Project 1, Part 1: Scrum Backlog ▶

Project 1, Part 2: Scrum Backlog

Part 1 of this assignment laid out the requirements for the Scrum Backlog application that you must create. Part 2 details the design that you must implement. For this part, you must create a complete Java program consisting at multiple source code files and JUnit test cases. You must electronically submit your files to NCSU GitHub by the due date.

Deliverables and Deadlines

Process Points 1 Deadline: Friday, February 22 at 11:45PM

Process Points 2 Deadline: Friday, March 1 at 11:45PM

Part 2 Due Date: Friday, March 8 at 11:45PM

Late Deadline: Sunday, March 10 at 11:45PM

Note

Project 1 Part 2 MUST be completed individually.

Set Up Your Work Environment

Before you think about any code, prepare your work environment to be compatible with the teaching staff automated grading system. The teaching staff relies on NCSU GitHub and Jenkins for grading your work. Follow these steps:

1. Create an Eclipse project named Project1: Right click in the package explorer, select New >

Java Project. Name the project Project1. (The project naming is case sensitive. Be sure that the project name begins with a capital P.)

2. If you have not already done so, clone your remote NCSU GitHub repository corresponding to this project. Note that each major project will have its own repository.

Requirements

The requirements for this project are described in Project 1 Part 1 in eight different use cases:

- UC1: Task XML File Interactions
- UC2: Task List
- UC3: Backlog State
- UC4: Owned State
- UC5: Processing State
- UC6: Verifying State
- UC7: Done State
- UC8: Rejected State

Design

Your program must implement our design, which we describe here. The design consists of twenty-one different classes (10 will be provided in the GUI), two enumerations, and one interface. We are providing the interface and the graphical user interface front end. Your program must use the TaskXML.jar library for processing XML files, which we provide also.

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

edu.ncsu.csc216.backlog.model.*

edu.ncsu.csc216.backlog.model.command. Sub-package of edu.ncsu.csc216.backlog.model containing the object representation of a command that a user would make in the Scrum Backlog system that might initiate a state change. This is an example of the Command Pattern.

Command. Encapsulates the information about a user command that would lead to a transition. Contains one inner enumeration, which will be provided for you in the Implementation section, below.

CommandValue. An enumeration contained in the Command class. Represents one of the six possible commands that a user can make for the Scrum Backlog FSM.

edu.ncsu.csc216.backlog.model.task. Sub-package of edu.ncsu.csc216.backlog.model containing the State Pattern implementation of the ScrumBacklog FSM.

- Note. Concrete class representing a note's contents and the author.
- TaskItem. Concrete class that represents a task item tracked in the Scrum Backlog system. A TaskItem keeps track of all task information including the current state. TaskItem is the *Context* class of the State Design pattern. The state is updated when a Command encapsulating a transition is given to the TaskItem. TaskItem encapsulates the the TaskItemState interface, six concrete *State classes and one enumeration:
  - TaskItemState interface. Interface that describes behavior of concrete TaskItemState classes for the Scrum Backlog FSM. **You cannot change this code in any way.** You MUST copy this interface code into the TaskItem class as an inner interface. All concrete *State classes must implement TaskItemState:
  - BacklogState: Concrete class that represents the *backlog* state of the Scrum Backlog FSM.
  - OwnedState: Concrete class that represents the *owned* state of the Scrum Backlog FSM.
  - ProcessingState: Concrete class that represents the *processing* state of the Scrum Backlog FSM.
  - VerifyingState: Concrete class that represents the *verifying* state of the Scrum Backlog FSM.
  - DoneState: Concrete class that represents the *done* state of the Scrum Backlog FSM.
  - RejectedState: Concrete class that represents the *canceled* state of the Scrum Backlog FSM.
  - Type: An enumeration contained in the TaskItem class. Represents the four possible categories of tasks.

edu.ncsu.csc216.backlog.model.scrum_backlog. Sub-package of edu.ncsu.csc216.backlog.model containing the list of TaskItems and the overarching ScrumBacklogModel functionality.

- TaskItemList. Concrete class that maintains a current list of TaskItems in the Scrum Backlog system.
- ScrumBacklogModel. Concrete class that maintains the TaskItemList and handles Commands from the GUI. ScrumBacklogModel implements the Singleton Design Pattern.

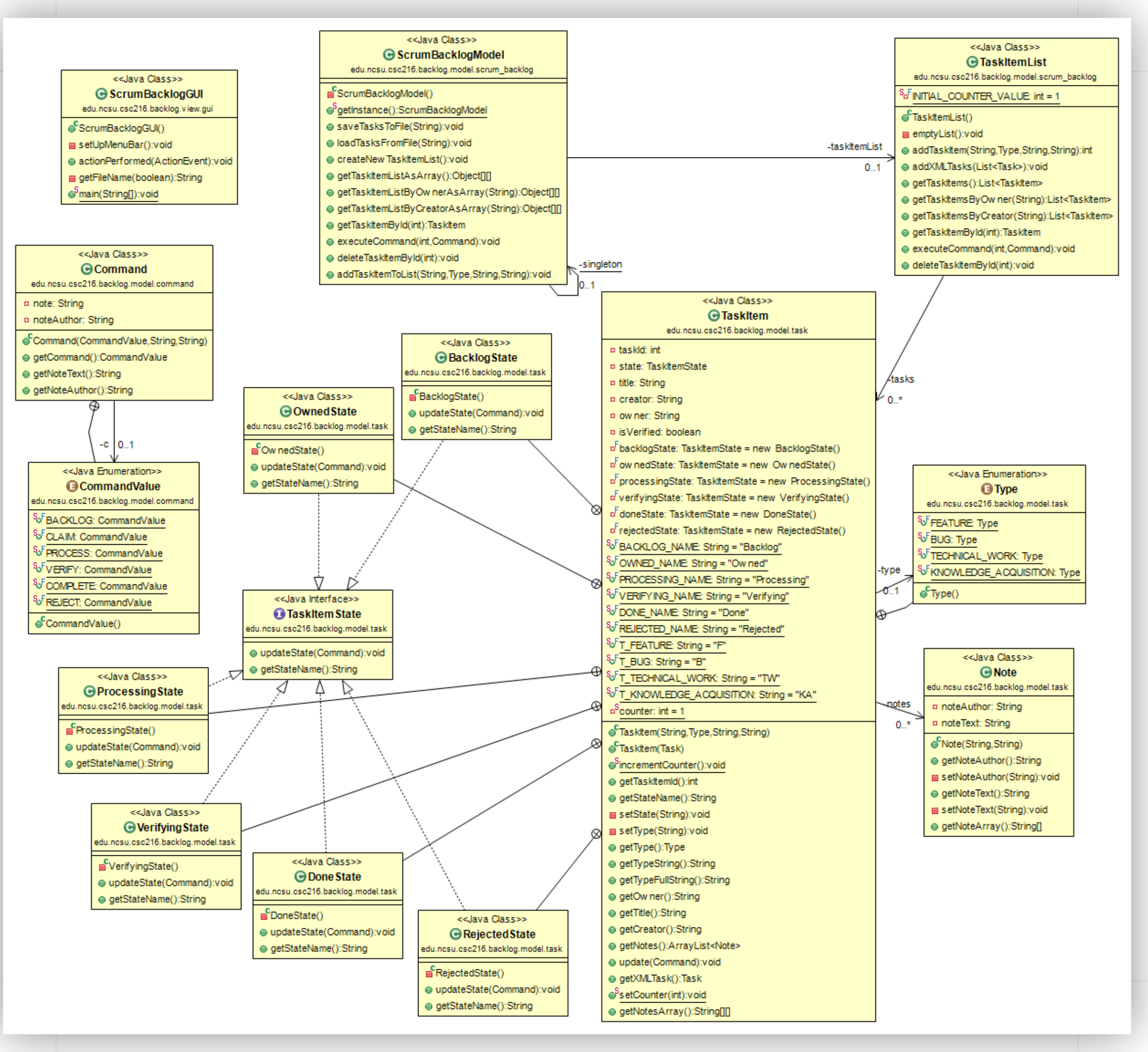
edu.ncsu.csc216.backlog.view.*

edu.ncsu.csc216.backlog.view.gui. Sub-package of edu.ncsu.csc216.backlog.view containing the view-controller elements of the Scrum Backlog Model-View-Controller pattern.

