Project 1, Part 2: ReqKeeper - Requirements Management Software

Part 1 of this assignment laid out the requirements for the requirements manager system that you must create. Part 2 details the design that you must implement. For this part, you must create a complete Java program consisting of multiple source code files and JUnit test cases.

Project 1 Part 2 Process Point Deadline (compiling skeleton): June 21 at 11:45 pm.

Project 1 Part 2 Due Date: June 28 at 11:45 pm.

> ### Note
>
> You will not be working with a partner for Project 1. All work must be strictly your own.

Design

Your program must implement our design, which we describe here. The design consists of eleven different classes, two enumerations, and one interface. We are providing the interface, the graphical user interface front end and a library (in a JAR file) for processing XML files.

> ### Important
>
> The project you push to NCSU GitHub must contain *unmodified* versions of the files that we provide for you.

Top

The design organizes the code into five different packages, listed below. You must implement the first three of those packages.

1. edu.ncsu.csc216.tracker.requirement

   edu.ncsu.csc216.tracker.requirement.RequirementState. Interface that describes behaviors of any concrete RequirementState for the ReqKeeper FSM. (See RequirementState.java) **You cannot change this code in any way.** All concrete State classes must implement RequirementState:
   AcceptedState. Concrete class that represents the *Accepted* state of the ReqKeeper FSM.
   CompletedState. Concrete class that represents the *Completed* state of the ReqKeeper FSM.
   RejectedState. Concrete class that represents the *Rejected* state of the ReqKeeper FSM.
   SubmittedState. Concrete class that represents the *Submitted* state of the ReqKeeper FSM.
   VerifiedState. Concrete class that represents the *Verified* state of the ReqKeeper FSM.
   WorkingState. Concrete class that represents the *Working* state of the ReqKeeper FSM.

   edu.ncsu.csc216.tracker.requirement.Requirement. Concrete class that represents a requirement tracked by the ReqKeeper system. Requirement is the *Context* class of the State design pattern. It maintains the requirement's current state and delegates commands for the current state to handle.
   edu.ncsu.csc216.tracker.requirement.Command. Concrete class that represents a command (assign, reject, and so on) that the user enters from the GUI to be handled by the internal FSM.

2. edu.ncsu.csc216.tracker.requirement.enums
   edu.ncsu.csc216.tracker.requirement.enums.CommandValue. Enumeration that names the commands that the user gives in the GUI for processing by the internal FSM.
   edu.ncsu.csc216.tracker.requirement.enums.Rejection. Enumeration that names the possible reasons for rejection of a requirement.

