20. Command Objects

We will see here that some of the concepts we use to define objects such as types (classes/interfaces) and actions (methods) can themselves be considered as objects with types and actions of their own. Such objects allow a tool such as ObjectEditor to understand and use, or reflect on, arbitrary objects, even instances of types defined after the tool was compiled.

We will also study a variation of the action object, called a command object, which has the same relationship with an action object that a command such as “Do your home work.” has with the verb “Do”. It encapsulates an action invocation.

Command objects, in turn, will allow us to define and understand threads, which are non blocking method invocations that support interleaved or concurrent activities. Threads, in turn, allow us to create animations that do not block or freeze the rest of the user-interface.

A variation of command objects will allow us to implement undo and redo of an action invocation. We will see that command objects allow us to create (part of) an undo/redo implementation that can work for arbitrary applications.

Reflection

To motivate action and type objects, consider the following user-interface.

![User Interface Image]

The program implementing this user-interface prints the values of all properties of an instance of BMISpreadsheet and an instance of Point. The property, Class, is inherited from Object and is filtered out by ObjectEditor when it displays an Object.
We would like to write a program, that like, ObjectEditor can display the properties of arbitrary objects. Thus, it should know nothing about the types BMISpreadsheet and Point. It should contain a printProperties() method that takes as argument instances of BMISpreadsheet, Point or any other type, as shown below.

```java
public static void main(String[] args) {
    BMISpreadsheet bmi = new ABMISpreadsheet(1.77, 75);
    printProperties(bmi);
    Point point = new ACartesianPoint(50, 34);
    printProperties(point);
}
```

This, in turn, means that that the argument of printProperties() must be Object. Yet we want it to call methods such as getBMI() and getX() that are not defined by Object. We can do so by casting the argument to specific types such as BMISpreadsheet or Point, but as mentioned above, printProperties() has no knowledge of these types. Action and type objects allow us to have our cake and eat it too by allowing invocations of type-specific operations on objects without using casts. The following code illustrates them.

```java
public static void printProperties(Object object) {
    System.out.println("Properties of:" + object);
    Class objectClass = object.getClass();
    Method[] methods = objectClass.getMethods();
    Object[] nullArgs = {};
    for (int index = 0; index < methods.length; index++) {
        Method method = methods[index];
        if (isGetter(method)) {
            Object retVal = methodInvoke(object, method, nullArgs);
            System.out.println(propertyName(method) + ":" + retVal);
        }
    }
    System.out.println();
}
```

The method invokes the getClass() method on its argument to determine the class of the object and stores the return value in objectClass. This method returns a type object describing the class of the object. The type of the type object is the Java class named Class. This class defines the getMethods() operation, which returns an array of action objects describing the methods of the class. The type of the action object is the class Method. printProperties() invokes getMethods() on objectClass, and passes each element of the returned array to isGetter() to determine if it describes a getter method. If it does, printProperties() calls propertyName() to get the name of the associated property, and methodInvoke() to get the value of the property.
Here objectClass, like a type parameter of a generic, is a variable that can be associated with different types. The difference is that it a generic type parameter is a compile-time variable while a variable of type Class is a runtime variable.

It is possible to write methodInvoke(), property Name(), and isGetter() because the class Method provides operations to invoke the method it describes, and determine the name, return type, and parameter types of the method, as shown below.

```java
public static String GETTER_PREFIX = "get";
pUBLIC static boolean isGetter (Method method) {
    return method.getParameterTypes().length == 0 & &
            method.getReturnType() != Void.TYPE & &
            method.getName().startsWith(GETTER_PREFIX);
}

public static String propertyName(Method getter) {
    return getter.getName().substring(GETTER_PREFIX.length());
}

public static Object methodInvoke(Object object, Method method, Object[] args) {
    try {
        return method.invoke(object, args);
    } catch (IllegalAccessException e) {
        e.printStackTrace();
        return null;
    } catch (InvocationTargetException e) {
        e.printStackTrace();
        return null;
    }
}
```

The substring(i) operation invoked by propertyName() on string, s, returns the string s.charAt(i)…s.charAt(s.length() – 1).

The invoke() operation provided by Method executes the method it encapsulates, and takes as arguments the object on which the method is to be invoked and an array of the formal arguments to be passed to the method. When we study exceptions, we will understand more thoroughly what the try/catch exactly means. For now it is enough to know that the invoke() throws IllegalAccessException error if the called method is not visible to its caller, and the InvocationTargetException if the called method does not match the arguments or throws any exception.

Invoking operations on a type (action) object is called type (action) reflection as it allows the caller to understand or reflect upon the type (action). Several object-oriented languages such as Smalltalk, Java, Python and C# support reflection. The figures below show some of the operations supported by these languages on type and action objects.
A type object encapsulating some type T cannot support an operation to get all of the subtypes of T as these are not known and could be distributed all over the world. If the associated type is an instantiatable class, then the type object can support an operation to instantiate the class.

