**Prerequisite**

- Animation MVC
Topics

- Animation
- Command Object
  - Object representing an action invocation such as “Do your homework”.
- Threads
  - Support non blocking action invocation.
ANIMATION AND MVC
public void animateFromOrigin(PlottedShuttle shuttle, 
        int animationStep, int animationPauseTime) {
    int originalX = shuttle.getShuttleX();
    int originalY = shuttle.getShuttleY();
    int curX = 0;
    int curY = 0;
    shuttle.setShuttleX(curX);
    shuttle.setShuttleY(curY);
    animateYFromOrigin(shuttle, animationStep, 
                       animationPauseTime, curY, originalY);
    animateXFromOrigin(shuttle, animationStep, 
                       animationPauseTime, curX, originalX);
}
protected void animateYFromOrigin(PlottedShuttle shuttle,
   int animationStep, int animationPauseTime,
   int startY, int endY) {

    // make sure we don’t go past final Y position
    while (startY < endY) {
        ThreadSupport.sleep(animationPauseTime);
        startY += animationStep;
        shuttle.setShuttleY(startY);
    }
    // move to destination Y position
    shuttle.setShuttleY(endY);
}
Observable Architecture

AnObservable PlottedShuttle

setShuttleX(Y)()

repaint()

Component

paint()

AShuttleAnimator

animateFromOrigin

Main Class

main
ANIMATION AND MVC
TWO INDEPENDENT ANIMATIONS
public static void main(String[] args) {
    PlottedShuttle shuttle1 = new AnObservablePlottedShuttle(50, 100);
    OEFrame oeFrame1 = ObjectEditor.edit(shuttle1);
    oeFrame1.hideMainPanel();
    oeFrame1.setLocation(0, 0);
    oeFrame1.setSize(400, 400);
    PlottedShuttle shuttle2 = new AnObservablePlottedShuttle(100, 50);
    OEFrame oeFrame2 = ObjectEditor.edit(shuttle2);
    oeFrame2.hideMainPanel();
    oeFrame2.setLocation(400, 0);
    oeFrame2.setSize(400, 400);
    ShuttleAnimator shuttleAnimator1 = new AShuttleAnimator();
    ShuttleAnimator shuttleAnimator2 = new AShuttleAnimator();
    shuttleAnimator1.animateFromOrigin(shuttle1, 5, 100);
    shuttleAnimator2.animateFromOrigin(shuttle2, 5, 100);
}
public static void main(String[] args) {
    PlottedShuttle shuttle1 = new AnObservablePlottedShuttle(50, 100);
    OEFrame oeFrame1 = ObjectEditor.edit(shuttle1);
    oeFrame1.hideMainPanel();
    oeFrame1.setLocation(0, 0);
    oeFrame1.setSize(400, 400);
    PlottedShuttle shuttle2 = new AnObservablePlottedShuttle(100, 50);
    OEFrame oeFrame2 = ObjectEditor.edit(shuttle2);
    oeFrame2.hideMainPanel();
    oeFrame2.setLocation(400, 0);
    oeFrame2.setSize(400, 400);
    ShuttleAnimator shuttleAnimator1 = new AShuttleAnimator();
    ShuttleAnimator shuttleAnimator2 = new AShuttleAnimator();
    concurrentDemoShuttleAnimation(shuttleAnimator1, shuttle1);
    concurrentDemoShuttleAnimation(shuttleAnimator2, shuttle2);
}
**INTERLEAVING EXAMPLE**

```java
while (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}
```
while (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}
INTERLEAVING EXAMPLE

```java
while (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}

while (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}
```
INTERLEAVING EXAMPLE

```java
while (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}
```
while (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}

while (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}
INTERLEAVING EXAMPLE

```java
while (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}
```
INTERLEAVING EXAMPLE

```java
while (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}
```

```java
while (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}
```
INTERLEAVING EXAMPLE

```java
while (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}
```

```java
while (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}
```
INTERLEAVING EXAMPLE

```java
while (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}
```
INTERLEAVING EXAMPLE

```java
while (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}
```

```java
while (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}
```
Each interleaved activity is associated with its own program counter, which marks the next statement to be executed for that activity.

At any one time, a single CPU executes the statement of only one activity called the current activity.

