2010

Auto-Generation of Use Case Controllers

Pushkar Marathe
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the [Computer Sciences Commons](#)

Recommended Citation

Marathe, Pushkar, "Auto-Generation of Use Case Controllers" (2010). *Master's Projects*. 27.
DOI: <https://doi.org/10.31979/etd.ve7e-56mr>
https://scholarworks.sjsu.edu/etd_projects/27

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

AUTO-GENERATION OF USE CASE CONTROLLERS

A Research Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Computer Science

by

Pushkar Marathe

Spring 2010

© 2010

Pushkar Marathe

ALL RIGHTS RESERVED

SAN JOSÉ STATE UNIVERSITY

The Undersigned Thesis Committee Approves the Project-Thesis Titled

AUTO-GENERATION OF USE CASE CONTROLLERS

by

Pushkar Marathe

APPROVED BY THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Jon Pearce,

Department of Computer Science

Date

Dr. Chris Pollett,

Department of Computer Science

Date

Dr. Agustin Araya,

Department of Computer Science

Date

APPROVED FOR THE UNIVERSITY

Associate Dean

Office of Graduate Studies and Research

Date

ABSTRACT

AUTO-GENERATION OF USE CASE CONTROLLERS

The thesis is based on the concept of generating machine executable code from UML diagrams. There has been some research done on the same topic but it is mainly focused on class diagrams. My main focus will be on the use case diagrams, and sequence diagrams. Sequence diagrams give us the details of any process in a step-by-step fashion which makes it easier to extract the required information regarding the use cases and the controllers which help us generate the code. The process follows three steps, first being generation of UML diagrams. Secondly, these are converted into XMI (XML Metadata Interchange) which is a XML format. In the end this XMI file is converted into machine executable Java code with the help of a tool that I will be generating.

We will be using StarUML as the UML editor for the thesis which also provides the functionality of converting UML diagrams into XMI format. The advantages of this process include the reduction of time complexity of the process and ease of coding for the software developers. In future, this project can be extended to map the requirements provided by the customer to the software developed so that the customer can understand the system very easily. This project could be further expanded so that it can generate code even considering the state chart diagrams for any system. The report includes the details of the implementation and the work I have done. I have named the software as ProtoGen and will be referring to it as ProtoGen throughout the report.

Acknowledgements

I would really like to thank Dr. Jon Pearce, for giving me an opportunity to work on this research project under his guidance. I also thank him for the patience, ideas, suggestions and inspirations without which this Master's research project would not have been possible. I would also like to thank Dr. Cay Horstmann, Dr. Chris Pollett and Dr. Agustin Araya for their time and feedbacks.

I would also like to thank Prof. Janelle Melvin for helping and guiding me with the writing of the research project.

Table of Contents

1. Introduction.....	1
2. Modeling Requirements	3
2.1 Requirements Model for an ATM	4
2.2 Requirements Model for Travel Mania	7
2.3 Scenarios	10
2.4 Requirements Model for ProtoGen	11
2.4.1 ProtoGen Snapshots	13
2.5 ProtoGen Analysis Model	16
3. XMI Description.....	18
4. Architectures	24
4.1 Model-View-Controller Architecture	24
4.2 Hexagonal Architecture.....	26
5. ProtoGen Prototypes.....	31
5.1 ATM Prototype.....	31
5.2 Travel Mania Prototype	38
6. Spectrum of Control	44
7. ProtoGen Design and Implementation	45
7.1 Handling Functions	48
7.2 Handling Loops and Conditions.....	48
7.3 Limitations	50
8. Future Work	51
9. Conclusion.....	52
10. References	53

List of Figures

Figure 1: Functionality of ProtoGen.	1
Figure 2: Use Case Model for ATM.	4
Figure 3: Analysis model for ATM.	5
Figure 4: Sequence Diagram for ATM.	6
Figure 5: Use Case model for Travel Mania.....	7
Figure 6: Analysis Model for Travel Mania.	8
Figure 7: Sequence Diagram for Reservation System.	9
Figure 8: ProtoGen Use Case.....	11
Figure 9: Sequence Diagram for ProtoGen.....	12
Figure 10: ProtoGen Start Page.	13
Figure 11: ProtoGen Select File option.	14
Figure 12: Generated Code by ProtoGen.....	15
Figure 13: Analysis Model for ProtoGen	16
Figure 15: Typical Project Model	17
Figure 16: MVC pattern	25
Figure 17: Hexagonal Architecture.....	27
Figure 18: ProtoGen Framework.	28
Figure 19: MVC framework.	28