The execute() operation provided by an action object (called invoke() by Java) takes as arguments the target object and actual parameters of the encapsulated method. A command object is a variation of a command object that has the target object and actual parameters embedded in it. The following animation example motivates such an object and the concept of a Java thread, which is built on top of the concept of a command object.

**Animating Shuttle**

Consider again the shuttle location object we created earlier. Suppose we create a variation of it, AnAnimatingShuttleLocation, that provides an additional method, animateFromOrigin (Figure 8), that animates the path the shuttle took from the origin to its current position. As we don’t know actually know this exact path, we will make the simplifying assumption that the shuttle first went vertically up to its current Y position (Figure 9), and then horizontally went to its current X position (Figure 10).

Before we try to code this object, let us first define what exactly it means for a method to animate the shuttle from one location to another one. It could imply that the method:

- moves the shuttle a distance, D, in a straight line towards the destination.
- checks if the shuttle has reached its destination. If yes, the method terminates; otherwise it repeats the above two steps.
However, this algorithm does not work as the computer will execute the steps so fast that the user will not see the intermediate positions of the shuttle – the shuttle will seem to reach its destination instantly. Thus, we need the method to pause for some time after step 1.

In summary, the method:

- moves the shuttle a distance, \( D \), in a straight line towards the destination.
- pauses for some time \( T \) to make the shuttle stays at its current location.
- checks if the shuttle has reached its destination. If yes, the method terminates; otherwise it repeats the above three steps.

We will assume that \( D \) is given by a named constant and time \( T \) is specified by the user-defined property, \texttt{AnimationPauseTime}.

```java
int animationPauseTime;

public int getAnimationPauseTime() {
    return animationPauseTime;
}
```
public void setAnimationPauseTime(int newVal) {
    animationPauseTime = newVal;
}

Figure 2. Animating the vertical shuttle motion
Figure 3. Animating the horizontal shuttle motion
In the remainder of this discussion, we will assume that time is specified in milliseconds. Thus, in Figure 8, the AnimationPauseTime specifies a pause time of 5000 milliseconds or 5 seconds. The large pause time gave us enough time to take a screen dump of each intermediate position of the shuttle. For the animation to appear smooth, the pause time should be about 30 milliseconds.

The repetition of the three steps shows the role loops play in solving this problem. However, what we have learnt so far is not sufficient to completely code it. How do we make the program pause, that is, do nothing for some period of time, pauseTime? The following is one way to do so:

```java
void sleep(int pauseTime) {
    int numberOfAssignments = pauseTime; //ASSIGNMENT_TIME
    for (int i = 0; i < numberOfAssignments; i++) {
        int dummy = 0; // nonsense assignment
    }
}
```

We could repeatedly make an unnecessary assignment in a loop until we have made enough to take time pauseTime. This “solution” has two problems. First, we would have to change our code each time we execute it on a computer of different power. Second, and more important, we unnecessarily use the computer executing the loop. It is for this reason that such a loop is called busy waiting.

What we really need is an operation that asks the operating system to suspend or put to sleep our program for pauseTime so that it can execute some other applications during this time. Java provides such an operation, `Thread.sleep(pauseTime)`, and the following recoding of our sleep method shows its use:

```java
void sleep(int pauseTime) {
    try {
        Thread.sleep(pauseTime);
    } catch (Exception e) {
        // program may be forcibly interrupted while sleeping
        e.printStackTrace();
    }
}
```

A sleeping method may be woken up not only when its regular alarm goes off, but also because of some unexpected condition such as the user terminating the program. To signal an abnormal waking up, `Thread.sleep(pauseTime)` throws an `InterruptedException`. We have therefore enclosed the call to it in a try-catch block.
In this course, we have tried to avoid using library calls we do not know how to implement. An exception is being made for the sleep() call above, as its implementation is extremely complex, and in fact, not possible in Java. Again, we will understand the try catch block when we learn about exceptions.

We can now code our animation algorithm, as shown below.

```java
public void animateFromOrigin() {
    animateFromOrigin(ANIMATION_STEP,
    getAnimationPauseTime());
}
```

```java
public void animateFromOrigin(int animationStep, int
animationPauseTime) {
    int curX = 0;
    int curY = 0;
    setLabelX(windowX(curX));
    setLabelY(windowY(curY));
    while (curY < getShuttleY()) {
        // loop make sure we don’t go past final Y position
        sleep(animationPauseTime);
        curY += animationStep;
        setLabelY(windowY(curY));
    }
    // move to destination Y position
    setLabelY(windowY(getShuttleY()));
    while (curX < getShuttleX()) {
        sleep(animationPauseTime);
        curX += animationStep;
        setLabelX(windowX(curX));
    }
    setLabelX(windowX(getShuttleX()));
}
```

We have a separate loop for moving in the X and Y direction. Each loop keeps track of the next X/Y position. In each iteration, this position is used to change the X/Y coordinate of the shuttle, and the position is incremented it by the distance ANIMATION_STEP. If the next X/Y position exceeds the final X/Y position, the loop exits, and thus does not change the shuttle coordinate. Therefore, after the loop, the method moves the shuttle to the final position.