The CPU does not wait for the current activity to complete before switching to another activity.
INTERLEAVING ACTIVITIES IN REAL LIFE

Smile at baby
Read email
Feed dog
Feed baby
Reply to email
Hug baby

Threads can have different priorities
**INTERLEAVED PROCESSES**

Process is an interleaved activity created by the Operating System each time a main method is run.
INTERLEAVED THREADS

Thread is an interleaved activity within a process. Threads in a process work cooperatively to do the job of the process. Java creates several threads when running our program for UI processing, garbage collection, main method, ....
THE PROGRAM COUNTER OF OTHER THREAD

PC of thread 1

Daemon Thread [AWT-Windows] (Running)
Thread [AWT-EventQueue-0] (Running)
Thread [ToolTip Thread] (Running)
Thread [ToolTip Thread] (Running)
Thread [DestroyJavaVM] (Running)

Thread [Shuttle Animation 2] (Suspended (breakpoint at line 11 in AShuttleAnimator)
  AShuttleAnimator.animateFromOrigin(PlottedShuttle, int, int) line: 11
  AShuttleAnimationCommand.run() line: 22
  Thread.run() line: not available

Thread [Shuttle Animation 1] (Suspended (breakpoint at line 11 in AShuttleAnimator)
  AShuttleAnimator.animateFromOrigin(PlottedShuttle, int, int) line: 11
  AShuttleAnimationCommand.run() line: 22
  Thread.run() line: not available

Program Files\Java\jre6\bin\javaw.exe (Apr 14, 2012 8:47:13 PM)
**Concurrent Activities**

```java
while (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}
```

```java
while (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}
```
While (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}

Processes/threads can execute concurrently in multi-processor, multi-core computers.

On single-core, single-processor, a single processor/core interleaves their execution.
Concurrency in “Real” Life
while (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}

while (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}

while (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}
while (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}

while (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}

while (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}
while (curY < originalY) {
    ThreadSupport.sleep(animationPauseTime);
    curY += animationStep;
    shuttle.setShuttleY(curY);
}
CONCURRENCY AND INTERLEAVING

Two hands serving three balls ~ two cores serving three threads.

While a ball is in the air the other balls can be served ~ while a thread is waiting for user input or sleeping, others can be served.
INTERLEAVING ACTIVITIES

- Each interleaved activity is associated with its own program counter, which marks the next statement to be executed for that activity.
- At any one time, a single CPU executes the statement of only one activity called the current activity.
- The CPU does not wait for the current activity to complete before switching to another activity.

How should we create a separate thread to execute `animateFromOrigin`?

- **Java creates the threads.**
- Requires us to tell it what method to execute.
public static void main(String[] args) {
    PlottedShuttle shuttle1 = new AnObservablePlottedShuttle(50, 100);
    OEFrame oeFrame1 = ObjectEditor.edit(shuttle1);
    oeFrame1.hideMainPanel();
    oeFrame1.setLocation(0, 0);
    oeFrame1.setSize(400, 400);
    PlottedShuttle shuttle2 = new AnObservablePlottedShuttle(100, 50);
    OEFrame oeFrame2 = ObjectEditor.edit(shuttle2);
    oeFrame2.hideMainPanel();
    oeFrame2.setLocation(400, 0);
    oeFrame2.setSize(400, 400);
    ShuttleAnimator shuttleAnimator1 = new AShuttleAnimator();
    ShuttleAnimator shuttleAnimator2 = new AShuttleAnimator();
    shuttleAnimator1.animateFromOrigin(shuttle1, 5, 100);
    shuttleAnimator2.animateFromOrigin(shuttle2, 5, 100);
}
A thread is an object that can be started, suspended, resumed, interrupted while sleeping, given lower/higher priority ...
A thread is an object that can be started, suspended, resumed, interrupted while sleeping, given lower/higher priority...

How to create a startable thread?
INDEPENDENT ANIMATIONS?

```java
public static void main(String[] args) {
    PlottedShuttle shuttle1 = new AnObservablePlottedShuttle(50, 100);
    OEFrame oeFrame1 = ObjectEditor.edit(shuttle1);
    oeFrame1.hideMainPanel();
    oeFrame1.setLocation(0, 0);
    oeFrame1.setSize(400, 400);
    PlottedShuttle shuttle2 = new AnObservablePlottedShuttle(100, 50);
    OEFrame oeFrame2 = ObjectEditor.edit(shuttle2);
    oeFrame2.hideMainPanel();
    oeFrame2.setLocation(400, 0);
    oeFrame2.setSize(400, 400);
    ShuttleAnimator shuttleAnimator1 = new AShuttleAnimator();
    ShuttleAnimator shuttleAnimator2 = new AShuttleAnimator();
    shuttleAnimator1.animateFromOrigin(shuttle1, 5, 100);
    shuttleAnimator2.animateFromOrigin(shuttle2, 5, 100);
}
```