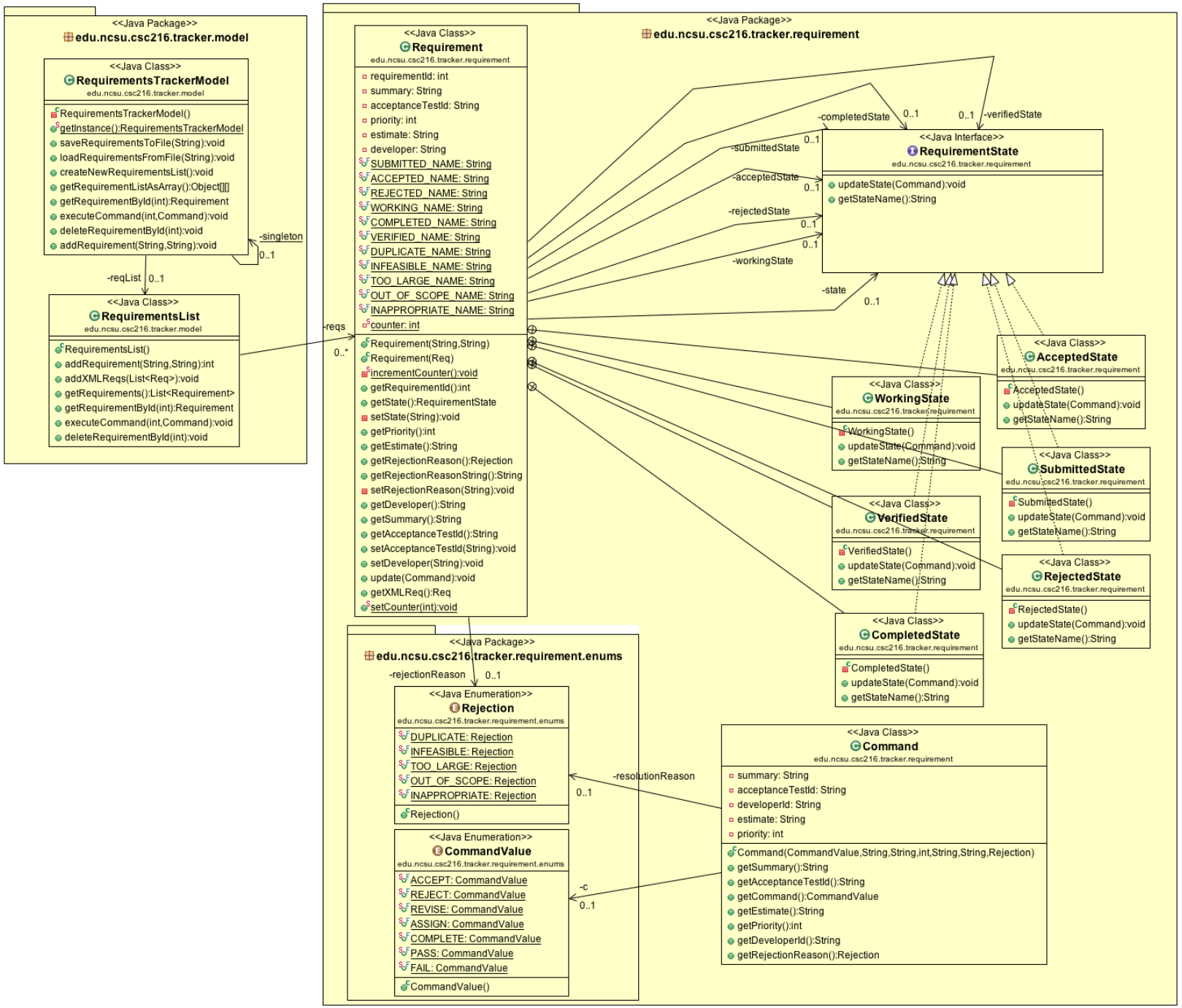
3. edu.ncsu.csc216.tracker.model
   edu.ncsu.csc216.tracker.model.RequirementsList. Concrete class that maintains the current list of Requirements in the ReqKeeper system.
   edu.ncsu.csc216.tracker.model.RequirementsTrackerModel. Concrete class that maintains the RequirementsList and handles CommandValues from the GUI.

4. edu.ncsu.csc216.tracker.gui
   edu.ncsu.csc216.tracker.gui.RequirementsTrackerGUI. The graphical user interface for the project. This is the class that starts execution of the program. **You cannot change this code in any way.**

5. edu.ncsu.csc216.tracker.xml. Given to you as a JAR file (Java archive), RequirementsTrackerXML.jar for processing XML files.

The UML class diagram for the design is shown in the figure below. For simplicity, the diagram omits the user interface class, RequirementsTrackerGUI, and its package, edu.ncsu.csc216.tracker.gui. The JAR file with its classes is not shown either. Except for those omissions, the diagram shows the minimum set of state and behavior required to implement the project. You may add other private methods, private data, and states in your implementation. You can modify the names of private variables and parameters. However, you MUST have the public methods (names, return types, parameter types and order) ***exactly*** as shown below for the teaching staff tests to run.

Top



This UML diagram was programmatically generated by ObjectAid on the instruction staff solution. The only associations this software shows a Top e in which one class has a class member that is the type of another class or enum. Such a member is not listed separately among the data for the first

class. For examples, see the association from Requirement to Rejection and the one from RequirementsModel to RequirementsList.

UML diagram notations

UML uses standard conventions for display of data, methods, and relationships. Many are illustrated in the UML diagram above. Here are some things you should note:

- in front of a class member means private.
+ in front of a class member means public.
Static members (data or methods) are underlined.
Methods that are declared but not defined (abstract methods or those declared in interfaces) are in italics.
The names of abstract classes are in italics.
Dotted arrows with triangular heads point to interfaces from the classes that implement them.
Solid arrows with triangular heads go from concrete classes to their parents.
Solid arrows with simple heads indicate *has-a* relationships (composition). The containing class is at the tail of the arrow and the class that is contained is at the head. The arrow is decorated with the name and access of the member in the containing class. The arrow is also decorated with the "multiplicity" of the relationship, where 0..1 means there is 1 instance of the member in the containing class and 0..* means there are many (usually indicating a collection such as an array or ArrayList).
A circle containing an "X" or cross sits on a the border of a class that contains an inner or nested class, with a line connecting it to the inner class. (For example, see the class RejectedState and its corresponding outer class Requirement.) classes nested within RequirementsTrackerGUI are not shown.

Our UML diagram has some additional graphic notation:

A square (empty or solid) in front of a name means private.
A green circle in front of a name means public.
SF in front of a name means static, final.
Methods embellished with C are constructors.

You can read this UML class diagram and see the names and types of all the classes and class members that you must create. The method signatures in your program ***must*** match the above diagram and provided interfaces ***exactly*** for the teaching staff tests to run.

Your values for class-level constants **must** also match to make our black-box tests pass. The expected values in the Requirement class are:

```
SUBMITTED_NAME = "Submitted";
ACCEPTED_NAME = "Accepted";
REJECTED_NAME = "Rejected";
WORKING_NAME = "Working";
COMPLETED_NAME = "Completed";
VERIFIED_NAME = "Verified";

DUPLICATE_NAME = "Duplicate";
INFEASIBLE_NAME = "Infeasible";
TOO_LARGE_NAME = "Too large";
```

Top

```
OUT_OF_SCOPE_NAME = "Out of Scope";
INAPPROPRIATE_NAME = "Inappropriate";
```

Important: Your code must maintain the same package structure as the UML diagram specifies. All class names, interface name, public constants, and public method signatures must match the UML diagram as well.

edu.ncsu.csc216.tracker.requirement.enums

We use an *enumeration* for the possible commands that can cause transitions in our FSM. We also use an enumeration for the possible rejection reasons. Our textbook defines enumeration as "a type that has only a small number of predefined constant values." Since there are a discrete number of command actions a user can take (essentially all buttons not labeled cancel) and a discrete number of rejection reasons, enumerations are quite appropriate to list those values. Use the following code for the two enumerations, each in their own file. To create an enum file right click and select New > Enum. Then fill in the names in the EXACT order provided below. CheckStyle will want you to comment each constant value. An example of a comment is provided below. You will be expected to comment the rest.

```
/** Comment here too */
public enum CommandValue {

/** Accepted requirement: Submitted -> Accepted*/
ACCEPT,
REJECT,
REVISE,
ASSIGN,
COMPLETE,
PASS,
FAIL

}
```

```
public enum Rejection { DUPLICATE, INFEASIBLE, TOO_LARGE, OUT_OF_SCOPE, INAPPROPRIATE }
```

To access a value in the enumeration, use the enumeration name followed by the value (for example, CommandValue.ACCEPT). Enumerated types are essentially a name given to an integer value. Therefore, you can use primitive comparison operators (== and !=) to compare enumerated type variables (of the same enumerated type - don't try to compare a CommandValue with a Rejection). Enumerated types can also be the type of a variable. For example, Requirement's rejectionReason field is of type Rejection. See pp. 1133-1134 in the Reges and Stepp textbook for more details on enumerated types.

Create the enums first. They are used in Command and other classes.

edu.ncsu.csc216.tracker.requirement

The requirement package holds the *context* (Requirement), *abstract state* (the interface RequirementState), and the *concrete states* (all the *State classes) of the State design pattern for the tracked requirement FSM.

Requirement represents a requirement tracked by our system. A requirement knows its requirementID, state (updated from Commands propag it from the UI), summary, estimate, developer, acceptanceTestId, and its Rejection reason. The instruction staff solution also defined Requiremer Top ix private constants corresponding to each possible state: submittedState, acceptedState, rejectedState, workingState, completedState, and verifiedState.

The six concrete state classes implement RequirementState. They should be inner classes of Requirement. Every concrete RequirementState must support two behaviors: 1) updateState when given a Command and 2) getStateName.

