The University of Sheffield

UML State Machines as Annotations for State-based Software

Author: David Goddard

July 8, 2020

Supervisor: Kirill Bogdanov

Module: COM3610

This report is submitted in partial fulfilment of the requirement for the degree of BSc in Computer Science

Declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

David Goddard

Abstract

The use of comments in software development is undeniable. They alleviate many of the problems developers experience when coding by concisely explaining what various aspects of the code are doing. This improves the efficiency of software development as less time is required to understand the code and therefore more time can be spent programming. Various tools exist to achieve this goal but many lack the desired performance.

This research project investigates issues with these tools with a close analysis on their strengths and weaknesses at annotating software with state machine diagrams. It subsequently outlines the design and implementation of a plugin for the Eclipse IDE which will extend one such tool called PlantUML, aiming to enhance its ability to annotate state-based software.

Acknowledgements

I would personally like to thank Dr Kirill Bogdanov as he has supported me throughout all aspects of this project, sometimes answering emails at 4.am, and for that I am truly grateful.

Contents

List of Figures

Chapter 1

Introduction

The involvement of annotations in the software development cycle is not a new-founded principle, with developers utilizing their advantages for decades. 1984 saw Donald Knuth stress their importance whilst introducing a new programming paradigm known as Literate programming[1]. This emphasised the advantages of treating a program as a piece of literature, addressed to humans rather than computers, transitioning the focus of development from writing code to writing explanations about the code. Knuth envisioned that this would progress the art of documentation by it leading to programs that were more reliable and easier to comprehend.

Thirty years have now passed since Knuths' paradigm was introduced, and software is more complex than ever. Software documentation aims to ease this understanding by concisely bestowing an overview of the programs structure. Jeff Raskin takes this further, stating that documentation "is one of the most-overlooked ways of improving software quality and speeding implementation[2]", which suffice to say are benefits in every developers interest.

One such form of documentation is the production of a state-machine diagram which can be used to model the dynamic behaviour of state-dependent objects in an intuitive way. This involves defining an objects finite number of states and then illustrating how events and/or actions can cause these to change. Figure 1.1 illustrates a state machine describing the simple system behind opening and closing a door.

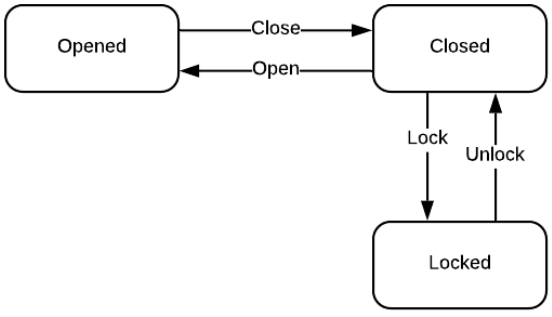


Figure 1.1: The system behind a door represented as a state machine

When used to describe software, state machines are an extremely useful resource because they provide a very different view on its behaviour when compared to the source code. In sum, they are crucial for developers wanting to adapt the code, understand the code, or even, and sometimes most importantly, write tests on the code.

Various tools exist with the purpose of creating state machines and their detailed examination can be found in Chapter 2. Unfortunately, when used to annotate state-based software, few achieve desired performance, with many exerting weaknesses that make the developer question whether the effort involved in creating and thereafter maintaining the diagram is worthwhile.

In order to elucidate a justification as to why this is, it is necessary to distinguish the difference between annotations and design documentation. Annotations in this context can be thought of as ways of labelling the source code in an effort to describe aspects of the code that are hard to comprehend, such as writing comments. Documentation, on the other hand, is the process of creating a sort of user-manual that is external to the source code and is often used to describe the overall design of a system such as with a state machine.

Therefore, although many tools are exemplary at creating state machine diagrams that will be used for documentation, if the user wants to use them to annotate software, they quickly fall short. This stems from the fact that they require the developer to create the diagram in an environment that is separate to the one they are writing their source code in. Additionally, this means that the state machine diagram can not be used as an annotation and itself must be stored separately from the code.

Nevertheless, there are some tools that ingeniously merge the concepts of annotation and documentation by giving the developers the freedom to create and display diagrams within the environment they are writing their code. PlantUML, is one such instance and is the tool that the plug-in made in this project will extend.

Project Aims

The fundamental objective of the project is to create a tool that helps users develop state-based software. This may manifest in various ways such as helping them to understand legacy state-based code or to verify that their systems behaviour works as intended.

This will be achieved through the design and implementation of a plugin for the Eclipse IDE that will extend PlantUML, improving its functionality in creating and maintaining state machine diagrams. The idea is that the user will point out source code that models a state-based system, and the plug-in will extract this and use it to automatically generate a state machine. Like PlantUML, it will then display this adjacent to the text editor (see Figure 1.2). This is as opposed to writing PlantUML commands within the source code to generate the diagram (see section 2.22).

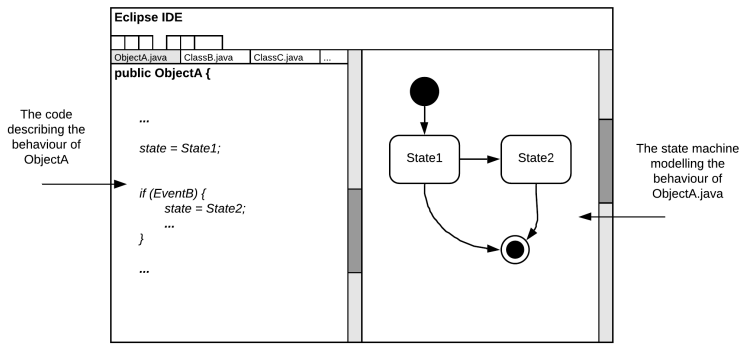


Figure 1.2: State machine automatically generated and displayed adjacent within Eclipse

In addition to this, the plugin will provide users with a way of associating the components in a state machine with the code they represent and vice versa. This will be achieved via navigation between components and lines of code where appropriate (see Figure 1.3).

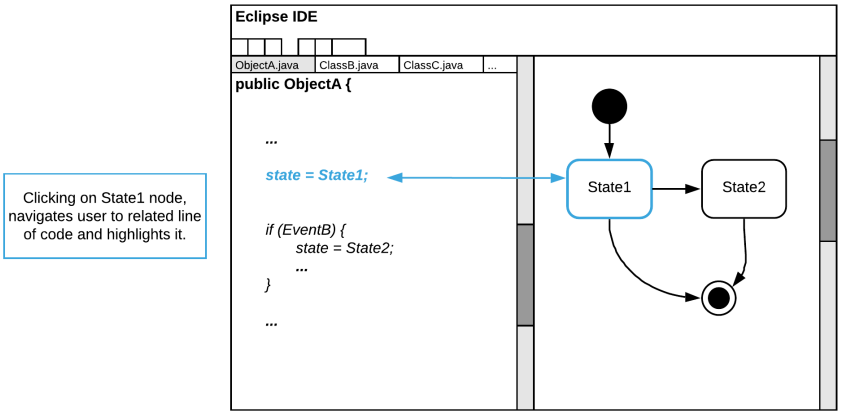


Figure 1.3: The association between a state machine and code via navigation

Firstly, this report will examine existing tools that can be used to annotate state-based software in an effort to motivate the requirements of this plug-in. After this, it will document the process of development, from design and implementation through to testing. Finally a discussion will be had on the results of using it to annotate state-based software, in order to evaluate its strengths and weaknesses over the tools discussed in Chapter 2.

Chapter 2

A Review of Existing Tools

The aim of this chapter is to evaluate the existing tools used to annotate software, with a focus on their strengths and weaknesses as these considerations should guide the development of this project. The chapter begins by noting methods used by developers to annotate code, after-which, there will be a more detailed investigation into the tools that can create state machine diagrams. Consideration will be taken as to whether these provide the capabilities of actually annotating the software as opposed to creating diagrams that can only be used externally.

2.0.1 FlightGear

In order to ensure consistency throughout the remainder of this report, a singular state-based system will be modelled using the tools evaluated in the next chapter as well as the plug-in made during the project. This will provide a fair basis from which to evaluate the tools as each will be exposed to the same software.

This system defines an autonomous flight from San-Francisco to Alcatraz used by the program FlightGear [3] (The source code is given in Appendix A). It has been primarily chosen because the software is relatively simple and therefore it is easy to infer where the components of the state machine are represented in the code. This allows for there to be a simple discussion of the relation between the two throughout the report.

More importantly, however, is the fact that the software was used in an assignment in which students where given an incomplete version of the system and then expected to fix/finish the autopilot and then document it with a corresponding state machine diagram. This makes it an ideal candidate to test the finished plug-in on as this will determine if it eases the process of developing state-based software.

2.1 Annotations as comments

Comments

As mentioned in the introduction, an annotation in the context of this project is a method of labelling the source code in an effort to improve its clarity.

Various ways exist of achieving this, the most basic of which is the writing of comments within the source code itself. They are usually initiated with a simple syntax, yet this

differs from language to language. Generally you can write a comment simply by prefacing it with '//'. For instance:

```
//Queries the plane for the current altitude
//if  > alt -75, plane turns to Alcatraz

If (a.getAltitiudeFt() > alt - 75) {
       state = valid_states.TURN_TO_ALCA;
}
```

Figure 2.1: Using a comment to annotate code

These are useful and easily amendable, and even though adding too many may seem to make the code verbose, Eclipse provides the capability to collapse various comment blocks to free up space and make the code feel less bloated.

There are, however, limitations to the breadth a single line comment can add in an effort to describe the code it was written for, with Jef Raskin stating "their real problem is their forced brevity."[2].

JavaDoc

Another method of annotating software is with a documentation comment which extends and improves the single-line comment. The Java Developmnet Kit (JDK), goes a step further and supplies an automatic generation of the supplied documentation in a neat, easy to read HTML format [4]. The Javadoc tool has a preset syntax to enable this generation and the variance of tags allows the author considerable freedom in describing their code. See Figure 2.2 for an example of how JavaDoc could be used to annotate the manoeuvre sub-routine.

```
/** Example manoeuvre III:
* Fly to Alcatraz at 800ft
* make a manoeuvre to the direction of Golden Gate Bridge
* go down to 100 ft
* fly under the Golden Gate Bridge
* @throws FgException if the communication with FlightGear breaks down
*/
synchronized public void manoeuvre() throws FgException
  {
```

Method Detail

aProcess

```
public java.lang.String aProcess(java.lang.String state)
```

This method takes a state as input and depending on its type, returns a string based on its toString() method. If the state is not identified then the program returns the empty string.

Parameters:

```
state - This is the state given as input
```

Returns:

```
String The toString method of the state.
```

Figure 2.2: Annotating the manoeuvre sub-routine using JavaDoc

Evidently, Javadoc makes a vast improvement to software annotation when compared with simple comments. This is because it goes a step further and illustrates the information in a more formal and visual way. Further to this, they, like single-line comments, are written within the source file themselves meaning they can be fairly easily maintained.

An additional functionality Eclipse adds to Javadoc that is unbeknownst to some people is the fact that the relevant JavaDoc descriptions appear as popups when you hover your mouse over various method bodies. This removes the need for the developer to leave their programming environment to reference the java documentation and makes it extremely easy to understand the structure of the subroutine. Users are then offered even more functionality, as Eclipse gives them the option to navigate to the point of origin of this subroutine within the appropriate class file. Both of these features serve the purpose of clarifying the flow of the code and complement Javadoc as a great tool to annotate software with.

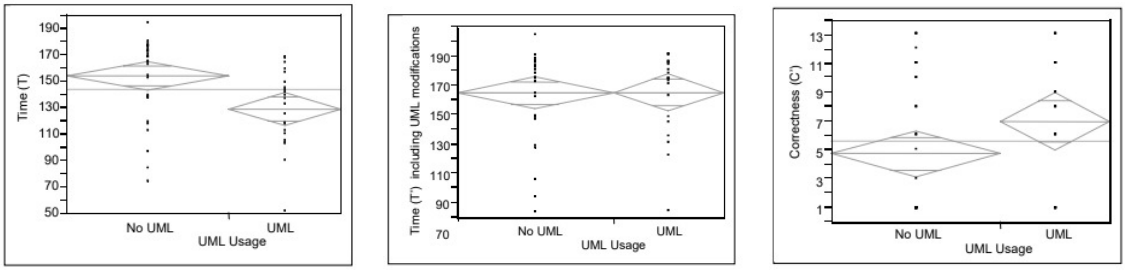
Despite this, there is a significant drawback to documenting code with tools such as Javadoc. They fail to offer diagrammatic capability and therefore cannot be used to annotate source code with state machines. The next section therefore investigates the various tools that can create such diagrams.

2.2 Annotations as UML diagrams

While annotations in software often appear as textual comments, sometimes there is a need to have a visual representation. Processes that involve the construction of UML models such as state machine diagrams, sequence diagrams or database schemas along with various other visualization practices, significantly benefit the development cycle. This is especially true when more than one individual is involved and/or the code is lengthy and complex [5].

Visualizations work so efficiently because "impressions created by diagrams last much longer than those created by the figures presented in a tabular form"[6] - a justification behind the phrase 'a picture paints a thousand words.' It is often the case that if the underlying code is complex, the annotations themselves become complex, making sense only to the original developer. Although perhaps still more narrative than the code, this still presents the problem of understanding the structure of the program. In this situation, an annotation would be far better served as a diagram representing the code in an intuitive way. Moreover, the construction of said diagram, enforces the developer to take a step back and explain design choices, which should ensure that the code they write is of quality. This is an outcome Donald Knuth and his paradigm of Literate Programming [1], would look upon favourably.

Despite this, it is imperative to take into account the fact that these diagrams are much harder to change than comments. In a paper titled 'The impact of UML documentation on Software Maintenance'[7], Erik Arisholm investigated the effect that UML diagrams had on the speed at which various programming tasks could be completed. He tested subjects under four tasks with increasing difficulty and supplied half with relevant UML diagrams (sample A) and half without (sample B). The observation made was that those from sample A, on average, completed the task faster (see Figure 2.2a). However, if one was to include the time required to change the UML model, no savings in time were visible (see Figure 2.2b). Despite this, it is important to note the interesting observation that those from sample A achieved far higher correctness (defined by passed test cases) with an average difference of 2.25 (units here are test cases) (Figure 2.2b).



(a) (b) (c)

Figure 2.3: Impact of UML diagrams on software development [5]

This concludes that, although the effort of constructing and maintaining the diagrams must be accounted for, this is no more than if the diagrams had not been created in the first place. The approach also comes with the added benefit of making the code more qualitative. Even so, when analyzing the following tools, it is important to consider the relative time that would be required to maintain the diagrams, as some of these options make this process trickier than others.

UML modelling tools can be split into two decisive categories: graphic-based UML tools and text-based UML tools. Whereas the first of these incorporates a user-friendly interface that might satisfy the average user, the textual variant, due to its programmatic nature, usually entices the programmer.

In order to evaluate these tools, four case studies have been chosen (two for each category of tool) based upon their workings - which are noticeably different. Their analysis should not only provide motivation, but an indication of what does and doesn't work well in tools that exist to annotate state based software.

2.2.1 Graphic-based UML tools

Graphic based UML tools allow the developer to drag and drop relevant components in order to construct the UML diagram they are attempting to create. Their advantage lies in their simplicity as little knowledge is required to begin modeling. Further to this, they give the developer unparalleled freedom in the customization of their diagrams, whether this be the size, color, shape or positioning of the components they are using.

In spite of this, the graphical tools exhibit various flaws when being used to annotate software. For one, the diagrams are seldom displayed next to the code. This makes it difficult to be used as a visual aid in an effort to understand source code as the user is forced to navigate away from their code to view the diagram. Secondly, once constructed, changing the diagrams to reflect any changes in the code is usually very time-consuming. For this reason, the question must be asked as to whether using these tools to annotate software is worth the effort.

Case-Study 1: Lucidchart

There are various online graphic-based UML tools that provide users with an easy way of generating various UML diagrams via a web application. Good instances of these include Lucidchart [7], Draw.io [8] and Gliffy [9] which all work superbly in making the process of creating these diagrams go as smoothly as possible.

Lucidchart is a variant worth spending particular time examining because it is one of the most widely used tools in the field of diagram drawing, claiming itself to have over 15 million users [10].

It provides a fantastic framework for developers to begin making their diagrams. Users can either choose to begin with a template or start from scratch and then import shapes by specifying which class of diagram they are drawing. The process of constructing the diagram then consists of the simple task of dragging and dropping the desired shapes onto the canvas. Figure 2.3 illustrates Lucidchart's interface and its attempt at modelling FlightGears auto-pilot system.

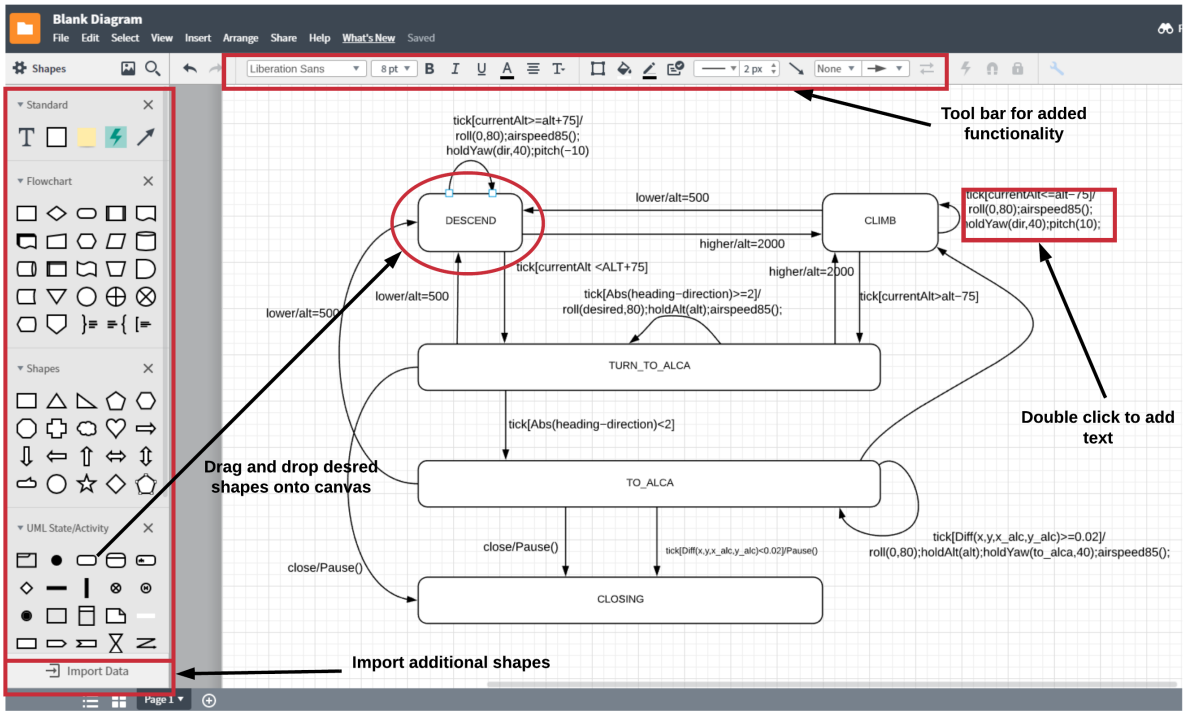


Figure 2.4: A state machine created with Lucidchart that describes the FlightGear auto-pilot manoeuvre

Strengths

The figure above illustrates concisely, the ease of using Lucidchart. Diagrams can be constructed quickly, with little technical knowledge needed - a clear indication of its ergonomic design. The same can not be said with text-based UMLtools, as writing the descriptions requires a plethora of syntactic knowhow. This, then, is the advantage these graphical-based tools possess.

