```

Gets the position that the squad was last sent to. Null if the last target was a unit or a right click target

- @return The position the squad was last sent to

7.5 getTargetUnit

```
public Unit getTargetUnit()
```

Gets the unit that the squad was last sent to attack. Null if the last target was a position or a right click target

- @return The unit the squad was last sent to attack

7.6 getRightClickTarget

```
public Unit getRightClickTarget()
```

Gets the target unit that the squad was to act as if a player had right-clicked on. Null if the last target was a position or a unit

- @return The target unit that the squad was to act as if a player had right-clicked

7.7 getWasComplete

```
public boolean getWasComplete()
```

If the squad was ever complete at some point then this returns true.

- @return True if all the squad goals have been met. False if not.

7.8 getIsDead

```
public boolean getIsDead()
```

Returns if the squad is dead or not. A squad is considered dead if it currently has no members and was once complete.

- @return True if squad is dead. False if not

7.9 getSquadNeeds

```
public HashMap<UnitType, Integer> getSquadNeeds()
```

Gets the HashMap of the disparity between assigned units and the squad target make up.

- @return HashMap UnitType: Integer of the types of units missing and the amount that are still needed.

7.10 setSquadMoveTarget

```
public void setSquadMoveTarget(Position movePosition, boolean attackMove)
```

Sets the squad move target to a given map position. All units in the squad will move there or as close as they can get. If the attack move option is given then the units will attack move instead. To set the squad target to a unit without making it attack, use this function and pass in the unit's position.

- @param movePosition the position to move the squad to.
- @param attackMove Whether the units should attack enemies they encounter along the way.

7.11 setAttackTarget

```
public void setAttackTarget(Unit toAttack)
```

Set the target of the squad to a unit. They will then go and attack that unit.

- @param toAttack The unit the squad will attack

7.12 setSquadRightClick

```
public void setSquadRightClick(Unit rightClickTarget)
```

For each squad member give it the command of right-clicking on the target unit

- @param rightClickTarget The target unit to right click on

7.13 addUnitToSquad

```
public void addUnitToSquad(Unit unitToAdd)
```

Assign a unit to the squad

- @param unitToAdd The unit to be assigned

7.14 removeUnitFromSquad

```
public void removeUnitFromSquad(Unit unitToRemove)
```

Will unassign a unit from the squad. The unit will no longer receive squad commands

- @param unitToRemove The unit to remove from the squad.

8 Example Bot

This example bot has been created to display a simple way of using each of the different packages of the library. The strategy employed by the bot will not win a game, but the current game plan will show how to do most of the actions that a user will want their bot to do. In this section is a simple breakdown of what is being used in each method.

8.1 onStart

To begin with, the base API onStart method is called using the super.onStart() method call. This sets up the base API terrain analyser along with setting the game flag for manual input so that testing is easier. This must be called in the onStart method of a users bot, otherwise no map analysis data will be available.

After that, along with a few testing variables being set-up, each of the managers for the library packages are initialised in accordance with their individual instructions. Then any other methods that the packages need for set-up are called.

8.2 onFrame

First in the onFrame method is all the onFrame methods of the three non production queue packages. Then the return of the production queue update method is captured for analysis. If the return is not null, then analysis is performed on the type of production order that has been returned and the necessary actions are determined which will be either calling the builder manager to add a building order, or finding the relevant production building and telling it to produce the unit, research or upgrade.

Next there is the housekeeping actions of making sure that a new supply depot is added to the production queue if the supply is getting low, and a new builder is added to the queue if there are not enough of them.

Finally the gameProgressionUpdate method is called that keeps track of where the bot is through its game plan and what actions it should be taking that don't need to be taken every frame.

8.3 gameProgressionUpdate

The game progression update tracker is set up so that there is a barrier that will track when certain game criteria are met for advancement. This barrier is different for each stage of the game plan.

- Level 1 - 10 Completed Builder Units
- Level 2 - Complete Bunker, Complete Bunker Squad
- Level 3 - Complete Academy, Complete Scouting Squad
- Level 4 - Complete Defence Squad, Complete Stim Pack Research

When each of the game stage criteria are met, the next stage of the bot's game plan is sent to the Production Queue and orders are given.

8.4 onUnitCreate

Calls the production queue's unitStructureStarted method and the Builder Manager's unitBuildingStarted method.

8.5 onUnitMorph

Calls the production queue's unitStructureStarted method and the Builder Manager's unitBuildingStarted method.

8.6 onUnitComplete

Calls the builder manager's unitComplete method.

Then for each squad that the Squad Manager holds, checks to see if that squad requires a unit of this type, and if it does, assigns the unit to that squad.

8.7 onUnitDestroy

Calls the production queue's unitStructureDestroyed method, the builder manager's unitBuildingKilled method, the enemy base tracker's unit destroyed method and the squad manager's onUnitDestroy method.

Then after that, checks to see if enough units are in production to fulfil all squad needs. If there isn't then it adds what's needed to the queue.

8.8 onUnitDiscover

Calls the enemy base tracker's unitFound method.

8.9 getBuildTile

This is an example method provided by http://sscaitournament.com/index.php?action=tutorial It will find the first available place to start a a building of a given type by outward searching from a point to see if there is an open place to build of that size.