Need multitasking
**Predefined Main Thread**

Each thread has a stack of method calls

A call to a method $M^2$ by $M^1$ gets pushed on top of the call to $M^1$ which blocks

The call to $M^2$ is popped when it returns, allowing $M^1$ to continue

Creating a new thread means asking a Thread object to create a new stack so no blocking occurs

→ Must specify to (the constructor of) the Thread object the first method call in the stack (method + parameters + object).
**Action Object (Method Object)**

- **execute (targetObject, params)**
- Provides an execute operation to perform some action.
- The execute operation takes as an argument the object on which the target operation is to invoked and an array of parameters of the target action.
- Can pass to a constructor a method object.
- A method that does not take method objects as arguments is a first-order function.
- A method that takes as an argument a method object is a second (higher) order function.
- A method that takes as an argument an N-order method object is an N+1 order method.
- Java reflection supports higher order functions.
**Using Reflection to Create Thread Object**

```java
public static void demoShuttleAnimation(
    ShuttleAnimator aShuttleAnimator, PlottedShuttle aShuttle) {
    aShuttleAnimator.animateFromOrigin(aShuttle, 5, 100);
}

public static void concurrentDemoShuttleAnimation(
    ShuttleAnimator aShuttleAnimator, PlottedShuttle aShuttle) {
    Thread thread = new Thread(aShuttleAnimator,
        animateFromOriginMethod, new Object[] {aShuttle, 5, 100});
    threadNumber++;
    thread.setName("SHUTTLE_THREAD_NAME + " + threadNumber);
    thread.start();
}
```

Reflection allows method parameters.

Exceptions can be thrown because a method can be invoked on an arbitrary object with arbitrary parameters.

A la grammar/vocabulary vs. phrase book.
public static void demoShuttleAnimation(
    ShuttleAnimator aShuttleAnimator, PlottedShuttle aShuttle) {
    aShuttleAnimator.animateFromOrigin(aShuttle, 5, 100);
}

public static void concurrentDemoShuttleAnimation(
    ShuttleAnimator aShuttleAnimator, PlottedShuttle aShuttle) {
    Thread thread = new Thread
        ((new AShuttleAnimationCommand
            (aShuttleAnimator, aShuttle, 5, 100));
    threadNumber++;
    thread.setName("SHUTTLE_THREAD_NAME + " + threadNumber);
    thread.start();
}

Bundle the target object and the three arguments into a single command object
associated with some action

The constructor of the command object checks that the target and parameters are
compatible with its action
ACTION OBJECT (METHOD OBJECT)

Use an object in which the action, target and params are bundled together.
**COMMAND OBJECT (METHOD OBJECT)**

### Action Object (Method Object)
- **execute**
  - (targetObject, params)

### Command (action with target + parameters)
- **execute ()**

### Constructor (targetObject, params)

**Provides a execute operation to perform some action.**

**The execute operation takes no arguments.**

**Constructor takes target object and parameters of operation as arguments.**

**Action is an operation that can be invoked on many different arguments**

**A command is a specific action invocation.**

**Action vs. command object ↔ “do” vs “Bob, do your homework”**

**Action vs. command object ↔ “move” vs “move Arthur 50 30”**
The interface is called Runnable instead of Command and the method is called run() instead of execute() because Thread designers probably did not realize they were using a command object.

The command object like an observable, model, view is a general design object that occurs in several contexts and was invented in the context of undo.
Runnable Implementation

```java
public class AShuttleAnimationCommand implements Runnable {
    ShuttleAnimator shuttleAnimator;
    PlottedShuttle shuttle;
    int animationStep;
    int animationPauseTime;
    public AShuttleAnimationCommand (ShuttleAnimator aShuttleAnimator,
                                      PlottedShuttle aShuttle,
                                      int anAnimationStep,
                                      int anAnimationPauseTime) {
        shuttleAnimator = aShuttleAnimator;
        shuttle = aShuttle;
        animationStep = anAnimationStep;
        animationPauseTime = anAnimationPauseTime;
    }
    public void run() {
        shuttleAnimator/animateFromOrigin(shuttle,
                         animationStep,    animationPauseTime);
    }
}
```

Action is hardwired: Separate implementation of Command Object Interface for each action.
public static void demoShuttleAnimation(ShuttleAnimator aShuttleAnimator, PlottedShuttle aShuttle) {
    aShuttleAnimator.animateFromOrigin(aShuttle, 5, 100);
}

public static void concurrentDemoShuttleAnimation(ShuttleAnimator aShuttleAnimator, PlottedShuttle aShuttle) {
    Thread thread = new Thread(new AShuttleAnimationCommand(aShuttleAnimator, aShuttle, 5, 100));
    threadNumber++;
    thread.setName(SHUTTLE_THREAD_NAME + " " + threadNumber);
    thread.start();
}

The run method calls animateFromOigin() on target object

Starts the thread, which Invokes the run method on the command object passed to constructor.

Creates a new Command

Creates a new Java thread
TWO SHUTTLES AND ANIMATORS, TWO THREADS

APlotted Shuttle

setShuttleX(Y)()

ASHuttleAnimator

animate Shuttle()

run()

Thread

start()

Main Class

main()
**Synchronous/Asynchronous Method Call**

```java
public static void demoShuttleAnimation(
    ShuttleAnimator aShuttleAnimator, PlottedShuttle aShuttle) {
    aShuttleAnimator.animateFromOrigin(aShuttle, 5, 100);
    System.out.println("animation finished");
}
```

```java
public static void concurrentDemoShuttleAnimation(
    ShuttleAnimator aShuttleAnimator, PlottedShuttle aShuttle) {
    Thread thread = new Thread(new AShuttleAnimationCommand
        (aShuttleAnimator, aShuttle, 5, 100));
    threadNumber++;
    thread.setName("SHUTTLE_THREAD_NAME + " + threadNumber);
    thread.start();
    System.out.println("animation finished");
}
```

Method caller waits for called method to finish.

Thread creator does not wait for created thread to finish.

Thread creation is typically the last step in a method call chain, so thread creator will probably be gone by the time the started thread executes.
Synchronous vs. Asynchronous

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>operation(&lt;parms&gt;)</td>
</tr>
<tr>
<td>animateFromOrigin();</td>
</tr>
<tr>
<td>System.out.println(“hello”)</td>
</tr>
</tbody>
</table>

**Synchronous**: Operation invoker **blocks** or waits until the operation finishes.

**Asynchronous**: Operation invoker does not block until completion.

Some other operation (e.g. observer notification) needed to wait for result or completion status.
Synchronous/Asynchronous Analogy

- **Synchronous**
  - I ask you a question in class and wait for answer.
  - I order food at subway and wait till my sandwich is made

- **Asynchronous**
  - I give you the question as homework.
  - I order food at a fancier restaurant with waiters.
**Command Design Pattern**

- **AnObservablePlotted Shuttle**
  - setShuttleX(Y)()

- **AShuttleAnimator**
  - animate Shuttle()

- **AShuttleAnimation Command**
  - run()

- **Thread**
  - start()

- **Main Class**
  - main()

---

**Command object**: Defines an execute method

**Command invoker**: invokes the execute method of a command object

**Command Client**: Instantiates the command object

---

- **Command object**: Defines an execute method
- **Command invoker**: invokes the execute method of a command object
- **Command Client**: Instantiates the command object

---

- **AnObservablePlotted Shuttle**
  - setShuttleX(Y)()

- **AShuttleAnimator**
  - animate Shuttle()

- **AShuttleAnimation Command**
  - run()

- **Thread**
  - start()

- **Main Class**
  - main()
THREADED ANIMATION DESIGN PATTERN

Animated object: performs each animation step, unaware of the animation

Animating object: Implements looping animation methods that take as parameters the animated object and optional animation controls

Animating command object: Its execute method calls an animating method in an animating object with parameters of the method. Constructor takes as parameters the animating object, animated object and animation controls.

Animating thread: Invoker of the execute method of an animating command object

Animating client: Client of an animating command object and creator and starter of an animating thread
THREAD-BASED ANIMATION DESIGN PATTERN

AnObservablePlottedShuttle

setShuttleX(Y)()

AShuttleAnimator

animateShuttle()

run()

AShuttleAnimationCommand

Thread

start()

MainClass

main()

Animated object

Animating object:

Animating command object

Combining objects?