Another of its strengths is that it can be used in any web browser making it very accessible for users. Furthermore, the documents created are saved within the cloud, allowing them to be accessed to view or edit at any time. The tool also provides functionality for team sharing and simultaneous editing on the same diagram. This could be helpful in situations where more than one person is working on a project at the same time - a situation most developers often find themselves in.

Weaknesses

The fact that it requires the developer to create the diagram in a web application separate from the source code, means that Lucidchart can not be seriously considered as a tool to annotate software with. Despite this, it should be noted that the quality illustrated in Figure 2.4 makes it ideal for creating state machines for documentation purposes.

Case-Study 2: Papyrus

Papyrus [11] is an example of a graphic-based UML tool that can be installed into the Eclipse development environment via plug-in. It is therefore worth mentioning as it overcomes Lucidchart's shortcomings by giving developers the freedom to create state machines in the same environment they are coding in.

Various instances of inbuilt graphic-based tools such as Papyrus exist, with notable mentions including but not limited to ArgoUML [12], Magic draw [13] and StarUML [14]. These all work in broadly similar ways and therefore a detailed analysis of one of them, say Papyrus, should provide adequate coverage of them all. Papyrus has been chosen as it is an option that extends the Eclipse environment and is therefore closely related to the context of this project. Also, examples such as ArgoUML, are less sophisticated and only support functionality up to UML 1.4 [15] with no option to undo [16]. Figure 2.5 depicts using Papyrus in Eclipse to model the same state machine as before.

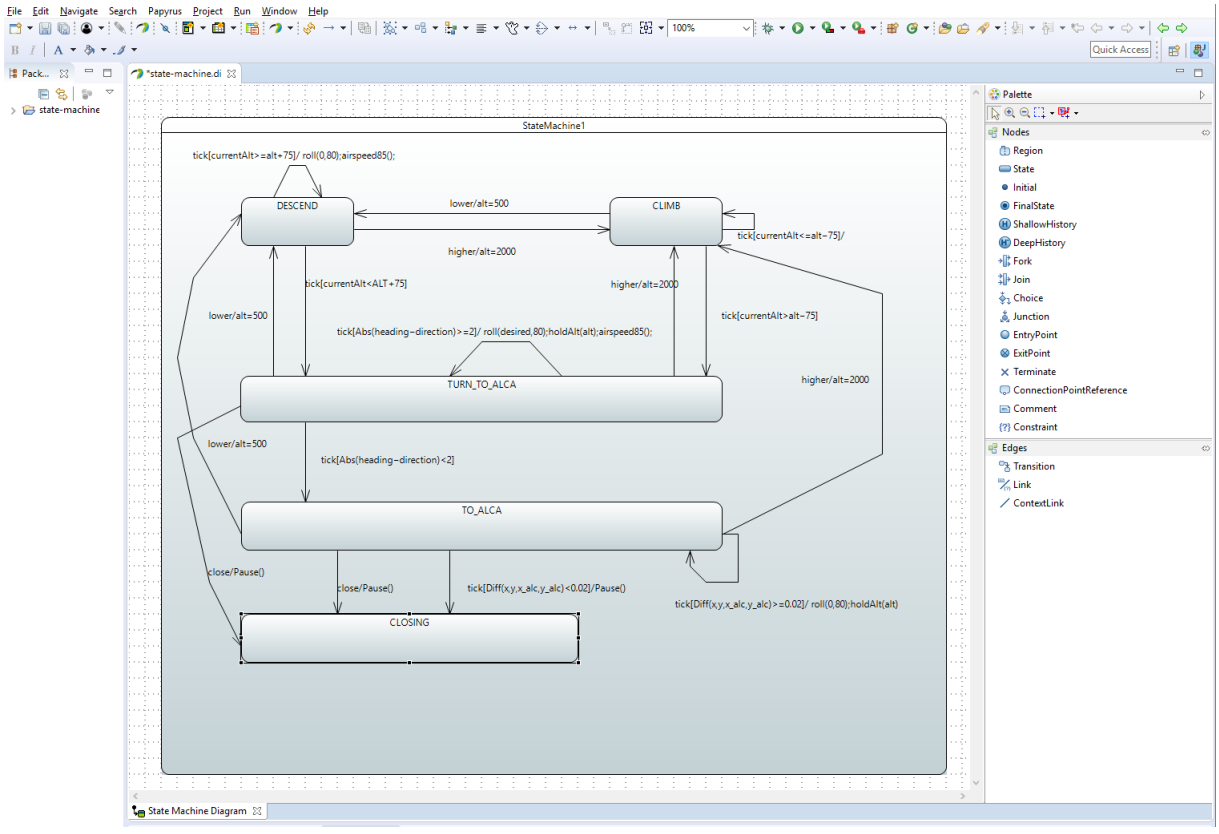


Figure 2.5: A state machine created using Papyrus which describes the FlightGear autopilot manoeuvre

Strengths

One of Papyrus' greatest strengths is the fact that a diagram can be constructed in the same environment that a developer is writing their source code. This means that a source file and its corresponding diagram can be shown simultaneously. This characteristic makes it a very promising candidate when evaluating it as a tool to annotate software with.

Moreover, Papyrus offer the functionality of automatically generating Java source code based on a given UML class diagram. Undoubtedly, if taken advantage of, this could significantly speed up the software development process.

Weaknesses

One downside of using Papyrus is that creating diagrams is more tricky than it is using Lucidchart. This is because the plugin has a daunting depth of usability with countless menus and toolbars, making the overall experience of creating a diagram less smooth. Moreover, although diagrams can be made and displayed in the same environment as the source code, it does not provide the functionality of linking between the two (i.e clicking a component does not navigate you to the code it models).

Additionally, although it can automatically generate code based off a class diagram, this does not work in the opposite direction and therefore editing the source code does not cause the diagram to change. Therefore, the issue of annotation maintenance still exists. Furthermore, the quality of the diagrams between the two options can not compare, with Figure 2.5 showing that Papyrus' appear significantly less professional.

Verdict: Graphic-based UML tools

While it's true that more time has to be spent maintaining the diagrams when compared with annotation techniques such as Javadoc comments, the graphic-based modeling tools still offer the alluring capability of creating good diagrammatic software annotations with ease. Nevertheless, their prevailing drawbacks - largely that diagrams are not linked to the code, hold them back from being an industrial standard.

2.2.2 Text-based UML tools

Frustrated of graphical UML tools and aware of the value it would lend to programming, some developers have taken the time to create tools that annotate software by generating UML diagrams via textual descriptions.

While some of these tools execute in a web application, many of them are integrated into various development environments so that they can work alongside your code. This means the text describing the diagrams can be written within the actual source files themselves (in a way not dissimilar to regular comments). This is a clear illustration of how they avoid the pitfalls of the previously discussed graphical-based modelling tools.

Case-Study 3: PlantUML

PlantUML [17] is without a doubt the most widely used example, with an estimated 65 million online generations since its release in 2009. This popularity has only further

contributed to its success, spawning a large community around it who adapt it to work in different environments. Written in Java, it makes uses of Graphviz to render its diagrams which is done in real-time next to the code. See PlantUML reference guide[18] for a great many tutorials on how various diagrams can be constructed.

Strengths

Like Papyrus, PlantUML has been extended to work within the Eclipse IDE (see Figure 2.6), yet it also gives developers the freedom to describe a state machine in the same class file that the diagram models. Although there are other text-based UML tools that work in Eclipse (such as Umple [17]), they force the user to write the description in a different file which essentially defeats the point of using the text based modelling tool in the first place.

Instinctively, this makes it an excellent tool to annotate software with. This is because it can (by the definition of annotation laid out in the introduction) *truly annotate* state-based software using state machines. Indeed, it is the only tool discussed that harnesses the ability to do this. This is the main reason why this project has chosen it as a foundation from which to extend from.

PlantUML also offers the alluring capability of generating class diagrams based on the source code alone. Although credit must be given for this, it would be far more impressive if other diagrams, such as a state machine, could be generated in this way, as this would save the developer time in writing and then maintaining its descriptions.

Weaknesses

As Figure 2.5 illustrates, the most obvious downside of using PlantUML is the lengthy descriptions required to generate a diagram. These can often take a while to write which also means the process of maintaining them is lengthened.

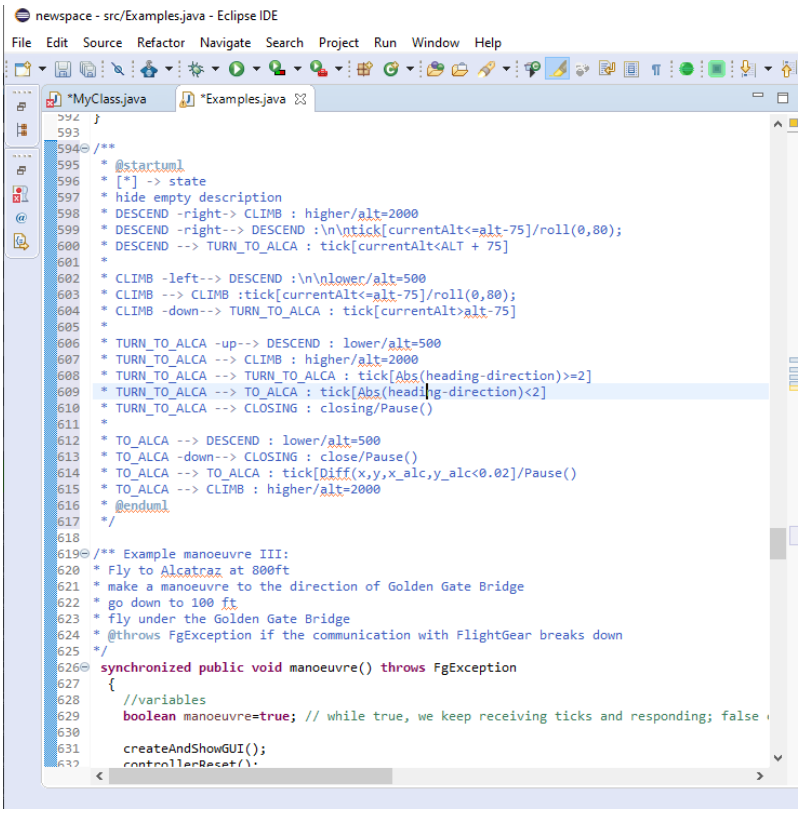
Furthermore, considering their disposition of being generated by command, it comes as no surprise that the quality of the diagrams can not be compared to the standards of tools such as Lucidchart. Indeed, Figure 2.6 is less neatly laid out when compared to Figures 2.5 and 2.4. It could also be argued that the diagram generated are harder to interpret.

PlantUML does however permit some customisation, with options to define colours and position various components, yet this choice is largely left to the renderers discretion. For example, with reference to Figure 2.6 A, it would have been ideal if the looping arrow on the state 'DESCEND' was positioned on the left side of the state, yet this was not an option. It's in this area, then, that text-based tools exert their deficiencies.

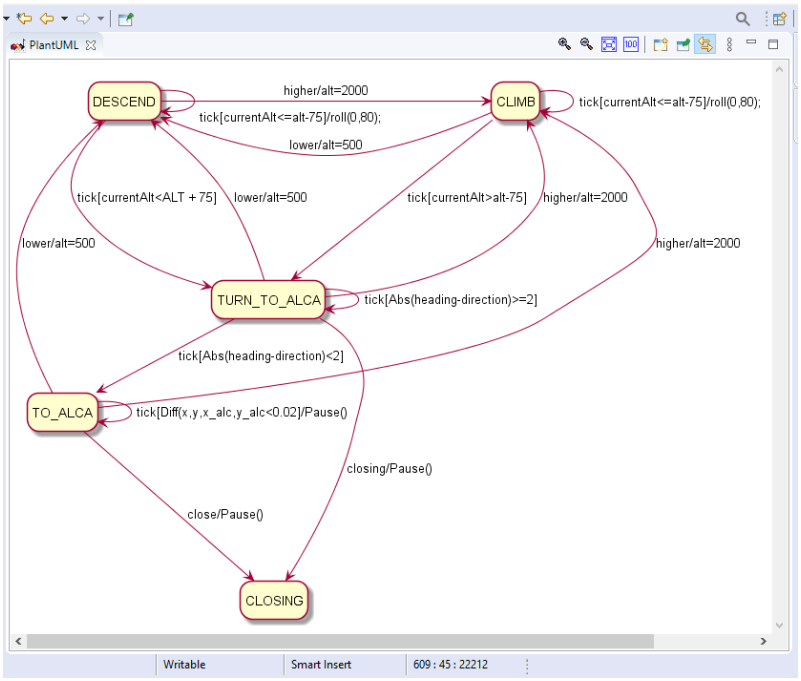
Figure 2.6 is a screenshot of using it within Eclipse to generate the same state machine as before. *Note:* the textual descriptions are written within the Flight gear source file.

Case-Study 4: GraphViz

Despite the fact the tools discussed in this chapter appear to compute the diagrams themselves, the reality is many of them act as an interface between the user and a separate tool known as GraphViz[19]. Similar to PlantUML, diagrammatic descriptions are defined as text and are written in a domain specific language (DSL) known as DOT. GraphViz reads these and converts them into their relative graphical representations. Notable tools



(a)



(b)

Figure 2.6: A state machine created using PlantUML within Eclipse describing the FlightGear auto-pilot manoeuvre

that use it are ArgoUML and Doxygen. *(Note: although PlantUML utilizes GraphViz, it simply uses it to determine the positions of the components in a diagram. It instead chooses to uses the Eclipse API to generate the diagrams).*

To render a diagram using GraphViz, graphic-based tools such as ArgoUML must first convert the UML diagrams from their XMI formats to a DOT file using an XSL transformation [20]. In contrast, text-based tools such as PlantUML instead transform their DSL files to their DOT counterparts by way of translation. Figure 4.2 depicts this more clearly.

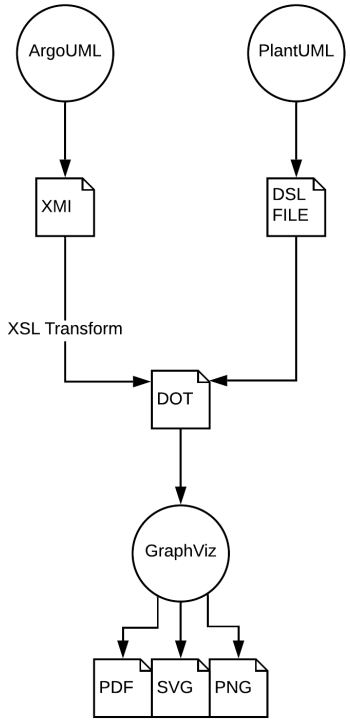
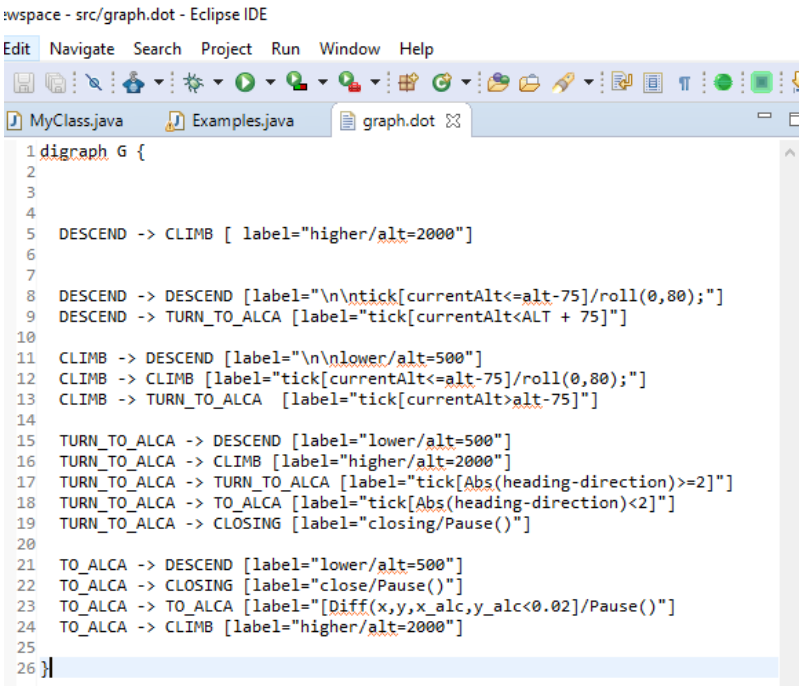


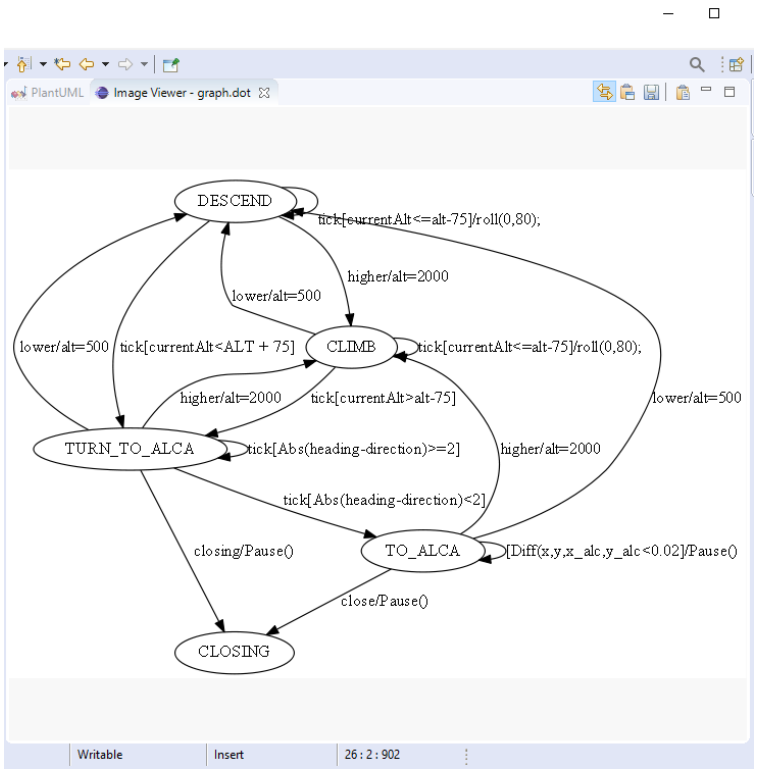
Figure 2.7: How UML tools use GraphViz to render their diagrams

It is important to note, however, that GraphViz does not require an external interface to be used and can act as an independent tool to annotate software with. Like PlantUML, a plug-in can extend it to work within the Eclipse IDE. As Figure 2.8 B illustrates, the diagram generated is fairly similar to the one in Figure 2.7, which likely stems from the fact that PlantUML uses GraphViz to determine the positions of its components. Despite this, a fundamental drawback of using GraphViz to annotate state based software is the fact that textual descriptions can not be written within the source files themselves. Instead, they must be written in a separate .dot file.

