EECS3311 Project 3
David Zakharian – 217117037 Sec A Lab 01,
Orion Nelson – 216305377 Sec A Lab 01,
Dennis Johnson – 210666915 Sec A Lab 01,
Hussain-Fatmi – Sec A Lab01

EECS 3311 Lab:
Third Software Project – Controller Project

EECS3311 Project 3
David Zakharian – 217117037 Sec A Lab 01,
Orion Nelson – 216305377 Sec A Lab 01,
Dennis Johnson – 210666915 Sec A Lab 01,
Hussain-Fatmi – Sec A Lab01

Contents

Part 1: Introduction 1
Part II: Design of the solution 2
Template Method Pattern: 2
Command Pattern: 2
Observer Pattern: 2
Implementation of Patterns: 3
Part III: Implementation of the Solution 4
Tools: 4
Process: 4
Part IV: Discussion/Conclusion 7
What went well in the software project? 7
What went wrong in the software project? 7
What have you learned from the software project? 7
What are the advantages and drawbacks of completing the lab in group? 7
Project breakdown in terms of output and deadlines: 7

EECS3311 Project 3
David Zakharian – 217117037 Sec A Lab 01,
Orion Nelson – 216305377 Sec A Lab 01,
Dennis Johnson – 210666915 Sec A Lab 01,
Hussain-Fatmi – Sec A Lab01

Part 1: Introduction

For this software project, the goal is to develop a Model-View-Controller (MVC) application that allows converting a value (specified in centimetres) to feet and meters respectively.

Challenges associated with this software project were slightly different from Project 2 as it also included the use of the mandated design patterns, Observer and Command. It also required a different line of thinking with respect to the design patterns mentioned above.

The MVC (Model-View-Control) architecture used in Project 2 helped refine the implementation considerably and thus, it required less time to create and refine the end product. A rough UML design was used to write out the code and a then final UML design with all the details was fleshed out once we reached 100% coverage.

Some of the Object-Oriented Principles (OOP) used to carry out the software project include *abstraction* and *inheritance*.

For our project:

- The Template Method pattern: Was used to generate the 3 measurement classes (Centimeter, Feet, Meter Conversion Area) and its most relevant operation, update.
- The Command pattern: Was used to generate the MenuOptions class and its most relevant operation save
- The Observer pattern: Was used to generate the ValueToConvert and ConcreteSubject class and its most relevant operations, save and notify (to get and update the Conversion Area classes)

These will be discussed in further detail in part II of our report

As this project was developed very similarly to Project 2, the report will also reflect the stages of development from design to implementation, modifications, testing coverage and discussion/conclusion. The final application created; will the goals have outlined at the start of this section. Our conclusion will discuss what went well, wrong and what could be improved.