Figure 20: GUI generated for the ATM.	32
Figure 21: ActionPerformed method of the GUI class.	33
Figure 22: Execute function of the controller.	34
Figure 23: Ask function of the GUI class.	35
Figure 24: Execute method of vbank class.	36
Figure 25: Ask method of transfer class.	37
Figure 26: View of Reservation System.	38
Figure 27: Reserve Car Controller.....	39
Figure 28: Execute method of Reserve Car controller.....	40
Figure 29: Ask method of the Reservation class.	41
Figure 30: Get Personal Info.....	42
Figure 31: Execute method of “getpersonalInfo” controller.....	43
Figure 32: Loops and Conditions.....	48
Figure 33: Conditions in UML.	49
Figure 34: Example of Combined fragment.	50

List of Tables

Table 1: XMI overview.....	19
Table 2: XMI Content.....	19
Table 3: Actor and Use case tag.	20
Table 4: Message Tag.	21
Table 5: Class Tag.	22
Table 6: Objects.	23
Table 7: Oracle Interface as generated by ProtoGen.	29
Table 8: Controller class generated by ProtoGen.	29
Table 9: Model class generated by ProtoGen.	30
Table 10: The “getElement” function.	46
Table 11: The “getChildren” function.	46
Table 12: Separating classes from the XMI file.	47

Auto-Generation of Prototypes from Requirements Models

1. Introduction

The main aim of this project was to generate a tool that would generate code from UML diagrams. The tool generated is called ProtoGen. ProtoGen is an experimental tool that enables developers to generate prototypes from requirements models.

Two techniques commonly used for specifying a system are prototyping and requirements modeling. A prototype can give stakeholders a better sense of what the system will do. It can also give developers a starting point to build upon. However, prototyping can be expensive. A UML requirements model is easy to generate, but doesn't give the same rich feedback to stakeholders, nor does it give a code-based starting point for developers. Also, sometimes the connection between requirements and code can be difficult to trace.

Figure 1: Functionality of ProtoGen.

ProtoGen connects the requirements model with the prototype. The requirements model is directly mapped to the code so that it is easier for the user to understand the exact functionality of the system. This tool also gives a starting point for the developer to work on making their work easier as they already have the base ready and they just have to fill in the code that they need to make the system efficient.

Many different tools that generate UML diagrams have the functionality of generating code but it only generates class definitions for the use cases present in the system. ProtoGen, on the other hand, generates a prototype of how a system would look like once it has been developed. It not only contains the class definitions but also the “ask ()” and “execute ()” functions for each of the class that has been generated.

The report takes us through the modeling requirements of any system followed by the XMI description and the Architectures. This part is then followed by the prototypes developed by ProtoGen and the implementation details of ProtoGen. The report ends with the future work and conclusion to ProtoGen. Let us start by taking a look at the modeling requirements for the software.

2. Modeling Requirements

The two models that describe any system are the design and requirements model. The requirements model explains the user's perspective while the design model explains us the developer's perspective. The requirements model consists of two parts: a use case model and an analysis model. ProtoGen takes in a XMI file as an input which is basically this requirements model converted to code that the machine can understand.

The use case model shows the system's functions and how actors interact with these functions. On the other hand the analysis model shows the entity, control, and boundary objects and how they interact with each other. Let us take a look at the requirements model for the two examples which we will be considering throughout the report.

2.1 Requirements Model for an ATM

The first example we will be considering is that of an ATM. Figure 2 shows the use case model for the ATM.

Figure 2: Use Case Model for ATM.

Figure 3 shows the analysis model. Figure 2 and Figure 3 together gives us the requirements model for the ATM.

Figure 3: Analysis model for ATM.