It has been decided that GraphViz will not be discussed in the evaluation between the other three case-studies. This is because its characteristics are strikingly similar when compared to PlantUML, and in many regards they fall short. For instance, the diagram generated is arguably of lower quality and further to this, the commands can not be written within a source file. Ultimately then, its evaluation would prove futile.



(a)



(b)

Figure 2.8: A state machine created using GraphViz within Eclipse describing the FlightGear auto-pilot manoeuvre

2.3 Evaluation of Case-Studies

It is important to have an evaluated comparison of the case studies discussed in this chapter, because many of their attributes correlate distinctly with the aims for this project(See "Requirements and Analysis"). Their analysis should therefore not only provide motivation but an indication of what works well and why.

Additionally, because they are industry standard, the below discussion will provide a neat environment for the testing of the plugin designed and developed in later chapters.

Criteria

In order to evaluate these tools in a fair and consistent manner, certain criteria must be established. The criteria have been set as follows:

1. **Ease of use**
   How hard is it to construct the diagram? If it is a text-based tool, is the syntax quick and easy to learn and then incorporate?

2. **Quality of the diagrams rendered**
   Are the rendered diagrams neat and easy to follow?

3. **Consistency of the diagrams rendered**
   Are the outputted diagrams predictable with what the user would expect to be rendered based on their description?

4. **Diagram encapsulation**
   How many diagrams does the tool support?

5. **Can it render diagrams automatically?**

6. **Has the tool been integrated into various development environments?**
   Can developers create diagrams in the same environment they are writing their code in? Can the diagram be shown alongside it?

7. **Difficulty to maintain diagrams**

8. **Difficulty of association**
   How hard is it to associate the diagrams with your code?

The above criteria have been chosen because they summarize the necessary components needed for a tool to be excellent at annotating software. Each tool will be examined against these, with each criteria awarding up to a maximum of 3 marks.

Criteria	Tool	Score	Reason
Ease of use	Lucidchart	3/3	Superbly smooth and intuitive framework allowing users to drag and drop shapes to create their desired diagrams quickly.
	Papyrus	2/3	Scores less than Lucidchart because the framework is more confusing and less smooth. Still gets a high score due to simplicity of drag and drop
	PlantUML	2/3	Scores surprisingly highly considering it requires knowledge of syntax because this is simple and intuitive
Quality of the diagrams rendered	Lucidchart	3/3	Produces the best quality diagrams due to high-resolution components and almost unlimited customization.
	Papyrus	2/3	The diagrams are of acceptable quality and the user has freedom in component positioning but the diagrams are not very customizable.
	PlantUML	2/3	Rendered diagrams are of good quality, however, positioning fairly limited due to being automatically generated. Allows limited customization
Consistency of the diagrams generated	Lucidchart	3/3	Very consistent and predictable because this is up to the user's discretion.
	Papyrus	2/3	Slightly less consistent than Lucidchart
	PlantUML	2/3	Due to its nature, diagrams are fairly consistent because given the same instruction twice the renderer should produce the same output
Diagram encapsulation	Lucidchart	3/3	Supports all UML 2.0 diagrams, and also has the available shapes to construct almost any other diagram.
	Papyrus	2/3	Support all UML 2.0 diagrams but not much more.
	PlantUML	2/3	Supports all UML 2.0 diagrams (but deployment and timing are in beta). It does also support various non-UML diagrams too. See [17] for a list.
Can it render diagrams automatically?	Lucidchart	0/3	N/A
	Papyrus	0/3	N/A
	PlantUML	1/3	Supports the automatic generation of class diagrams based on source code

Criteria	Tool	Score	Reason
Has the tool been integrated into various development environments?	Lucidchart	0/3	N/A
	Papyrus	1/3	Build for Eclipse
	PlantUML	3/3	Extended to work in a huge number of environments (around 105), and works smoothly in these. [21] for a list. Works great for annotating source code in Eclipse as diagrams are shown next to the code and their relative descriptions embedded within the code
Difficulty to maintain diagrams	Lucidchart	1/3	Can prove very difficult to edit the diagrams once they have been made - especially if the user is not the individual who made them in the first place.
	Papyrus	2/3	Because Papyrus works in Eclipse, it means editing the diagrams to reflect changes in the code is fairly easy. However, as the diagram creation is done in a separate file, the tool can't achieve full marks on this criteria
	PlantUML	3/3	Very easy to change the diagrams to reflect changes in code because their textual descriptions are located in the same source file.
Difficulty of association	Lucidchart	0/3	Difficult to relate diagram to code as they would have to be opened in two seperate windows with no links between the two.
	Papyrus	1/3	Difficult to relate diagram to code as they would have to be opened in two seperate windows with no links between the two.
	PlantUML	2/3	Clicking on a description links you directly to the diagram, opening it in a seperate view, yet there is no functionality for linking back.
Results	Lucidchart	13/24	
	Papyrus	11/24	
	PlantUML	17/24	

2.4 Summary

For a tool that proves challengingly laborious if used to annotate software, Lucidchart scored surprisingly high. Indeed, at its core, it works tremendously at providing users with a way to create professional UML diagrams with ease. This caused it to achieve high marks when subject to the early criteria, yet when set in the context of annotating software, it fell short. If it was extended to work in a development environment such as Eclipse, it would overcome many of its harshest criticisms.

Luckily, these criticisms are solved by PlantUML and Papyrus as diagrams can be created within the programming environment itself. The former also goes a step further by allowing the developer to achieve this within the source file - an adaptation that makes it far easier to maintain the diagrams along with the code. In addition to this, it provides a link between the description and the diagram making the association between the two clear. Unfortunately, this link only works in one direction, and clicking on components of the diagram fails to direct you to its description.

Despite this, PlantUML demands a more complex approach to diagram rendering. Even though the syntax is fairly intuitive and quick to learn, it could never compare to the intrinsically simple process of dragging and dropping components that tools such as Lucidchart and Papyrus offer. Its textual descriptions can often become lengthy, resulting in a 'bloated' source file.

Ultimately then, the analysis above has outlined the flaws and technical gaps that industry standard tools for annotating software possess, thereby simultaneously motivating many of the requirements this project should satisfy.

Chapter 3

Requirements and Analysis

Fundamental requirements

As stated in the introduction, the aim of the project is to create a plugin for Eclipse that can be used to annotate state-based software by modelling it with state machines. The idea is that these will be automatically generated and displayed next to the source code at the users discretion. This functionality will help the plug-in overcome the harshest criticism the tools in chapter 2 faced - the effort involved in maintaining them. By automatically creating the diagram, the developer will no longer need to manually update or change the state machine when the software evolves, as any changes in the code will be directly reflected in the diagram. For a list of formal requirements related to this, see requirement topic 1.

Another fundamental requirement that was extracted through the evaluation of the tools in the previous chapter is the idea that components in the generated state machine should be linked to the lines of code that describe them and vice versa. For a list of formal requirements related to this, see requirement topic 2.

While time could be exhausted developing such a plug-in from scratch, it appears to make more sense to utilize PlantUML as a source from which to extend from. Doing so, will allow more time to be spent on the difficult tasks many of the requirements possess, allowing for more progress to be made. For a list of formal requirements defining the plug-ins compatibility with PlantUML, see requirement topic 3.

3.1 Requirement Topic 1: Automatic generation of a state-based diagram

Due to the constraints of the project with regards to time, it is unrealistic to expect the plug-in to be able to automatically generate a state machine that can model the behaviour of any system. The scope of this project will instead prioritise the implementation of a plug-in that can model systems similar to the one used by FlightGear to auto-pilot a plane from San-Francisco to Alcatraz. The fulfillment of this will allow conclusions to be drawn at the inherent use of annotating state-based software in this way. If the results are positive, then the process of extending this to work for more complex systems can be noted under further work.

A Review on State-based behaviour and the code used to model it

As stated in the introduction, a state machine consists of states linked by transitions. The varying states in a state machine symbolise an objects current condition while it either waits for an event, or carries out an action[21]. A transition encapsulates a relationship between two states that is triggered by the occurrence of an event (that may have a guard), which itself occurs because of an action. If a guard is present, then its conditional must evaluate to true in order for the event to cause the transition.

Using the transition between Climb and Descend in the flight gear state machine as an example (see Figure 3.1), the *action* is setting the pitch to -10, the *event* is the lowering of the plane and the *guard* the altitude being greater than 2000m.

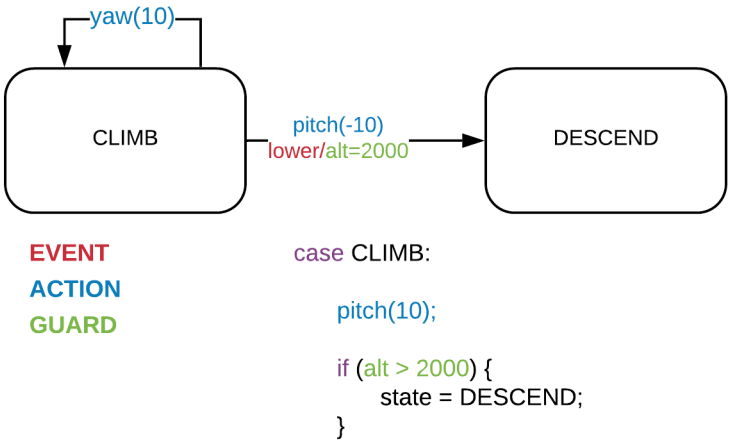


Figure 3.1: Snippet of the flight gear code and its state machine

The nature of events means they are extremely difficult to infer. Unlike guards, they rarely have a literal definition in state based software. For example, the auto-pilots source code shows no indication of the event *lower* shown in Figure 2.1. For this reason, although the plug-in will be expected to infer actions and guards, the deduction of events will not fall under its scope. If necessary, users can specify an event for a transition by using standard PlantUML commands.

The self-loop transition shown in the diagram occurs when actions and/or events don't require the object to leave the state it is currently in. In state-based systems these often occur as a result of various loops that require a condition to be met in order for the loop to be broken. This occurs in the auto-pilot system through the use of the switch-statement being placed within a 'while loop'.

Unfortunately, State machines can become even more complex due to the fact that states, under certain conditions, need not be visible. For instance, if a state fails to make a call to another visible object between it being initialized and the next state being declared, then that state will not appear in the diagram. The reason for this is that the transition between the two states occurs atomically and therefore any external entities are not able to access or even observe the source state. It is only when an object (either the one currently being modelled or another visible one) makes a call to another object, that a state is registered in a machine. *Note:* A call can, in turn, call another object which itself may make a callback to the original object or call another object. Otherwise it can simply return a value. Figure 3.2 illustrates the affect of callbacks (highlighted in red) on state visibility.

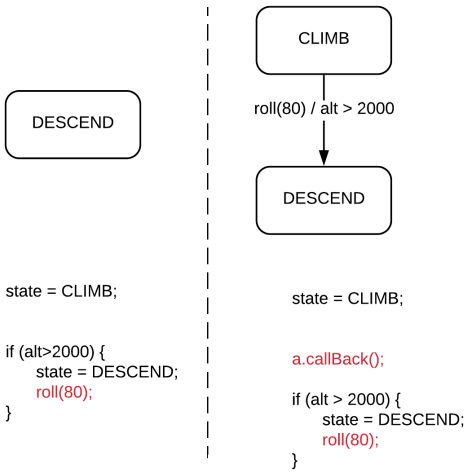


Figure 3.2: The effect of callbacks on state visibility

Figure 3.1 formalises a list of requirements that encompasses the functionality necessary to generate a model for software similar to FlightGear (See Appenix A for its source). Each requirement below has been scored according to two scales - desirability and complexity. These criteria will help determine the timeline and structure of the project, by prioritising requirements.

#	**Requirement**	**Desirability**	**Complexity**
1	Every possible state is inferred from the source code.	5/5	3/5
2	Every possible transition is inferred from the source code.	5/5	5/5
3	Guards (See Figure 3.1) are inferred from source code and label relevant transitions.	5/5	4/5
4	Actions (see Figure 3.1) are inferred from the source code and label relevant transitions.	5/5	5/5
5	Where appropriate,an initial state transitions to the first state.	5/5	1/5
6	Where appropriate, states correctly transition to the exit state.	5/5	4/5
7	Self-loop transitions are inferred if an action/event doesn't require the object to leave its current state.	4/5	3/5
8	A states visibility is inferred (see Figure 3.2), and this determines whether it is shown in the diagram	4/5	5/5

Table 3.1: The requirements that encompass the automatic generation of a simple state machine

3.2 Requirement Topic 2: Creating links between diagram and code

This requirement has been mentioned extensively and reaching it is of upmost importance. In this context, a link refers to illustrating an association between components in a diagram and the code that describes them. This will be achieved in both directions, i.e from code to diagram and diagram to code. Creating a link from diagram to code will involve highlighting components in the diagram when a line that was used to model them is clicked on. Creating a link from code to diagram will involve physically navigating the users cursor to the line/section of code that describes the component that was clicked (See Figure 1.3 to see a concept of this.)

Providing this functionality will give the developer a comprehensive view of the source code and the behaviour it models, consequently allowing them to adapt the code or write extensive tests more easily. Therefore if satisfied, this requirement will make the plug-in successful in its objective of easing the process of developing state-based software.

Various options exist to fulfill this requirement and these will be explored in chapter 4 during the 'Design and Implementation' of the plugin. The actual requirements that these options must meet are listed in Figure 3.2:

#	**Requirement**	**Desirability**	**Complexity**
9	Clicking on a state node links you to its definition in the source.	5/5	4/5
10	Clicking an action label on a transition links you to its definition in the source.	5/5	4/5
11	Clicking a guard label on a transition links you to its definition in the source	5/5	4/5
12	Clicking on a line that defines a state highlights the relevant node in the diagram	5/5	3/5
13	Clicking on a line defining an action that causes a transition, highlights the transition.	5/5	5/5
14	Clicking on a line defining a guard for a transition, highlights the transition it guards.	5/5	4/5

Table 3.2: Requirements for creating a link between diagram and code

3.3 Requirement Topic 3: The plugins compatibility with PlantUML

The fact that this plugin aims to extend PlantUML means that it must work seamlessly alongside it. This comes with the added benefit of using and adapting some of its features to complement the generation of state-based diagrams. For example, explicitly defining a

transition if it has failed to be inferred.

Furthermore, because the specification of this project is to create a plugin, the end product should be efficiently packaged up and then deployable, so that it can be used in practice.

#	Requirement	Desirability	Complexity
15	Work seamlessly alongside PlantUML	5/5	3/5
16	Ability to explicitlywrite states and transitions in case they are missed.	5/5	1/5
17	Ability to write a command that removes unwanted state and transitions	5/5	2/5
18	Clicking on an explicitly written PlantUML statement links you to the relevant component in the diagram.	5/5	3/5
19	Where appropriate, clicking on a component in a state machine navigates you to the PlantUML command describing it.	5/5	3/5
20	Fully packaged plug-in that can be installed and used within Eclipse	4/5	3/5

Table 3.3: Requirements listing how the plug-in should work with PlantUML

3.4 Optional requirements

Again, the following requirements have been scaled against desirability and difficulty to implement. This has been done in order to prioritise the list in case there is time to develop additional functionality.

#	Requirement	Desirability	Complexity
1	Functionality that infroms the user of any trap states/transitions when a button is pressed. (States that are impossible to reach/exit)	3/5	3/5
2	If non-determinism is introduced in the source, multiple diagrams are displayed, adhering to the options	2/5	5/5

Table 3.4: A table listing the Optional requirements

3.5 Testing & Evaluation

Three different approaches to testing will be used to ensure the functionality of the plug-in meets the requirements laid out in this chapter. Namely: unit tests, integration tests, and end to end tests. Each of these will be implemented using the JUnit Framework [22].

Unit Tests

Unit tests are an example of a simple test that can be conducted on a system. This is because they test methods independently against a predisposed number of inputs in an attempt to try and assert their outputs. Ideally, each function should be tested to ensure that they are all working correctly. This will simply require iteratively working through the classes and testing every method that can be called in isolation. *Note:* due to the planned system design (see "Design and Implementation"), there may be some routines that are fairly highly coupled (very dependent on the functionality of other routines). These examples will instead be tested during the integration phase of testing.

Integration Testing

Integration testing is a higher level of testing in which units are combined and tested as a group. This is crucial in order to expose any faults in their interaction. As stated above, some classes will be highly dependent on the functionality of others and it is in this phase that they will be tested.

End-to-End Testing

An even higher level of testing that will involve validating that the behaviour of the plug-in works as expected from start to finish. This phase will be useful in determining whether the system meets the requirements promised in this chapter and therefore will be one way of evaluating the plug-in.

Evaluation

The evaluation of the plug-in will involve conclusively investigating the extent to which it achieves its fundamental aims. The introduction specified that the project description was to create *"a plugin for the EclipseIDE that will extend PlantUML, improving its functionality in creating and maintaining state machine diagrams"*. This is, however, different to the project aim and can be mostly tested during the three phases above.