The Command class creates objects that encapsulate user actions (or transitions) that cause the state of a Requirement to update.

Table 1: Requirements State Transition Table

	Submitted		Accepted		Rejected	Working		Completed				Verified	
Command	ACCEPT	REJECT	ASSIGN	REJECT	REVISE	COMPLETE	REJECT	FAIL	PASS	ASSIGN	REJECT	ASSIGN	REJECT
Requirement State	Accepted	Rejected	Working	Rejected	Submitted	Completed	Rejected	Working	Verified	Working	Rejected	Working	Rejected
summary					Y								
acceptanceTestId					Y								
estimate	Y	null		null			null				null		null
priority	Y	null		null			null				null		null
developer		null	Y	null			null			Y	null	Y	null
rejectionReason		Y		Y			Y				Y		Y

edu.ncsu.csc216.tracker.tracker

The tracker package contains the two classes that manage all tracked requirements during program execution. A RequirementsList maintains a List of Requirements. RequirementsList supports the following list operations:

add a Requirement to the list
remove a Requirement from the list
search for a Requirement in the list
update a Requirement in the list through execution of a Command
return the entire list or sublists of itself (for example, the RequirementsList can return a List of Requirements filtered by owner)

RequirementsTrackerModel implements the Singleton design pattern. This means that only one instance of the RequirementsTrackerModel can ever be created. The Singleton pattern ensures that all parts of the RequirementsTrackerGUI are interacting with the *same* RequirementsTrackerModel at all times.

RequirementsTrackerModel works with the XML files that contain the Requirements in a file when the application is not in use. Therefore, RequirementsTrackerModel works closely with the RequirementReader and RequirementWriter classes in the RequirementTrackerXML.jar. Separation of RequirementsList from RequirementsTrackerModel means each class can have a very specific abstraction: RequirementsList maintains the List of Requirements and RequirementsTrackerModel controls the creation and modification of (potentially many) RequirementsLists.

Top

Implementation

Every Java file for this assignment must be in the package edu.ncsu.csc216.tracker.*, where * is a sub-package for different areas of the system. The four sub-packages that you must create in your Eclipse project are requirement, requirement.enums, model, and gui.

Follow these steps and strategies as you implement your solution:

Compile a skeleton. The class diagram provides a full skeleton of what the implemented ReqKeeper program should look like. You should start by creating an Eclipse project named **Project1.** Then set up the RequirementTrackerXML.jar library (see below), packages, copy in provided code, and create the skeleton of the classes you will implement. If a method has a return type, put in a place holder (for example, return 0; or return null;) so that your code will compile. Push to GitHub and make sure that your Jenkins job shows a yellow ball. A yellow ball on Jenkins means that 1) your code compiles and 2) that the teaching staff tests compile against your code, which means that you have the correct code skeleton for the design.

Compiling Skeleton

A compiling skeleton is due at least one week before the final project deadline to earn the associated process points.

Comment your code. Javadoc your classes and methods. When writing Javadoc for methods, think about the inputs, outputs, preconditions, and post conditions.

Fully Commented Classes

Fully commented classes and methods on at least a skeleton program are due at least one week before the final project deadline to earn the associated process points.

Commit with meaningful messages.

Meaningful Commit Messages

The quality of your commit messages for the entire project history will be evaluated for meaning and professionalism as part of the process points.

Run your CheckStyle and PMD tools locally and make sure that all your Javadoc is correct. Make sure the tools are configured correctly (see configuration instructions) and set up the tools to run with each build so you are alerted to style notifications shortly after creating them.
Practice test-driven development. Start by writing your test cases. This will help you think about how a client would use your class (and wil ou see how that class might be used by other parts of your program). Then write the code to pass your tests.

Top

Setting up RequirementTrackerXML.jar for use

RequirementTrackerXML.jar is a library provided for you to marshal and unmarshal XML files. Parts of the RequirementTrackerXML library were created automatically by using the req_tracker.xsd schema as input to the JAXB library, which is part of the Java API. Each class in the RequirementTrackerXML library (Req and ReqList) directly corresponds to a tag in the schema. The edu.ncsu.csc216.tracker.xml.ReqList is different from the edu.ncsu.csc216.tracker.tracker.RequirementsList class you must write. You will not need to access edu.ncsu.csc216.tracker.xml.ReqList directly in the code that you write.

The Req class in the RequirementTrackerXML library is used by the Requirement class in edu.ncsu.csc216.tracker.requirement. Your Requirement class should create a new Requirement from a Req object. When saving, the Requirement should generate a Req object for writing to a file. The methods setState(String), setRejection(String), and getRejectionString() help translate between the RequirementState and Rejection objects and their string equivalents (which are all stored as constants in Requirement).

RequirementsTrackerModel interacts with the RequirementReader and RequirementWriter classes. RequirementReader unmarshals XML files and creates Req objects, which can then be used to create Requirements. RequirementWriter marshals Req objects into valid XML and writes them to a file.

For examples of creating and working with Req objects and for examples of using RequirementWriter and RequirementReader, see the provided tests for the RequirementTrackerXML library.

Follow these steps to add RequirementTrackerXML.jar to your Project1 build path:

1. Open or create your Project1 in Eclipse.
2. Save RequirementTrackerXML.jar to a directory named lib/ in your RequirementTracker project. If you create the directory and add the file through the file system, refresh your project in Eclipse so the changes are reflected in your workspace.
3. Right click the project and select **Build Path > Configure Build Path**. A dialog box opens.
4. In the resulting dialog box, open the **Libraries** tab.
5. Click **Add JARs**.
6. Select RequirementTackerXML.jar in the lib/ directory, then click **OK**

Implementing the State design pattern

Start your coding by implementing the Requirement and a Requirement's underlying state pattern. Create the concrete *State classes as private inner classes of Requirement. That enables those RequirementStates to access the private fields of Requirements, like the final instances of each RequirementState (for example, submittedState). Additionally, the concrete RequirementStates are associated only with Requirements. As private inner classes, these concrete RequirementStates are encapsulated from direct instantiation and use by other classes.

The RequirementState interface *cannot* be an inner interface of Requirement. Since Requirement can return a RequirementState object, RequirementState must be visible so that other classes (such as the teaching staff test classes) can access it.

User actions are encapsulated in a Command object. Implement Command first since it is used in several places throughout the project and will help with testing the FSM.

The following conditions result in an IllegalArgumentException when constructing a Command object:

Top

A Command with a null CommandValue parameter.

A Command with a CommandValue of ACCEPT and a null estimate, empty string estimate, or priority that is not between 1 and 3, inclusive.
A Command with a CommandValue of REJECT and a null Rejection.
A Command with a CommandValue of ASSIGN and either a null developer id or an empty string for the developer id.
A Command with a CommandValue of REVISE and a null summary, empty string summary, a null acceptance test, or an empty string acceptance test.

If an invalid Command is passed to a state (such as an ASSIGN command is given to the SubmittedState), an UnsupportedOperationException should be thrown by the concrete state. If this exception is thrown, it should be allowed to propagate through the call chain all the way back to the GUI. The GUI will handle this exception by presenting the user with an error message. If this exception is thrown, no internal state of the Requirement should change (for example, no fields or Requirement should be modified)!

RequirementsList

After you have completed defining Requirement and its state pattern, start on RequirementsList. RequirementsList maintains a List of Requirements. You can use either a LinkedList or ArrayList from the Java Collections Framework.

When creating a new RequirementsList, reset the Requirement's counter to 0. When creating a RequirementsList from the contents of an XML file, set the Requirement's counter to the maxId in the list of XML requirements plus 1.

When working with methods that receive a requirementId parameter, there is no need to error check or throw an exception if the requirementId does not exist in the list. For getRequirementById, return null. For all other methods, do not change the internal state of the list.

RequirementsTrackerModel

RequirementsTrackerModel maintains the current RequirementsList and handles activity around loading, saving, and creating new RequirementsLists. For this, RequirementsTrackerModel will use RequirementReader and RequirementWriter from the RequirementTrackerXML library. If a RequirementIOException is thrown, RequirementsTrackerModel should catch it and throw a new IllegalArgumentException.

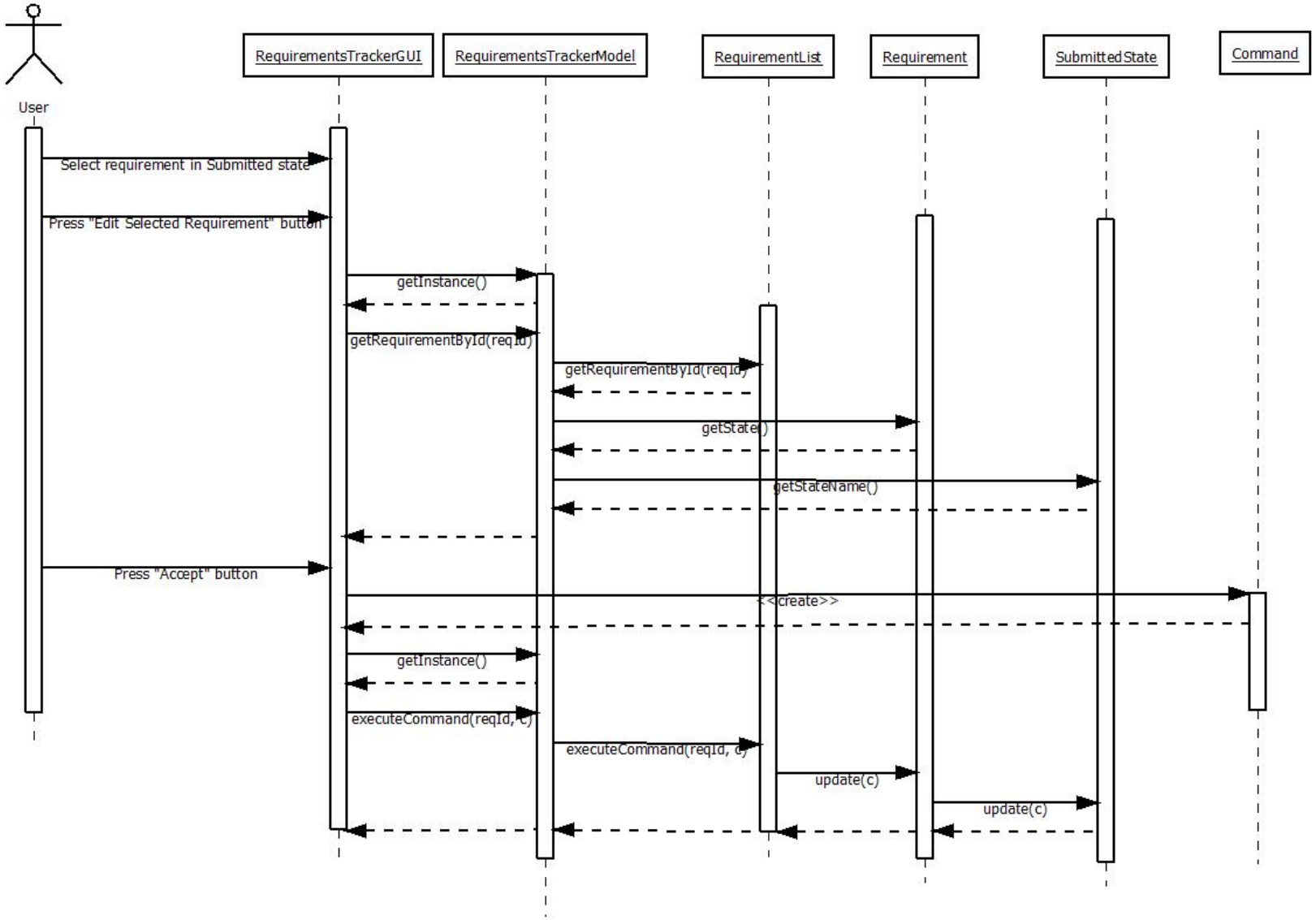
RequirementsTrackerModel also provides information to the GUI through methods like getRequirementsListAsArray(), getRequirementsListByOwnerAsArray(String), and getRequirementById(). The first two of these methods return a 2D Object array that is used to populate the RequirementTableModel (inner class of the RequirementsTrackerGUI) with information. The 2D Object array stores [rows][columns]. The array should have 1 row for every Requirement that you need to return. There should be 3 columns:

Index 0. Requirement's id number
Index 1. Requirement's state name
Index 2. Requirement's summary

Overall flow of control

The sequence diagram shown below models the flow of Use Case 2, Subflow 3 to Use Case 3 and back to Use Case 2. The assumption is that a RequirementsList has already been created or loaded and populated with at least one Requirement in the SubmittedState. The flow of control in the sequence diagram is similar for other functionality.

Top



Testing

For Part 2 of this project, you must do white box testing via jUnit AND report the results of running your black box tests from Part 1.

Test Files

The RequirementsTrackerXML.jar contains many different XML files use for testing. They are located under the test_files directory of the RequirementTrackerXML.jar. Additionally, you can download the 12 test XML requirement files for testing RequirementReader.

Top

Black box testing and submission

Use the provided black box test plan template to describe your tests. Each test must be repeatable and specific; all input and expected results values must be concrete. All inputs required to complete the test must be specified either in the document or the associated test file must be submitted. Remember to provide instructions for how a tester would set up, start, and run the application for testing (what class from your design contains the main method that starts your program? What are the command line arguments, if any? What do the input files contain?). The instructions should be described at a level where anyone using Eclipse could run your tests.

Follow these steps to complete submission of your black box tests:

1. Run your black box tests on your code and report the results in the Actual Results column of your BBTP document.
2. Save the document as a pdf named **BBTP_P1P2.pdf**.
3. Create a folder named **bbtp** at the top level in your project and copy the pdf to that folder.
4. Push the folder and contents to your GitHub repository.

White box testing

Your JUnit tests should follow the same package structure as the classes that they test. You need to create JUnit tests for *all* of the concrete classes that you create. At a minimum, you must exercise every method in your solution at least one by your JUnit tests. You will likely cover the simple getters, setters, and constructors as part of testing more complex functionality in your system. Start by testing all methods that are not simple getters, simple setters, or *simple* constructors for all of the classes that you must write and check that you're covering all methods. If you're not, write tests to exercise unexcecuted methods. You can test the common functionality of an abstract class through a concrete instance of its child.

When testing void methods, you will need to call other methods that do return something to verify that the call made to the void method had the intended effects. For example, if you add a requirement to the system, you can then return the RequirementsList and check that the added requirement is in the list.

For each method and constructor that throws exceptions, test to make sure that the method does what it is supposed to do and throws an exception when it should.

Testing the State pattern

We highly recommend that you implement the concrete RequirementState classes as private inner classes of Requirement. Since the concrete RequirementStates are private inner classes, you will not be able to test them directly. Instead, you will need to update a particular Requirement with a Command that will cause a change of state (or not) and then check the Requirement's state and other fields to ensure the values were updated correctly.

How much do you need to test?

At a minimum, you must exercise every method in your solution at least one by your JUnit tests. You will likely cover the simple getters, setters, and constructors as part of testing more complex functionality in your system.

At a minimum, you must exercise at least 80% of the statements/lines in all non-GUI classes. You will likely cover the simple getters, setters, and constructors as part of testing more complex functionality in your system. You must have 95% method coverage to achieve a green ball on Jenkins (assuming everything else is correct).

Top

We recommend that you try to achieve 100% condition coverage (where every conditional predicate is executed on both the true and false paths for all valid paths in a method). We also recommend that you test your state pattern to consider every possible transition (the teaching staff tests cover all possible transitions).

Deployment

For this class, deployment means submitting your work for grading. Submitting means pushing your project to NCSU GitHub.

Before considering your work complete, make sure:

1. Your program satisfies the style guidelines.
2. Your program behaves as specified in this document. You should test your code thoroughly. Be sure to know what messages should be displayed for each major scenario.
3. Your program satisfies the gradesheet.
4. You generate javadoc documentation on the most recent versions of your project files.
5. You push any updated bbtp, doc, and test-files folders to GitHub.

Deadline

The electronic submission deadline is precise. Do not be late. You should count on last minute failures (your failures, ISP failures, or NCSU failures). Push early and push often!

◀ Project 1, Part 1: Scrum Backlog

Top