In the first example, we have 3 different use cases that have been implemented which are “transferfunds”, “deposit”, and “withdraw”. These use cases have associated scenarios which we have used to develop our code. ProtoGen uses the requirements model to identify the actors and use cases which are involved in the system to be prototyped. The design model provides information about the classes and the interaction between the different entities involved in the system. While building ProtoGen, the first draft generated one single controller which has a couple of if else statements to differentiate the function which needs to be executed. We can see this in the ATM example. Figure 4 on the next page shows the sequence diagram for the ATM. As we can see there is only one controller named “transfer” which takes care of all three use cases related to the ATM which is “transferfunds”, “withdraw”, and “deposit”. This diagram is used by ProtoGen to generate code for the controller.

Figure 4: Sequence Diagram for ATM.

2.2 Requirements Model for Travel Mania

The second example that we will be referring to would be the Reservation System which is used to rent cars, reserve a room, and book a flight. We will name this system as Travel Mania For this example; “rent-car” and “reserve-flight” use cases have been completely implemented as we will see in the later part of the report.

Figure 5 below shows the use case model for Travel Mania.

Figure 5: Use Case model for Travel Mania.

The analysis model will be similar to the one shown above and is shown in Figure 6 below.

Figure 6: Analysis Model for Travel Mania.

Figure 7 shows the sequence diagram for Travel Mania example. In this example we can see that there is an individual controller for each use case that is present in the system and so creating a separate Java file for each of these unlike the ATM example where only one controller gets generated. This is better as it becomes clearer as to how the sequence diagram gets mapped onto the Java code. Figure 7 is not the complete sequence diagram but a part of the reserve car controller. The part that is not shown is that of reserve flight and that of making a payment, which is similar to the part that is shown. It also shows the type of all objects present in the diagram. The first class named “reservation” is of type “View” which tells us that it is the boundary

that interacts with the user for that system. The last class is of type model and the remaining entities are of type controller which indicates that these are the controllers that will be generated as the output of ProtoGen.

Figure 7: Sequence Diagram for Reservation System.

2.3 Scenarios

Let us take a look at the withdraw scenario for ATM:

```
Account Holder: inserts card
ATM: asks for PIN
Account Holder: enters PIN
ATM: asks for source account
Account Holder: specifies source account
ATM: asks for amount to be withdrawn
Account Holder: specifies amount
ATM: requests account to be debited
Bank: debits account
ATM: dispenses cash
```

The second example is of a Travel Mania whose sample scenario would look like the following:

```
User: opens website
Reservation System: displays options
User: selects to reserve a car
Reservation System: asks for related information
User: specifies the required information
Reservation System: provides a list of available cars
User: selects a car
Reservation System: asks for personal and payment information
User: enters the information
Server: books the car
Reservation System: returns success message
```

2.4 Requirements Model for ProtoGen

The above sections gives us the requirements models for the two examples which we will be considering but now we should also take a look at the requirements model of the ProtoGen system which we will be using to generate code for the above shown examples. This software basically takes in an XMI file as an input and gives Java code as output so it has only one use case as shown in Figure 8. UML model can be viewed by humans as a diagram, but is viewed by programs as an XML document. XMI is the XML standard for UML models. XMI has been described in section three of this report.

Figure 8: ProtoGen Use Case.

There will only be one scenario in this case as the only functionality this software has to generate Java code form the XMI file. Figure 9 shows that scenario.

Figure 9: Sequence Diagram for ProtoGen.

The figure above shows how the developer interacts with the ProtoGen. It is basically a short scenario consisting of few steps which include specifying the requirements model for the system for which u need the prototype to be generated and then saving the generated code in the specified destination folder. One restriction to operating this system is that the requirements model which is the input to ProtoGen is in XMI format. Almost all of the UML tools that we have now have the functionality of transforming the UML diagram to XMI. We are using StarUML for this project. There are some other restrictions that we need to take care while generating the UML diagram which we will discuss later. As of now the only thing to take care is that the input should be an XMI file.

2.4.1 ProtoGen Snapshots

Let us take a look at some snapshots of ProtoGen which helps you understand the sequence diagram it follows. Figure 10 shows the main page of ProtoGen from which the developer selects the input file.