More specifically then, the project aim encompasses the reason behind making the plug-in in the first place - to help users develop state-based software, either by helping them understand legacy systems or to verify that their system works as intended. The procedure to evaluate this will be similar to the one carried out by Erk Arisholm [7] (see Section 2.2). Two sample groups of programmers will be given the FlightGear assignment (See 2.0.1, or Appendix A for the source), and asked to complete the auto-pilot under timed conditions, with one group having access to the plug-in. The quality and correctness (does the systems behaviour work as expected?) of their submissions will be indicative of the value of the plug-in and an excellent way to evaluate its use.

Chapter 4

Tools used in the Development of the Plug-in

This chapter will give a general insight to the Eclipse IDE, explaining why it is the chosen environment to develop the plug-in within. There will also be a discussion on the inner-workings of PlantUML and its extension into Eclipse, along with a brief definition of the various components in a state machine. These are necessary to set the stage for the design choices discussed in the next Chapter.

4.1 The Eclipse IDE

Since 2001, the Eclipse Integrated Development Environment (IDE) [23] has solidified itself as a staple for aspiring Java programmers (9.62% marketshare [24]) due to the extent to which the resource can be used to develop Java applications (yet this is not whats it's restricted to [25]). Truly, its proficiency lies in its modularity, giving users the freedom to configure and adapt their work-space through the use of countless plug-ins. Then, if functionality is not as expected, the open source nature of the software permits personal changes so that the editor can be customized to suit any need. In the words of Jeff McAffer, it 'is like the enormous rockets that carry NASA's robots into space: powerful, sophisticated, essential, but ultimately just the launch vehicle that propels our creations safely to their destinations"[26].

(Note; With the release of Eclipse 3.0 a new component model was introduced (OSGi) which now referred to Plug-ins as bundles. These can be used interchangeably, yet for consistency plug-in will be used throughout this report.)

Strictly abiding by open-source principles, Eclipse encourages its community to extend its architecture through the implementation of plug-ins. This process is made simple with the Eclipse Software Development Kit (SDK) - an architecture comprising three major components. These are the Platform, the Java development tools (JDT), and the Plug-in development environment (PDE). The latter provides developers with the tools required to create plug-ins.

The accurate definition of a 'plug-in' within Eclipse is a JAR file compiled of its functionality along with a manifest file (MANIFEST.MF) which describes its dependencies (on other plug-ins), yet also its extensions (by other plug-ins). These extensions, known as extension points, act as a blueprint by identifying the interfaces developers need to use when writing an extension. It's important to note here that everything in Eclipse can be

defined as a plug-in. Categorically, this stresses its impressive modularity, and highlights the extent to which this platform is truly configurable. Each component can be seen to act as a socket that can be plugged into the Eclipse application, as and when they are needed. Sockets can therefore either use or extend the functionality of their neighbours.

The PDE can also be used to deploy and test developing plugins by launching a second instance of Eclipse, known as a 'runtime instance', within the current instance. On its launch, the PDE examines the dependencies of the plug-in and builds it in the new instance, providing a quick and easy way to test implemented features.

Overall, the PDE and its accompanying components such as the JDT, make the Eclipse IDE the perfect choice for which to build the plugin. With continuous iterative refinement, the Eclipse SDK has simplified the development process by way of an extensive framework with complementary documentation that can be exploited to design a plug-in for any purpose.

4.2 Why PlantUML?

Although briefly touched on, there are various reasons why using PlantUML as a base from which to extend from makes logical sense.

As discussed, it's a solid tool for annotating software, achieving a score of 17/24 during the evaluation in Chapter 2. On top of this, it satisfies much of the projects specification:

- Extended to work in Eclipse using a plug-in, with intuitive extension-points set-up to encourage expansion.
- Dynamically renders diagrams next to the source code.
- Builds an association to GraphViz so that it can be used as a renderer.
- Open source project which encourages experimentation.

Despite the fact these factors make it an ideal candidate, there are some downsides when extending open-source software. The obvious one is the degree to which the software must be comprehensively understood before any changes can be made. Inferring its workings can be a difficult process, and in particular PlantUML seldom uses comments to help ease this understanding. Furthermore, once understood, the design paradigms used in the software often have to be followed by way of interface or for other reasons. If the code you are working from is cumbersome, this in turn requires your code to be cumbersome.

The alternative would be to develop a standalone plug-in that meets the criteria above, plus the requirements laid out in Chapter 3. Whilst this would relieve some of the heartache above, it would involve a great deal of time with not much of the effort directly resulting in novel features. For instance, building the User Interface, making an association with GraphzViz, and rendering the diagram are features that exist in all of the tools examined in chapter 2. While their implementation could be re-built and unique, the time required to achieve this would be large, and their evaluation barely note-worthy. By making use of PlantUML, more time can be spent working on the novel features, especially those that overcome its weaknesses, hopefully increasing their overall quality.

4.3 PlantUML and its integration with Eclipse

Although PlantUML is a standalone tool separate from Eclipse, it has been integrated with Eclipse so that users can define diagrams in the editors they are writing their code. This means there are two libraries that need to be considered: PlantUML-Eclipse, which acts as a plug-in within the IDE, and PlantUML-core. The former deals with the Eclipse side of things, such as extracting the text from the editor and displaying the diagram in a view, while the latter, using GraphViz, deals with the actual rendering of the diagram. Figure 4.1 below illustrates this more clearly.

(Note: although the plug-in consists of 10 packages, only the Eclipse and text packages are required to carry out the process in Figure 4.1)

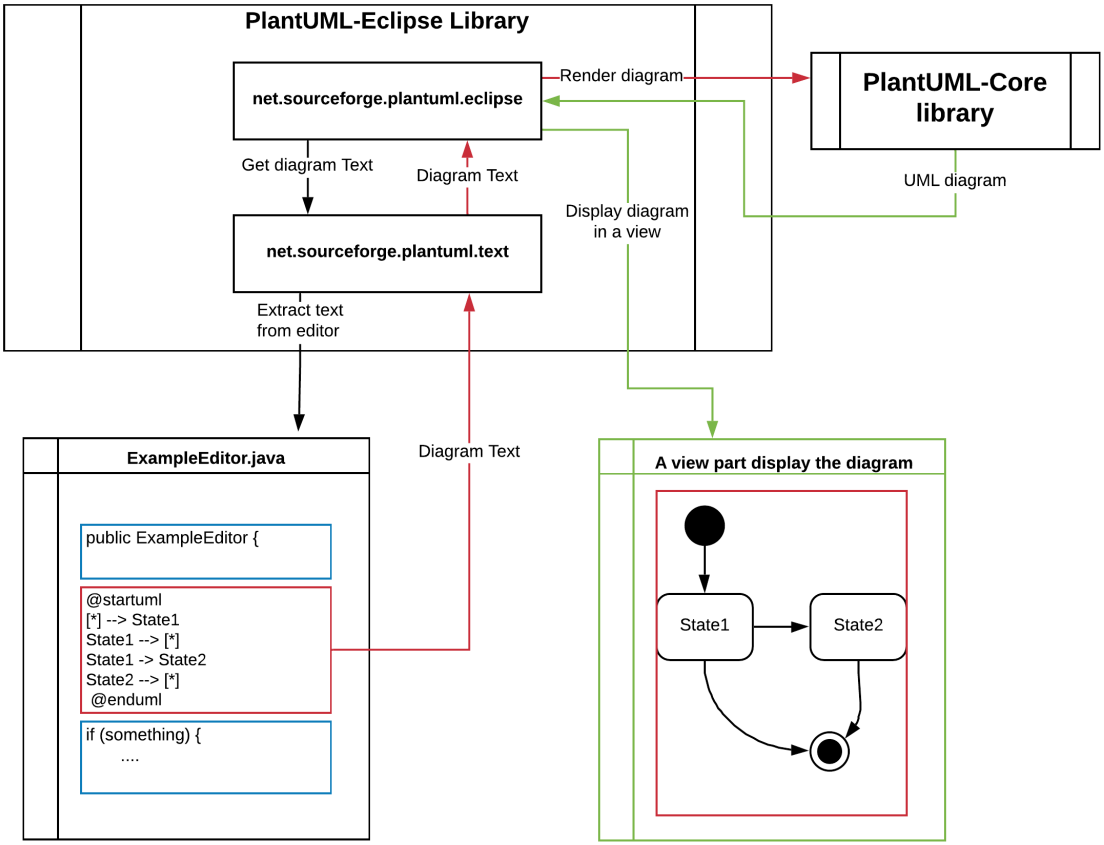


Figure 4.1: Flow diagram illustrating the process of rendering a diagram

The process above is executed when a diagram is issued to be drawn. The primary way to initiate such an execution is by clicking within a PlantUML description block, such as the one outlined in red. Then, if any changes are made in that description area (such as a new transition), the process above is run again. This results in a dynamically rendered diagram that responds to change with relative speed.

To extract the diagram text from the editor, the text package calls TextDiagramHelper.java. Starting from the location of the cursor, this class scans the active editor for the diagram text prefix (@startuml) and suffix (@enduml). If these are both found, then the class knows the user has clicked within a diagram text section and therefore extracts the text between the two. This text is then sent to the core library to deal with the task of drawing

the actual diagram.

4.3.1 Extension points

As mentioned, the PlantUML-Eclipse plug-in utilizes extension points to encourage expansion. Some of these can be used to enhance the text extraction process described above. The ones that achieve this functionality are coined DiagramTextProvider and TextDiagramProvider. Whilst the former specifies an interface which deals with the methods of obtaining the text, the latter is used to define custom diagram markers such as @startuml.

A further extension point that will be of particular interest to us during the design phase, is termed linkOpener. Intuitively, this defines an interface which deals with how links are opened.

Chapter 5

Design & Implementation

This chapter outlines the process of developing the State Diagram plug-in. Each section begins by detailing early design concepts and then illustrating how and why these ideas were finalised and thereafter implemented.

(Note: From this point onwards, the plug-in that will be made in this project will be refereed to as SDP (State Diagram plug-in) and the PlantUML-Eclipse plug-in reffered to as PEP. The hope is that this will improve clarity.)

5.1 General concept

Similar to the techniques used by the PEP, the SDP will extract the text from the editor describing the diagram. It will then iterate over the extracted lines with a parser in order to elicit the information necessary to construct the links and infer the diagram. Once complete, the generated text can be appended to any explicitly written PlantUML commands and then sent to the core library to be rendered and displayed in the adjacent view as usual.

5.2 Working alongside PlantUML

The requirement that the SDP should be a unit separated entirely from the PEP, means that it must implement its own text extraction techniques. Fortunately, this process is made easy with the 'net.sourceforge.plantuml.eclipse.diagramTextProvider' extension point, which acts as a pointer to a class that must implement the extension points' interface. By using it in the SDPs manifest file, the PEP is informed by the Eclipse IDE of the extension and therefore knows that a new class to extract text has been defined. With this association created, the SDP can now override the methods used to extract the text by the PEP thereby executing its code rather than the code in the PEP.

The problem arising from this, however, is for each of the plug-ins to know when it is suitable for their code extraction classes to be called. By utilizing a further extension point (net.sourceforge.plantuml.text.textDiagramProvider), the SDP can define its own diagram prefix making this segregation clear.

Once the code has been successfully extracted, the SDP can make any modifications to the text it wishes, and then pass this text to the PEP which itself sends to the core PlantUML

library for it to be rendered. The PEP will then be used once more to display the rendered diagram in a view next to the code. Figure 5.1 illustrates this idea with a flowchart.

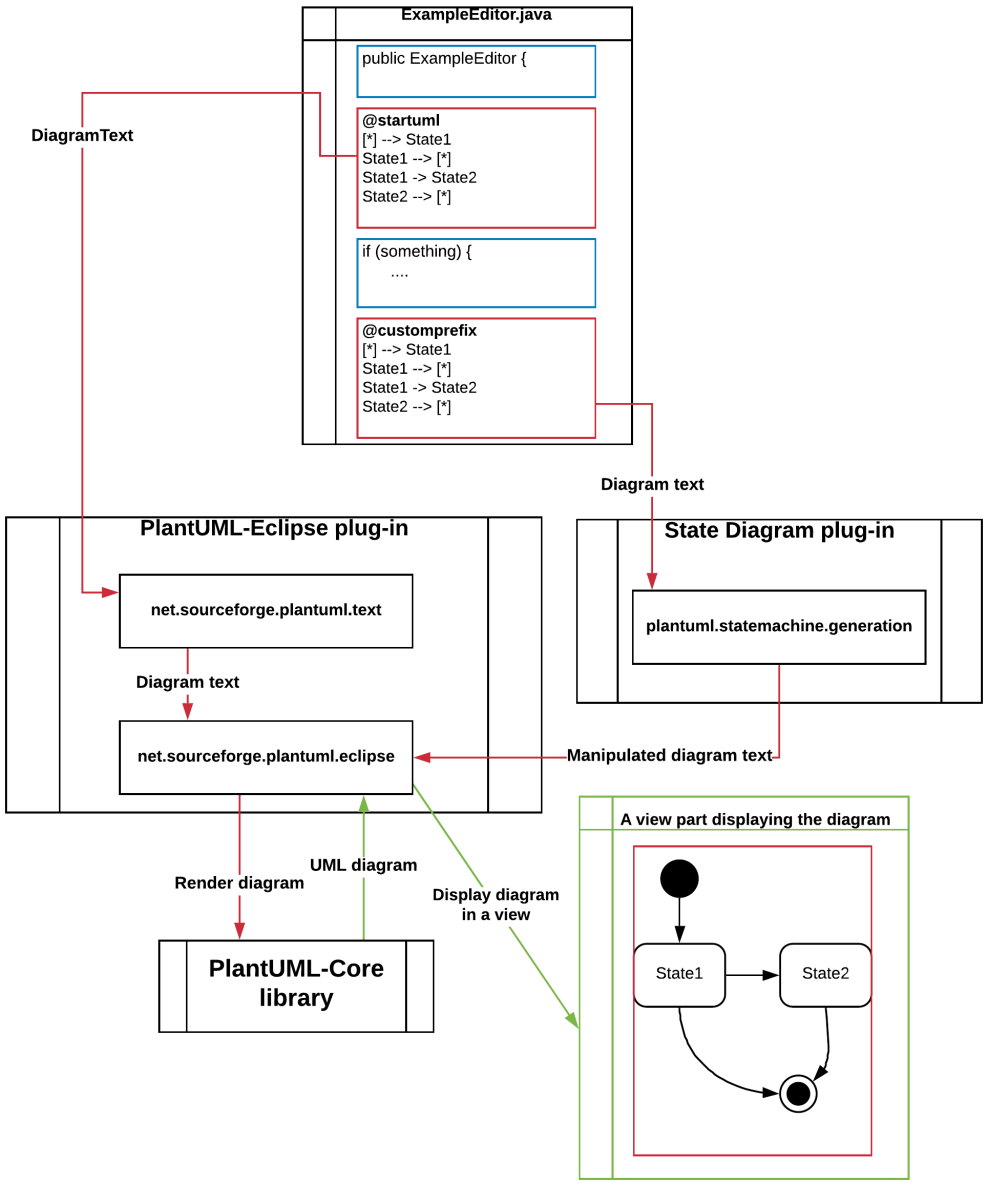


Figure 5.1: Flowchart illustrating the text extraction process of the two plug-ins

5.3 The Parser

In sum, the parser has the crucial job of inspecting the code, inferring what each line means. After this has been determined it methodically initializes data-structures in order to store the behaviour of the system which it uses to generate the state machine after it has finished parsing the text.

Initially, it must determine whether an inputted line is an explicitly written PlantUML command, or a line of code. To make this task easy, the SDP requires users to preface PlantUML commands with '//FSM:'.

If passed a PlantUML command, the parser simply needs to call the subroutine used to

create links with the line as input. Once this is done, it can append it to the string used to store the entire diagram.

If passed a line of code, the parser must govern whether it describes a behaviour component in a state machine. One way of achieving this is by using regular expressions to match input strings against a pattern known to describe a particular component. For example, the following regular expression can be used to filter a state declaration:

```
(state)\s*\=\s*([a-zA-Z0-9_\\-.()]+)\s*\;\s*
```

If the string matches the pattern, then the parser knows that the line was a state declaration and can extract its name by specifying which group (defined by parentheses) it wants extracted.

5.3.1 If statements

This concept can be extended further to work with multiple regular expressions at the same time. Another instance of a pattern is given below, this time for if statements:

```
(if|\}?\s*else\sif)\s*\(\s*([a-zA-Z0-9\s\[\];|&.,()!=_\-<>+*]*)\s*\)\s*(\{?)
```

If this pattern is matched, the parser knows the code following it is part of an if/then block. This information can be pushed to a stack and later popped when the parser matches a closing '}'. If the parser needs to decide whether a state is protected by a guard, it can cross reference it with the stack to determine whether it is within the scope of a conditional (See Figure 5.2).

Utilizing a stack in this way is a fundamental aspect that will be used throughout the parser's design. It's strict push/pop functionality, is a perfect way to track the control and scope of any code block.

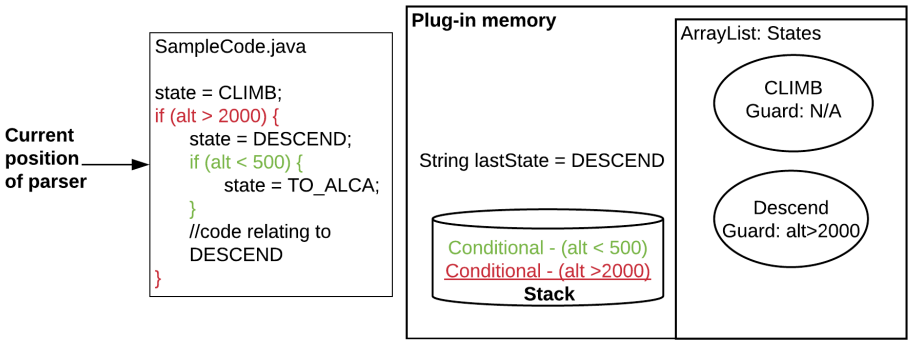


Figure 5.2: A preliminary insight into the data structures initialized by the parser

As figure 5.2 should begin to point out, the idea is that the parser will iteratively initialize data structures that will be used to generate a diagram once the text has been successfully parsed.

5.3.2 Computing visible states

The method of computing a state's visibility can follow a similar approach. Every state read by the parser can be pushed to a stack. Then, if a callback pattern is matched by the parser, it can set the state on top of the stack to visible. This can be done because the top of the stack will always refer to the state the object is in at that point of time in the program. Figure 5.3 illustrates the enhancements of the design now that a state's visibility has been defined.

Note: Although in more complex programs an object has the *potential* to be in a number of states, the top of the stack will always refer to the state an object can *definitely* be in at that time.