Part II: Design of the solution

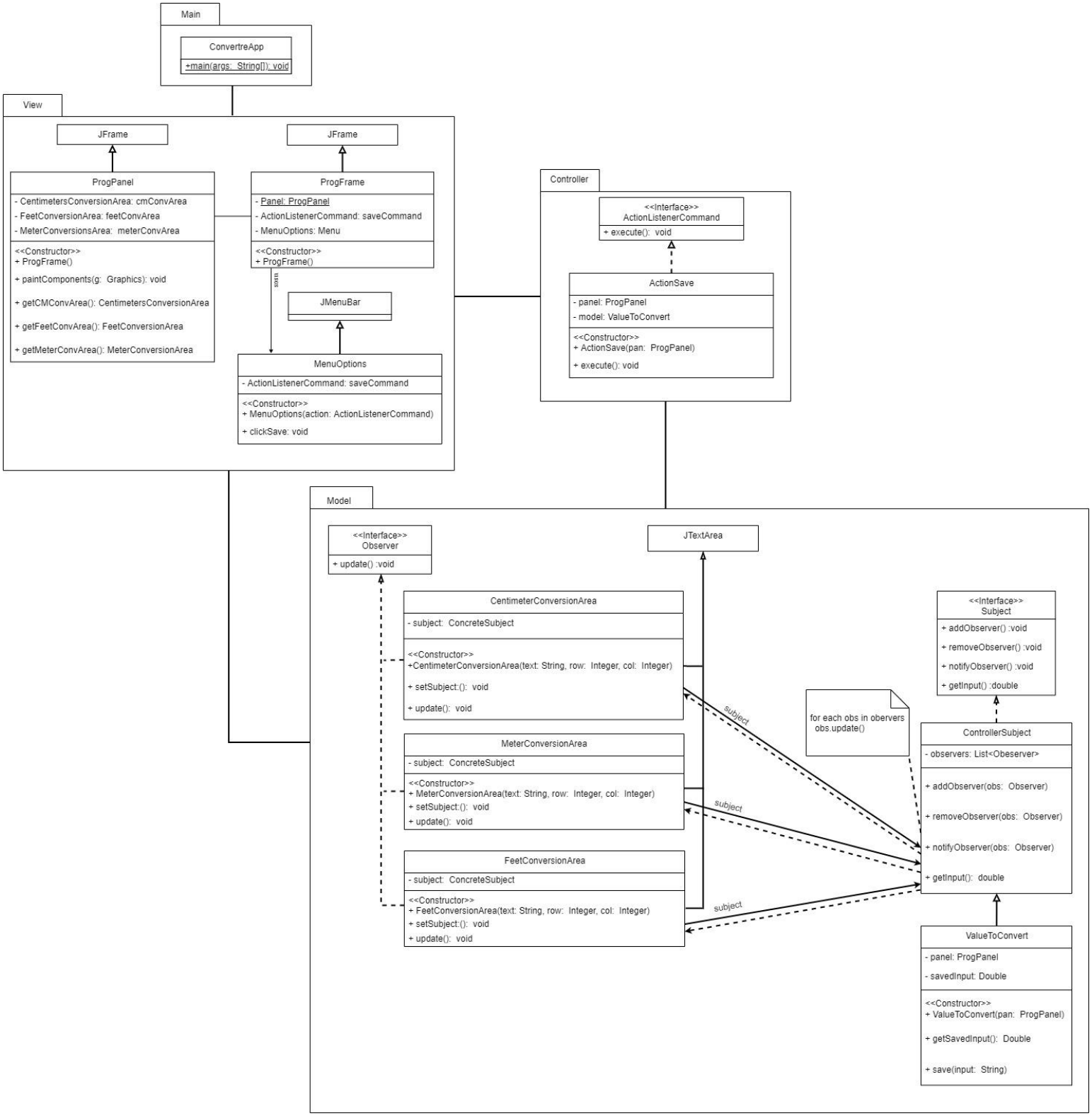


Figure 1: UML Diagram of Controller Project showing the MVC components (drawn in Draw.io)

EECS3311 Project 3
David Zakharian – 217117037 Sec A Lab 01,
Orion Nelson – 216305377 Sec A Lab 01,
Dennis Johnson – 210666915 Sec A Lab 01,
Hussain-Fatmi – Sec A Lab01

As mentioned in Part I, there are at least three design patterns shown in the Unified Model Language (UML.)

Template Method Pattern:

The Template Method Pattern was used to define the skeleton of the conversion which was encapsulated in their own classes: CentimeterConversionArea, FeetConversionArea, and MeterCoversionArea.

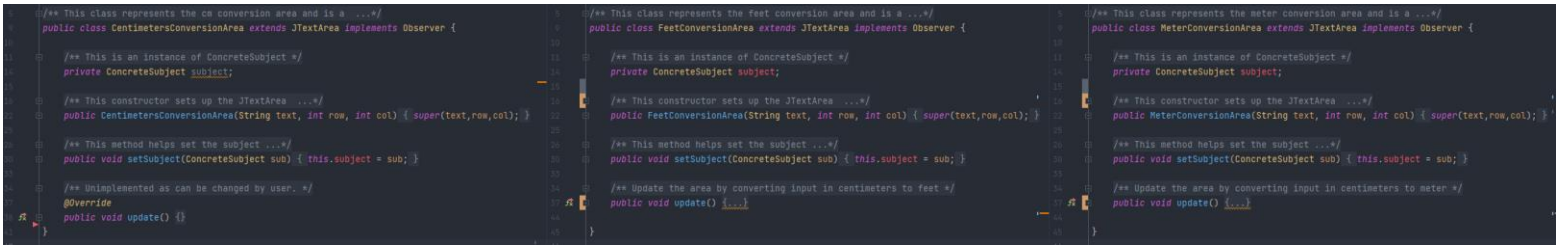


Figure 2: Example of Template Method Pattern

Each Update function was either re-defined to convert the value given in the Centimeter class to Feet and Meters respectively or was left empty since it was meant to be changed by the user.

