Team Orange Midterm Report

Anil Acharya, Eddie Davis, and Iker Vazquez

November 6, 2017

Contents

1 Introduction

The goal of this project is to apply object-oriented design and agile development principles to develop a graphical environment for the development and simulation of Petri Nets. A Petri Net (PN) is a mathematical modeling language represented as a directed, bipartite graph. The graphs can be used to model software processes, state machines, formal language semantics, parallel and distributed systems, and many other computer science applications. The PN abstraction provides a higher level than finite state automata, able to model nondeterministic or stochastic processes.

A PN graph consists of two types of nodes, with *places* represented by circles, and the *transitions* between them by rectangles. Places and transitions are connected by directed arrows called *arcs*. An arc connecting a place to a transition is called a *source* arc, and one connecting a transition to a place is known as a *destination*. Places can contain a number of *tokens*, represented by an integer value, $t \geq 0$. An arcs can also have a *weight*, an integer value, $w \geq 1$, that defaults to 1, and determines the number of tokens that move from one place to another between the transition. A transition may have any number of input arcs, and is *firable* when all of the connected places contain the number of tokens specified by the arc weights. When the transition fires, tokens are moved from the destination place to the source place. A PN configuration, i.e., the number of tokens in each place is known as a *marking*, and each PN has an initial one. An ordered set of transition firings is called a *firing sequence*.

The formal definition of a Petri Net, PN with a set of places P of size m, set of T transitions of size n, set of transition functions F representing the arcs, and weight function W, assigning a weight to each arc, is given below. The initial marking is denoted as M_0, and all subsequent markings as $M_1, M_2, ...$ The firing sequence with p firings is denoted by σ.

$$
\begin{aligned}
PN &= (N, M_0) \\
N &= (P, T, F, W) \\
P &= \{p_1, p_2, ..., p_m\} \\
T &= \{t_1, t_2, ..., t_n\} \\
F &\subseteq (P \times T) \cup (T \times P), F \neq \emptyset \\
W &: F \to \{1, 2, 3, ...\} \\
M_0 &: P \to \{1, 2, 3, ...\} \\
\sigma &= (M_0, t_1 M_1, t_2 M_2, ..., t_p M_p)
\end{aligned}
$$

The remainder of this report describes the design and development of the Petri Net simulator. Section 2 covers the team organization, Section 3 the product backlog, i.e., user stories, Section 4 details the object oriented design, Section 5 the pair development summary, Section 6 the overall development process, and finally in Section 7, each author provides his lessons learned thus far.

2 Team organization and buddy rating

The team was organized under the Scrum framework. The roles were designated as follows.

- Product owner: *Shared*
- Scrum master: *Shared*
- Development team: *Everyone*

The project was divided into three primary components:

1. Core Petri Net Modeling (Business Logic): Anil
2. Graphical User Interface (GUI): Iker
3. XML Input and Output Handling (I/O): Eddie

Each member of the development time was primarily responsible for one of the three components, and the writing of the corresponding unit tests. However, whenever one team member was too busy, another would step in to help with the development of a key area. In general, level of collaboration between team members was quite high, and no member in particular emerged as a leader of the overall project.

2.1 Buddy rating

Member	**Rating**
Anil Acharya	0.98
Iker Vazquez	0.98

Table 1: Participation of each member in the project.

3 Product backlog

In this section, are listed the requirements for the completion of the first phase of the project. These requirements, are generated based on the client's specifications for the program and thus they cover all the features for this stage.

As shown in Table 2, all the user stories of the program are focused on providing to the user all the necessary functionalities to edit a PetriNet. Each user story in Table 3 defines a task (identified by its *ID*) and these tasks must to be complete in order to finish the first stage. Table 3 shows each tasks' priority and its time estimation in hours. There are some invisible tasks that are not part of the user stories but which are important for developing the product. The users stories define the functionalities of the program and thus they do not take into account core modules (e.g. the design of the PetriNet and its components).