Figure 10: ProtoGen Start Page.

Figure 11 below shows the folder from which the input file is selected and the in our case the source and destination folder is same so it also shows the folder where the generated files are being stored. As soon as the output files are generated ProtoGen is ready to generate code for the next available input. Once we are done using the ProtoGen we can click on the exit button to exit from the software. As you can see below the input file we are using is “atm1.xml” which is the UML diagram of ATM converted into the XMI format.

Figure 11: ProtoGen Select File option.

As soon as we select the input file, ProtoGen takes this XMI file and starts parsing it. It then figures out the use cases, classes, and actors from the file which helps the ProtoGen to generate the code. Figure 12 shows the folder where the

generated code is stored. We will discuss about this generated code in detail in the later part of the report and also go through the flow of the generated code.

Figure 12: Generated Code by ProtoGen.

2.5 ProtoGen Analysis Model

The analysis model for ProtoGen would not be as complicated as that of the systems that it develops the prototype for. Figure 13 shows the analysis model for ProtoGen.

Figure 13: Analysis Model for ProtoGen [9].

The entities package shown in the figure above is basically the requirements model of the system for which the prototype is to be developed. There is a format in which ProtoGen expects the input to be. The input file should have a use case diagram, a class diagram that specifies the framework for the system, and a sequence diagram which actually helps in developing the code. Figure 2 above gives an example of the use case model for the ATM example we are considering. Similarly Figure 5 shows the use case diagram for Travel Mania example. The use cases for the ATM are “transferfunds”, “deposit”, and “withdraw” while for Travel Mania we have “rent-car” and “reserve-flight” as the use cases being developed.

The class diagram is the main building block in object oriented modeling. The classes in these diagrams tell us actually which all classes are to be implemented. In our cases it will be the controller, model and the view class as we are using the model-view-controller framework which is discussed below.

Once the UML model has been developed which basically consists of a requirements model as shown above and a design model which has the sequence diagram showing the interaction between the different controllers and actors, it is then converted to XMI using the inbuilt functionality of any UML tool (In my case it is StarUML).

Figure 14 below shows a typical project.

Figure 14: Typical Project Model [9].

3. XMI Description

“The XML Metadata Interchange (XMI) is an Object Management Group (OMG) standard for exchanging metadata information via Extensible Markup Language (XML)” [11].

A typical XMI file has the following tags associated with it:

- Header: It describes the type of UML and XMI used.
- Content: It consists of the requirements as well as design model for the system.
- Actor: It specifies the actors in the system.
- Use Case: It gives us a list of use cases related to the system.
- Message: The flow of operation in the sequence diagram is represented as message in XMI file.
- Class: It specifies the classes associated with the system.
- Classifier-Role: It specifies the entities in the sequence diagram.

Tables on the next page give us brief idea of how the XMI for the Reservation System example looks like.

This is the overview of the XMI file. It basically contains header information and everything else is stored in the Content tag.

```
<XMI xmi.version = "1.1" timestamp = "Thu Feb 11 10:28:8
2010">

<XMI.header> </XMI.header>

<XMI.content> </XMI.content>

</XMI>
```

Table 1: XMI overview.

The packages shown above separate the analysis model and the design model.

```
<UML:Package xmi.id="UMLPackage.2" name="analysis model"
visibility="public" isSpecification="false"
namespace="UMLProject.1" isRoot="false" isLeaf="false"
isAbstract="false">

</UML:Package>

<UML:Package xmi.id="UMLPackage.136" name="design model"
visibility="public" isSpecification="false"
namespace="UMLProject.1" isRoot="false" isLeaf="false"
isAbstract="false">

</UML:Package>
```

Table 2: XMI Content.

```

<UML:Actor xmi.id="UMLActor.109" name="Customer"
visibility="public" isSpecification="false"
namespace="UMLPackage.2" isRoot="false" isLeaf="false"
isAbstract="false" participant="UMLAssociationEnd.126"/>

<UML:UseCase xmi.id="UMLUseCase.115" name="Balance"
visibility="public" isSpecification="false"
namespace="UMLPackage.2" isRoot="false" isLeaf="false"
isAbstract="false"
generalization="UMLGeneralization.131"/>

<UML:UseCase xmi.id="UMLUseCase.116" name="Transfer Funds"
visibility="public" isSpecification="false"
namespace="UMLPackage.2" isRoot="false" isLeaf="false"
isAbstract="false"
generalization="UMLGeneralization.132"/>

```

Table 3: Actor and Use case tag.