Command Pattern:

The Command pattern helped encapsulate and parametrize the clients with the save request. The participants of the Command Pattern are:

- Command: The ActionListenerCommand interface is the Command Participant. It's declared as an interface which handles the execution of save operation through execute() method.
- ConcreteCommand: The ActionSave class implements Execute by invoking the relevant method (save) on the receiver class, CentimeterConversionArea
- Invoker: The MentuOptions class, asks the Command to perform the request, save
- Receiver: The receiving class CentimeterConversionArea, updates the relevant text

See Figure 3 (show below) to see how it was implemented in our code:

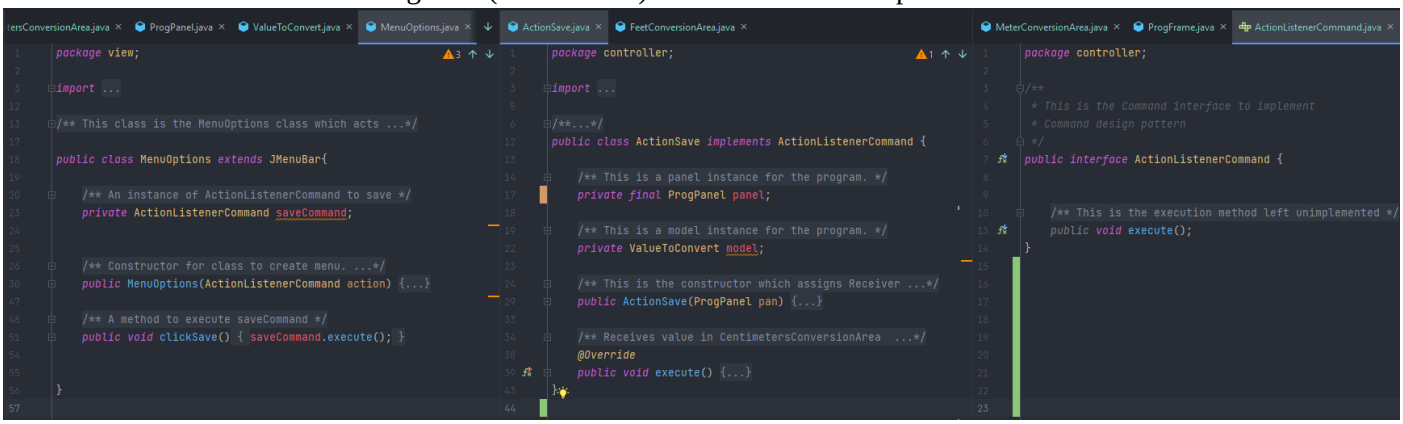


Figure 3: Example of Command Pattern

Observer Pattern:

The Observer pattern was used to specify a one-to-one relationship between the objects, Centimeter Conversion Area, Meter Conversion Area and Feet Conversion Area so that when the user uses the Menu bar

and clicks Save (or uses the shortcut Alt+F), all the objects would also be notified and updated automatically. As shown in Figure 4 (shown below), the code allows the client/user to manipulate all 3 objects mentioned above by notifying and updating the objects automatically. The participants of the Observer Pattern are:

- Subject: The interface Subject contains operations (add, remove, notify and getInput) needed for notifying and updating methods
- Observer: The class ValueToConvert observes the CentimeterConversionArea class and is notified when it is updated
- ConcreteSubject: The class ConcreteSubject stores the state of Centimeter, Meter and Feet Conversion Area Class since those classes are updated when Centimeter is modified
- ConcreteObserver: The classes shown in Figure 2 (CentimeterConversionArea, MeterConversionArea and FeetConversionArea) all maintain a reference to a ConcreteSubject class and update themselves to remain consistent

Figure 4: Example of Observer Pattern

Implementation of Patterns:

To use all these patterns, OOP such as inheritance, encapsulation and abstraction were used. For example, both *ProgPanel* and *ProgFrame* inherit methods and properties from their public class *JFrame* while adding their own properties and methods. Polymorphism is used when the class, *ActionSave*, overrides the default execute in ActionListenerCommand. Abstraction is used when the classes (Centimeter conversion, Meter conversion, and Feet conversion shown in Figure 2) are implemented using the Observer interface. Lastly, encapsulation is used with the conversion classes, as the field, *ConcreteSubject*, is made private and only class methods can be used to access it. Encapsulation is alos used to encapsulate the cenitmeter value given by the user in the CentimetersConversionArea class

We included additional design patterns/classes in the model as mentioned in the requirements (and above):

- Observer design Pattern
- Command design Pattern

Part III: Implementation of the Solution

Tools:

The project was implemented using a combination of tools:

- GitHub – Software Version Control (SVN)
- Diagrams.net – Unified Modelling Language (UML)
- Maven 3.8.1– Project Management Tool
- Junit 5.7.0– Unit Testing Framework
- Jacoco 0.8.7 – Java Code Coverage
- CodeCov.io – Test Coverage Analyzer
- Eclipse 20210910-1417 – Java Integrated Development Environment (IDE)
- IntelliJ IDEA 2021.2.3 – Java IDE
- Discord 103871 – Communication

Process:

A rough design was used to reflect the requirements of Project 3. The code was then uploaded in GitHub while Eclipse/IntelliJ were configured to use GitHub as the SVN to keep track of code/issues.

All the additional design patterns requested (and described in Part 2) were implemented using the tools mentioned above. We implemented the classes from main onwards and gradually added code to controller (actionListenerCommand), model (Observer and Subject) and view (ProgFrame and ProgPanel) classes progressively. Once the skeleton structure was working, we added the ActionSave, Centimetres/Feet/MeterConversionArea.
The initial build allowed us to hash out a solution and achieve an 80% test coverage very quickly. Our code runs without triggering any exceptions and precise documentation (generated by JavaDocs) given here: https://or9.ca/Lab6/ and in shown in GitHub.

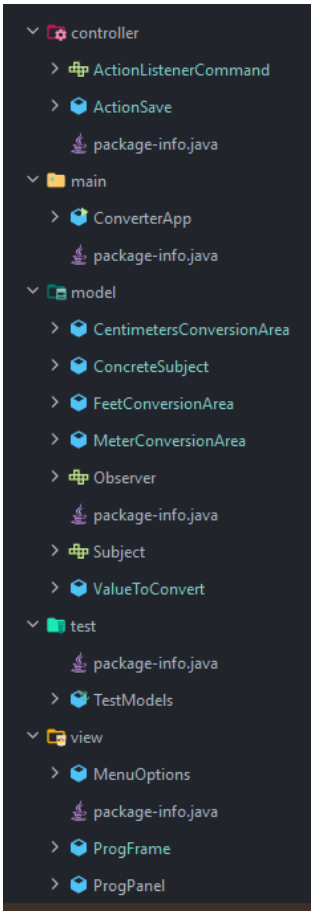


Figure 5: Five Packages (model, view, controller and test)

Our code contains 5 packages: model, view, controller, main and test (See Figure 5 for program packages.) The view package consists of a ProgFrame (which uses JFrame), ProgPanel (which uses JPanel) and MenuOptions (which uses JMenuBar.) The model package consists of a class named ValueToConvert which encapsulates cm value specified by the user. ValueToConvert class notifies and updates FeetConvesionArea and MeterConversion area when the save option is triggered. The Controller package contains a class (ActionSave) that saves the value specified by the user and sets the state of ValueToConvert class. The main package contains the main class which launches the application. The test package contains the testModels and tests various components to achieve a 100% test coverage.

We used JUnit to test and make sure the classes we implemented work as expected. The test package is shown below to test out the different components:

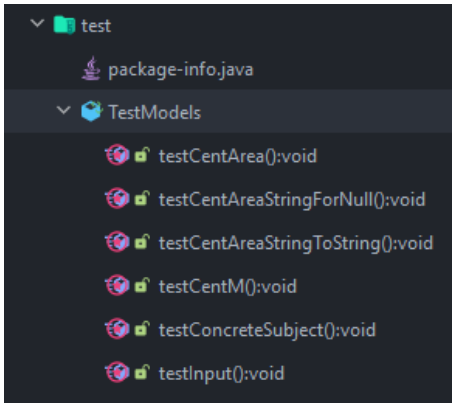


Figure 6: Test package using JUnit5 to test various components of the project

We then used JaCoCo/CodeCov to measure code coverage. We achieved a 100.0% coverage and started working on the final UML, JavaDocs and report.

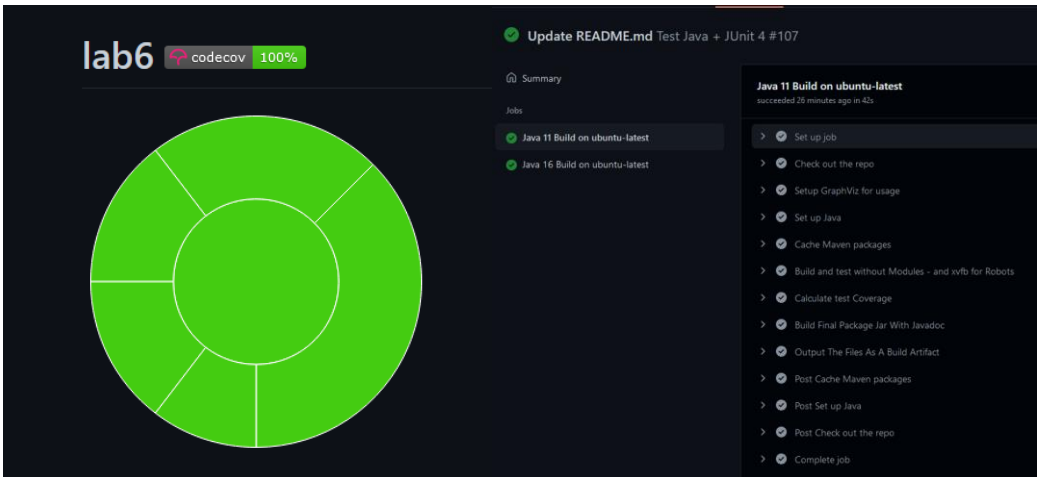


Figure 7: Codecov automated test output on Ubuntu machines running Java 11/16

The entire process (compile, build, test and generate docs & UML) was automated using Maven. By using option, "Maven Install", we get this output

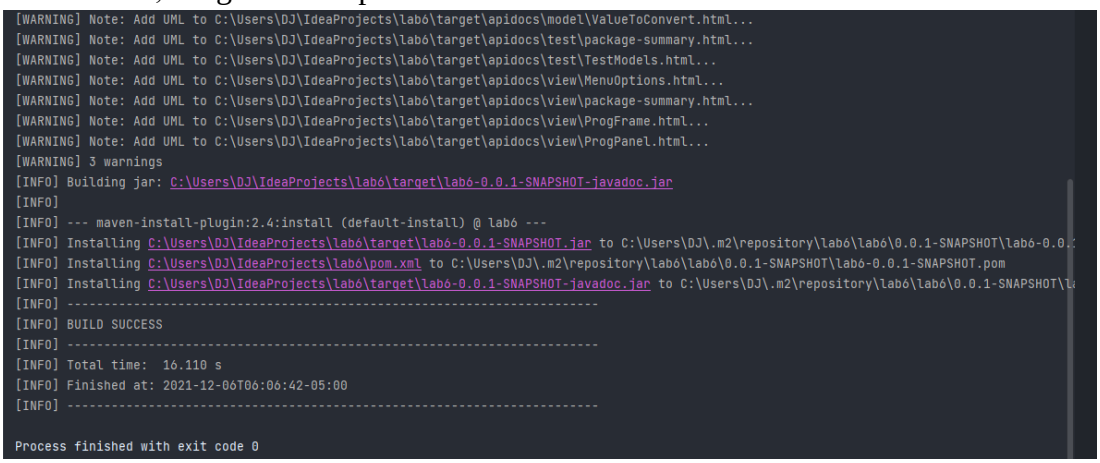
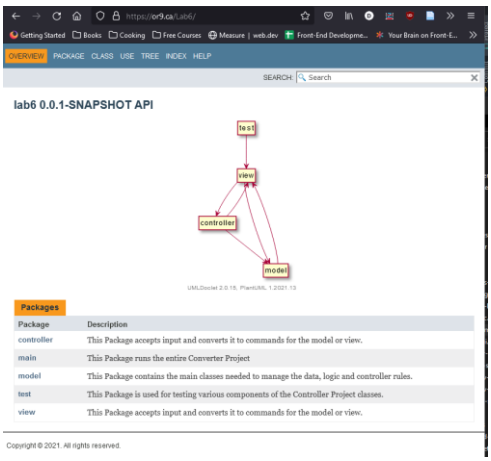


Figure 8: Output of using Maven Install using Maven & pom.xml

The Java Docs & an automated UML diagram were re-generated using Maven and the build output can be found here in this link: https://or9.ca/Lab6/. Finally, we were able to create an application that does the following:

- Create an application that displays an interface with one menu: MenuOptions
  - MenuOptions allows for a user to save and convert cm input in yellow square to feet and meters
- Create an UI that displays three conversion areas: centimeter, feet and meters
  - The green and orange views observe the model
  - The controller stores the input value in the model and the model automatically notifies green and orange views
- Use Maven to automatically build, test, generate JavaDocs and UML diagrams every time we made a change to the code:



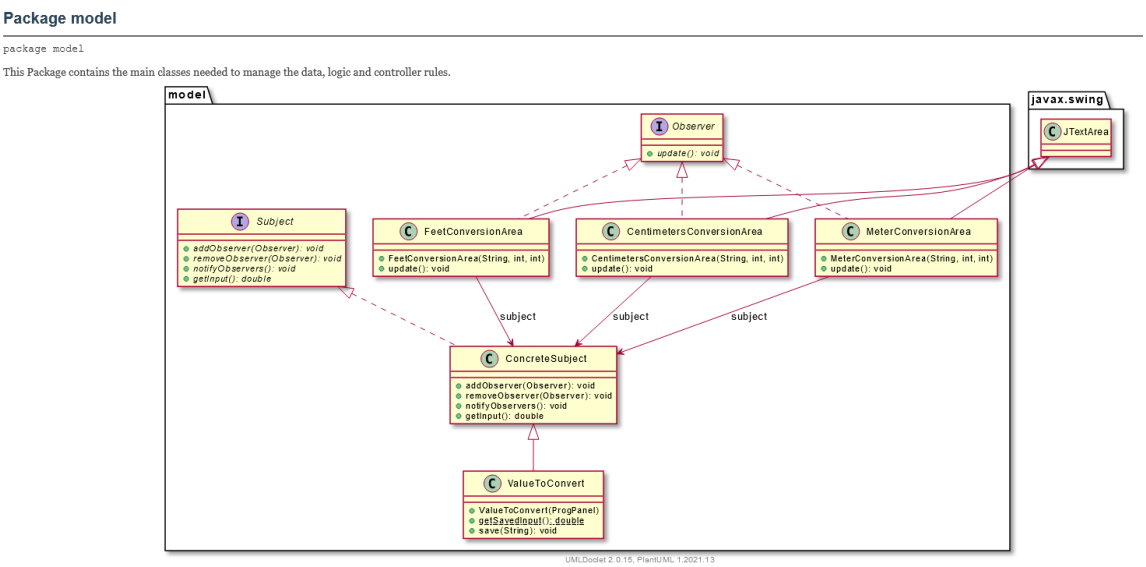


Figure 9: Automatic generation of UML/JavaDocs upon every Maven build

- The video can be found as a link in the GitHub link: *https://github.com/orionnelson/lab6* or directly at *https://youtu.be/BoLrKZb8shM*

EECS3311 Project 3
David Zakharian – 217117037 Sec A Lab 01,
Orion Nelson – 216305377 Sec A Lab 01,
Dennis Johnson – 210666915 Sec A Lab 01,
Hussain-Fatmi – Sec A Lab01

Part IV: Discussion/Conclusion

What went well in the software project?

Overall, the project was collaborative in the sense each group member worked on what they were strongest in and contributed where possible.

What went wrong in the software project?

While we were able to reach 100% code coverage, we had one member join the day before and was unable to contribute meaningfully to the project

What have you learned from the software project?

While writing code is its own challenge, every person has their own form of contribution and feedback. As a result, the strongest aspects of each person come out and refines final project even further.

What are the advantages and drawbacks of completing the lab in group?

Some of the advantages of working in a group is that it becomes easy to divide on the workload. Groups progress very quickly with mutual respect and understanding as no one person makes an assumption of the other. Each person fully committed to the tasks made it easier to complete the project ahead of time. We were also able to discuss some final design changes with further discussion (not to combine cm class with concrete subject and create a mini-god class for example.)

Project breakdown in terms of output and deadlines:

Task Output

	Design	Implementation	Testing	Documentation
David Z	x	x		
Orion N		x	x	x
Dennis J			x	x
Hussain-F	N/A	N/A	N/A	N/A

Deadlines

	Design / (due)	Implementation / (due)	Testing / (due)	Documentation / (due)
David Z	Nov 3, 21	Dec 2, 21		
Orion N		Dec 5, 21	Dec 5, 21	
Dennis J				Dec 6, 21
Hussain-F	N/A	N/A	N/A	N/A

We had one member in our team who joined our Discord group by the time the code was done and all other parts were finished shortly after.