Two rules are defined to ensure the integrity of the state-stack. The first of these rules pops a state A from the stack if A was declared behind a guard, the closing of which('}'), was just matched by the parser. This can be done because this signifies the parser is now out of the states scope. The second rule states that if an unconditional state is discovered (a visible state, not declared behind a guard), then the stack of visible states is cleared and this state is pushed to the bottom. This is safe because all states declared before that point must intuitively transition to that state regardless.

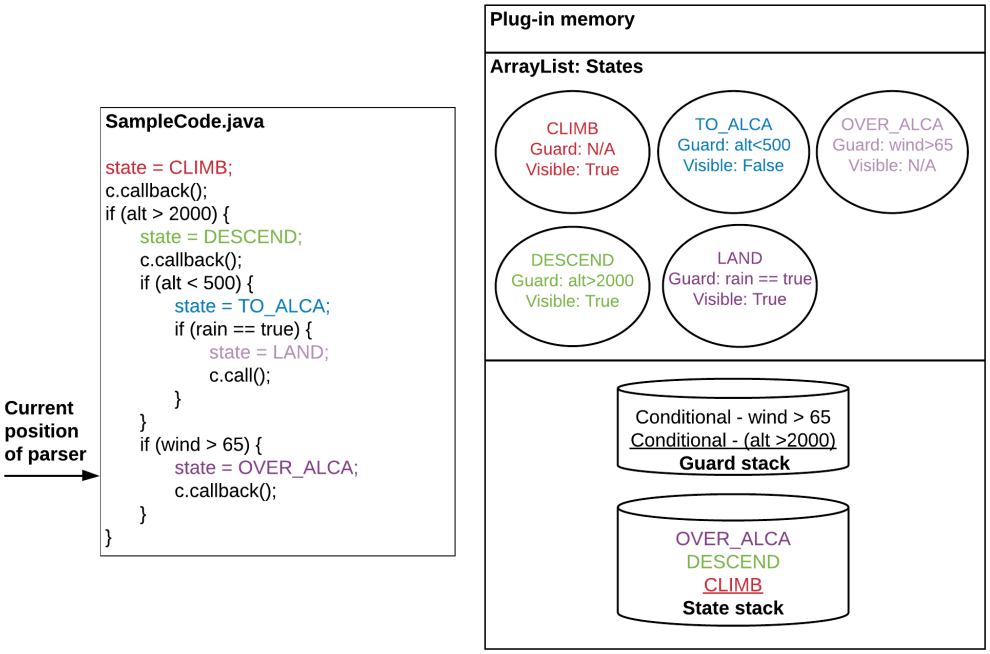


Figure 5.3: Enhancement of Figure 5.2, with the introduction of a stack to store states

5.4 Generating PlantUML commands

Using the designs above, a state is modelled and stored independently, with no regard to the structure of the code such as the relations that exist between the states. To store this knowledge, a new data structure is required that can model the flow of control in a system.

This requirement can be addressed using a tree structure in which:

1. The nodes relate to the different state models, storing their name, visibility, and any actions/events or guards that led to its declaration.
2. The root is the first *visible* node found by the parser.
3. A parent of a node is the state, which at the time of adding the node, was on top of the state stack.

Using the same code as in Figure 5.3, Figure 5.4 illustrates how a tree is structured.

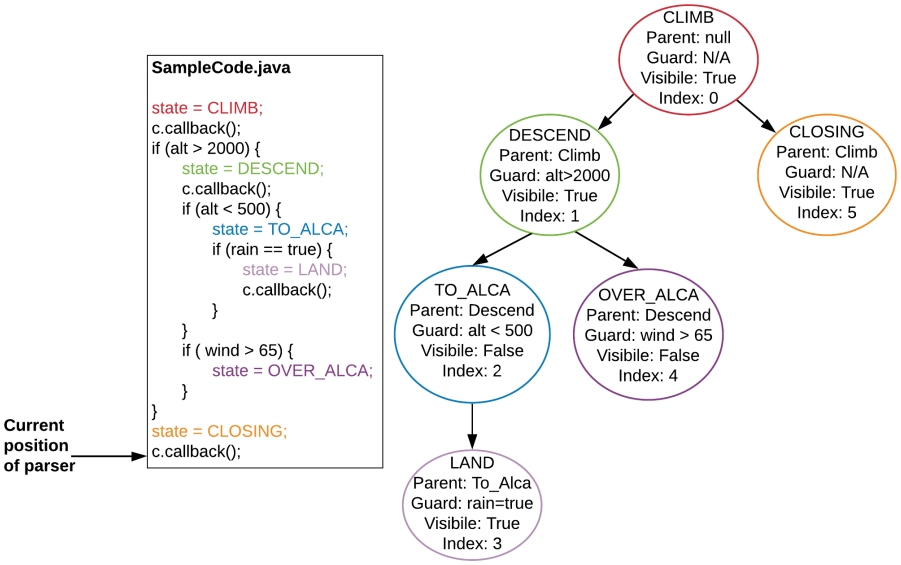


Figure 5.4: Concept of storing a state machine as a Tree

Using the tree structure above, a sub-routine was designed to compute the transitions between various states. After some calibration, the algorithms in Figure 5.5 have been written to find a path between any two nodes as well as initialize an array with the nodes met on the way.

```
GET-TRANSITION(from, to, tree):

1: route[] = {}
2: nodesMet[] = {}
3: rootToTarget = INITIALIZE-ROOT-TO-TARGET(to);
4: while(!targetFound) do

5:     for child ∈ tree.getChildren(from) do

6:         if (child.index > from.index) then
7:             if (FIND-TARGET(to, child, tree)) then
8:                 targetFound = true;
9:                 break;

10:    if (from != root) then
11:        from = from.parent;

12: return (route, nodesMet);
```

```
FIND-TARGET(to, child, tree):

1: if (child == to) then
2:     route.add(child);
3:     if (!to.visible) then
4:         nodesMet.addAll(tree.getVisibleChildren(to));
5:     return true;
6: else if (child ∈ rootToTarget) then
7:     route.add(child);
8:     for child ∈ tree.getChildren(from) do
9:         if (FIND-TARGET(to, child, tree)) return true;
10: else if (child.visible) then
11:    nodesMet.add(child);
12: else
13:    if (!to.visible) nodesMet.add(child);
14:    for child ∈ tree.getChildren(from) do
15:        if (FIND-TARGET(to, child, tree)) return true;
16: return false;
```

```
INITIALIZE-ROOT-TO-TARGET(to):

1: rootToTarget[] = {};
2: while (to != root)
3:     rootToTarget.add(to);
4:     to = to.parent;
5: return rootToTarget;
```

Figure 5.5: Functions used to traverse the tree structure

By appending the guard values from *route* and negating and then appending the values from *nodesMet* to a string, the plug-in can create a label for the transition between two states. Then, to generate the actual PlantUML command, it simply needs to enter these values into the following expression:

$$from \ -> \ to \ : \ label;$$

For instance, calling Get-Transition(Descend, Closing, tree) will output:
{Closing},{Land}, creating the label: (!rain) and the command: Descend $->$ Closing : !rain.

The formula to construct every transition that would exist in a particular state machine then becomes;
$Get-Transition(\forall state \in visibleStates, \ \forall anotherState \in allStates : where \ anotherState.index > state.index, \ tree)$

To generate the state commands, the plug-in simply iterates through a list of the visible states, building a string by prefacing each one with "state ", and then appends this to the final string.

5.4.1 Actions

Actions can be easily determined as they are usually represented through method calls. If an action causes the state of the object to meet the condition of a guard, then the action is appended to the label on the transition to the new state, else it is appended to the self-loop. Working this into the plug-ins design, the state model now initializes an empty *action-array*. If an action is matched by the parser it must be acknowledged and stored by any of the states the object could be in at that point time. This is achieved by appending the action to the array of the state at the top of the state-stack, doing the same for each of its descendants. Then, when the label for a transition is being generated (traversing the tree), it can append the actions stored in the source state, along with the guards.

5.4.2 Dynamically rendering trees

At certain points during the life-cycle of a state-based object, the object, regardless of the events before, can only be in one state. Intuitively, none of the actions, events or guards declared after this point can affect the states declared above. This means that a new tree can be initialized each time an unconditional state is declared which keeps the number of states in each tree to a minimum. This in turn means the algorithms used to traverse a tree are more efficient resulting in a more performant plug-in.

5.5 Constructing the links

A fundamental prerequisite when constructing links, is the positional information about a line such as file name and line number. This knowledge is readily accessible to the parser and can therefore be stored in the state model. As well as updating the current state model, this requires the introduction of a new model to represent guards. Figure 5.6 illustrates these new models. *Note: the new action field in the state model.*

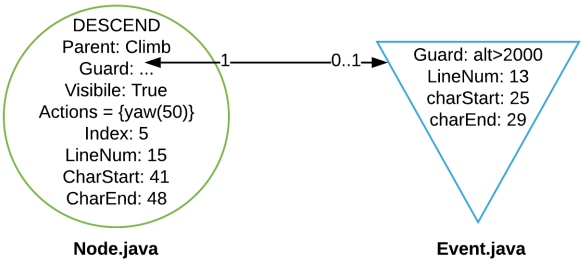


Figure 5.6: Association between the state model and the guard model

5.5.1 Linking from the code to the diagram

To achieve this direction, one approach could be to enlarge the section of the diagram corresponding to the code the user is editing. The same result could also be achieved by displaying the component in a different color to the rest of the diagram. Both of these options can be achieved using simple string manipulation on the generated PlantUML commands.

PlantUML defines the syntax; $CLIMB -[\#Green]-> DESCEND$, to color a transition green and *state CLIMB #Green* to color a state green;

Implementation

To automate this process, the plug-in compares the line number the cursor is on with the line number of every component generated. If a transition or a state has an equivalent line number FORWARD-TRANSITION-LINK or FORWARD-STATE-LINK, are called respectively.

```
FORWARD-TRANSITION-LINK(selectedLine):
1: colorTransition = selectedLine.substring(0, indexOfArrow) +
        "-[thickness=5,"+transColor+"]>" +
            selectedLine.substring(indexOfArrow +2, selectedLine.length());
2: return colorTransition;
```

```
FORWARD-STATE-LINK(stateName):
1: return "state " + stateName + " 'Cyan";
```

Figure 5.7: A method that can be used to color a state or transition

5.5.2 Linking from the diagram to the code

A link in this direction is more difficult because the components of a diagram are computed in the core library and therefore their relative positions will not be accessible.

A solution for this is to create a custom Eclipse marker[27] for each state and transition. A list of these can then be displayed in a view next to the diagram, allowing the user to click on an item describing a component in order to be linked to it in the text. This solution is rather cumbersome and lacks the finesse of having the links directly on the components themselves.

While experimenting with different approaches for this feature, it was discovered that the

PEP enables a user to open a website by clicking on a node in a class diagram. It achieves this by putting the URL of the website in-between two square brackets and appending this to a command describing a class node like so:

$$class\ Car\ [[http://google.com]]$$

It was clear that this idea could be extended to work with the components in a state diagram.

Implementation

Much like diagramTextHelper, the extension point 'net.sourceforge.plantuml.eclipse.linkOpener", allows the SDP to implement a class (StateLinkOpener) to handle the process of opening a link. This class defines how the text written between the nested square brackets is interpreted when a component with a link is clicked. The idea is that this handler will instruct the Eclipse IDE to open a marker on the line that describes the component that was clicked. Intuitively, this means StateLinkOpener requires the positional information of a line.

Whilst it is unable to reference the models, StateLinkOpener can be passed the information by inserting it as a string between the two square brackets (See Figure 5.8). Then, through string manipulation, StateLinkOpener can create a marker and subsequently open it, to be directed to the desired line.

```
BACKWARD-TRANSITION-LINK(transition):
1: linkedTrans = transition + " : "+
        "[[#FSM#"+ stateDiagram.className +"#"+ transition.lineNum +"#"+
                    transition.charStart +"#"+ transition.charEnd + "]] ";
2: return linkedTrans;
```

```
BACKWARD-STATE-LINK(state):
1: return "state " + state.stateName + "[[#FSM#" + stateDiagram.className +"#"+
            state.lineNum +"#"+ state.charStart +"#"+ state.charEnd + "]] ";
```

Figure 5.8: Method of passing line positions to StateLinkOpener

Chapter 6

Testing

"Requirements and Analysis" briefly outlined that there would be three phases of testing: Unit, Integration and End-to-End. Whilst using JUnit to test isolated units is intuitive, configuring ways to automate the latter two is more complex. This chapter will outline the approach taken to configure the test cases for each phase using examples to illustrate the point. For a full list of the tests and their outputs refer to Appendix B.

6.1 Unit Tests

Independent methods were tested *with* all logical inputs and *for* all logical outputs. The method getNodeAndAllDescendants() found in the StateTree.java class will be used as an example. Figure 6.1 shows its definition.

```
public ArrayList<Node> getNodeAndAllDescendants(Node parent) {
    ArrayList<Node> allDescendants = new ArrayList<Node>();
    if (parent.visible) allDescendants.add(parent);
    for (Node node : nodes) {
        if (node.index > parent.index && node.visible) {
            allDescendants.add(node);
        }
    }
    return allDescendants;
}
```

Figure 6.1: The method getNodeAndAllDescendants from StateTree.java

Taking a single node as input in this context means that three different types of node must be tested; A node which doesn't exist in the tree, a leaf node, and a node with children(s).

Logical case	**Expected return**	**Pass/Fail**
Input node which doesnt exist in the tree	NULL	Pass
Input leaf node	EmpyList	Pass
Input node with one child "childA"	(childA)	Pass
Input node with two children; "childA", "childB"	(childA, childB)	Pass
Input node with one child "childA", which itself has a child "descendantA"	(childA, descendantA)	Pass
Input node with two children "childA","childB", "childA" also has a child "descendantA".	(childA, childB, descendantA)	Pass

Table 6.1: A table listing the test cases for method getNodeAndAllDescendants()

A similar approach to 6.1 was taken for the remainder of the units tests (see Appendix B).

6.2 Integration & End-to-End testing

The tests required for these phases, require a test suite to be set-up that simulates using the plug-in for different scenarios.

As mentioned in section 4.1, the Eclipse PDE allows users to run a nested instance of Eclipse (running the plug-in under development), within the current instance. This functionality is extended further by allowing JUnit tests to be run on the nested instance. This means plug-in functionality can be automated and the results tested. In the context of this plug-in, this means using the plug-in on different instances of source code (simple example of state-based software), and evaluating the resultant string that is generated and sent to the PlantUML-Eclipse plugin (recap Figure 5.1).

When setting up the test suite, certain objects representing aspects of the nested Eclipse instance must be initialized as the plug-in requires these to function. As Figure 6.3 illustrates, this can be implemented in a method prefaced with '@Before' which means that it is executed once, before any of the tests in the class are run.

```
@Before
public void createEnvironment() {


    IWorkspace workspace = ResourcesPlugin.getWorkspace();
    IWorkspaceRoot wsRoot = workspace.getRoot();
    className = "ExampleClass.java";

    IPath location = new Path("/ExampleProject/src/ExampleClass.java");
    IFile file = wsRoot.getFile(location);
    IWorkbench wb = PlatformUI.getWorkbench();
    IWorkbenchWindow window = wb.getActiveWorkbenchWindow();
    IWorkbenchPage page = window.getActivePage();
    editor = page.getActiveEditor();
    input = editor.getEditorInput();

    HashSet<String> plantMarkerKey = new HashSet<String>();

    try {
        IDocumentProvider provider = new TextFileDocumentProvider();
        provider.connect(file);
        document = provider.getDocument(file);
        FindReplaceDocumentAdapter finder = new FindReplaceDocumentAdapter(document);
        IEditorInput editorInput = editor.getEditorInput();
        root = StateTextDiagramHelper.getRoot(editorInput);
        IPath path = ((IFileEditorInput) editorInput).getFile().getFullPath();

        stateTextDiagramHelper = new StateTextDiagramHelper(stateDiagram, "", 0, plantMarkerKey);
        StateTree theTree = null;
        stateMachineGen = new StateMachineGenerator(theTree);

        System.out.println(document.get());
    } catch (CoreException e) {
        e.printStackTrace();
    }
}
```

```
ExampleClass.java

public class ExampleClass {
    //@start_OSM_generation
    state = valid_states.ExampleState;
    c.call();

    if (anEvent) {
        state = valid_states.AnotherExampleState;
        c.call();
    }

    state = valid_states.AThirdState;
    c.call();
    //@end_OSM_generation
```

Figure 6.2: The method used to set-up the test suite

The variable 'location' in the Figure above, refers to a class that contains preset scenarios (examples of state-based code), that the plug-in will be used on.

Because the plug-in works by clicking on sections of state-based code, a method is required to simulate a user doing this. The method in Figure 6.3 A can be called to simulate a

user clicking on one of the scenarios in ExampleClass.java. This method also returns a string representing the state machine generated by the plug-in. This string can therefore be compared with an expected string to test if the output from the plug-in was correct and therefore that the functionality works (See Figure 6.3 B).

```
public StringBuilder clickExample2_BothVisible() {
    int selStart = 889; //cursor position in text editor
    return stateMachineGen.getDiagramTextLines(document, selStart, input);
}
```

(a)

```
@Test
public void checkTwoStatesBothVisible() {
    initializeResult();

    result.append("[*] -> ExampleState" + "\n");
    result.append("ExampleState -down-> AnotherState :  / call();" + "\n");
    result.append("AnotherState -down-> [*] :  / call();" + "\n");
    result.append("state AnotherState[[ExampleClass.java#FSM#state#47]]" + "\n");
    result.append("state ExampleState[[ExampleClass.java#FSM#state#45]]" + "\n");
    assertEquals(result.toString(), clickExample2_BothVisible().toString());
}
```

(b)

Figure 6.3: A state machine created using GraphViz within Eclipse describing the FlightGear auto-pilot manoeuvre

The Actual Tests

Similar to the Unit tests, Integration and End-to-End testing requires testing all logical inputs/outputs. In the context of this plug-in, this means testing the core variations of the scenarios. For example, in a system consisting of two states there are the following scenarios:

1. No callbacks after each state
2. A callback following *just* the first state
3. A callback following *just* the second state
4. Callbacks following each state
5. A guard around each state
6. A guard around *just* the first state
7. A guard around *just* the second state
8. Repeat 1-4 for items 5, 6, 7

As shown, there are many variations for each scenario, which only grows larger the more states that are involved. Fortunately, larger state systems essentially consist of a combination of these core scenarios and their variations. Therefore, whilst its true that these systems may be composed of far more states and transitions, their underlying logic is the same. Overall, this means rigorously testing the core examples results in the inclusive testing of these more complex systems.

6.3 Coverage

In total, 123 tests were written with all passing successfully. This can be broken down into 57 Unit tests (see Figure 6.3A), 22 Integration Tests (see Figure 6.3B) and 44 End-to-End tests (see Figure 6.3C).

Figure 6.4: JUnit test cases

Figure 6.5 illustrates that the 123 tests provide an impressive 85.3% coverage, which shows that the plug-in functions as intended.

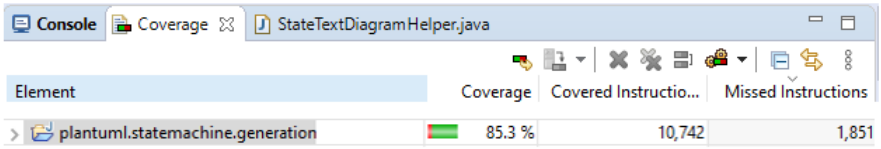


Figure 6.5: The coverage of all JUnit tests

Chapter 7

Results

The introduction of this report stated that the aim of this project was to *"create a tool that helps users develop state-based software"*. It specified that this would be accomplished by creating a plug-in for the Eclipse IDE that would extend PlantUML and give users a way to annotate their code with state machines. This chapter evaluates if the plug-in has been successful in achieving this.

7.1 Meeting requirements

Through the examination of existing tools in chapter 2, requirements were drawn up that described the functionality that the tool should have. This section explicitly lists whether or not these requirements have been met, whilst the testing in the previous chapter (and appendix B), illustrates that this functionality works as intended.

Requirement Topic 1: Automatic generation of a state-based diagram

#	**Requirement**	**Satisfied**
1	Every possible state is inferred from the source code	MOSTLY
2	Every possible transition is inferred from the source code	MOSTLY
3	Guards are inferred from source code and label relevant transitions.	TRUE
4	Actions are inferred from the source code and label relevant transitions.	TRUE
5	Where appropriate, an initial state transitions to the first state	TRUE
6	Where appropriate, states correctly transition to the exit state	TRUE
7	Self-loop transitions are inferred if an action/event doesn't require the object to leave its current state.	TRUE
8	A states visibility is inferred, and this determines whether it is shown in the diagram	TRUE

When used to generate complex state machines, the plug-in proves extremely promising. Below an example has been specifically configured in an attempt to showcase as much of the plug-ins functionality in one diagram. See the appendices for more examples.

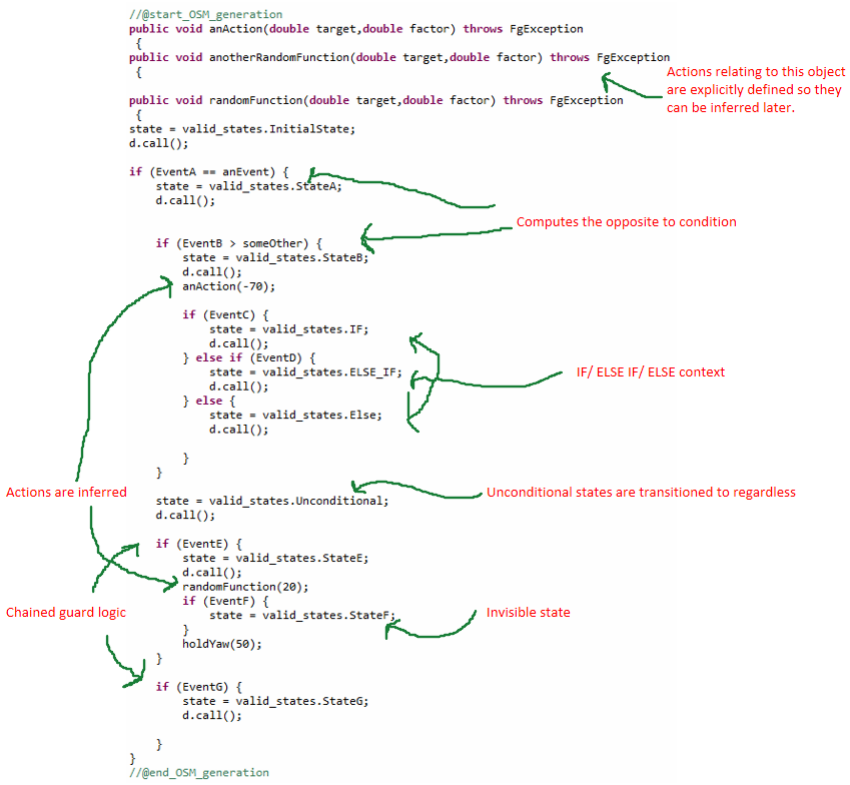


Figure 7.1: Code that describes a complex state-based system

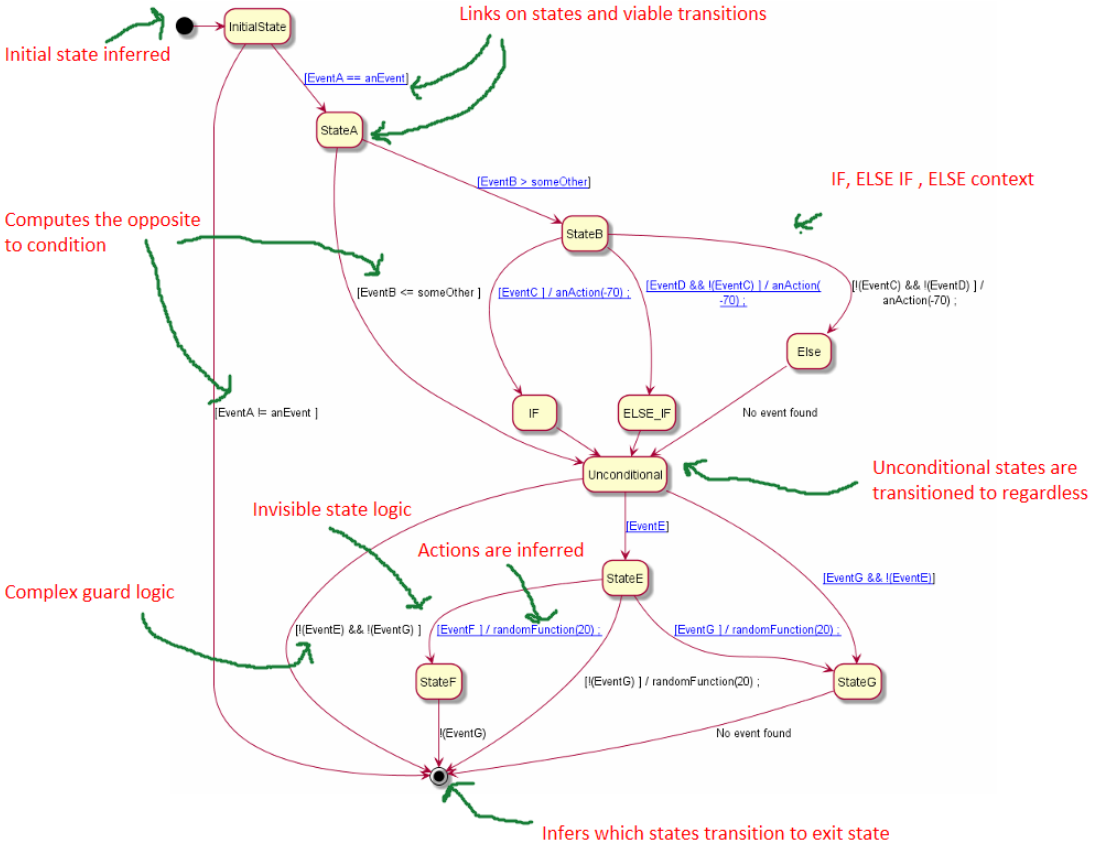


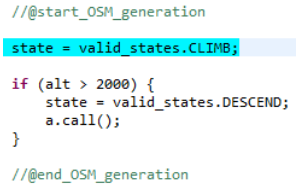
Figure 7.2: The diagram generated by the plugin

Requirement Topic 2: Linking between diagram and code

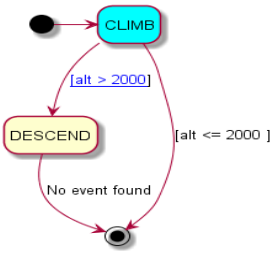
#	Requirement	Satisfied
9	Clicking on a state node links you to its definition in the source	TRUE
10	Clicking an action label on a transition links you to its definition in the source.	FALSE
11	Clicking a guard label on a transition links you to its definition in the source	TRUE
12	Clicking on a line that defines a state highlights the relevant node in the diagram	TRUE
13	Clicking on a line defining an action that causes a transition, highlights the transition.	TRUE
14	Clicking on a line defining a guard for a transition, highlights the transition it guards.	TRUE

Note: During implementation, it was decided that clicking on an action would not navigate you to its definition in the source. This is because complications arose when a transition had a guard and an action as a label.

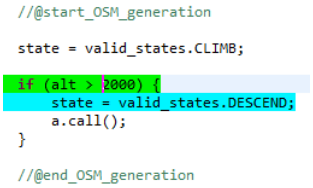
The final implementation of the plugin allows users to click on a component in a state-machine to be directed to the line of code which describes it. Figure 7.3 illustrates the results.



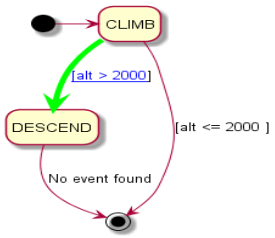
(a) Clicking the node CLIMB



(b) Clicking on a CLIMB decleration



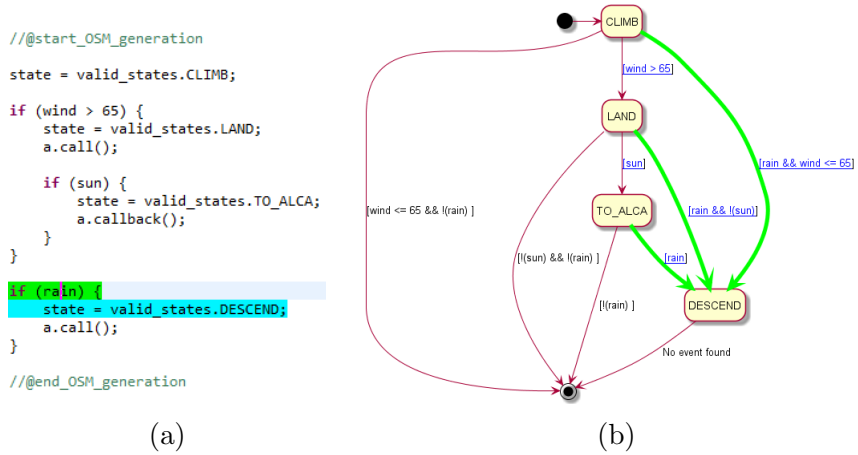
(c) Clicking on a transition



(d) Clicking on an if conditional

Figure 7.3: Association between diagram and code

If a state has many incoming transitions and the user clicks on the conditional guarding the state, all of the incoming transitions relating to that guard are highlighted. Similarly, all references to a state are highlighted when the user clicks on one of those references (See Figure 7.4).



(a) (b)

Figure 7.4: Clicking on an if-statement which can be used by many states

Although the state machine models simple stated-based behaviour, Figure 7.4(b) illustrates that implementing associations between diagram and code simplifies the process of understanding the behaviour of the system.

Requirement Topic 3: The plug-ins compatibility with PlantUML

#	Requirement	Satisfied
15	Work seamlessly alongside PlantUML	TRUE
16	Ability to explicitly write states and transitions in case they are missed.	TRUE
17	Ability to write a command that removes unwanted state and transitions	TRUE
18	Clicking on an explicitly written PlantUML statement links you to the relevant component in the diagram	TRUE
19	Where appropriate, clicking on a component in a state machine navigates you to the PlantUML command describing it	TRUE
20	Fully packaged plug-in that can be installed and used within Eclipse	TRUE

Providing the user with the functionality to use PlantUML commands within the source not only gives them more freedom when annotating their code but also equips them with a way to handle any bugs.

New PlantUML commands

If the plug-in incorrectly infers a state or transition then the user can type the following commands to remove them:

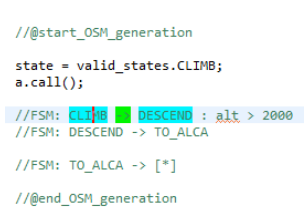
$$//FSM : REMOVE \ \ stateName$$
$$//FSM : REMOVE \ \ state- > anotherState : thelabel$$

Another command implemented allows the user to specify exit conditions/calls deliminated with a '/':

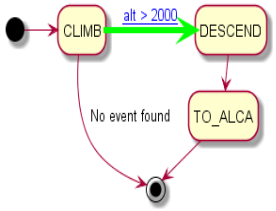
$$//FSM : EXIT - a.pause(true)/stop();$$

Links between PlantUML and diagram

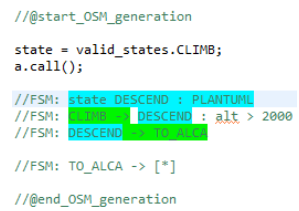
The plug-in creates links between PlantUML commands and the components they represent. Transitional links work in the same way as usual, however, clicking on a PlantUML command which defines a state highlights all references to that state:



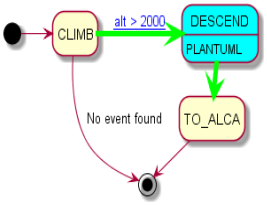
(a) Clicking on a transition



(b) Clicking on a PlantUML transition command



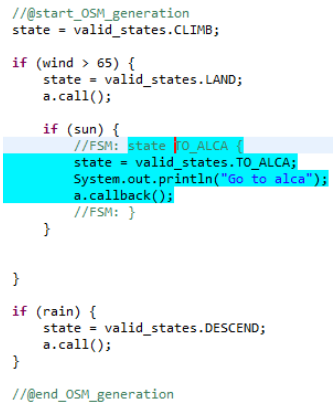
(c) Clicking on the DESCEND node



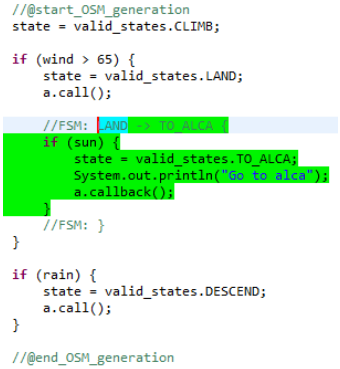
(d) Clicking on PlantUML state command

Figure 7.5: Association between diagram and code

It also provides the user with some customization in this process. By appending '{' to the end of a command and then inserting a closing '}' where appropriate, the user can specify an association between multiple lines of code and a component:



(a) State component



(b) Transition component

Figure 7.6: Linking a component to multiple lines of code

7.1.1 Optional requirements

Unfortunately, due to the complexities of implementing the above requirements, there was no time to satisfy the optional requirements. These can be duly noted under further work.

7.2 Using the plug-in on FlightGear

Figure 7.7 is the result of using the plug-in on the on the manoeuvre (incomplete) function in Examples.java (See appendix)

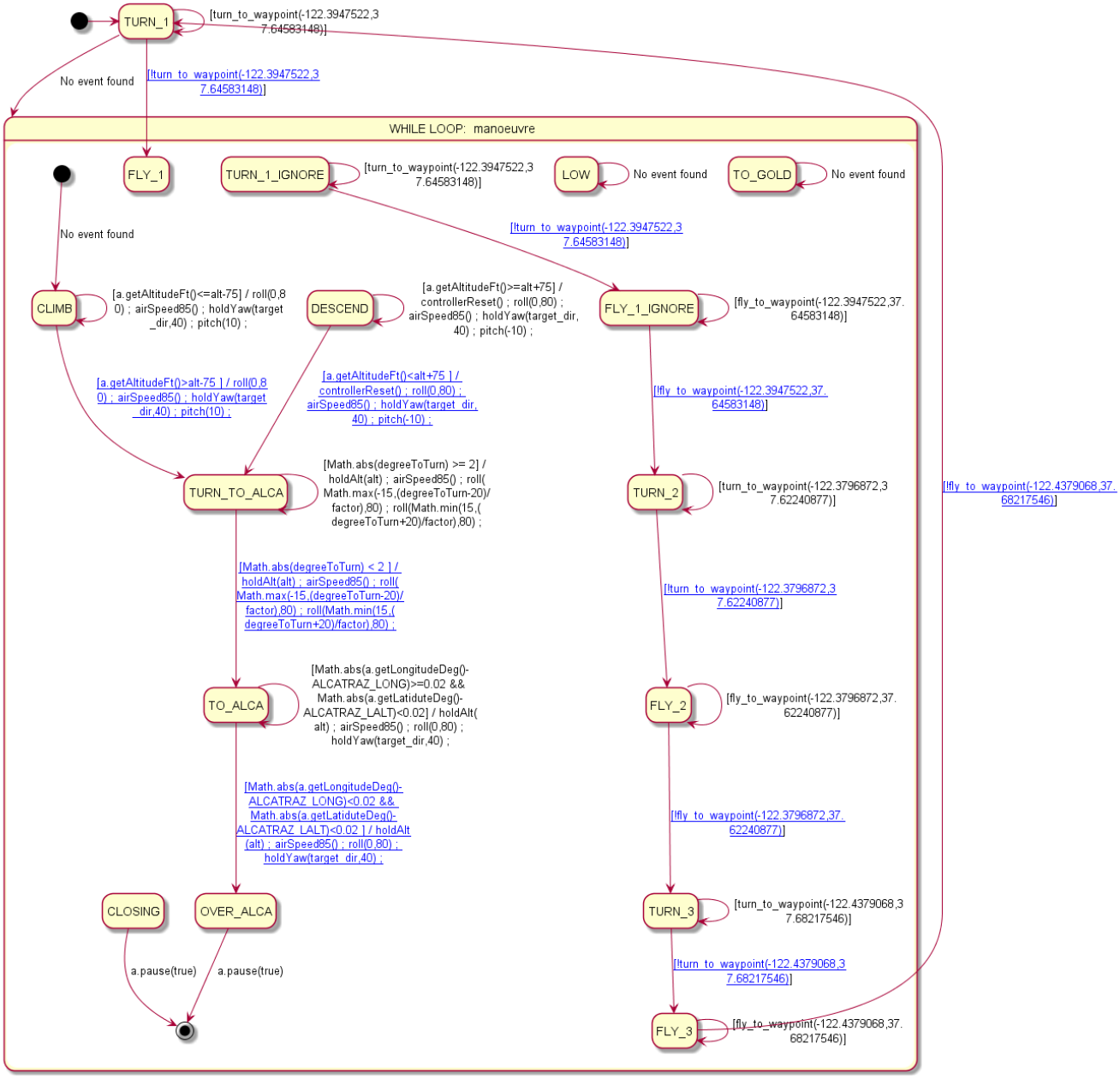


Figure 7.7: The diagram generated using the plug-in on flight-gears manoeuvre

The user is instantly informed about the flow of control between behaviours and yet also about the states that are impossible to enter and exit, and therefore in this context, those

that will infinitely loop. This aptly shows the immediate use of the plug-in. A developer faced with this information can begin working on fixing the logic behind the system. This job will be conveniently helped by the fact that they can click on any component to find its location, which in a class consisting of 851 lines, is extremely advantageous.

Evaluation

Using the plug-in on FlightGear is useful when comparing the plug-in to the tools examined in chapter 2. This is because many of the illustrations, such as the diagrams drawn by the case studies, have referenced snippets of the code. This code (which is incomplete) was used in an assignment that asked students to complete/fix the auto-pilot and then document this with a corresponding state-transition diagram. This makes the program an ideal candidate to evaluate the plug-in with as it answers two important questions:

1. Does the plug-in overcome the weaknesses of the case studies discussed in chapter 2
2. Does the plug-in help with the development of state-based software?

In order to answer these questions, an investigation similar to the experiments carried out by Erk Arisholm [7] (see Section 2.2) was to be carried out. Two sample groups of programmers were to be given the FlightGear assignment (See 2.0.1, or Appendix A for the source), and then asked to complete the auto-pilot under timed conditions, with one group having access to the plug-in. The quality and correctness (does the systems behaviour work as expected?) of their submissions would then be indicative of whether the questions above could be answered positively.

Unfortunately, due to the constraint of time, this investigation was not undertaken. Despite this, its use as an evaluation criteria still stands, and therefore it too has been noted under further work.

For this reason, an alternative solution was required to evaluate the tool. In an effort to maintain consistency, yet also to provide a fair comparison, the plug-in will now be evaluated with the criteria used to assess the case studies in section 2.4.

1. **Ease of use** (3/3)b
   No difficulty. Creating a diagram simply requires the code to be inserted between @start_OSM_generation and @end_OSM_generation.
2. **Quality of the diagrams rendered** (2/3)
   The overall quality of the diagram has not been enhanced, therefore this mark remains the same.
3. **Consistency of the diagrams rendered** (2/3)
   Diagrams are consistent because the parser examines the text line by line, therefore the commands are issued in the same order each time
4. **Diagram encapsulation** (1/3)
   The plug-in is only implemented to work for state-machines, however because it extends PlantUML, it has access to all of the diagrams that it supports (UML 2.0)
5. **Can it render diagrams automatically?** (2/3)
   Yes and no. It can generate diagrams for various contexts seen in state-based systems, yet, because it does not work in all contexts i.e try/catch/finally, it cannot score 100% here.

6. **Has the tool been integrated into various development environments?** (1/3)
   This extension of PlantUML only works in Eclipse.

7. **Difficulty to maintain** (3/3)
   The fact that diagram are automatically generated means that changes in the code are dynamically reflected in the diagram and therefore no maintenance is required.

8. **Difficulty of association** (3/3)
   Links exist in both directions and therefore associating components to code and vice versa is intuitive.

7.3 Conclusion

The absolute motivation for the project was to develop a plug-in that helped user to develop state-based software. Through the evaluation of various tools in chapter 2, numerous requirements were formalised that aimed to help guide its development and achieve this goal. The results in this chapter illustrate that the plug-in was successful in satisfying these.

Whilst carrying out an investigation like Erik Arisholm's would have provided strong evidence that the plug-in had reached its fundamental aim of easing the process of developing state-based software, the evaluation above is sufficient at determining its use over the existing tools. Although the overall score obtained is equal to the mark awarded to PlantUML in chapter 2, disheartening conclusions should not be drawn from this. This is because the criteria above were specifically chosen to assess the extent to which a tool can be used to annotate software of all types. The fact that the plug-in specializes in state machines, results in it losing marks when subjected to areas of brevity. This inference is reflected in the next section where future work is discussed.

More critically then, the conclusion that can be garnered from the evaluation is that, when exposed to state-machines, the plug-in capably overcomes many of the weaknesses exerted by industrial standards in the same field. Ultimately, then, the tool is successful in helping developers write state-based software.

7.4 Future work

The positive conclusion above suggests that future work on the plug-in makes sense. Naturally, this would begin by extending its functionality to generate diagrams for all types of state-based software. Alternatively to this, the results regarding association between diagram and code have proven to be practical when developing code and as shown with state-machines, of great use. It therefore stands to reason that the development of extending this to work with other diagrams, such as in a sequence diagram, should provide equivalently advantageous.

Chapter 8

Appendices

Appendix A

The Manoeuvre function in Flight-gear (4 parts)

```
synchronized public void manoeuvre() throws FgException
 {
   //variables
   boolean manoeuvre=true; // while true, we keep receiving ticks and responding;

   createAndShowGUI();
   controllerReset();


   state = valid_states.TURN_1;//a.load("turned_to_ALCA.sav");
   frame.setTitle(state.toString());

   double target_dir=0, degreeToTurn=0;

   //the control loop
   while(manoeuvre)
   {
       //check position every few hundred milliseconds
       sleep(300);

       if (eightsecTimeout > 0)
       {
           eightsecTimeout-=300;
           if (eightsecTimeout <= 0)
           {
               btnState = buttonStates.RELEASED;
               frame.setTitle(state.toString()+" "+btnState.toString());
           }
       }

       switch(state)
       {
       case INIT:  //initial state (reset all variables)
           a.setThrottle(1);
           a.setFlaps(0);
           target_dir=a.getHeadingDeg();
           //change to the "first real" control mode
           state=valid_states.CLIMB;
           System.out.println("climbing to cruise altitude");
           frame.setTitle(state.toString());
           break;

       case CLIMB:  //climb up to alt
           roll(0,80);
           airSpeed85();
           holdYaw(target_dir,40);
           pitch(10);

           if(a.getAltitudeFt()>alt-75)
           {
               state=valid_states.TURN_TO_ALCA; //set new state
               System.out.println("reached the cruising altitude, turning to ALCA");
               controllerReset();
               frame.setTitle(state.toString());
           }
           break;
```

(Part 1)

```
case TURN_TO_ALCA:  //turn to fly to Alcatraz
    holdAlt(alt);
    airSpeed85();
    double currentHdg = a.getHeadingDeg();
    target_dir = calc_dir(a.getLongitudeDeg(),a.getLatiduteDeg(),
              ALCATRAZ_LONG,ALCATRAZ_LALT);

    degreeToTurn = (target_dir-currentHdg);
    if (degreeToTurn < -180) degreeToTurn +=360;
    else
        if (degreeToTurn > 180) degreeToTurn-=360;
    double tolerance = 4;

    double factor=3;

    if(degreeToTurn<-tolerance)
    {

        roll(Math.max(-15,(degreeToTurn-20)/factor),80);
    }
    else if (degreeToTurn>tolerance)
    {

        roll(Math.min(15,(degreeToTurn+20)/factor),80);
    }

    if (Math.abs(degreeToTurn) < 2)
    {
        state = valid_states.TO_ALCA;  //set new control state
        System.out.println("turn complete, flying to ALCA");
        controllerReset(); //reset yaw integrator/differentiator values
        frame.setTitle(state.toString());
    }
    break;

case TO_ALCA:  // fly to Alcatraz
    holdAlt(alt);
    airSpeed85();
    roll(0,80);
    target_dir = calc_dir(a.getLongitudeDeg(),
           a.getLatiduteDeg(),ALCATRAZ_LONG,ALCATRAZ_LALT);
    holdYaw(target_dir,40);

    if(Math.abs(a.getLongitudeDeg()-ALCATRAZ_LONG)<0.02 &&
           Math.abs(a.getLatiduteDeg()-ALCATRAZ_LALT)<0.02)
    {
        state=valid_states.OVER_ALCA;  //set new state
        frame.setTitle(state.toString());
    }
    break;
```

(Part 2)

```
case OVER_ALCA:  //manoeuvre over Alcatraz
    System.out.println("ALCA reached");
    a.pause(true); //pause the flight simulator
    manoeuvre=false; //leave the control loop
    frame.setTitle("at Alca");
    System.out.println("END");
    break;
case TO_GOLD: //sink and fly in direction of Golden Gate Bridge
    break;
case LOW:  //low-level flight
    //missing at the moment
    break;
case DESCEND:  // descend to alt
    controllerReset();
    roll(0,80);
    airSpeed85();
    holdYaw(target_dir,40);
    pitch(-10);

    if(a.getAltitudeFt()<alt+75)
    {
        state=valid_states.TURN_TO_ALCA; //set new state
        System.out.println("reached the cruising altitude, ALCA");
        controllerReset();
    }
    break;

case TURN_1:
    if (!turn_to_waypoint(-122.3947522,37.64583148))
    {
        state = valid_states.FLY_1;
        frame.setTitle(state.toString()+" "+btnState.toString());
    }
    break;
case TURN_1_IGNORE:
    if (!turn_to_waypoint(-122.3947522,37.64583148))
    {
        state = valid_states.FLY_1_IGNORE;
        frame.setTitle(state.toString()+" "+btnState.toString());
    }
    break;
case FLY_1:
case FLY_1_IGNORE:
    if (!fly_to_waypoint(-122.3947522,37.64583148))
    {
        state = valid_states.TURN_2;
        frame.setTitle(state.toString()+" "+btnState.toString());
    }
    break;
```

(Part 3)

```
case TURN_2:
    if (!turn_to_waypoint(-122.3796872,37.62240877))
    {
        state = valid_states.FLY_2;
        frame.setTitle(state.toString()+" "+btnState.toString());
    }
    break;
case FLY_2:
    if (!fly_to_waypoint(-122.3796872,37.62240877))
    {
        state = valid_states.TURN_3;
        frame.setTitle(state.toString()+" "+btnState.toString());
    }
    break;
case TURN_3:
    if (!turn_to_waypoint(-122.4379068,37.68217546))
    {
        state = valid_states.FLY_3;
        frame.setTitle(state.toString()+" "+btnState.toString());
    }
    break;
case FLY_3:
    if (!fly_to_waypoint(-122.4379068,37.68217546))
    {
        state = valid_states.TURN_1;
        frame.setTitle(state.toString()+" "+btnState.toString());
    }
    break;

case CLOSING:
    a.pause(true); //pause the flight simulator
    manoeuvre=false; //leave the control loop
    System.out.println("Application closed on user request");
    break;
default:
    assert false;
    break;
}
}
}
}

//@end_OSM_generation
```

(Part 4)

Appendix B

Unit Testing

Event.java - 1/1 passed

```
@Test
public void checkSetLineEnd() {
    Event event = new Event("Event", "eventLine", 0, 0, 0);
    event.setLineEnd(50);
    assertEquals(50, event.multiLineEnd);

}
```

Node.java - 5/5 passed

```
@Test
public void checkSetNodeIndex() {
    Node node = new Node("ExampleNode", "exampleNode", true, 0, 0, 0, new Event("Event"));
    node.setIndex(5);
    assertEquals(5, node.index);
}

@Test
public void checkSetNodeVisible() {
    Node invisibleNode = new Node("ExampleNode", "exampleNode", false, 0, 0, 0, new Event("Event"));
    invisibleNode.setVisible();
    assertTrue(invisibleNode.visible);
}

@Test
public void checkSetParent() {
    Node node = new Node("ExampleNode", "exampleNode", true, 0, 0, 0, new Event("Event"));
    Node parent = new Node("parentNode", "parentNode", true, 0, 0, 0, new Event("Event"));
    node.setParent(parent);
    assertEquals(parent, node.parent);
}

@Test
public void checkEqualsReturnsTrueIfSameNode() {
    Node node = new Node("ExampleNode", "exampleNode", true, 0, 0, 0, new Event("Event"));
    assertTrue(node.equals(node));
}

@Test
public void checkEqualsReturnsFalseIfDifferentNode() {
    Node node = new Node("ExampleNode", "exampleNode", true, 0, 0, 0, new Event("Event"));
    Node parent = new Node("parentNode", "parentNode", true, 0, 0, 0, new Event("Event"));
    assertFalse(node.equals(parent));
}
```

PatternIdentifier.java - 3/3 passed

```
@Test
public void checkAddAppendsNewPatternToPatternStore() {
    Pattern samplePattern = Pattern.compile("Apattern");
    PatternIdentifier patternIdentifier = new PatternIdentifier();
    patternIdentifier.add(samplePattern, 15);
    boolean bool = false;
    for (RegexInfo regex : patternIdentifier.patternStore) {
        if (regex.pattern.equals(samplePattern)) bool = true;
    }
    assertTrue(bool);
}

@Test
public void checkAddDoesntAddIfIdentifierInUse() {
    Pattern samplePattern = Pattern.compile("Apattern");
    PatternIdentifier patternIdentifier = new PatternIdentifier();
    patternIdentifier.add(samplePattern, 14);

    assertEquals(15, patternIdentifier.patternStore.size());
}

@Test
public void checkConstructorInitializesAllPatterns() {
    PatternIdentifier patternIdentifier = new PatternIdentifier();
    assertEquals(15, patternIdentifier.patternStore.size());

}
```

StateTextDiagramHelper.java - 11/11 passed

```
////////////////////////////////////STATES/////////////////////////////////////
@Test
public void checkColorState() {

    String expected = "state ExampleState #Cyan";
    String stateName = "ExampleState";
    assertEquals(expected, stateTextDiagramHelper.forwardStateLink(stateName));
}



//////////////////////////////////TRANSITIONS/////////////////////////////////////


@Test
public void checkColorTransition() {

    String transition = "ExampleState -> AnotherExampleState";
    String expected = "ExampleState -[thickness=5,#Lime]> AnotherExampleState";
    assertEquals(expected, stateTextDiagramHelper.forwardTransitionLink(transition));
}
```

```
////////////////////////////NEGATION METHODS/////////////////////////////////////////////////

@Test
public void checkNegateEquality() {
    String stringToNegate = "a == b";
    String expected = "a != b";
    assertEquals(expected, stateTextDiagramHelper.negateCondition(stringToNegate));
}

@Test
public void checkNegateInequality() {
    String stringToNegate = "a != b";
    String expected = "a == b";
    assertEquals(expected, stateTextDiagramHelper.negateCondition(stringToNegate));
}

@Test
public void checkNegateGreaterEqualsTo() {
    String stringToNegate = "a >= b";
    String expected = "a < b";
    assertEquals(expected, stateTextDiagramHelper.negateCondition(stringToNegate));
}

@Test
public void checkNegateLessEqualsTo() {
    String stringToNegate = "a <= b";
    String expected = "a > b";
    assertEquals(expected, stateTextDiagramHelper.negateCondition(stringToNegate));
}

@Test
public void checkNegateGreaterThan() {
    String stringToNegate = "a > b";
    String expected = "a <= b";
    assertEquals(expected, stateTextDiagramHelper.negateCondition(stringToNegate));
}

@Test
public void checkNegateLessThan() {
    String stringToNegate = "a < b";
    String expected = "a >= b";
    assertEquals(expected, stateTextDiagramHelper.negateCondition(stringToNegate));
}


@Test
public void checkStateDescriptorRemovesFSMLower() {
    String fsmLine = "//fsm: State1 -> State2";
    String expected = "State1 -> State2";

    assertEquals(expected, StateTextDiagramHelper.stateDescriptor(fsmLine));

}
```

```
@Test
public void checkStateDescriptorRemovesFSMUpper() {
    String fsmLine = "//FSM: State1 -> State2";
    String expected = "State1 -> State2";

    assertEquals(expected, StateTextDiagramHelper.stateDescriptor(fsmLine));

}

@Test
public void checkStateDescriptorReturnsNullIfNotFSM() {
    String fsmLine = "State1 -> State2";

    assertNull(StateTextDiagramHelper.stateDescriptor(fsmLine));
}
```

StateTree.java - 36/36 passed

Helper methods

```
////////////////////////////////////CREATE NODE OBJECTS/////////////////////////////////

public Node createExampleNode() {
    String stateName = "ExampleState";
    String editorLine = "state = valid_states.ExampleState";
    return new Node(stateName, editorLine, true, 0, 0, 0, new Event("an event"));
}

public Node createExampleNode2() {
    String stateName = "ExampleState_2";
    String editorLine = "state = valid_states.ExampleState_2";
    return new Node(stateName, editorLine, true, 0, 0, 0, new Event("an event_2"));
}

public Node createExampleNode3() {
    String stateName = "ExampleState_3";
    String editorLine = "state = valid_states.ExampleState_3";
    return new Node(stateName, editorLine, true, 0, 0, 0, new Event("an event_3"));
}

public Node createInvisibleNode() {
    String stateName = "InvisibleNode";
    String editorLine = "state = valid_states.InvisibleNode";
    return new Node(stateName, editorLine, false, 0, 0, 0, new Event("invisible event"));
}

public Node createUnconditionalNode() {
    String stateName = "unConditional";
    String editorLine = "state = valid_states.unconditional";
    return new Node(stateName, editorLine, true, 0, 0, 0, new Event("unconditional"));
}

public void initializeTree() {
    String stateName = "rootNode";
    String editorLine = "state = valid_states.EditorLine";
    Node root = new Node(stateName, editorLine, true, 0, 0, 0, new Event("an event"));
    stateTree = new StateTree(root);
}
```