The tags in Table 3 above are the actors and use cases described in the Reservation System example. These tell us the number of controllers to be generated as the output of ProtoGen.

\

The message tag in Table 4 below is the interaction between the different actors and entities within the system as depicted in the sequence diagram.

```
<UML:Interaction.message>
  <UML:Message xmi.id="UMLStimulus.140" name="transferfunds"
  visibility="public" isSpecification="false"
  sender="UMLObject.210" receiver="UMLObject.211"
  interaction="UMLInteractionInstanceSet.139">
    <UML:Message.action>
      <UML:SendAction xmi.id="UMLSendAction.141" name=""
      visibility="public" isSpecification="false"
      isAsynchronous="false" stimulus="UMLStimulus.140"/>
    </UML:Message.action>
  </UML:Message>
```

Table 4: Message Tag.

The table below gives us a list of all classes that are present in our system. This basically shows our framework which is the model-view-controller framework.

```
<UML:Class xmi.id="UMLClass.214" name="Model"
visibility="public" isSpecification="false"
namespace="UMLPackage.137"
clientDependency="UMLRealization.235" isRoot="false"
isLeaf="false" isAbstract="true"
participant="UMLAssociationEnd.225 UMLAssociationEnd.228"
isActive="false"/>

<UML:Class xmi.id="UMLClass.215" name="Boundary"
visibility="public" isSpecification="false"
namespace="UMLPackage.137"
clientDependency="UMLRealization.233" isRoot="false"
isLeaf="false" isAbstract="true"
participant="UMLAssociationEnd.221 UMLAssociationEnd.224"
isActive="false"/>
```

Table 5: Class Tag.

These are the objects as represented by the sequence diagram.

```
<UML:ClassifierRole xmi.id="UMLObject.212"
name="transfer" visibility="public"
isSpecification="false" base="UMLClass.216"
message2="UMLStimulus.150 UMLStimulus.148 UMLStimulus.146
UMLStimulus.172 UMLStimulus.174 UMLStimulus.176
UMLStimulus.194 UMLStimulus.196 UMLStimulus.198
UMLStimulus.144 UMLStimulus.166 UMLStimulus.188
UMLStimulus.192 UMLStimulus.202 UMLStimulus.152
UMLStimulus.158 UMLStimulus.178 UMLStimulus.200
UMLStimulus.162 UMLStimulus.164 UMLStimulus.184
UMLStimulus.186 UMLStimulus.206" isRoot="false"
isLeaf="false" isAbstract="false">
  <UML:ClassifierRole.multiplicity>
    <UML:Multiplicity xmi.id="X.292">
      <UML:Multiplicity.range>
        <UML:MultiplicityRange
xmi.id="X.293" lower="1" upper="1" multiplicity="X.292"/>
      </UML:Multiplicity.range>
    </UML:Multiplicity>
  </UML:ClassifierRole.multiplicity>
</UML:ClassifierRole>
```

Table 6: Objects.

4. Architectures

ProtoGen follows the model-view-controller architecture where the view asks the controller to execute commands that the customer needs to perform. The code that I have generated now follows the hexagonal architecture which suggests the adapter-port structure where the adapter is least concerned about the port that has requested the action. The port is just the link between the users and the controllers. Let us take a look at both of them.

4.1 Model-View-Controller Architecture

Model-View-Controller is the concept of encapsulating some data together with its processing (the model) and isolates it from the manipulation (the controller) and presentation (the view) part that has to be done on a [User Interface](#). Figure 15 shows the MVC pattern.

- A **model** is an object encapsulating application data and logic, e.g. a database table or even some plant-floor production-machine process.
- A **view** is some form of visualization of the state of the model.
- A **controller** offers facilities to change the state of the model.