**Interleaving Activities**

Unfortunately, the above implementation of animateFromOrigin() does not work. When we ask ObjectEditor to execute a method, it waits for the method to finish execution, just as any calling method waits for a called method to finish execution. During this period the user-interface is frozen – the display is not updated, and we cannot use any of the ObjectEditor menus. We have not noticed this so far.
because our methods returned quickly. The method animateFromOrigin is different because of the
time-consuming loops it executes. Figure 11 shows what happens.

While animateFromOrigin is executing, the menu is frozen and the display is not updated. When the
method finishes executing, the display is updated, at which point the shuttle is back at its original
position. Thus, the effect of executing the method is a long pause as ObjectEditor suspends its activities
for the time it takes to complete the “animation,” that is, the time it takes to complete all the sleeps in
the method.

During animation two interleaved activities must take place. One activity executes the loops, setting the
intermediate X and Y positions of the label. The second activity (executed by ObjectEditor) displays the
intermediate X and Y positions. These activities are interleaved in that one activity does not wait for the
other one to finish. Instead, after executing one or more steps of one activity, the computer can execute
some steps of the other activity, and then return to the first activity to execute its remaining steps. This
is shown in the figure below.

Figure 4. Instance of AnAnimatingShuttleLocation when animateFromOrigin is called
Here we see two activities, one that changes the X and Y coordinates of the shuttle, and another that refreshes the display to make the shuttle move. The arrows point to the next steps to be executed by each of these activities. The CPU can next executes a few steps of the first activity.

It can then switch to the second activity and execute steps in it.
It can then resume execution of the first activity, executing a few of the remaining steps in it.
Real-Life Analogy, Priorities and Thread Blocking

Interleaving of computer activities is analogous to real-life multi-tasking. The figure below shows a supermom switching between doing work and looking after the kid and dog.

In this scenario, we can expect the mother to give higher propriety to the child’s needs than to her work or the dog’s needs. Similarly, computer activities can have different proprieties. An interactive task such as editing is given higher than a non-interactive task such as scanning the disk for viruses. Moreover, just as the mother would switch from looking after the kid when he sleeps, the computer would switch from a computer activity when it sleeps or makes executes some other blocking or waiting operation such as waiting for information from a server.

Interleaving vs. Concurrency

It is possible to not only interleave but also, on a multi-processor/multi-core computer, execute the activities concurrently on different CPUs. Taking the above example, given the following execution state of the two activities:

```java
public synchronized void animateFromOrigin(int animationStep, int animationPauseTime) {
    int curX = 0;
    int curY = 0;
    setLabelX(windowX(curX));
    setLabelY(windowY(curY));
    while (curY < getShuttleY()) {
        sleep(animationPauseTime);
        curY += animationStep;
        setLabelY(windowY(curY));
    }
    // move to destination Y position
    setLabelY(windowY(getShuttleY()));
    while (curX < getShuttleX()) {
        sleep(animationPauseTime);
        curX += animationStep;
        setLabelX(windowX(curX));
    }
    setLabelX(windowX(getShuttleX()));
}
```

```java
propertyChange(PropertyChangeEvent event) {
    refresh(event);
}
```
In the next computer execution step or cycle, both pointers can advance concurrently.

Concurrent execution is analogous to us using different limbs to perform different activities simultaneously, as shown below.

Such concurrency is expected to be the key to harness the power of future computers. It is getting increasingly difficult to increase the power of a single CPU - therefore the power of computers is being primarily increased by adding additional CPUs. These CPUs can be kept busy only if there are enough threads to execute concurrently.

A computer can both interleave and concurrently execute activities. Given the following three activities:
In the next execution cycle, the instructions of the top two activities can be simultaneously executed by two CPUs.

In the next cycle, the top and bottom activities can be executed simultaneously by the two processors.
This scenario corresponds to a juggler using two hands to juggle three balls.

An extra hand would not be useful in this scenario because when a ball is in the air, it does not need service from a hand. Similarly, when a computer activity is waiting for user input or sleeping, it does not need service from a processor. Thus, we often do not sacrifice performance by having fewer CPUs than executable activities.
Processes vs. Threads
There are two kinds of interleaved/concurrent activities – processes and threads.

Each time a main class is executed, a process is created for executing the program. In the figure above, the two javaw.exe entries correspond to two main classes executed by the user.

We can use Eclipse debugger to show the threads created for executing a program.

If we put a breakpoint on a statement, we can see the various threads active when control transfers to the statement. Here we see not only the thread that will execute the statement, but also threads, such as the AWT-Windows and AWT-Shutdown threads, that perform background activities started by Java.

Thread and Command Objects
Like methods and classes, threads are a fundamental part of Java needed for creating executable programs. They are not simply library packages that can be replaced by our own code. However, as in the case of methods and classes, Java provides runtime objects to manipulate them. A thread object is an instance of the class Thread and is associated with an executing thread. It provides operations to start, suspend, resume, interrupt (the sleep of), and change the priority of the associated thread.
To understand how we can create a Thread object and an associated activity, let us return to the animation example.

Recall that the animation required two *interleaved* threads. One thread executes the loops, setting the intermediate X and Y positions of the label. The second thread displays the intermediate X and Y positions and processes menu and other actions of the user-interface.

Our implementation of the parameterless `animateFromOrigin()` executed the parameterized `animateFromOrigin()` as part of the user-interface thread. The former method blocked the user-interface thread until the latter finished executing loops.

```java
public void animateFromOrigin() {
    animateFromOrigin(ANIMATION_STEP,
    getAnimationPauseTime());
}
```

The solution is to create a new thread for executing the parameterized `animateFromOrigin()`, allowing the original user-interface thread to update the display and process user-commands while the animation loops execute as part of the new thread. Thus, we need a way to tell Java to execute the call `animateFromOrigin(ANIMATION_STEP, getAnimationPauseTime())` in a separate thread, and associate this thread with a new instance of `Thread`.

A straightforward approach is to create an instance of `Thread` and pass it information about the call to be executed as a separate thread. A call could be described by a `Method` instance describing the method to be executed and an array containing the actual parameters of the call, as shown in the figure.

```java
public void animateFromOrigin() {
    Thread thread = new Thread(animateFromOriginMethod, this,
    makeParams(ANIMATION_STEP, getAnimationPauseTime()));
    thread.start();
}
```

However, this approach is not supported in Java, for several reasons. It uses reflection, which was introduced in Java after threads were invented, and more important, is expensive and checks the compatibility between a method and its actual parameters at runtime, throwing
InvocationTargetException if the check fails. A better approach, then, is to pass to the Thread constructor a single object that represents a checked method call. Such an object is called a command object. Like an action object, it encapsulates an embedded operation, and provides an execute() method to invoke the operation. In addition, it encapsulates the actual arguments and target object of the operation, which are passed to it in its constructor. Thus, while the execute() operation of an action object takes the target object and arguments as parameters, the execute() operation of a command object takes no parameters. It simply invokes the embedded operation with the encapsulated arguments on the embedded target object. The following figures show the difference between the two kinds of objects.

In other words, an action object represents an operation, while a command object represents an operation invocation. A command object has the same relationship to an action object that an imperative sentence such as “Do your homework.” has to the verb “Do” in the sentence.

Let us see how we can use the concept of a command object to construct a Thread instance. Java provides a command object type called java.langRunnable, whose execute method is called run().

```java
package java.lang;
public interface Runnable {
    public void run();
}
```

If we wish to execute a method call in a separate thread, we should encapsulate the call in an instance of Runnable and pass this object to the constructor of Thread. In response, Java creates a new thread to execute the encapsulated call and associates it with the Thread instance. The start() operation can next be invoked on the Thread instance to start the method call encapsulated by the command object. This is illustrated below for our example.
Here, AShuttleAnimationCommand is a class implementing the Runnable interface that encapsulates the parameterized animateFromOrigin() method. Its constructor takes as argument the object on which the embedded operation is to be executed and the actual arguments of the call. The Thread instantiation is passed an instance of this class as an argument. When we invoke the start() operation on the Thread instance, Java starts asks the associated thread to execute the run() method of the command object.

This method, in turn, calls animateFromOrigin(this, animationStep, animationPauseTime), as shown below.

```java
public void animateFromOrigin() {
    Runnable animateCommand = new AShuttleAnimationCommand(this, animationStep, animationPauseTime);
    Thread thread = new Thread(animateCommand);
    thread.start();
}
```

Thus, we have managed to execute this call in a separate thread.

The following figure shows the difference between executing this call in the user-interface thread and in a separate thread.
In the first case, the parameterless animateFromOrigin() waits for the method call to terminate. Thus what it prints is correct. In the separate thread case, it creates a separate thread to make the method call. It does not wait for the call to complete. Thus, what it prints is not correct! Thus, the first method call blocks the caller while the second one does not. A blocking call corresponds to waiting at the counter of a fast food restaurant for the food to be delivered to you, while a non-blocking call corresponds to giving the order in a fancier restaurant and returning to your seat without collecting the food, which will later be delivered by a waiter when it is done. Similarly, a blocking call corresponds to a professor waiting for a student to answer a question, while a non-blocking call corresponds to asking the student to answer the question as a homework exercise. As these analogies show, if the caller needs to know the progress of a non-blocking call, it somehow needs to be explicitly notified about the progress. Let us continue with the animation example to illustrate the need for this notification.

**Incremental Display Update**

If we execute the new, thread-creating, version of the parameterless animateFromOrigin(), the user-interface will not freeze while the loops execute. However, as before, we will not see intermediate positions of the shuttle. In other words, the shuttle will appear not to move.

As mentioned before, normally ObjectEditor updates the display at the end of the execution of each method it calls. In the case of an animating method, ObjectEditor should update the display after each animation step. However, it does not know when an animation step has completed. Therefore, after each animation step, the animation method should explicitly tell ObjectEditor that the state displayed has changed. In other words, the object containing the animation method should behave as an observable that allows ObjectEditor and other observers to be registered and informs them whenever the animated state changes. Thus, the methods that change the shuttle location should notify all of the stored observers about the changed property. Consider how we can animate the X location – animation of the Y location is similar.

The X location is changed by the statement:

```java
setLabelX(windowX(curX)); // need to update display
```
in the first loop. As the statement does not directly change the shuttle location, we need to look at the implementation of `setLabelX`:

```java
void setLabelX(int x) {
    Point oldLocation = shuttleLabel.getLocation();
    Point newLocation = new ACartesianPoint(x, oldLocation.getY());
    shuttleLabel.setLocation(newLocation);
}
```

Again, the method does not directly change the shuttle location. Therefore, we need to look at the implementation of `setLocation` in class `ALabel`:

```java
public void setLocation(Point newVal) {
    location = newVal;
}
```

This is the method in which the shuttle location changes. Therefore, we need to make its class, `ALabel`, an observable in the fashion described above. We can add the `addPropertyChangeListener` and `notifyAllListeners` implementations given above without any changes to the class, as these are standard and do not depend on the property being changed. Now we can call `notifyAllListeners` from `setLocation`:

```java
public void setLocation(Point newVal) {
    Point oldVal = location;
    location = newVal;
    notifyAllListeners(new PropertyChangeEvent(
        this, "Location", oldVal, newVal));
}
```

Here, `setLocation` assigns the instance variable, `location`, a new instance of `ACartesianPoint`. Suppose that `Point` was not immutable, that is, it provided setter methods for the X and Y properties. In this case, if `setLocation` simply changed the X and Y coordinate of the existing location, we would need to change the `setX` and `setY` methods of `ACartesianPoint` class in the manner described above.

With these changes, our animation finally works. If we invoke the new `animateFromOrigin()`, the shuttle starts animating from the origin towards its current position. Let us trace this call to understand what happens.

When we execute the parameterless `animateFromOrigin()`, it creates a new thread, which starts executing the parameterless `animateFromOrigin()`, as shown below.
The method resets the label position, and notifies ObjectEditor, which moves the label to the origin.

It then executes the first loop, which repeatedly executes the steps of incrementing the Y position, notifying ObjectEditor, and sleeping, thereby gradually moving the shuttle along the Y axis.
Synchronization of Concurrent Threads

Let us get more adventurous and see what would happen if, while the animating method is sleeping, we execute the parametless `animateFromOrigin()` again from the ObjectEditor menu. This method will create another thread, which will execute the same code as the previous thread. Thus, the parameterized `animateFromOrigin()` will be executed by two interleaving/concurrent threads, as shown below.

Let us assume that the execution of these threads is interleaved on a single CPU. As the first thread is currently sleeping, the CPU will execute the first thread. As this thread has made a different call to the parameterized `animateFromOrigin()`, it gets its own copy of the local variables, `curX` and `curY`, in this:

```java
int curX = 0;
int curY = 0;
setLabelX(windowX(curX));
setLabelY(windowY(curY));
while (curY < getShuttleY()) {
    // loop make sure we don't go past final Y position
    sleep(animationPauseTime);
    curY += animationStep;
   setLabelY(windowY(curY));
}
// move to destination Y position
```
method. It sets these variables to zero and then copies their values to the X and Y coordinates of the global shuttle label. As a result, the shuttle will move back to the origin.

Next it will sleep. Thus, at this point, both threads are sleeping. The first thread will wake up first, and resume from where it left out. It will set the Y coordinate of the shuttle to its copy of the curY variable, thus causing the shuttle to jump to this location.

Thus, a single shuttle will switch between the two animations – not something we desire. The reason is that the two interleaving animation threads access the same global variable – in this case the shuttle label, thereby interfering with each other.

It seems we can easily avoid this problem by using a global boolean variable that is set to true at the start of the animation and to false at the end of the animation. If a thread finds the variable to be true, it repeatedly executes the steps of sleeping and checking the variable until it finds the variable false, as shown below.
boolean animationInProgress = false;

public void animateFromOrigin(int animationStep, int animationPauseTime) {
    while (animationInProgress)
        sleep(POLLING_TIME);
    animationInProgress = true;
    //animation code
    ...
    animationInProgress = false;
}

As it turns out, this solution does not work, because both threads might try to simultaneously set the variable. In fact, it is not possible for us to write library code to prevent this problem. Therefore, Java provides language support for preventing this problem. If we want a method to be executed by at most one thread at one time, we should declare it as synchronized, as shown below.

    public synchronized void animateFromOrigin(int animationStep, int animationPauseTime) {
            int curX = 0;
            ...
    }

If a thread tries to execute the method while some other thread is executing it, Java makes the second thread wait until the first thread finishes executing the thread. This is displayed in Figure 12 (bottom), which shows Thread-3 and Thread-4 executing the two calls to animateFromOrigin. As we see, each thread is associated with its own stack of calls, with a calling method waiting for a called method to finish. The call at the top of both stacks is animateFromOrigin, the ones below are calls to ObjectEditor code, about which we don’t have to worry. Figure 12 (bottom) shows that the two calls are executing at different locations of animateFromOrigin. The next statement to be executed by Thread 3 is the sleep call in the first loop, while the next statement to be executed by Thread 4 is the very first statement of the method.

As the thread numberings imply, these are not the only threads in the system. One of the other two threads is the ObjectEditor thread that processes user commands and updates the display. The others thread(s) are system threads that do “garbage collection,” that is, gets rid of object we no longer need and other book-keeping/clean-up activities.
As it turns out, the **synchronized** keyword cannot be used in a method declared in an interface. Thus, in the interface of the class, we must declare the header of `animateFromOrigin` as:

```java
public void animateFromOrigin()
```

even though, in the implementation of the interface, we declare it as:

```java
public synchronized void animateFromOrigin()
```

Normally, Java requires matching of all components of corresponding method headers in interfaces and the classes implementing it, but not in this case, probably to give the implementers of an interface the flexibility of deciding if they want to allow for concurrency and pay the cost of synchronization.

When a class declares several synchronized methods, only one of them can execute at any time. Thus, if we make `setShuttleX()` also synchronized, and a thread is executing the synchronized `animateFromOrigin()`, the a thread that executes `setShuttleX()` waits until the first thread finishes executing `animateFromOrigin()`. This is exactly what we want, as both methods manipulate a global label, and would interfere with each other if executed by two interleaving or concurrent threads.

What if, instead of making the parameterized `animateFromOrigin()` synchronized, we made the parameterless `animateFromOrigin()` synchronized?

**Figure 5. Making a call to animateFromOrigin before the previous one has finished**
This method simply creates threads. Serializing access to this method does not achieve our goal, as the threads created by this method are then free to execute the parameterized animateFromOrigin() simultaneously. Put in another way, we make a method synchronized if it accesses some global variable. However, this method does not access any global variable. More intuitively, the goal of synchronized is to make the caller wait. As this method creates threads to do its work, it does not block.

Thus, the synchronized keyword above serves no purpose. We should, as before, make the parameterized animateFromOrigin() synchronized.

This example shows it is useful to allow only a subset of the methods of a class to be synchronized.

**Summarizing Animation**

In general, an animating method performs one or more animation steps, where an animation step changes one or more animating (graphical) properties such as size, location, icon of one or more graphical objects and then pauses execution for some time. Execution can be paused using busy waiting or a sleep call provided by Java/operating system. Busy waiting has the problem that it is platform-specific and does not allow other activity to proceed while the animation is paused. Therefore using the sleep call is preferable.

After each animation step, all displays of the animation must be updated. The observable-observer concept can be used to ensure these updates are made. This means that we must ensure that for each graphical property changed by the animation, the class of the property allows observers to be registered and the setter of the property informs the observers about the update. ObjectEditor requires the JaveBeans observer-observer approach based around the PropertyChangeListener interface.

An animating method should be executed in a separate thread as otherwise the user-interface thread will wait for it to finish execution before performing any screen update. This means that it is possible to start multiple executions of the method concurrently. We should use the keyword synchronized in the declaration of the method to ensure that it is executed serially by the thread, that is, to ensure that if a thread is in the middle of executing the method, other threads wait for it to finish. We should use the synchronized keyword also in all other methods that access the global variables accessed by the animation method.

In general, a method that performs the animation steps and a method that changes the value of some animating property may be in different classes such as AnAnimatingShuttleLocation and ALabel.

A thread is created by passing to the constructor of the Thread class a command object that encapsulates the method call to be executed by the thread.
Undoable/Re-doable Command Objects

Command objects have applications not only in the creation of threads but also in the implementation of undo and redo. In fact, they were first discovered in the context of undo/redo. To illustrate, let us add the simplest possible undo/redo support to ABMISpreadsheet. Let us say we first change the weight property, then the height property, and execute undo. The undo command will undo the change weight command. However, if we execute undo again, nothing happens. This is because our implementation can undo only the last command. Put another way, the history of undoable/redoable commands it maintains consists of only one command. Thus, if the next command is a redo, we re-execute the set weight command. If the next command is another redo, it has no effect.

Notice that in the description of the undo/redo semantics, we talked about (re)executing and undoing the set weight command. This usage of ordinary English illustrates the role command objects have in undo and redo. The command objects we see here will be extensions of the one we saw in the context of threads. They will not only support an execute() operation but also an undo() operation. Moreover, the execute() operation will be used to both execute for the first time and re-execute the operation call encapsulated by the command object. As Java does not define such a command, let us create our own interface to do so.

```java
public interface Command {
    public void execute();
    public void undo();
}
```

The undo/redo commands execute by the user will not themselves be associated with command objects as they will be processed by calling the undo() / redo() operations on regular command objects.
Notice also that the description of the undo/redo mechanism was independent of the specific command, as long as the command provided an `execute()` and `undo()` operation. The fact that the history was a single command and that user’s invocation of undo/redo causes the execution of the `undo()`/`redo()` operation on the undoable command was independent of the implementation of these two operations. It is common to say that Word and PowerPoint provide the same undo/redo mechanism. This does not mean that the commands undone/redone by them are the same. It means they add and remove commands from their history in the same way and invoke the `execute()` and `undo()` operations on these commands in the same way. Put another way, the only difference between the two implementations is in the commands processed by them.

This means it should be possible to create an undo/redo implementation that is independent of the exact commands, and depends only on the Command interface described above. This is analogous to creating a Thread implementation that is independent of the command object passed to its constructor, and depends only on the Runnable interface.

The following interface describes the methods of such an undoer/redoer, to which we will refer to simply as an undoer.

```java
public interface Undoer {
    public void undo();
    public void execute(Command command);
    public void redo();
}
```

The `execute()` method is invoked in the undoer to submit an unexecuted command to it. The undoer responds to it by storing it and executing it for the first time. The `undo()` and `redo()` methods are invoked in it to undo/redo the last executed/undone command. Different implementations of this interface will differ in the size of the command history they create and when they clear the history (for example on file save or never). Our simple undoer keeps a single history of a single command, as shown below.
Because the history is of size 1, a single variable, lastCommand, is used for it. Both the redo and undo operations do nothing if this variable is uninitialized. Moreover, the redo operation does nothing if the previous operation was not an undo, and the undo operation does nothing if the previous operation was an undo. A variable, lastOperationWasUndo determines if the last operation was an undo.

Let us see how we can use this class to support undo in ABMISpreadsheet. Rather than changing this class, let us add undo awareness to a new class that delegates to an implementation of BMISpreadsheet such as ABMISpreadsheet, and an instance of Undoer such as LastCommandUndoer. These two instances are passed to class through a constructor, and stored in bmiSpreadsheet and undoer, respectively. The read methods of this class are not undoable, so they are delegated to bmiSpreadsheet without additional processing. The write methods, on the other hand, are undoable. Therefore, instead of directly delegating to bmiSpreadsheet, these methods create appropriate command objects, which are then sent to undoer for execution. The target object passed to the constructor of each command object is bmiSpreadsheet, not this, the delegator, as passing this would result in an infinite recursion: To execute the command, the undoer would execute a write method in the delegator, which would send another command to the undoer, which would execute it again by calling the same write method, and so on.

```java
public class LastCommandUndoer implements Undoer {
    Command lastCommand = null;
    boolean lastOperationWasUndo = false;
    public void execute (Command c){
        c.execute();
        lastCommand = c;
        lastOperationWasUndo = false;
    }
    public void undo() {
        if (lastOperationWasUndo || lastCommand == null)
            return;
        lastCommand.undo();
        lastOperationWasUndo = true;
    }
    public void redo() {
        if (!lastOperationWasUndo || lastCommand == null)
            return;
        lastCommand.execute();
        lastOperationWasUndo = false;
    }
}
```
This class also provides undo() and redo() methods, invoked in the figures above from the menu, which simply call the corresponding methods in the undoer.

The command objects created by the two write methods are instances of two different classes.
Like all command classes, such as AShuttleAnimationCommand we saw earlier, each of these classes encapsulates an embedded operation, and provides a constructor that takes the target object on which the operation is to be invoked and the actual arguments of the operation. The embedded operation in ASetHeightCommand (ASetWeightCommand) is setHeight() (setWeight()). The target object in both cases is an instance of BMISpreadsheet, which defines the embedded operation, and the actual argument is a double representing the new height/weight. As expected, the execute operation simply calls the embedded operation on the target object with the actual argument. The undo operation is more interesting. It must execute the inverse of the embedded operation invocation. When the embedded operation is a setter, the inverse of its invocation is another invocation of it with the value of the associated property before the first invocation was made. Therefore, the constructor of each command class invokes the corresponding getter in the target object to record the value of the property before it is changed by execute(). The undo() method simply calls the embedded operation with this value.

As we see above, the only difference between the two command classes is the setter and getters they invoke. It is possible to use reflection to combine their implementations. In fact, this is exactly what ObjectEditor does, providing a single command class to undo all setters.

Finally, we can write the following main method to create the user interface shown earlier.

```java
public static void main (String[] args) {
    ObjectEditor.edit (new AnUndoableBMISpreadsheet (new ABMISpreadsheet (1.77, 75), new LastCommandUndoer ());
}
```

**Proxy and Undo Pattern**

Our undo implementation involves several classes: the original class ABMISpreadsheet, the undo-aware class, AnUndoableBMISpreadsheet, the undoer, Undoer, the two command classes, ASetWeightCommand and ASetHeightCommand. Together, they follow a general undo pattern, shown below.
Here, we refer to the class whose methods are to be undone as an *undoable*. In our example, the class `ABMISpreadsheet` is an undoable. We do not add any undo awareness directly to the undoable. Instead, we add this awareness to a proxy class we create for it. A proxy class, like an adapter class, is a class that sits between some subject class and its clients.

While an adapter class filters out or transforms the methods of the subject class, thereby providing a different interface than the subject class, a proxy implements an extension of the subject interface. As a result, a client may not even be aware that it is interacting through the proxy. In our example, `AnUndoableBMISpreadsheet` extends the interface of `BMISpreadsheet` with the undo() and redo() methods. Like an adapter, a proxy may be delegate to or inherit from the subject class, with delegation providing more flexibility. `AnUndoableBMISpreadsheet` is a delegating proxy, which allows it to provide undo/redo support for any implementation of `BMISpreadsheet` such as `AnotherBMISpreadsheet`. This would not have been possible had it been a subclass of `ABMISpreadsheet`.

We did not implement the entire undo/redo functionality in the proxy. Instead, we implemented most of it in the undoer and command objects. This approach allows us to change the undoer without changing the proxy object or the command object. For instance, we can replace the simplistic `LastCommandUndoer` with a more sophisticated `HistoryUndoer` that stores a variable length history of undordable/redoable commands.
At one time, there was significant debate regarding the undo/redo semantics. Thus, it made sense to experiment with different undoers. Today, the common applications such as word processors, programming environments, and graphics applications all provide the same undo/redo semantics. Thus, it does not make much practical sense to change the undoer of an application. However, by keeping the undoer independent of the application object, we can use the same undoer for different kinds of undoable objects and their undo proxies. For example, our LastCommandUndoer can be used to undo methods of ACartesianPoint.

As long as we define an appropriate proxy for and command objects for it, we can use exactly the same undoer class.

Even more interesting, the same undoer instance can be used simultaneously to undo and redo methods of objects of different classes such as ABMISpreadsheet and ACartesianPoint. The operations invoked on all of these objects would be maintained by this single undoer.

This is the biggest reason for having a separate undoer. Most applications allow users to manipulate a variety of objects. For example, a Word processor allows users to manipulate text, graphics, and various kinds of preferences. A single undo history is maintained for operations invoked on all of these objects.

To better understand how the various components of the undo pattern work, let us return to the BMI spreadsheet example. Consider what happens when the user changes the weight field to 66.0.
Let us now consider what happens when the user executes the undo command.

1. ObjectEditor calls setWeight() in AnUndoableBMISpreadsheet.
2. setWeight(), in turn, creates an instance of ASetWeightCommand and asks the undoer to execute it.
3. The undoer invokes the execute() method on the command.
4. The command invokes the setWeight() method with the value 66.0 on the undoable BMISpreadsheet.
ObjectEditor calls the undo method in AnUndoableBMISpreadsheet.

This method calls the undo method in the undoer.

The undoer undo method calls the undo method in the command object.

The command object calls the setWeight() method in the undoable BMISspreadsheet with the value of weight before the command was executed: 75.0.

Finally, consider what happens when the user now calls redo.

ObjectEditor calls the redo() method in AnUndoableBMISpreadsheet.

This method calls the redo() method in the undoer.

The undoer redo method calls the execute() method in the command object.

The command object calls the setWeight() method in the undoable BMISspreadsheet with the value of the argument passed to its constructor: 66.0.

Thus, we see an important difference between the command object passed to an Undoer and the one passed to a Thread instance. The latter calls the execute() method of the command a single time. The former calls it once when the command is created and then each time the command is undone.

**Inheritance vs. Delegation in Proxies and Command Objects**

Our undo pattern includes the proxy pattern, which is a very common pattern. Proxies are provided for adding logging, collaboration, caching, web server redirection, access control, and several functions to some existing subject such as a web server.
It is possible to create a proxy by inheriting from or delegating to the subject class.

**Inheritance-based Proxy**

- **ABMISpreadsheet** IS-A **BMISpreadsheet**
- **AnUndoableBMISpreadsheet** HAS-A **ABMISpreadsheet**
- **Proxy Class** HAS-A **Subject Class**
- **Subject Class** implements **Subject Interface**
- **ABMISpreadsheet** implements **BMISpreadsheet**
- **AnUndoableBMISpreadsheet** implements **ABMISpreadsheet**

**Delegating Proxy**

Delegation has the advantage that it allows the proxy to work for all implementations of the subject interface. For example, we can create one caching proxy for all web server implementations. On the other hand it requires more work.

The choice between delegation and inheritance must also be made for command objects. In the examples shown above, a command invoker HAS-A reference to the command objects.

**Command Invoker** HAS-A **Command Invoker Interface**
- **Command Class** implements **Command Interface**
- **Command Invoker** implements **Command Invoker Interface**
- **Command Invoker Interface** implements **Command Interface**
For example, AHistoryUndoer has a reference to a ASetWieght command instance.

Similarly, a Thred has a reference to AShuttleAnimationCommand.

The designers, initially, used an inheritance based approach in which a command object was a subclass of the class that invoked it.

This approach is still provides by Thread. We can make AShuttleAnimationCommand a subclass of Thread. Thread implements the Runnable interface by providing a null implementation of run(). If it is not passed a command object in its constructor, its start() method invokes run() on itself (this). If a command class extends it, then it is the run() method in this class that is executed. This approach makes no logical sense as an IS-A relationship does not exist between a command and a command invoker. However, it is still used by many programmers because of the convenience of inheritance.
The undo/redo semantics acceptable to users are fairly standard – the popular applications we use study such as the Microsoft Office applications an