ID	Entity	Description	Status
T1	User	As a user I want to CREATE a PetriNet	Complete
T2	User	As a user I want to LOAD a PetriNet from an XML file	Complete
T3	User	As a user I want to SAVE a PetriNet to a XML file	Complete
T4	User	As a user I want to EDIT a PetriNet via GUI	Almost done
T5	User	As a user I want to UNDO my actions	Almost done
T6	User	As a user I want to REDO my actions	Almost done
T7	User	As a user I want to COPY component of the PetriNet	Complete
T8	User	As a user I want to PASTE PetriNet already copied	Complete
T9	User	As a user I want to SIMULATE step by step the PetriNet transitions	Complete
T10	User	As a user I want to see the VENDING MACHINE example	Complete
T11	User	As a user I want to see the DINNING PHILOSOPHERS example	Complete
T12	User	As a user I want to see COMMUNICATION PROTOCOL example	Complete
T13	User	As a user I want to see READERS-WRITERS example	Complete

Table 2: User story identification.

4 Object-Oriented design

4.1 Design Overview

The object oriented design was motivated by the class elicitation principle. For example, there are classes for all of the key Petri net components, e.g., `Place`, `Transition`, `SourceArc`, `DestinationArc`, `Tokens`, and `PetriNetState` (an individual marking). Since a Petri net is a mathematical construct, it is a natural fit for an ADT structure.

The project is divided into four packages as described in the following subsections. The `logic.model` package handles all of the core logic for the Petri net abstraction as described above. The `logic.utilities` package is responsible for managing Petri net objects at run time, such as firing transitions and transferring tokens between places, i.e., representing markings, and generating the coverability tree. The `io` package contains the classes responsible for serializing Petri net objects to and from an XML representation. Finally, the `ui` package encapsulates the user interface for visualizing Petri net objects, implemented using the Java Swing framework.

4.2 Logic Modeling

Text describing the Petri Net logic modeling class hierarchy presented in Figure 1.

4.3 Logic Utilities

Text describing the Petri Net logic utilities class hierarchy presented in Figure 2.

ID	Priority	Estimate (h)
T1	1	4
T2	2	6
T3	3	6
T4	4	10
T7	5	6
T8	6	6
T9	7	14
T5	8	4
T6	9	4
T10	10	6
T11	11	6
T12	12	6
T13	13	6

Table 3: Priority of each task and its time estimation in hours.

4.4 XML I/O

The Petri net XML I/O class hierarchy is presented in Figure 3. The overall design is modeled after the *Serialization Proxy* design pattern. The purpose of this design pattern is to decouple the serialization of an object from its implementation. For example, if the application must implement the *Serializable* interface without modifying the original source code. In this case, the I/O package interacts with the classes in the `logic.model` package, while treating it is a black box. In other words, it must serialize the objects while only accessing their public methods.

The core of the I/O package is the abstract class `PetriSerializable`, responsible implementing the `Serializable` interface. It does not implement the required `readObject` and `writeObject` methods directly, but defers them to its concrete subclasses. Each of the subclasses is responsible for serializing a particular Petri net component, and acts as an adapter to each object via object composition, i.e., each serialization object obtains a reference to the component it serializes.

The `PetriNetSerializer` class is the primary class in the hierarchy as it is responsible for serializing or deserializing an entire Petri net object to or from an XML stream. There is a corresponding concrete class for each subcomponent, e.g., `PlaceSerializer`, `TransitionSerializer`, `SourceArcSerializer`, and `DestArcSerializer`.

There are also two utility classes that are essentially implementations of the *Factory Method* and *Singleton* design patterns. The `IOComponentManager` singleton maintains the sets of components needed for a Petri net object currently under construction while the `PetriNetSerializer.readObject` method is running, and constructs new components as necessary. The `XmlFactory` object performs the same role for XML components during execution of the `PetriNetSerializer.writeObject` method.

The XML schema format implemented is a subset of PNML, the Petri Net Markup Language. This enabled reuse of any Petri nets already defined in that format, such as *Producers and Consumers*, *Dining Philosophers*, and *Communication Protocol*. Only the