Figure 15: MVC pattern [6].

4.2 Hexagonal Architecture

Hexagonal architecture follows the port-adapter configuration where the ports are usually the interface for the users to communicate with the use case controllers. The adapter could be something that would translate the users command into a language that the ports can understand and request the controllers to perform the corresponding action. Hexagon corresponds to 6 sides which is totally arbitrary for this architecture and could be changed.

We can relate this hexagonal architecture to the ProtoGen. The innermost part is the core or the model in our case. The next layer is of the controllers and the third layer is that of the boundaries. The black dots between the boundaries and the controllers are the adapters which translate the output of boundaries to the expected input of the controllers. Outside of the hexagon are the actors. The sample user interface which is generated by ProtoGen is the port or boundary referred in the above architecture and the controllers which it generates are the controllers that execute the commands generated by the user. So with this architecture the port which is our GUI need not know about how the controller or the core operates. Figure 16 on the next page shows the hexagonal architecture.

Figure 16: Hexagonal Architecture.

Prototypes generated by ProtoGen are customizations of the ProtoGen framework.
This framework instantiates the Model-View-Controller architecture.

Figure 17: ProtoGen Framework.

Let us now look at how the model-view-controller pattern is implemented by ProtoGen. Figure 18 below shows the framework of the system for which the prototypes are to be developed. Both the examples in our case use the same pattern so this figure will be common for both the ATM and Travel Mania.

Figure 18: MVC framework.

Let us take a look at the code that has been generated for the framework shown above. Table 7 shows the code for the interface oracle which is generated by ProtoGen taken the framework into consideration.

```
interface Oracle {  
  
    String ask(String query) throws Exception;  
  
}
```

Table 7: Oracle Interface as generated by ProtoGen.

The next table which is Table 8 shows the controller class generated by ProtoGen. This class is implemented by all the use case controllers generated by ProtoGen and thus implementing the execute method of this class.

```
abstract class Controller implements Oracle {  
  
    protected Model myModel;  
  
    abstract public String execute(Oracle caller, String  
        cmmd) throws Exception;  
  
}
```

Table 8: Controller class generated by ProtoGen.

All the use-case controllers generated by ProtoGen extends this Controller class and thus implement the execute function as described in the later sections.

The third file generated from the framework is the model which is as described but the only thing here is that all the code for the model cannot be generated as UML diagram does not describe what the model would behave when the final execute command is called. So in the code that is generated which we will be seeing in the next part some part is commented out so that the flow of the program could be explained. Table 9 shows the model class generated.

```
abstract class Model extends Observer implements
Serializable {

public void subscribe(){

}

}
```

Table 9: Model class generated by ProtoGen.

The last class that is generated is the view which is the user interface. ProtoGen generates a sample user interface to test the flow of the program for the functions specified in the sequence diagram for the system. In ATM, these functions are “deposit”, “withdraw”, and “transfer” while in Travel Mania these are “rent-car and “reserve-flight”. We will be able to see this in the next part when we take a look at the prototype generated by ProtoGen.

5. ProtoGen Prototypes

In this section, we will take a look at the prototypes generated by ProtoGen. Let us begin with ATM example.

5.1 ATM Prototype

The models that are generated using StarUML are converted to XMI file using the StarUML inbuilt tool. This XMI file is then passed through ProtoGen which produces a number of Java files.

Let us take a look at the code being generated and how it flows between the different files which are generated. Figure 19 shows the code for the GUI file which displays the options which are implemented by the ATM. Here as you can see this is the file named `Boundry.java` which displays the GUI as seen in the figure. Let us now click on the “transferfunds” button from the GUI and follow what happens.

Figure 19: GUI generated for the ATM.

Figure 20 shows the “actionPerformed” method which is called when we click on the “transferfunds” button. As we can see an object of class transfer gets created which is used to call the respective functions, in our case it would be “transferfunds”. The objects for all the controllers get generated only once avoiding duplicate object creation. This object automatically gets created based on the controller in our sequence diagram. As we have clicked “transferfunds” button the highlighted part in the figure below shows that the execute method of the controller has been called with the parameter “transferfunds” which informs the controller of the action it needs to perform. Now let us take a look at the execute method of the controller.