- ScrumBacklogGUI class. 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.**

UML Diagram

The UML class diagram for the design is shown in the figure below. The fields and methods represented by green circles (public) or yellow diamonds (protected) represent the minimum set of visible state and behavior required to implement the project. You may add other methods and attributes/state in your implementation, but all must be private. You can modify the names of private variables and parameters. *However, you MUST have the public and protected methods (names, return types, parameter types, and order) exactly as listed below for the teaching staff tests to run.*



<<Java Class>> ScrumBacklogGUI edu.ncsu.csc216.backlog.view.gui
ScrumBacklogGUI()
setUpMenuBar():void
actionPerformed(ActionEvent):void
getFileName(boolean):String
main(String[]):void
<<Java Class>> ScrumBacklogModel edu.ncsu.csc216.backlog.model.scrum_backlog
ScrumBacklogModel()
getInstance():ScrumBacklogModel
saveTasksToFile(String):void
loadTasksFromFile(String):void
createNewTaskItemList():void
getTaskItemListAsArray():Object[][]
getTaskItemListByOwnerAsArray(String):Object[][]
getTaskItemListByCreatorAsArray(String):Object[][]
getTaskItemById(int):TaskItem
executeCommand(int,Command):void
deleteTaskItemById(int):void
addTaskItemToList(String,Type,String,String):void
-singleton
0..1
-taskItemList
0..1
<<Java Class>> TaskItemList edu.ncsu.csc216.backlog.model.scrum_backlog
INITIAL_COUNTER_VALUE: int = 1
TaskItemList()
emptyList():void
addTaskItem(String,Type,String,String):int
addXMLTasks(List<Task>):void
getTaskItems():List<TaskItem>
getTaskItemsByOwner(String):List<TaskItem>
getTaskItemsByCreator(String):List<TaskItem>
getTaskItemById(int):TaskItem
executeCommand(int,Command):void
deleteTaskItemById(int):void
-tasks
0..*
<<Java Class>> Command edu.ncsu.csc216.backlog.model.command
note: String
noteAuthor: String
Command(CommandValue,String,String)
getCommand():CommandValue
getNoteText():String
getNoteAuthor():String
-c
0..1
<<Java Enumeration>> CommandValue edu.ncsu.csc216.backlog.model.command
BACKLOG: CommandValue
CLAIM: CommandValue
PROCESS: CommandValue
VERIFY: CommandValue
COMPLETE: CommandValue
REJECT: CommandValue
CommandValue()
<<Java Class>> BacklogState edu.ncsu.csc216.backlog.model.task
BacklogState()
updateState(Command):void
getStateName():String
<<Java Class>> OwnedState edu.ncsu.csc216.backlog.model.task
OwnedState()
updateState(Command):void
getStateName():String
<<Java Interface>> TaskItemState edu.ncsu.csc216.backlog.model.task
updateState(Command):void
getStateName():String
<<Java Class>> ProcessingState edu.ncsu.csc216.backlog.model.task
ProcessingState()
updateState(Command):void
getStateName():String
<<Java Class>> VerifyingState edu.ncsu.csc216.backlog.model.task
VerifyingState()
updateState(Command):void
getStateName():String
<<Java Class>> DoneState edu.ncsu.csc216.backlog.model.task
DoneState()
updateState(Command):void
getStateName():String
<<Java Class>> RejectedState edu.ncsu.csc216.backlog.model.task
RejectedState()
updateState(Command):void
getStateName():String
<<Java Class>> TaskItem edu.ncsu.csc216.backlog.model.task
taskId: int
state: TaskItemState
title: String
creator: String
owner: String
isVerified: boolean
backlogState: TaskItemState = new BacklogState()
ownedState: TaskItemState = new OwnedState()
processingState: TaskItemState = new ProcessingState()
verifyingState: TaskItemState = new VerifyingState()
doneState: TaskItemState = new DoneState()
rejectedState: TaskItemState = new RejectedState()
BACKLOG_NAME: String = "Backlog"
OWNED_NAME: String = "Owned"
PROCESSING_NAME: String = "Processing"
VERIFYING_NAME: String = "Verifying"
DONE_NAME: String = "Done"
REJECTED_NAME: String = "Rejected"
T_FEATURE: String = "F"
T_BUG: String = "B"
T_TECHNICAL_WORK: String = "TW"
T_KNOWLEDGE_ACQUISITION: String = "KA"
counter: int = 1
TaskItem(String,Type,String,String)
TaskItem(Task)
incrementCounter():void
getTaskItemId():int
getStateName():String
setState(String):void
setType(String):void
getType():Type
getTypeString():String
getTypeFullString():String
getOwner():String
getTitle():String
getCreator():String
getNotes():ArrayList<Note>
update(Command):void
getXMLTask():Task
setCounter(int):void
getNotesArray():String[][]
-type
0..1
<<Java Enumeration>> Type edu.ncsu.csc216.backlog.model.task
FEATURE: Type
BUG: Type
TECHNICAL_WORK: Type
KNOWLEDGE_ACQUISITION: Type
Type()
-notes
0..*
<<Java Class>> Note edu.ncsu.csc216.backlog.model.task
noteAuthor: String
noteText: String
Note(String,String)
getNoteAuthor():String
setNoteAuthor(String):void
getNoteText():String
setNoteText(String):void
getNoteArray():String[]

Figure 1: Scrum Backlog UML class diagram

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:

- - or a red square in front of a class member means private.
- + or a green circle in front of a class member means public.

# or a yellow triangle in front of a class member means protected.
Static members (data or methods) are underlined.
Methods that are declared but not defined (abstract methods or those declared in interfaces) are in italics, unless they are in an interface.
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. (See taskItemList on the arrow from ScrumBacklogModel to TAskItemList.) A multiplicity of 0..* means there are many, usually indicating a collection such as an array or collection class. (See tasks on the arrow from TaskItemList to TaskItem.) If the field in teh association should be static, the field name is underlined. (See the singleton field on ScrumBacklogModel.
A circle containing an "X" or cross sits on a the border of a class that contains an inner (nested) class, with a line connecting it to the inner class. (See the class BacklogState and its corresponding outer class TaskItem. See also the enumerations.)

Our UML diagram has some additional graphic notation:

A red square (empty or solid) in front of a name means private. Solid squares are methods and empty squares are data members. (See TaskItem.taskId.)
A green circle (empty or solid) in front of a name means public. (See TaskItemList.addTaskItem().)
A yellow diamond (empty or solid) in front of a name means protected.
SF in front of a name means static, final. (See TaskItem.BACKLOG_NAME.)
Methods embellished with C are constructors. (See TaskItem.TaskItem().) Private methods embellished with a C are constructors in private inner classes. (See ScrumBacklogModel.ScrumBacklogModel().) Note that while all classes require a constructor, not all constructors require implementation. In some cases the default constructor is sufficient.

Access Modifiers

Note: You can modify the names of private variables, parameters, and methods. However, you MUST have the non-private data and methods (names, return types, parameter types and order) exactly as shown for the teaching staff tests to run. YOU MAY NOT ADD ANY ADDITIONAL PUBLIC OR PROTECTED CLASSES, METHODS, OR STATES.

Implementation Process

Professional software development is more than building working software; the process of building software is important because that can support teams of developers building complex systems efficiently. We expect you to follow good software engineering practices and processes as you create your software to help you become a better software developer and to help with understanding your progress on the project.

Project-wide Process Points

There are certain processes that you should follow throughout project development.

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***.

Process Points 1 Milestone

Process Points 1 Milestone gets you started with the project and ensures that you meet the design and that you start commenting your code.

Compile a skeleton. The class diagram provides a full skeleton of what the implemented Scrum Backlog program should look like. Start by creating an Eclipse project named **Project1**. Copy in provided code and create the skeletons of the classes you will implement, **including the test classes**. Ensure that the packages, class names, and method signatures match the class diagram *exactly!* 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, 2) that the teaching staff tests compile against your code, which means that you have the correct code skeleton for the design, and 3) you have a skeleton of your test code in the `test\` folder..

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 two weeks before*** the final project deadline to earn the associated 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 will help you see how that class might be used by other parts of your program). Then write the code to pass your tests.

Process Points 2 Milestone

Process Points 2 Milestone keeps you moving on the project. By the Process Points 2 deadline you should have implemented AND tested all of your classes with at least 80% statement coverage. By doing these things, the teaching staff unit tests will run on your code and provide additional feedback.

You will be assessed on 1) revealing the teaching staff tests and 2) writing high quality tests.

Implementation

We suggest you implement Project 2 Part 2 one piece at a time. The details below provide the suggested implementation order and details about the implementation.

Setting up TaskXML.jar for Use

TaskXML.jar library is a library provided for you to marshal and unmarshal XML files. Parts of the TaskXML library were created automatically by using the task.xsd schema as input to the JAXB library, which is part of the Java API. The classes in the TaskXML library (Task, TaskList, NoteItem, and NoteList) all directly correspond to a tag in the schema. Note that edu.ncsu.csc216.task.xml.TaskList is different from the edu.ncsu.csc216.backlog.model.scrum_backlog.TaskItemList class you must write.

The Task class in the TaskXML library is used by the TaskItem class. The TaskItemList class should create a new TaskItem from an Task object (see TaskItem.TaskItem(Task) constructor). When saving, the TaskItem should generate a Task object for writing to a file (see TaskItem.getXMLTask()). The private method setType(String) helps with translating between the enumeration type and their string equivalents (which are all stored as constants in

TaskItem). The private method setState(String) helps with translating between a state name as a string and the TaskItemState object that represents the current state.

ScrumBacklogModel interacts with the TaskReader and TaskWriter classes. TaskReader unmarshals XML files and creates Task objects, which can then be used to create TaskItems. TaskWriter marshals Task objects into valid XML.

For examples of creating and working with Task objects and for examples of using TaskReader and TaskWriter, see the provided tests for the TaskXML library. The source for all of the TaskXML library classes and tests are included in the provided jar file. Open up the jar under the **Referenced Libraries** item in your Eclipse project and double click on a class file to see the source code in your editor.

Follow these steps to add TaskXML.jar to your project build path:

In your Project1 Eclipse project, create a new folder called lib/ by right-clicking on your project and selecting **New > Folder**.
Copy TaskXML.jar into the lib/ directory of your Project1. You can do this either through Eclipse's UI or through the file system. If you copy via the file system, you will need to refresh your Eclipse project by right-clicking on the project and selecting **Refresh**.
Right click the project and select **Build Path > Configure Build Path**. A dialog box opens.
In the resulting dialog box, open the **Libraries** tab.
Click **Add JARs**.
Select TaskXML.jar in the lib/ directory, then click **OK**.
Commit/push the jar file to GitHub. Your TaskXML.jar is required to be in the lib/ directory of your project for your project to successfully build on Jenkins.

Package edu.ncsu.csc216.backlog.model.command.

Command

The Command class creates objects that encapsulate user actions (or transitions) that cause the state of a TaskItem to update. We recommend that you start with the Command class because the TaskItem class and the concrete States rely on a correctly working Command class.

Command should include enumerations for for the possible commands (which correspond to a button click by a user) that can cause transitions in your FSM (e.g., CommandValue). Our textbook defines enumeration as "a type that has only a small number of predefined constant values." Since there are a discrete number of actions a user can take (essentially all buttons not labeled cancel), an enumeration is quite appropriate to list those values.

Copy the code below should into the Command class exactly as given. You can include the enumeration code with the fields of Command. Enumerations are pseudo-objects (like arrays). **Constructors are listed in the class diagram, but are not needed in the implementation**.

```
/** Lists possible commands */
public enum CommandValue { BACKLOG, CLAIM, PROCESS, VERIFY, COMPLETE, REJECT }
```

To access a value in the enumeration, use the enumeration name followed by the value (for example, CommandValue.BACKLOG). 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 Type!). Enumerated types can also be the type of a variable. For example, Command's c field is of type CommandValue. See pp. 1133-1134 in the Reges and Stepp textbook for more details on enumerated types.

Command has several constants that provide the string values for each of the enumerations as listed in the class diagram.

A Command constructor has 3 parameters: CommandValue c, String noteAuthor, String noteText. All parameters are required for any Command. Any of the following conditions result in an IllegalArgumentException when constructing a Command object:

- A Command with a null CommandValue parameter. A Command must have a CommandValue.
- A Command with a noteAuthor that is null or an empty string.
- A Command with a noteText that is null or an empty string.

The remaining methods are standard getters for the fields. There are no setter methods in Command. The fields are only set during construction.

Package edu.ncsu.csc216.backlog.model.task

Note

Note encapsulates the author and note text of a user's interaction with a TaskItem. A noteAuthor string (first parameter) and noteText string (second parameter) should not be null or be an empty string. If one of these values is null or an empty string during construction, an IllegalArgumentException is thrown. The only time that the noteAuthor and noteText should be set is during construction. The private setter methods in the Note class are not required. You may not create public setter methods for these fields!

Note.getNoteArray() returns an array of length 2 with the author at index 0 and the text at index 1. This is used when creating a list of ntoes to display in the GUI.

TaskItem

The TaskItem class is the implementation of the Task FSM. It is the context class of the State pattern and holds the abstract state (the interface TaskItemState), and the concrete states (all the *State classes).

Type Enumeration

TaskItem includes an enumeration for the possible types of tasks (e.g., bug, feature, technical work, and knowledge acquisition). Since there are a discrete number of task types an enumeration is quite appropriate to list those values. Use the following code for the enumeration.

```
/** Lists possible task types */
public enum Type { FEATURE, BUG, TECHNICAL_WORK, KNOWLEDGE_ACQUISITION }
```

TaskItem Fields

TaskItem represents a task tracked by our system. A TaskItem knows its taskId, state, type, title, creator, owner, isVerified, and all of its notes. Each TaskItem has its own state, which is updated from Commands propagated to it from the UI. Other fields may be updated when a Command is issued as per the definition of the Scrum Backlog FSM.

TaskItem has a static field, counter, that maintains the taskId number of the next TaskItem. The counter field is static so that it can be shared across all instances of the TaskItem class. That means an increment by one instance would change the value seen by all instances. This allows you to use the counter as an id generator for our TaskItems. The counter should be initialized to 1 and should never contain a value of zero or less (this was a small update to the original requirements). The counter may be set by working with the setCounter() static method. setCounter() should throw an IllegalArgumentException if the new counter is zero or less. The other static method for the counter is the incrementCounter() method. This method should be called from the constructor AFTER assigning the current counter value to the new TaskItem's taskId. The increment increases the counter for the next TaskItem's id.

When testing, you can use the setCounter() method in the test class' setUp() method to ensure that your ids always start at 1.

TaskItem Constructors

A TaskItem can be constructed with the following parameters: title (String), type (Type), creator (String), note (String). The creator is the noteAuthor of the first Note in the TaskItem. If any of the constructor parameters are null or empty string (as appropriate for the parameter type) an IllegalArgumentException is thrown. A newly constructed TaskItem starts in the BacklogState.

A TaskItem may also be constructed from a Task. The information in the Task is stored in the TaskItem. Private helper methods setState() and setType() can help transform a string input into the needed type. If the TaskItem is created from a Task, the taskId is the value from the Task object. The TaskItem(Task) constructor should NOT work with the counter field. That will be handled in another part of the system.

TaskItem Behaviors

You should start by creating TaskItem's main behaviors: constructor, getters, and the special methods for working with the state and type. Once those are working, you can move on to the

State pattern. TaskItem.setState(String stateValue) should throw an IllegalArgumentException if the parameter representing the String representation of the state is null or not one of the correct state names (which are public constants of the TaskItem class). TaskItem.setType(String typeValue) should throw an IllegalArgumentException if the parameter representing the short String representation of the type is null or not one of the correct type names (which are public constants of the TaskItem class). getTypeString() returns the TaskItem type as the short hand used in the XML. For example "B" for Bug. getTypeFullString() is used by the GUI to display the information about the type like "Bug".

Implementing the State Pattern

To start implementing TaskItem's underlying state pattern, first add the TaskItemState interface as a private inner interface of TaskItem. You will need to copy and paste the code into the TaskItem class.

The six concrete *State classes implement TaskItemState. They should be inner classes of TaskItem. Every concrete TaskItemState must support two behaviors: 1) update when given a Command and 2) know its name.

TaskItem maintains one instance of every concrete TaskItemState class. Any of these instances may be the current state of the TaskItem at any given time. All of the TaskItemState instances should be final.

- backlogState: Final instance of the BacklogState inner class.
- ownedState: Final instance of the OwnedState inner class.
- processingState: Final instance of the ProcessingState inner class.
- verifyingState: Final instance of the VerifyingState inner class.
- doneState: Final instance of the DoneState inner class.
- rejectedState: Final instance of the RejectedState inner class.

TaskItem has six inner classes that each implement TaskItemState. The inner classes each represent the six states in the Scrum Backlog FSM. Each concrete TaskItemState handles updating when given a Command is provided. If the Command is in appropriate for the current state, then the concrete TaskItemState's update() method throws an UnsupportedOperationException. Otherwise, the state is updated as appropriate for the Command and as defined in the Scrum Backlog FSM. A concrete TaskItemState also knows its name. The use case associated with each concrete TaskItemState is below:

- UC3: Backlog State
- UC4: Owned State
- UC5: Processing State
- UC6: Verifying State
- UC7: Done State
- UC8: Rejected State

Package edu.ncsu.csc216.backlog.model.scrum_backlog

The scrum_backlog package contains the two classes that manage all TaskItems during program execution.

TaskItemList

A TaskItemList maintains a List of TaskItems (you may use either an ArrayList or a LinkedList from the Java Collections Framework as the constructed type). TaskItemList supports the following list operations:

add a TaskItem to the list
add a List of Tasks from the XML library to the list
remove a TaskItem from the list
search for and get a TaskItem in the list by id
update a TaskItem in the list through execution of a Command
return the entire list or sublists of itself (for example, the TaskItemList can return a List of TaskItems filtered by owner or creator)

When creating a new TaskItemList, reset the TaskItem's counter to 0. When creating an TaskItemList from the contents of an XML file, set the TaskItem's counter to the maxId in the list of XML tasks plus 1 (this uses the min/max looping paradigm).

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

ScrumBacklogModel

Separation of TaskItemList from ScrumBacklogModel means each class can have a very specific abstraction: TaskItemList maintains the List of TaskItems and ScrumBacklogModel controls the creation and modification of (potentially many) TaskItemLists.

ScrumBacklogModel implements the Singleton design pattern. This means that only one instance of the ScrumBacklogModel can ever be created. The Singleton pattern ensures that all parts of the ScrumBacklogGUI are interacting with the same ScrumBacklogModel at all times. Note that the ScrumBacklogGUI does not have a global reference (or a field) to ScrumBacklogModel. Since ScrumBacklogModel is a Singleton, the GUI can access that single instance at any time with the static getInstance() method.

ScrumBacklogModel works with the XML files that contain the TaskItems in a file when the application is not in use. Therefore, ScrumBacklogModel works closely with the TaskReader and TaskWriter classes in the TaskXML.jar. If an TaskIOException (defined in the TaskXML library) is thrown, ScrumBacklogModel should catch it and throw a new

IllegalArgumentException to the GUI.

ScrumBacklogModel also provides information to the GUI through methods like getTaskItemListAsArray(), getTaskItemListByOwnerAsArray(String), getTaskItemListByCreatorAsArray(String), and getTaskItemById(). The first three of these methods return a 2D Object array that is used to populate the TaskItemTableModel (inner class of the ScrumBacklogGUI.TaskItemListPanel) with information. The 2D String array stores [rows][columns]. The array should have 1 row for every TaskItem that you need to return. There should be 3 columns:

Index 0. TaskItem's id number
Index 1. TaskItem's state name
Index 2. TaskItem's title

If the owner/creator String passed to getTaskItemListByOwner/CreatorAsArray(String) is null, throw an IllegalArgumentException.

For all the remaining methods, delegate to the TaskItemList class. All ints are taskIds. The parameters for ScrumBacklogModel.addTaskItemToList(String title, Type type, String creator, String note) match the corresponding method in TaskItemList and one of the TaskItem's constructors.

Package edu.ncsu.csc216.backlog.view.gui

ScrumBacklogGUI

After you have completed the model code that contains the Scrum Backlog logic, you'll add the ScrumBacklogGUI.java class to the edu.ncsu.csc216.backlog.view.gui package.

There is a known bug in the ScrumBacklogGUI where the contents of any of the note text JTextAreas will not wrap to a second line unless you use the enter key. If the teaching staff resolves the issue, we will provide an updated GUI. Otherwise, work with the current GUI. Your model implementation should handle any text entered into the JTextArea, wrapped or not.

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 an TaskItemList has already been created or loaded and populated with at least one TaskItem in the BacklogState. The flow of control in the sequence diagram is similar for other functionality.

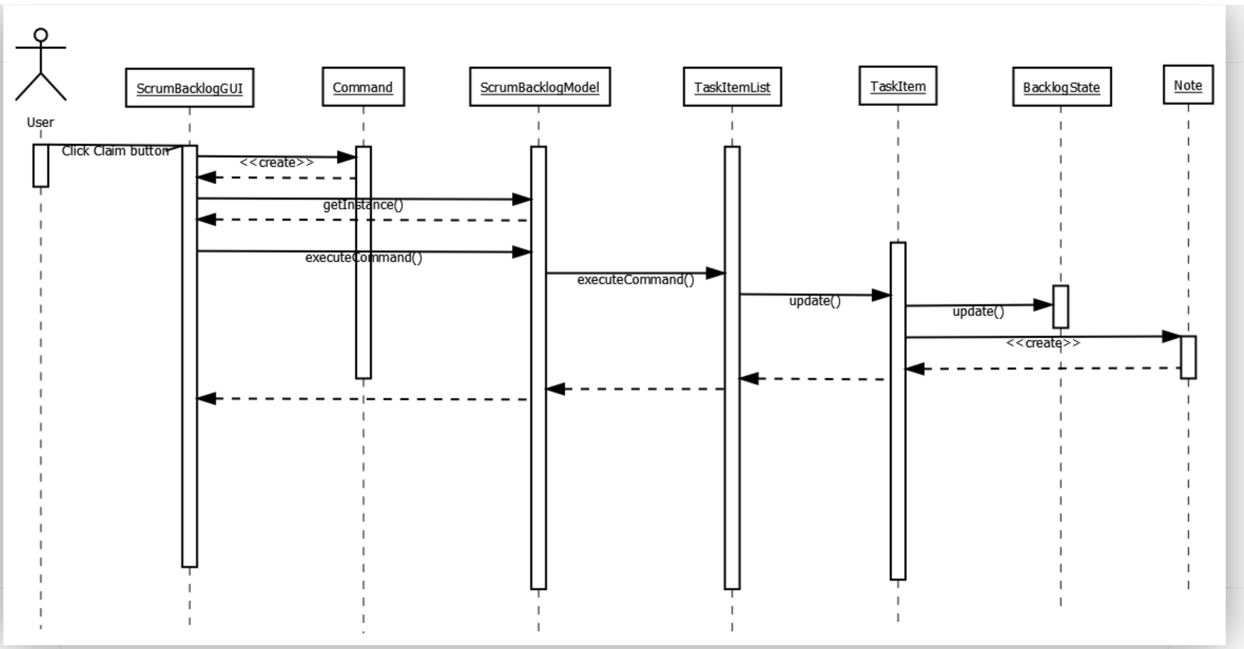


Figure 2: Scrum Backlog Sequence Diagram for UC2, S3 to UC3

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

We have provided several test files that will be helpful when testing the Scrum Backlog system.

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 (even inner classes). At a minimum, you must exercise every method in your solution at least once by your JUnit tests. 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 are covering all methods. If you are not, write tests to exercise unexecuted 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 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.

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 80% method coverage to achieve a green ball on Jenkins (assuming everything else is correct).

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).

Testing the State Pattern

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

If you want to test the id of a TaskItem, make sure that you reset the counter to zero at the start of each test BEFORE creating a new TaskItem. We suggest using the setUp() method in your test class to reset the counter.

Remember that when testing an FSM, you want to test each transition out of each state. That means you must first get to the state (by issuing Commands) and then test the transition out (by issuing another Command). Then you need to check that the state of the TaskItem is correct.

An example test scenario for testing transition ProcessingB would be the following:

1. Create a new task item
2. Check that newly created TaskItem is Backlog
3. Issue a CLAIM Command and assign owner “owner” with note “assigning owner”.
4. Check that TaskItem is Owned and assigned to the given owner.
5. Issue a PROCESS Command.
6. Check that TaskItem is in Processing with the correct information.
7. Issue a VERIFY Command.
8. Check that TaskItem is in Verifying with the correct information.

You should create similar tests for all the other transitions and then consider different patterns of transition.

Testing the Singleton

The ScrumBacklogModel is a singleton class, which means there is only one instance, and once that instance is created, that’s the instance that is used. So if one test adds TaskItems to a list, then those TaskItems will be there for the next test. To make each test atomic so that it can run in isolation (because run order is NOT guaranteed), you can use the ScrumBacklogModel.createNewTaskItemList() method to remove any TaskItems that exist in the ScrumBacklog. Since the taskItemList is the only other state (besides the singleton instance), using that method resets the Singleton to an empty TaskItemList.

For the TaskItem’s id to work correctly, the TaskItemList constructor MUST reset the counter to 1 (for an empty TaskItemList) or to maxId + 1 for an TaskItemList generated by reading from a file.

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.

If you are planning for your tests to read from or write to files, you need to provide details about what the files so that the teaching staff could recreate them or find them quickly in your project. Similar to Part 1 submissions, the test files may be one of the provided test files.

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_P2P2.pdf**.
3. Create a folder named **project_docs** at the top level in your project and copy the pdf to that folder.
4. Push the folder and contents to your GitHub repository.

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. You can estimate your grade from your Jenkins feedback.
4. You generate javadoc documentation on the most recent versions of your project files.
5. You push any updated project_docs, 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 ▶