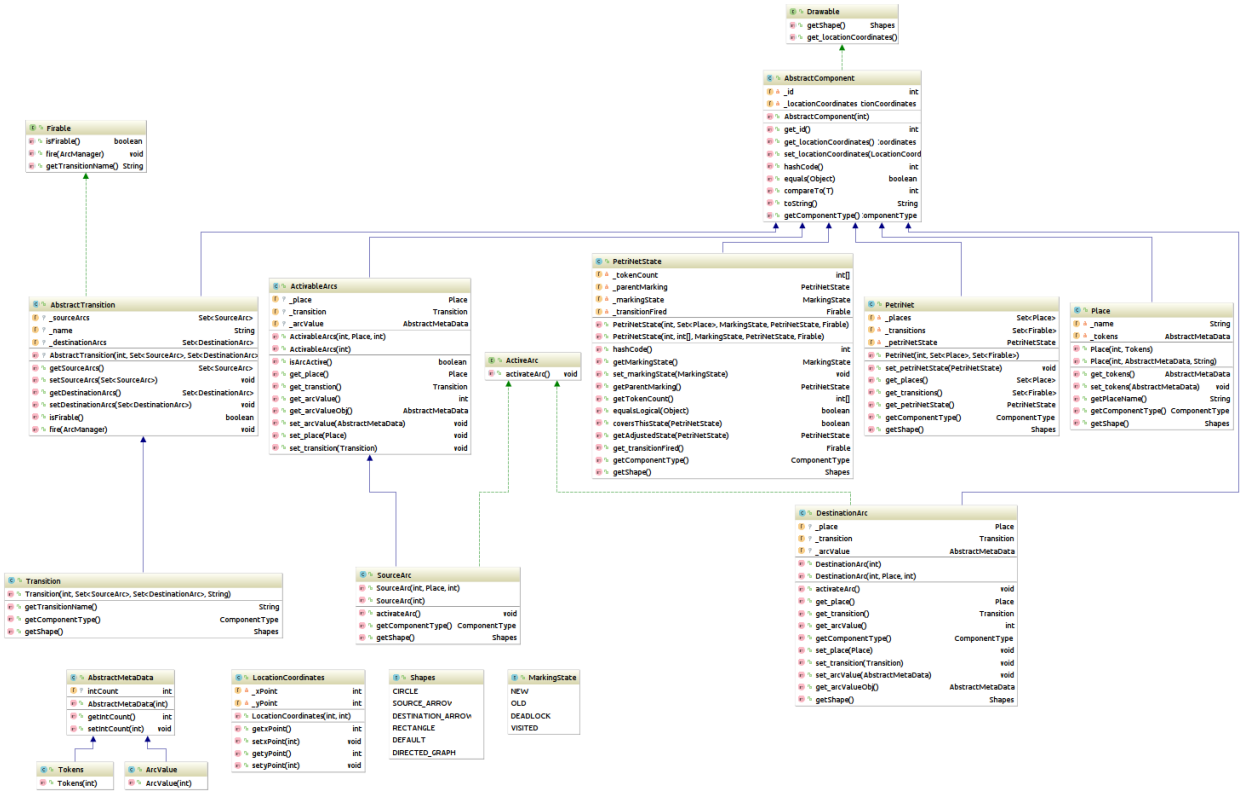

Figure 1: UML class diagram for Petri Net logic modeling package.

portions of the specification needed for this class design were implemented. There are many XML parsers available for Java, however the *java.xml.parsers* package in the standard library was selected for full compatibility and reliability.

When reading an XML stream, the `PetriNetSerializer` collects all place, transition, and arc nodes into sets. It then processes places, arcs, and transitions respectively to accommodate the `logic.model` implementation. The parent-child relationships between components are mirrored in the XML document by writing places, transitions, source arcs, and destinations. New nodes are constructed by the `XmlFactory` as needed and appended to parent nodes.

4.5 Graphical User Interface

The Petri Net graphical user interface (GUI) class hierarchy is presented in Figure 4.

As we know, neither Design By Contract nor Defensive programming is best suited for the class design. It solely depends on matter of time, the application context, class responsibility and above all, the performance issue. So, we exploited both paradigms when we are distributing class responsibility, wherever they are feasible. The three key classes that we pick to justify our conscience on how the class design follows the defined paradigm, will be explained below one by one.

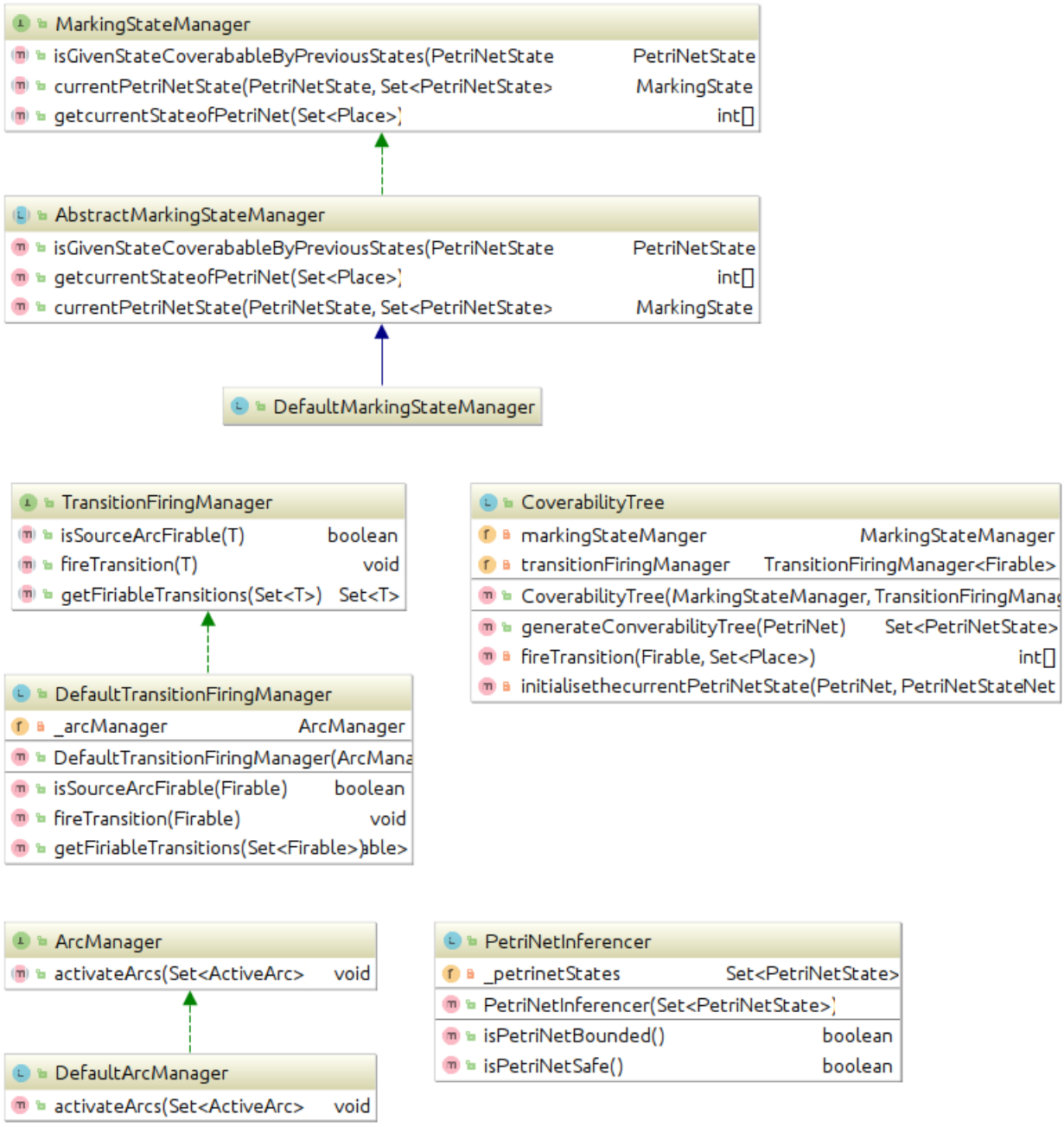

Figure 2: UML class diagram for Petri Net logic utilities package.

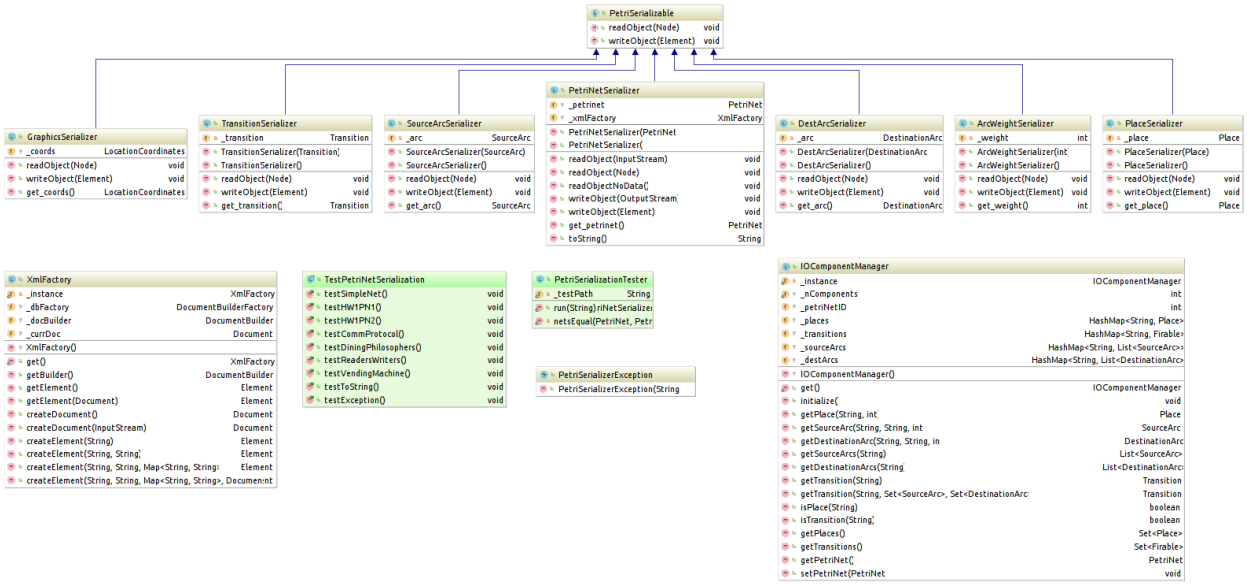

Figure 3: UML class diagram for Petri Net XML I/O package.

4.6 CoverabilityTree class

Class responsability: The main responsibility of this class is to implement the coverability tree drawing algorithm and return collection of all possible marking state along with their respective marking.

How the class follows the ADT: ADT is the specification of the class such that it defines the top level structure of the class. The actual class is just the implementation of the ADT. According to ADT, class must have functions with distinct behaviour and each function must be bounded by set of preconditions, that the client ensures, and the set of postconditions for which the function is obliged. The **CoverabilityTree** has three that has distinct functionality and each of them has set of precondition and postcondition.

public Set<PetriNetState> generateCoverabilityTree(PetriNet petriNet)

Method description: This method generates the coverability tree of the provided PetriNet. The coverability tree is represented as the collection of all possible PetriNet-State.

PreCondition: This method expects the PetriNet instance, provided as a parameter, to be the valid PetriNet. The client must ensures that the PetriNet, he wants to pass to this method, is valid. The PetriNet class has a method named **validate()**, which acts as a precondition check method.

PostCondition: This method returns the coverability tree as a collection of PetriNet-State, such that each entry in the collection will be a distinct PetriNetState which the petri net can be in, by firing a sequence of firable transition. The returned collected is formatted in such a way that client could create a graph structure from it. Each

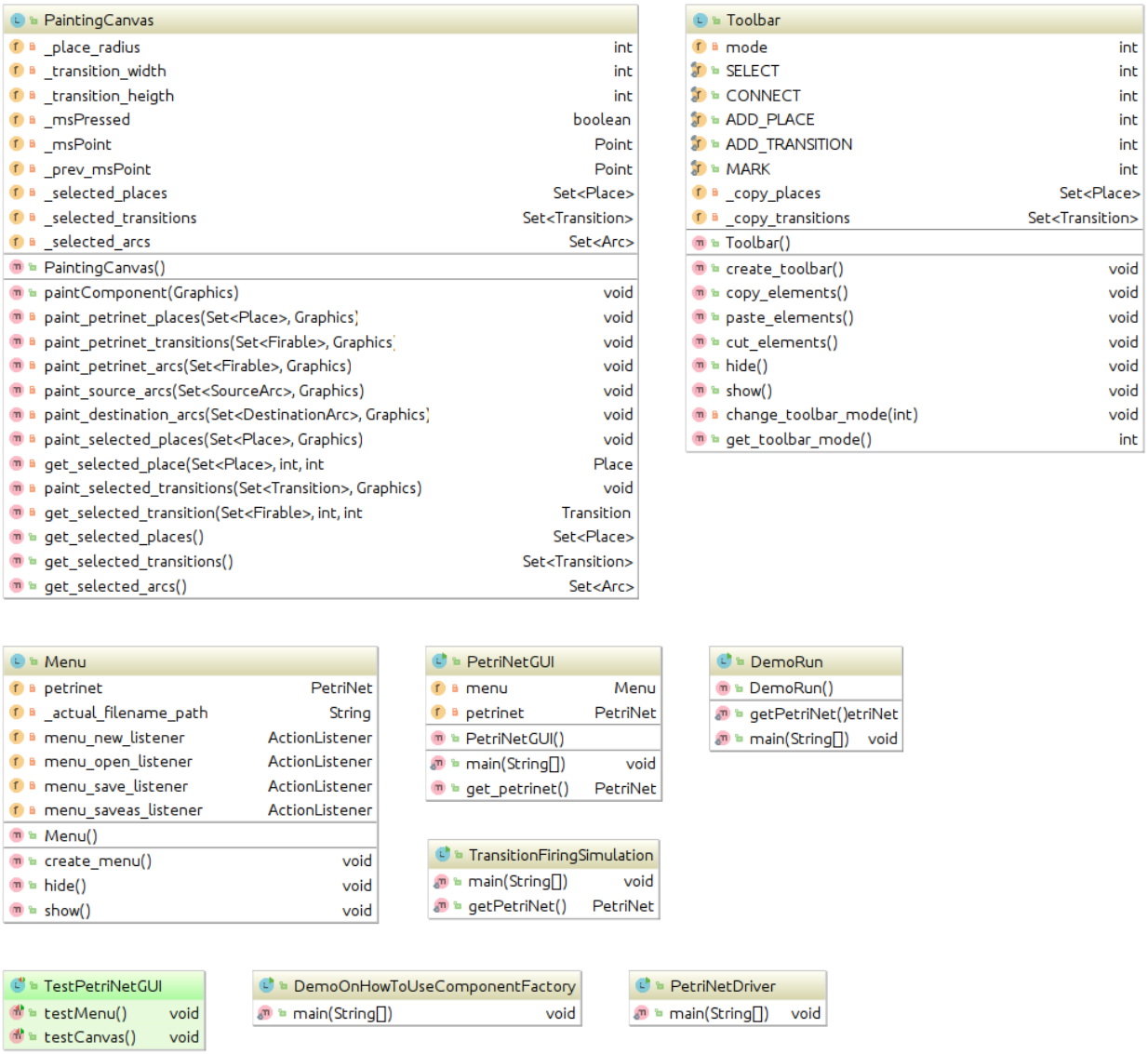

Figure 4: UML class diagram for Petri Net graphical user interface (GUI) package.

PetrinetState instance in the collection, will hold a reference to its parent while the first instance will point to null.

private int[] fireTransition(Firable firable, Set<Place> places)

Method description: This method fire the provided transition and returns the array of newly created token value.

PreCondition: The method expects the firable instance to be firable. The client must check either the instance Firable Interface is firable or not by calling **isFirable()** method.

PostCondition: The array return by the method has length equal to the number of places in the petri net and will contain current token count is each places after the provided transition is fired.

public PetriNet initialisethecurrentPetriNetState(PetriNet petriNet, PetiNetState petriNetState)

Method description: This method initialize the provided PetriNet with the provided PetriNetState.

PreCondition: The provided petriNetState must be valid. PetriNetState class provide a method **validate()**, to test if the PetriNetSate is valid or not. The client must call this method before calling the above method as a precondition check.

PostCondition: The method insures that the provided PetriNet is a valid PetriNet and initialize the token count in each places with the provided token value in petriNetState.

Seeing above we can say the the Coverability tree class is in accordance with the ADT specifications. Each method in this class is design with design by contract approach. When means that, in the documentation, the postcondition and precondition is clearly specified in the documentation and method that provides precondition check is also expose publicly, so that the client could call it to validate the precondition. All three method of the class is in accordance with the command query separation principle. The first method **generateConverabilityTree()** is Query Method, because the method return the coverability tree of the petri net, without altering the state of the PetriNet. This method only process the the current state of the petri net to generate the coverability tree. The second method **fireTransition** is command method despite the fact that the method is returning the integer array. The method fires the provided transition, and update the token value of each affected places. In that sense, the method is updating the state of petriNet. Thus, this method is acting as a command method. The third method is also a command method, because it is also updating the PetriNet state with the provided value of token for each places. The result of this method is change in the state of PetriNet state and hence the method act as a Command method.

4.7 PetriNetInferencer class

Class responsability: The class hold the responsibility of performing various test on the petri net. It functionality to test various properties of the PetriNet.

How the class follows the ADT: This class contains two method, and each of the method has distinct functionality and set of precondition and postcondition, which are explained below.

public boolean isPetriNetBounded() throws InvalidPetriNetStateException

> **Method description:** The method test if the provided coverability tree is bounded or not. The coverability tree is represented as a collection of PetriNetState.
>
> **PreCondition:** The method expect each PetriNetState within the collection to be the valid PetriNetState. PetriNetState class expose a method validate() that act a precondition check method.
>
> **PostCondition:** The method returns false only if the coverability tree contains w in any of the states token.

Seeing above, we can say that PetriNetInferner class is implemented in accordance with ADT specification. This class is design with defensive programming approach. The documentation do not expose the precondition, rather the method itself take care of validating the precondition. The method only expose the postcondition. The first method, while iterating through the collection of PetriNet States, implicitly checks if the PetrinetState instance is valid or not and throws the exception if it is not valid. The second method is also doing the same. Defensive programming is computationally efficient for this class because, if we see the implementation of the two method, we will notice that, the implementation is iterating over each and every possible PetriNetState. So, it is far more efficient to make the single call to check, if the state is valid or not, than to make client of this method, check the state validity for each PetriNetState instance manually as a precondition check.

We have a super class by the name of AbstractComponentManager and 4 base class that extends this superclass. Each base class act as a concrete component manager for each element of PetriNet. For example, we have PlaceManager class that manages creation, edit, remove and search of the place, also we have TransitionManager that manages creation, edit, remove and search of the transition and so on. We formally define AbstractComponentManager as a class designated for handling various operations related to elements of Petrinet. AbstractComponentManager is a generic class, it provides, on the house, various common functionality that is reusable by concrete component manager classes. Functionality like findByComponentId, getLastComponentId, clear are reused by all the base classes. It also expose some abstract methods like getComponent, copyComponent, attachComponentToItsParent, updateComponentMetaData, getComponents. The implementation of these abstract method is dependent upon the element type the base class is supposed to work with. For example, the base class that works with Places, will implement these method, in accordance to Place element, and by the base class that works with DestinationArc element.

Each subclass in the defined inheritance structure implies both model-view as well as type-view structure. First the each base class can reuse the method provided by the base class and if it wants to modify the behaviour of any existing method, it can do so by overriding it. The inheritance structure follows the open close principle and all the base class is being facilitated because of this. In order to explain the implication of type-view model, consider the **ComponentFactory** class. Component factory class is design based on factory design pattern. This class contain basic method that allows basic operation to be done on the petri net element in generic way. Lets consider the simple method called **getComponent()**. If we see the implementation of getComponent() method, we can see, how it has exploited the concept of dynamic binding in it. The type of component return by this method is independent of the component manager type it dealing with at compile time. Java runtime call the right getComponent method on right base class of AbstractComponentManager at the execution time, and return the appropriate component, the client is expecting. This dynamism is achieved because of remarkable feature of object oriented language called **Polymorphism**. Thus, we can visualize the implication of both model-view and type-view concept being used, because of above inheritance behaviour.

5 Summary of pair development

The team had problems to meet due to schedule differences. There was a meeting every week on Fridays afternoon but that was the only time the full team could meet. To perform pair development of this project, the team could not perform a full pair development as defined by the project requirements. Due to this problem the team used Fridays meetings to work on some team development at the same time they were discussing the next milestones of the project.

Along the week and before the meetings, the team was in constant contact via Slack. This helped the team to be organized and perform some pair programming. Conceptual problems, bugs or meetings' misunderstandings were solved using this platform. The team considered this interaction as a pair programming development but did not track the tasks or the hours.

In the last meeting before the deadline, there were some bugs that had to be solved. The team was entirely pair working for the entire meeting to solve those bugs and have the product ready for the demo presentation. This pair programming last for 2 hours which almost satisfied the project requirements of each team member's pair programing hours.

6 Summary of Development Process

The development model selected was Agile Scrum. The team conducted weekly scrum meetings on Friday afternoons, to plan the activities for the next sprint. The development team collaborated daily via the Slack online web tool. Developers also collaborated via email and repository commit messages.

The version control system (VCS) used was a Bitbucket repository with git command tool. The object-oriented programming language selected is Java, and the build system is

Maven. Unit tests were written in JUnit 4, and code coverage was calculated with the Eclipse IDE. The UML class diagrams were generated with the IntelliJ IDE.

The coding standard adopted for this project includes many conventions that are common in Java applications. Variable names should be descriptive, i.e., the code should be self-commenting. Class and variable names follow the camel case convention, with alternating upper- and lowercase characters, to signify new words. Class and interface names all begin with an uppercase character. Class names are always nouns, and interfaces adjectives (e.g., `Firable`, `PetriSerializable`).

Method names should be verbs, query (accessor) method names beginning with *get*, properties with *set*, and the rest assumed to be commands (mutators). Instance variables should be private if the class is closed for inheritance, or protected if the class can be extended (Open-Closed Principle). Instance variable names should begin with an underscore "_", to distinguish them from local method variables, an example of writing for later readability.

Opening braces should be placed on the same line as the method, loop, or conditional definition, to reduce the total lines of code. Blank lines should be used sparingly, to separate methods, and logical blocks of code within methods, much like paragraphs in written prose.

The classes and methods are documented with Javadoc comments. Some preconditions, postconditions, class invariants are documented with Java Modeling Language (JML), embedded within the Javadoc comments. Class headers and method headers should be well commented, but within method comments should be sparse, relying instead on descriptive class, variable, and method names to self-document the code.

As indicated above, the intent of the development process has been to follow the Agile Scrum paradigm. However, the actual implementation path bears some similarity to the waterfall model. The first stage of the process was requirements gathering, following by the initial design, and first pass at implementation. It has been an iterative process as well, because it has been necessary to revisit the requirements, design, or implementation along the way. The development of unit tests along the way could be considered part of the verification phase, and this has also followed a rather iterative process as well.

Overall, the goal has been to produce quality, working code as quickly as possible, by following agile development processes and SOLID design principles. The work has been organized into sprints, emphasizing collaboration, with frequent feedback from the development team (also the customers in this scenario). Though the development has been quite successful, as the user story status in Table 2 indicates, opportunities for improvement remain. Collaboration via Slack and Bitbucket have been instrumental, but there are other tools (Scrum task boards for example), that had been intended for use but proved not to be time effective. Other Confluence tools, such as Kanban boards and their corresponding swimlanes might be helpful as well. In general, more frequent communication would be the most significant improvement.

7 Lessons learned

At the beginning of this project, I had only recently become aware of Petri Nets, while researching related work for a paper on macro dataflow graphs modeling distributed memory applications. In the era of big data, heterogeneous computing platforms, and the apparent

bottleneck of the memory subsystem in multi-core architectures, modeling systems based on the availability of data are becoming increasingly important. To that end, I have learned much more about the applications of Petri nets to complex problems such as *Dining Philosophers* and *Producers-Consumers*.

In development terms, I have learned much more about dividing a large project into multiple components and the challenges of developing those components in a team environment. During much of my professional and academic life, much of my work has taken place in isolation, collaborating mostly at a high level, either having complete control to modify code in an application, or none at all. The need to collaborate at a low level in a tight knit development group has been enlightening.

The way in which the concepts covered during lecture are able to be applied to the class project enable a hands-on learning approach that make the course more effective. One specific lesson learned is the need to be cautious when developing international code. It is important to pay attention to details like number formats, a comma is used to represent the radix in the EU, instead of a decimal point as in the US. A file written with the comma format for a floating point number will fail when read by a system expecting a decimal point instead. Such details highlight the need for comprehensive unit testing, particularly for human-computer interaction.