Figure 20: ActionPerformed method of the GUI class.

Figure 21 shows the execute method of the `transfer.java` class. As we can see in the figure below I have highlighted the name of the current class and also the method which is being called at the moment. As we can see an object of the model, which is `vbank` in our case is automatically generated based on the sequence diagram. The code for the execute method shows that the controller asks the caller for some basic information like the source account, destination account, and the amount to be transferred. This call would take us back to the GUI which should ask the user for all these entries. After the user enters the information the execute method for the `vbank` gets called which finalizes the transaction.

Figure 21: Execute function of the controller.

Figure 22 shows the ask function of the GUI class which prompts the user to enter some basic details like the account numbers and the amount to be transferred. This ask method is in the user interface class so its purpose is to prompt the user to enter the input which is shown by the dialog box that pops up at the user interface as seen in the figure below.

Figure 22: Ask function of the GUI class.

After the user enters all the required information the controls goes back to the execute method of transfer controller as shown in Figure 21 where the next step gets executed which is calling the execute method of the vbank which is an object of the bank server. What goes on at the bank cannot be generated as it is a process not described by the UML diagrams. Figure 23 below shows this execute function of vbank class that is generated by ProtoGen.

Figure 23: Execute method of vbank class.

As we can see in this class the execute method of vbank class calls the ask methods of transfer controller which has all the information stored already in its local variables.

The ask method of transfer class just returns this value as shown in the Figure 24.

When the control returns to this class it calls the withdraw method which you can see is commented as the UML diagram does not specify the internal operations of any bank. It also generates if else conditions based on the branch conditions specified in the UML sequence diagram. If everything works properly it returns success to the caller and waits for the next input from the user.

Figure 24: Ask method of transfer class.

The above given snapshots depict how the system would perform once it is really developed. The user interface developed by the ProtoGen can easily be replaced by a richer one as this is just a prototype. As I have talked earlier, the ATM example uses a single controller that uses a multi-way conditional to execute all the commands.

5.2 Travel Mania Prototype

Similar to the ATM this also has a main function in the `Reservation.java` class which is of type “view” in the framework. It calls the constructor for the class which deploys the user interface as shown in Figure 25 below. In this example we have 3 use cases which are “rent-car”, “reserve-room”, and “reserve-flight”. Apart from these main use cases we have “getpersonalInfo” and “Makepayment” use cases which are used by all the main use cases.

Figure 25: View of Reservation System.

As soon as we click on the reserve car button it takes the control over to the “rent-car” controller as shown in Figure 26. As we can see the class name is highlighted and the action which gets performed is also highlighted so that you can see the flow of the program clearly. As we can see this calls the execute method of `ReserveCar.java` class. Unlike the ATM example, here we have different controllers so each one of them has an object initialized before being called. In this case “ReserveCar” object has been initialized. These controllers only get initialized once during the whole execution to avoid duplicate objects being generated and causing confusion for accessing the data.

Figure 26: Reserve Car Controller.

As we can see in the figure given below the execute method of the controller asks the caller for information that is required to book a car. The caller in this case being the `reservation.java` class which is the view will get this info from the user.

Unlike the ATM example here there is no if else condition determining the use case which needs to be implemented as each use case has a separate file generated for itself. Figure 27 below depicts this scenario.

Figure 27: Execute method of Reserve Car controller.

The Figure 28 below shows the ask method generated for the view. As this class interacts with the user the ask method should prompt the user to enter the required inputs for booking a car or performing any of the available operations. As you can see below the ask method pops up a dialog box asking the user to enter the required input.

Once this step is completed the control goes back to the execute method of the controller.

Figure 28: Ask method of the Reservation class.

In this example there are two controllers which are called by the main controllers which are “getpersonalInfo” and “Makepayment”. These controllers are called after the required information is gathered from the user about the reservations they need to make. These controllers are important for getting personal information as well as the payment details of the reservations being made. Figure 29 on the next page shows the execute method of the controller where the “getpersonalInfo” is called.

ERROR: stackunderflow
OFFENDING COMMAND: ~

STACK: