Appendix A Animation Framework

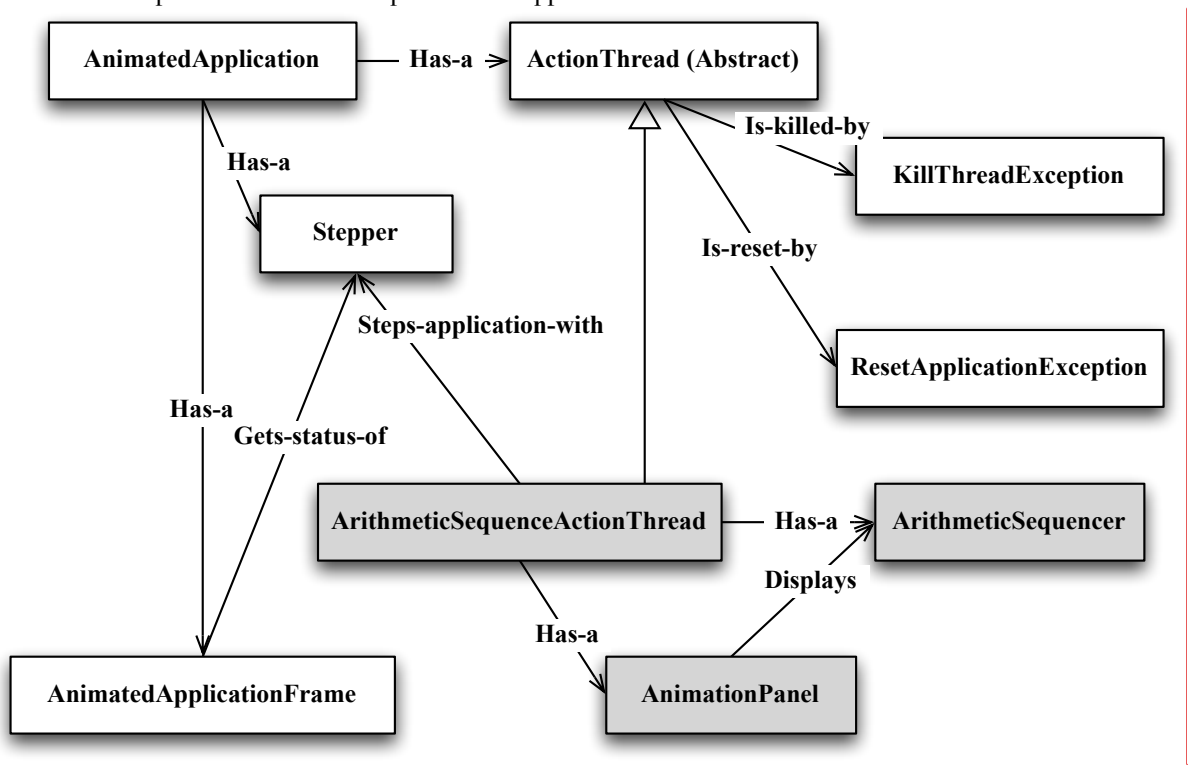
Animated Application

This appendix describes the framework used by a number of the labs to implement an animated application. This framework supports applications that have a set up phase where application specific variables are set. The application then runs and has checkpoints that the user can step between where the state of the application is displayed graphically. The application can be reset at any point and restarted. While this framework is not designed to handle applications like games that periodically require user input, an experienced programmer could modify it to do a simple game. A general description of the framework and how it works is given first. After that, directions are given for implementing a new animated application using the framework classes.

Instead of just giving the framework with blanks to be filled in by the programmer, a simple application demonstrating the framework is shown. It computes the terms of an arithmetic sequence.

The Framework

The following class diagram shows the relationship among the classes in the application. The classes that are shaded are the specific classes that implement the application. The other classes are the framework classes.



AnimatedApplication

This class has the `main()` method used to invoke the application. It is responsible for creating

1. an `AnimatedApplicationFrame`, which contains the controls for stepping the application,
2. a `Stepper`, which is how the application will signal that it is waiting, and
3. an `ActionThread` that executes the application.

Once these objects are created, `main` starts the `ActionThread`. This is the only class in the framework that needs to change for different applications. (An action thread of the appropriate type needs to be created.)

Stepper

This class acts as an intermediary between the control components of the frame and application. It has a number of methods where the action thread can wait for a step to occur. The state of the action thread (setup phase, initial phase, stepping, final phase) will be indicated by the specific call that gets made. The animation frame will set the controls based on the state of the stepper. When stepping, the stepper will keep track of the step that the application is on.

`getStep()`—This method is used by the application frame when displaying the controls. It returns the number of application steps since the last reset.

`getStatus()`—This method is used by the application frame when displaying the controls. It returns the phase.

`setupStep()`—This method will be used by the action thread. It will change the status of the stepper and then wait for notification it can continue. It should only be used to indicate that the application specific controls can be used to set the parameters for the application.

`initialStateStep()`—This method is used by the action thread. It will change the status of the stepper and then wait for notification it can continue. This step occurs after initialization has been performed for the application, but the application has yet to start running.

`animationStep()`—This method is used by the action thread. It will change the status of the stepper and then wait for notification it can continue. These steps are counted and are the actual steps in the execution of the application.

`finalStep()`—This method is used by the action thread. It will change the status of the stepper and then wait for notification it can continue. In this step the application has finished execution and the final state of the animation display is held so it can be viewed.

It is not intended that the application thread interact directly with the stepper, but instead will use methods defined in the abstract class `ActionThread` to affect the stepper.

This class should not be changed for new applications.

AnimatedApplicationFrame

This class is subclass of `JFrame` and holds the entire application. It creates the user interface components for stepping the application. The control components are placed in a panel, which in turn is placed in the north position of the frame. The specific animation panel for the application is retrieved from the action thread and placed in the center position of the frame. This class will use a `Timer` that fires and generates a step at given intervals. The timer will be started if go is pressed. It will be stopped if pause or reset is pressed. A step will notify the waiting action thread that it can continue.

This class should not need to be changed for new applications.

ActionThread

This abstract class specifies certain responsibilities that an `ActionThread` must satisfy to "play nicely" with the `AnimatedApplicationFrame`. The major responsibilities of an actionThread are to

1. define the general run method for executing the thread,
2. have a stepper object to control the animation,
3. have a panel that the application will do its animation on, and
4. have a mechanism to kill or reset the thread.

The heart of the `ActionThread` class is the `run()` method.

> **`run()`**—Any thread must define this method to specify what happens when the thread is executed. For the `ActionThread` class, the thread will execute the application of interest one or more times. There will be a setup phase, where the user is allowed to change the input parameters for the application. The application will be initialized and then executed. There is a final step, which allows the final state of the application to be displayed. It is not intended that this method be modified.

The `ActionThread` class has some methods that its subclasses can use when implementing the animation

> **`animationPause()`**—This method should be called any time that the state of the application has changed and the animation needs to be updated. It automatically calls `makeThreadWellBehaved()`. It is not intended that this method be modified.

> **`forceLastPause()`**—If you want to kill or reset the animation, an exception needs to be thrown. Before invoking the exception, you should do an animation pause. This method will indicate that the animation should go to the last step and it should be called before the exception is thrown. It is not intended that this method be modified.

> **`makeThreadWellBehaved()`**—Because of the cooperative technique that Java uses to kill a thread (explained below) the thread should regularly check to see if it needs to kill itself. It is automatically as part of every animation pause. If there is a part of the application that is computationally intensive, it is a good idea to call this function throughout the computation. It is not intended that this method be modified.

> **`getAnimationPanel()`**—Get the panel that is holding the application specific animation. It is not intended that this method be modified.

> **`applicationControlsAreActive()`**—Returns true if the animation is in the setup phase and the application specific controls should be active. It is not intended that this method be modified.

As an abstract class, `ActionThread` specifies some methods that are the responsibility of its specific subclasses to implement. These are related to the general mechanism for executing the thread.

> **`getApplicationTitle()`**—Return the title of this application.

> **`createAnimationPanel()`**—Create and return the panel that the application will do its animation on.

> **`setUpApplicationSpecificControls()`**—Create all of the components and panels for the user interface for the application. Place them on the animation panel. (Use the `getAnimationPanel()` method.)
>
> Since the panel will also be used to display the animation, you should be careful about where the components are placed. It is probably best to leave the center free. Placing the controls on the right side or the bottom may also make it easier to draw the animation. Creating the handlers for the controls can be done in a number of different ways. One fairly typical technique (which is used in the example) is to create an anonymous inner class that acts as a listener for the component. It calls a handler method, which is easier to locate than code embedded in the inner class.

> **`init()`**—This method will be run before the application is executed in the setup phase. It will initialize any variables that the application will use.

`executeApplication()`—A method that does a single execution of the application. The entire code for the application may be contained within this method, but it is not required. In fact, good program design encourages abstraction via the creation of other methods. For thread safety, the application should never change or access any of the components in the user interface. Instead, it will change display variables. The display will be redrawn using the new status of the display variables when the `animationPause()` method is invoked.

One of the requirements is that the application that is being run should be killable. The approved technique in Java is to use cooperative action. The outside object, which wishes to kill the thread, calls a method that sets a variable. Periodically the thread must check this variable and if needed kill itself.
The thread that is created will run the application one or more times. This necessitates the ability to reset the application in the thread. Using cooperative action, a variable will be set. As before, the thread is responsible for checking periodically to see if it needs to be reset.

`resetExecution()`—Allow an outside object to signal this thread to reset the application it is currently running.

`killThread()`—Allow an outside object to signal this thread to kill itself.

ResetApplicationException

The `makeThreadWellBehaved()` method in `ActionThread` will throw this exception if the current execution of the application should be halted. It is caught by `run()` and appropriate action is taken. This class should not need to be changed for new applications.

KillThreadException

The `makeThreadWellBehaved()` method in `ActionThread` will throw this exception if the thread needs to die. It is caught by `run()` and appropriate action is taken. This class should not need to be changed for new applications.

The Arithmetic Sequence Application

There are three classes that implement the application that computes arithmetic sequences. These three classes are specific to the sample application, but they can serve as a model for the creation of new applications. One issue that must be addressed in these classes is thread safety.

Thread Safety

The animated application will have two threads running concurrently. One thread will be the application that is being executed and the other will be a thread that deals with the graphical user interface. Problems can occur when both threads can access a shared object. If the object is mutable, a situation can arise where one thread starts to change the state of the object and then is interrupted by the thread manager. If the second thread accesses the object, it can be in an invalid state. If both threads mutate the object, they can interfere with one another in unpredictable ways.

ArithmeticSequenceActionThread

This class is a concrete subclass of `ActionThread` and has the primary responsibility of defining how the arithmetic sequence application operates. The major work in using the framework for a different application lies in creating a class like this.

There are two kinds of private variables that an application will typically have. The first kind of variable will be referred to as a parameter of the application. These variables can be changed by the user interface of the application in the setup phase. They are then used to initialize any private variables that the application needs to run. They should not be changed during the running of the application. Examples of this kind of variable in the arithmetic sequence application are `start` and `delta`. These variables should always be initialized when they are declared.

The second kind of variable will be referred to as display variables. They are variables that will affect the graphics display of the animation. Display variables can be accessed by both of the threads in the animated application so thread safety is an issue. They should either be primitives, immutable, or specially designed objects. Examples of this kind of variable in the sample application are `mySequencer` and `count`. The responsibilities of the specially designed objects will be shown later with the `ArithmeticSequencer` class.

Since `ArithmeticSequenceActionThread` is a concrete subclass of `ActionThread`, it must define the abstract methods of `ActionThread`. These methods are specialized for the arithmetic sequence application:

`getApplicationTitle()`—Returns "Arithmetic Sequences (Sample Application)".

`createAnimationPanel()`—Creates and returns a new `AnimationPanel`.

`setUpApplicationSpecificControls()`—Creates and places two text fields on the animation panel, one for the initial value in the sequence and the other for the difference between terms in the sequence. Creates and places three labels on the animation panel.

`init()`—Initializes all the variables that the application needs to execute. It will be invoked immediately after the setup phase and may use the application parameters to initialize the other variables. In the arithmetic sequence application, there are just the two display variables, `count` and `mySequencer`, that need to be initialized.

`executeApplication()`—A method that does a single execution of the application. It contains a loop that runs ten times. The body of the loop adds the next term in the sequence using the `mySeqencer` object and then pauses the animation.

The class has two handler methods for the text fields. Thread safety is an issue with these methods since they will be invoked from the user interface thread. The handlers should only change the parameters for the application. In addition, only the handlers are allowed to access the user interface components.

`countByTextFieldHandler()`—A handler method that gets the string from `countByTextField`, converts the string into an integer, and finally initializes the `count` parameter.

`startAtTextFieldHandler()`—A handler method that gets the string from `startAtTextField`, converts the string into an integer, and finally initializes the `start` parameter.

To be really well behaved, the handlers should only change the parameters if the application is in the setup phase. The method `applicationControlsAreActive()` can be used to check if the application is in the setup phase.

The application thread (any methods that are invoked from `executeApplication()`) should not access any of the user interface components, as they are not thread safe.

AnimationPanel

It is expected that every subclass of `ActionThread` will have an inner class that has the single responsibility to draw the animation frame. Thread safety is a concern, so it should only access display variables.

`paintComponent()`—This method draws the animation frame. It must call the super class method first. This guarantees that any components on the panel are drawn. It is expected that specially

designed display classes will do most of the actual drawing. Before drawing, it is important to check that any nonprimitive display variables are nonnull.

For the arithmetic sequence application, the number of terms added from the sequence is drawn in a string. If `mySequencer` has been initialized, it will draw itself on the panel.

ArithmeticSequencer

The display variable `mySequencer` is of this type. It is a specialized auxiliary class that holds information that will be displayed by the application during the animation. This kind of class will have mutator methods that the application calls.

`addNext()`—A mutator that directs the object to compute the next term in the sequence. A string with that term will be added to the list of terms that will be displayed.

The other responsibility of this kind of class is to draw a representation of itself on a given graphics context.

`drawOn()`—This method draws on the given graphics context. Besides the graphics context, it often will have as parameters *x* and *y* coordinates that shift the origin of what it draws. It may also have a scale parameter that changes the size of the drawing.

For the `ArithmeticSequencer` application, a star like object will be drawn with one spoke for each of the terms computed so far. Each of the strings will then be displayed, one to a line. (Each will have a term in the sequence.)

These auxiliary animation classes are shared by both the application thread (mutator methods) and the user interface thread (the `drawOn()` method). Therefore, thread safety is important. All methods in the auxiliary display classes should be synchronized.

Using the Framework to Create New Applications

The following is offered as a guideline for using the framework to create a new animated application. It uses the arithmetic sequence application as a starting point.

Preparation

Step 1. Decide on a name for your application. (The name will be referred to as XXXX in the rest of the guidelines.)

Step 2. What information must your application have before it can start? Only list those things that the user can change.

Examples are

numerical values—such as the size of an array or number of values to generate
strings—such as the name of a file containing data
Boolean values—such as a flag indicating the amount of information to be displayed
Enumerations—such as the color of a displayed component

Step 3. What should be displayed at each step of the animation? Create rough sketches.

Step 4. Is the animation display composed of pieces? Write down class names for each of the pieces (auxiliary classes). (These are analogous to `ArithmeticSequencer` in the sample application.) Give a brief list of responsibilities of these classes. Think carefully about the states that the classes can be in. List methods that change the state of these classes.

Step 5. Copy the files in the *`Arithmetic Sequence`* folder into a new folder.

Step 6. Make a copy of `ArithmeticSequenceActionThread` and call it `XXXXActionThread`. Change the class declaration and the constructor to match the new name for the class.

Step 7. Make a copy of `AnimatedApplication` and call it `XXXXApplication`. Change the class declaration to match.

Changes to AnimatedApplication

Step 8. Change the creation of `myThread` to use `XXXXThread` instead of `ArithmeticSequenceActionThread`.

At this point, no more changes are needed to XXXXApplication. Before working on the action thread, it is a good idea to create the auxiliary classes that will be used to draw the animation. The class ArithmeticSequencer can be used as a model.

Auxiliary Animation Classes

Step 9. Create each of the auxiliary classes listed earlier.

Step 10. Create constructors for each class.

Step 11. Create the accessor methods for each of the classes.

Step 12. Create the mutator methods for each of the classes.

Step 13. Create a `drawOn()` method for each class. This class must have a parameter `Graphics g`, which is the context that the class will draw itself on. It is strongly recommended that a position (*x* and *y* coordinates) be passed in as well that the drawing will be relative to. A scale parameter may also be useful.

Step 14. Test the auxiliary classes. These tests do not need to be exhaustive and do not need to test `drawOn`.

Remember to make all the methods of these classes `synchronized` so they are thread safe.

The major changes will be to the action thread. These changes will be approached in small chunks. First the arithmetic sequence applications code will be cleared to make way for the new application. Keep the original code in comments so it can be referred to if needed.

Clearing Out the Code in XXXXThread

Step 15. Comment out the declaration of the private variables `start` and `delta` that were parameters for the application.

Step 16. Comment out the declaration of the private variables `mySequencer` and `count` that were display variables for the application.

Step 17. Comment out the body of the methods `init()`, `executeApplication()`, and `paintComponent()` of the inner class `AnimationPanel`.

XXXXThread should compile with no errors. The next change to make is to add in the application specific controls.

Creating the Application-Specific Control Components

Step 18. Refer back to your preparations and create a private variable for each parameter (required piece of information to start the application). Put them in the place of the declarations for `start` and `delta`. Make sure that they are initialized.

Step 19. Create user interface components for each of the parameters to allow them to be set. The arithmetic sequence application puts all of the components into a panel named `setupPanel`, which it puts in the south location of the animation panel. More panels can be used and other locations used as desired. Do not put anything in the center, since that will be used for displaying the animation.

Step 20. Use the existing code as a template for setting up the listeners for each of the components. Create handler methods for each of the components. (Use the existing handlers as examples.) Make sure the handlers set the appropriate private variables and provide feedback via the `setupStatusLabel`.

XXXXThread should compile with no errors and the application should run. Check that the application specific controls operate. The next change will be to initialize the application and make sure that the animation is displayed correctly.

Initialization of the Application

Step 21. Change the title returned by `getApplicationTitle()`.

Step 22. Create variables for the items that will be displayed graphically by the application. (These go where `mySequencer` and `count` were.)

Step 23. Initialize each of the variables that were just created in the `init()` method.

Step 24. Add code to the `paintComponent()` method, which will draw each of the pieces. Most of the time this will involve calling `drawOn()` methods for the specific objects. Two notes: First, make sure that `super.paintComponent(g)` is called. The component will not draw correctly without it. Second, this method may be called before `init()` has had a chance to work. So before using any object, make sure that it is nonnull.

XXXXThread should compile with no errors and the application should run. Set the variables and then click on Step. The initial state of the application should be displayed. Debug the code as needed so that the display is correct. If needed, you can add calls to the mutators for the display variables in init() and verify the displays and methods work as expected.

Stepping the Application

Step 25. Add in code to `executeApplication()` that specifies what the application is to do. Create new methods as needed. Create new private variables as needed. As with any project, it is a good idea to work incrementally.

Whenever you want the application to change its display, add in the following line of code.

```
animationPause();
```

This will make the application wait until it is stepped (either manually, or after a given delay if the go button has been pressed). It also makes the thread well behaved (checks to see if the application thread should be killed or reset).

If you have sections of the code that are computationally intensive (they do not make many calls to `animationPause()`), it is a good idea to add extra calls to `makeThreadWellBehaved()`. For example, it can be placed in the body of an iteration or recursive function.