Tests

```
////////////////////////////////////TREE CREATION/////////////////////////////////

@Test
public void checkTreeCreated() {
    Node theNode = createExampleNode();
    StateTree theTree = new StateTree(theNode);
    assertEquals(theNode, theTree.nodes.get(0));
}

@Test
public void checkNodeSuppliedBecomesRoot() {
    Node theNode = createExampleNode();

    StateTree theTree = new StateTree(theNode);
    assertEquals(theNode, theTree.root);
}
////////////////////////////////////GET NODE/////////////////////////////////////////

@Test
public void checkgetNodeReturnsNode() {
    Node theNode = createExampleNode();

    StateTree theTree = new StateTree(theNode);
    assertEquals(theNode, theTree.getNode("ExampleState"));
}

@Test
public void checkGetNodeReturnsNullIfInvalidNodeSupplied() {
    Node theNode = createExampleNode();

    StateTree theTree = new StateTree(theNode);
    assertNull(theTree.getNode("FalseNode"));

}
```

```
,
////////////////////////////////////ADD NODE/////////////////////////////////////////

@Test
public void checkAddingNodeToRoot() {
    initializeTree();
    Node theNode = createExampleNode();
    stateTree.addNode(stateTree.root, theNode);
    assertTrue(stateTree.nodes.contains(theNode));
}

@Test
public void checkAddingNodeToAnotherNode() {
    initializeTree();
    Node aNode = createExampleNode();
    Node aNode_2 = createExampleNode2();

    stateTree.addNode(stateTree.root, aNode);
    stateTree.addNode(aNode, aNode_2);

    assertTrue(stateTree.links.get(aNode).contains(aNode_2));

}
////////////////////////////////////REMOVE LAST NODE/////////////////////////////////

@Test
public void checkRemoveLastNodeRemovesLastNode() {
    initializeTree();
    Node aNode = createExampleNode();
    stateTree.addNode(stateTree.root, aNode);
    stateTree.removeLastNode();
    assertFalse(stateTree.nodes.contains(aNode));

}

@Test
public void checkRemoveLastNodeDoesNothingIfJustRoot() {
    initializeTree();
    stateTree.removeLastNode();
    assertNotNull(stateTree.root);
}



////////////////////////////////////GET CHILDREN/////////////////////////////////////

public @Test void checkGetChildrenWithNoChildReturnsNoChild() {
    initializeTree();
    assertTrue(stateTree.getChildren(stateTree.root).size() == 0);
}

public @Test void checkGetChildrenWithInValidNodeReturnsEmpty() {
    initializeTree();
    assertTrue(stateTree.getChildren(createExampleNode()).size() == 0);
}

@Test
public void checkGetChildrenWithOneChildReturnsOneChild() {
    initializeTree();
    Node aNode = createExampleNode();
    stateTree.addNode(stateTree.root, aNode);
    assertTrue(stateTree.getChildren(stateTree.root).size() == 1);

}

@Test
public void checkGetChildrenWithOneChildReturnsCorrectChild() {
    initializeTree();
    Node aNode = createExampleNode();
    stateTree.addNode(stateTree.root, aNode);
    assertTrue(stateTree.getChildren(stateTree.root).contains(aNode));
}

@Test
public void checkGetChildrenWithMoreThanOneChildReturnsCorrectChildren() {
    initializeTree();
    Node aNode = createExampleNode();
    Node aNode_2 = createExampleNode2();
    stateTree.addNode(stateTree.root, aNode);
    stateTree.addNode(stateTree.root, aNode_2);
    ArrayList<Node> sampleChild = new ArrayList<Node>();
    sampleChild.add(aNode);
    sampleChild.add(aNode_2);
    assertEquals(sampleChild, stateTree.getChildren(stateTree.root));

}
```

```
////////////////////////////////////FIND LAST UNCONDITIONAL/////////////////////////////////

@Test
public void checkFindLastUnconditionalReturnsLastUnconditional() {
    initializeTree();
    Node unconditionalNode = createUnconditionalNode();
    stateTree.addNode(stateTree.root, unconditionalNode);
    assertEquals(unconditionalNode, stateTree.findLastUnconditionalState());
}

@Test
public void checkFindLastUnconditionalReturnsRootIfNoOtherInTree() {
    initializeTree();
    Node aNode = createExampleNode();
    stateTree.addNode(stateTree.root, aNode);
    assertEquals(stateTree.root, stateTree.findLastUnconditionalState());
}

@Test
public void checkFindLastUnconditionalReturnsLastUncondWith2Unconditionals() {
    initializeTree();
    Node unconditionalNode = createUnconditionalNode();
    stateTree.addNode(stateTree.root, unconditionalNode);
    Node aSecondUnconditional = createUnconditionalNode();
    stateTree.addNode(unconditionalNode, aSecondUnconditional);

    assertEquals(aSecondUnconditional, stateTree.findLastUnconditionalState());
}

@Test
public void checkFindLastUnconditionalReturnsLastUnconditionalIfConditionalPreceedesIt() {
    initializeTree();
    Node unconditionalNode = createUnconditionalNode();
    Node conditionalNode = createExampleNode();
    stateTree.addNode(stateTree.root, unconditionalNode);
    stateTree.addNode(stateTree.root, conditionalNode);

    assertEquals(unconditionalNode, stateTree.findLastUnconditionalState());
}
```

```
////////////////////////////////////GET NODE AND ALL DESCENDANTS/////////////////////////////////

@Test
public void checkGetNodeAndAllDescendantsReturnsNullIfNodeDoesntExistInTree() {
    initializeTree();
    ArrayList<Node> expected = new ArrayList<Node>(
            Arrays.asList(stateTree.root)

            );
    assertEquals(expected, stateTree.getNodeAndAllDescendants(stateTree.root));
}

@Test
public void checkgetNodeAndAllDescendantsReturnsNoneIfNone() {
    initializeTree();
    ArrayList<Node> expected = new ArrayList<Node>(
            Arrays.asList(stateTree.root)

            );
    assertEquals(expected, stateTree.getNodeAndAllDescendants(stateTree.root));
}

@Test
public void checkGetNodeAndAllDescendantsReturnsOneDescendantsIfOne() {
    initializeTree();
    Node aNode = createExampleNode();
    stateTree.addNode(stateTree.root, aNode);
    ArrayList<Node> expected = new ArrayList<Node>(
            Arrays.asList(stateTree.root,
                    aNode)

            );
    assertEquals(expected, stateTree.getNodeAndAllDescendants(stateTree.root));

}

@Test
public void checkGetNodeAndAllDescendantsReturnsTwoDescendantsIfTwo() {
    initializeTree();
    Node aNode = createExampleNode();
    Node aNode_2 = createExampleNode2();
    stateTree.addNode(stateTree.root, aNode);
    stateTree.addNode(stateTree.root, aNode_2);
    ArrayList<Node> expected = new ArrayList<Node>(
            Arrays.asList(stateTree.root,
                    aNode,
                    aNode_2)

            );
    assertEquals(expected, stateTree.getNodeAndAllDescendants(stateTree.root));

}
```

```
@Test
public void checkGetNodeAndAllDescendandsReturnsNestedDescendants() {
    initializeTree();
    Node aNode = createExampleNode();
    Node aNode_child = createExampleNode2();
    stateTree.addNode(stateTree.root, aNode);
    stateTree.addNode(aNode, aNode_child);
    ArrayList<Node> expected = new ArrayList<Node>(
            Arrays.asList(stateTree.root,
                          aNode,
                          aNode_child)

            );
    assertEquals(expected, stateTree.getNodeAndAllDescendants(stateTree.root));
}

@Test
public void checkGetNodeAndAllDescendantsReturnsNestedAndUnestedDescendants() {
    initializeTree();
    Node aNode = createExampleNode();
    Node aNode_2 = createExampleNode2();
    Node aNode_2_Child = createExampleNode3();
    stateTree.addNode(stateTree.root, aNode);
    stateTree.addNode(stateTree.root, aNode_2);
    stateTree.addNode(aNode_2, aNode_2_Child);
    ArrayList<Node> expected = new ArrayList<Node>(
            Arrays.asList(stateTree.root,
                          aNode,
                          aNode_2,
                          aNode_2_Child)

            );
    assertEquals(expected, stateTree.getNodeAndAllDescendants(stateTree.root));
}
```

```
////////////////////////////////////CHECK FOR UNCONDITIONAL/////////////////////////////////

@Test
public void checkForUnconditionalReturnsFalseIfTargetHasNoChildren() {
    initializeTree();
    Node unconditionalNode = createUnconditionalNode();
    stateTree.addNode(stateTree.root, unconditionalNode);
    assertFalse(stateTree.checkForUnconditional(stateTree.root, unconditionalNode, unconditionalNode));

}

@Test
public void checkForUnconditionalReturnsFalseIfAllChildrenConditional() {
    initializeTree();
    Node aNode = createExampleNode();
    Node aNode_2 = createExampleNode2();
    stateTree.addNode(stateTree.root, aNode);
    stateTree.addNode(aNode, aNode_2);
    assertFalse(stateTree.checkForUnconditional(stateTree.root, aNode_2, aNode));

}

@Test
public void checkForUnconditionalReturnsFalseIfAllChildrenInvisible() {
    initializeTree();
    Node aNode = createExampleNode();
    Node aNode_2 = createInvisibleNode();

    stateTree.addNode(stateTree.root, aNode);
    stateTree.addNode(aNode, aNode_2);
    assertFalse(stateTree.checkForUnconditional(stateTree.root, aNode_2, aNode));

}

@Test
public void checkForUnconditionalReturnsTrueIfUnconditionalIndexGreaterThanToButLessThanFrom() {
    initializeTree();
    Node aNode = createExampleNode();
    Node aNode_2 = createExampleNode2();
    Node uncondtionalNode = createUnconditionalNode();
    stateTree.addNode(stateTree.root, aNode);
    stateTree.addNode(aNode, uncondtionalNode);
    stateTree.addNode(aNode, aNode_2);

    assertTrue(stateTree.checkForUnconditional(stateTree.root, aNode_2, aNode));

}
```

```
@Test
public void checkForUnconditonalReturnsTrueIfUnconditionalIndexGreaterThanToAndToInvisible() {
    initializeTree();
    Node aNode = createExampleNode();
    Node invisibleNode = createInvisibleNode();
    Node uncondtionalNode = createUnconditionalNode();
    stateTree.addNode(stateTree.root, aNode);
    stateTree.addNode(aNode, invisibleNode);

    stateTree.addNode(aNode, uncondtionalNode);

    assertTrue(stateTree.checkForUnconditional(stateTree.root, invisibleNode, aNode));
}

@Test
public void checkForUnconditonalReturnsFalseIfUnconditionalIndexGreaterThanToAndToVisible() {
    initializeTree();
    Node aNode = createExampleNode();
    Node aNode_2 = createExampleNode2();
    Node uncondtionalNode = createUnconditionalNode();
    stateTree.addNode(stateTree.root, aNode);
    stateTree.addNode(aNode, aNode_2);

    stateTree.addNode(aNode, uncondtionalNode);

    assertFalse(stateTree.checkForUnconditional(stateTree.root, aNode_2, aNode));
}

////////////////////////////////////ROOT TO DESTINATION/////////////////////////////////

@Test
public void checkRootToDestinationReturnsRouteIfNoVisibleStateFound() {
    initializeTree();
    Node from = createExampleNode();
    Node invisibleNode = createInvisibleNode();
    Node invisibleNode_2 = createInvisibleNode();
    Node to = createExampleNode();
    stateTree.addNode(stateTree.root, from);
    stateTree.addNode(stateTree.root, invisibleNode);
    stateTree.addNode(invisibleNode, invisibleNode_2);
    stateTree.addNode(invisibleNode_2, to);

    ArrayList<Node> expected = new ArrayList<Node>(
            Arrays.asList(invisibleNode_2,
                          invisibleNode,
                          stateTree.root)

            );
    assertEquals(expected, stateTree.rootToDestination(from, to, to));
}
```

```
@Test
public void checkRootToDestinationReturnsNullIfVisibleStateFound() {
    initializeTree();
    Node from = createExampleNode();
    Node aNode =  createExampleNode();
    Node invisibleNode_2 = createInvisibleNode();
    Node to = createExampleNode();
    stateTree.addNode(stateTree.root, from);
    stateTree.addNode(stateTree.root, aNode);
    stateTree.addNode(aNode, invisibleNode_2);
    stateTree.addNode(invisibleNode_2, to);


    assertNull(stateTree.rootToDestination(from, to, to));
}

@Test
public void checkRootToDestinationReturnsNullIfUnconditionalFound() {
    initializeTree();
    Node from = createExampleNode();
    Node aNode =  createExampleNode();
    Node invisibleNode_2 = createInvisibleNode();
    Node to = createExampleNode();
    stateTree.addNode(stateTree.root, from);
    stateTree.addNode(stateTree.root, aNode);
    stateTree.addNode(aNode, invisibleNode_2);
    stateTree.addNode(invisibleNode_2, to);


    assertNull(stateTree.rootToDestination(from, to, to));
}
```

```
////////////////////////////////////GET ROUTE/////////////////////////////////////
@Test
public void checkGetRouteReturnsNullIfToIndexLessThanFromIndex() {
    initializeTree();
    Node from = createExampleNode();
    Node to = createExampleNode();
    stateTree.addNode(stateTree.root, to);
    stateTree.addNode(to, from);
    assertNull(stateTree.getRoute(from, to));

}

@Test
public void checkGetRouteReturnsRouteFromParentAndChild() {
    initializeTree();
    Node from = createExampleNode();
    Node to = createExampleNode();
    stateTree.addNode(stateTree.root, from);
    stateTree.addNode(from, to);
    ArrayList<Node> expected = new ArrayList<Node>(
            Arrays.asList(to)
            );
    assertEquals(expected, stateTree.getRoute(from, to).route);
}

@Test
public void checkGetRouteReturnsRouteFromParentToDescendant() {
    initializeTree();
    Node from = createExampleNode();
    Node inbetweenInvisible = createInvisibleNode();
    Node to = createExampleNode();
    stateTree.addNode(stateTree.root, from);
    stateTree.addNode(from, inbetweenInvisible);
    stateTree.addNode(inbetweenInvisible, to);
    ArrayList<Node> expected = new ArrayList<Node>(
            Arrays.asList(inbetweenInvisible,
                          to)
            );
    assertEquals(expected, stateTree.getRoute(from, to).route);
}

@Test
public void checkGetRouteReturnsRouteBetweenTwoChildren() {
    initializeTree();
    Node from = createExampleNode();
    Node to = createExampleNode();
    stateTree.addNode(stateTree.root, from);
    stateTree.addNode(stateTree.root, to);
    ArrayList<Node> expected = new ArrayList<Node>(
            Arrays.asList(to)
            );
    assertEquals(expected, stateTree.getRoute(from, to).route);
}
```

Integration & End-to-End Testing

As stated in the testing chapter, all integration and end-to-end tests passed. In order to save time and space, instead of listing all of the test cases with their code and their output here, a class listing all of the different scenarios that cover the functionally of the plug-in (i.e what state-based software it can generate diagrams for) has been attached to the project code. This is located in [*src/test/end_to_end/plantuml.statemachine.generation*], in a file called ContextTestClass.java. With this class, two things can be achieved:

1. If you have the plug-in installed, then simply clicking on each scenario will show you their output and you can test the functionality this way.
2. If you want to run the JUnit tests located in IntegrationTestSuite and End2EndTestSuite, then within a nested Eclipse instance, you must create a class called ExampleClass.java in a project called ExampleProject (see Figure 6.2). You can then run both of these test suites by running them as a JUnit plug-in test.

Bibliography

[1] Donald Knuth. "Literate Programming". In: *The Computer Journal* 27.2 (1984), pp. 97–111. DOI: https://doi.org/10.1093/comjnl/27.2.97.

[2] Jef Raskin. "Comments Are More Important Than Code". In: *Acmqueue* 3.2 (2005), pp. 64–65. DOI: 10.1145/1053331.1053354.

[3] FlightGear Flight Simulator [online] Available at: https://www.flightgear.org/ [Accessed 24 April. 2020].

[4] *Docs.oracle.com*. Accessed on 7 Dec. 2019. URL: https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html.

[5] Erik Arisholm. *The Impact of UML Documentation on Software Maintenance: An Experimental Evaluation*. Carleton University, Department of Systems and Computer Engineering, 2005.

[6] Krishi Shiksha. *Significance of diagrams and graphs*. 22nd March 2012. Accessed on 7 Dec. 2019. URL: http://ecoursesonline.iasri.res.in/mod/page/view.php?id=34200.

[7] Lucidchart.com. (2019). [online] Available at: https://www.lucidchart.com/pages/ [Accessed 7 Dec. 2019].

[8] Draw.io. (2019). Flowchart Maker Online Diagram Software. [online] Available at: https://www.draw.io/ [Accessed 7 Dec. 2019].

[9] Gliffy. (2019). Diagram Software Team Collaboration Tools — Gliffy. [online] Available at: https://www.gliffy.com/ [Accessed 6 Dec. 2019].

[10] *Lucidchart.com*. Accessed on 23 Nov. 2019. URL: https://www.lucidchart.com/pages/landing/uml_diagram_tool.

[11] Eclipse.org. (2019). Papyrus. [online] Available at: https://www.eclipse.org/papyrus/ [Accessed 9 Dec. 2019].

[12] Methodsandtools.com. (2019). ArgoUML - Open Source Unified Modeling Language UML Tool. [online] Available at: https://www.methodsandtools.com/tools/tools.php?argouml [Accessed 9 Dec. 2019].

[13] Inc, N. (2019). MagicDraw. [online] Nomagic.com. Available at: https://www.nomagic.com/products/magicdraw [Accessed 9 Dec. 2019].

[14] Staruml.io. (2019). StarUML. [online] Available at: http://staruml.io/ [Accessed 9 Dec. 2019].

[15] Argouml.tigris.org. (2019). argouml: ArgoUML Features. [online] Available at: http://argouml.tigris.org/features.html [Accessed 9 Dec. 2019].

[16] Argouml.tigris.org. (2019). argouml: Issue 1834. [online] Available at: http://argouml.tigris.org/issues/show$_bug.cgi?id = 1834[Accessed9Dec.2019]..

[17] PlantUML.com. (2019). Open-source tool that uses simple textual descriptions to draw beautiful UML diagrams.. [online] Available at: https://plantuml.com/ [Accessed 9 Dec. 2019].

[18] PlantUML.com. (2019). PlantUML Language Reference Guide. [PDF] Available at: http://plantuml.com/guide [Acessed 15 Feb. 2020].

[19] GraphViz - Renderer of choice (2020). [online] Available at: https://www.graphviz.org/ [Accessed 28 March. 2020].

[20] Tigris.org - XSL transformation (2020). [online] AAvailable at: http://argouml-graphviz.tigris.org/ [Accessed 15 Feb. 2020].

[21] Guidelines: Statechart Diagram [online] Available at: https://sceweb.uhcl.edu/helm/RationalUnifiedProcess/process/modguide/md$_stadm.htm[Accessed16Feb.$2020].

[22] JUnit 5 (2020). [online] Available at: https://www.haskell.org/ [Accessed 13 May. 2020].

[23] Eclipse (2020). [online] Available at: https://www.eclipse.org/ [Accessed 28 March. 2020].

[24] IDE markert share (2020). [online] Available at: https://www.datanyze.com/market-share/ide–444 [Accessed 28 March. 2020].

[25] Eclipse MarketPlaces (2020). [online] Available at: https://marketplace.eclipse.org/category/categories/languages [Accessed 28 March. 2020].

[26] Jeff McAffer. *Eclipse Rich Client Platform*. Eclipse RT, 2010.

[27] Eclipse - Task Tag preferences. [online] Available at: https://www.eclipse.org/pdt/help/html/task$_tags.htm[Accessed20April.$2020].