Combining animated and animating object?
Combining Animated and Animating

**AnObservablePlotted Shuttle**
- `setShuttleX(Y)()`
- `animate Shuttle()`

**AShuttleAnimation Command**
- `run()`

**Thread**
- `start()`

**Main Class**
- `main()`

**combine animated and animating object?**

- **Animated object**
- **Animating object:**
- **Animating command object**
- **Animating thread**
- **Animating client**
Combining Animated and Animating

- Must duplicate animating code for AnotherPlottedShuttle

AnotherPlotted Shuttle

- setShuttleX(Y)()
- animate Shuttle()

- AShuttleAnimation Command

- run()

- Thread

- start()

- Main Class

- main()

- Animated object
- Animating object:
- Animating command object
- Animating thread
- Animating client

---

- Must duplicate animating code for AnotherPlottedShuttle

AnotherPlotted Shuttle

- setShuttleX(Y)()
- animate Shuttle()

- AShuttleAnimation Command

- run()

- Thread

- start()

- Main Class

- main()
THREAD-BASED ANIMATION DESIGN PATTERN

Original Code working with AnObservablePlotted Shuttle

main()

Thread

start()

Main Class

AShuttleAnimation Command

run()

Animated object

Animated object:

ANI

animate Shuttle()

run()

setShuttleX(Y)()
**Thread-based Animation Design Pattern**

- **AnotherPlottedShuttle**
  - `setShuttleX(Y)()`

- **AShuttleAnimator**
  - `animateShuttle()`

- **AShuttleAnimationCommand**
  - `run()`

- **Thread**
  - `start()`

- **Main Class**
  - `main()`

- **Animated object**
- **Animating object**:
- **Animating command object**
- **Animating thread**
- **Animating client**

**Reusing animating code for AnotherPlottedShuttle**
THREAD-BASED ANIMATION DESIGN PATTERN

Using a different animator for same animated object

Combine animating object and command?

AnotherPlotted Shuttle

setShuttleX(Y)()

AnotherShuttle Animator

animate Shuttle()

AShuttleAnimation Command

run()

Thread

start()

Main Class

main()

Animated object

Animating object

Animating command object

Animating thread

Animating client
Combining Animator and Command?

- AnObservablePlottedShuttle
  - setShuttleX(Y)()
  - animateShuttle()

- AShuttleAnimator
  - run()

- ShuttleAnimationThread 1
  - start()

- MainClass
  - main()

- Animated object
  - Animated object:
    - Animating object:
    - Animating command object
    - Animating thread
    - Animating client

May want forward and backward animation in one object
Or may want to run animation synchronously
Run executes asynchronously exactly one method
Our pattern allows a separate command object to be created for each method of an object that is to be executed asynchronously.
**Subclass Thread?**

- **AnObservablePlotted Shuttle**
  - `setShuttleX(Y)()`

- **AShuttleAnimator**
  - `animate Shuttle()`

- **Thread**
  - `start()`
  - `run()`

- **Main Class**
  - `main()`

**Animated object**

**Animating object:**

**Animating command object**

**Animating thread**

**Animating client**

Subclass Thread rather than give it a constructor argument

A command is executed by a thread and is not a thread!

Subclassed thread cannot inherit from another class

AnAnimating Command

- **IS-A**

AnAnimating Command

- **IS-A**

AShuttleAnimation Command

- **IS-A**

Thread

AShuttleAnimator

- **run()**

AnObservablePlotted Shuttle

- **run()**

Animated object
INHERITANCE AMONG COMMAND OBJECTS

Animated object

Animating object:

Animating command object

Animating thread

Animating client

Our pattern allows inheritance among Command objects

AnObservablePlottedShuttle

setShuttleX(Y)()

AnAnimatingCommand

AnObserver

setShuttleX(Y)()

AsShuttleAnimator

animateShuttle()

AsShuttleAnimationCommand

run()

MainClass

main()

Thread

start()

run()
Abusing IS-A and Instantiating Unnecessary Code

- **AnObservablePlottedShuttle**
  - `setShuttleX(Y)()`
  - **Animated object**

- **AShuttleAnimator**
  - `animateShuttle()`
  - **Animating object**

- **Parser/Undoer**
  - **HAS-A**
  - **Other objects such as parser and undoer might need to share a thread command objects**

- **AShuttleAnimationCommand**
  - `run()`
  - **Animating command object**

- **Thread**
  - `start()`
  - **Animating thread**

- **Main Class**
  - `main()`
  - **Animating client**

- **Unnecessary run code in these classes if we combine thread with command object**
**Not instantiating Unnecessary Code**

- **AnObservablePlotted Shuttle**
  - `setShuttleX(Y)()`

- **AShuttleAnimator**
  - `animate Shuttle()`

- **AShuttleAnimation Command**
  - `run()`

- **Thread**
  - `start()`
  - `run()`

- **Main Class**
  - `main()`

- **Parser/Undoer**
  - HAS-A

- **Animated object**

- **Animating object:**
  - AShuttleAnimator

- **Animating command object:**
  - run()

- **Animating thread**

- **Animating client**

No instantiation of unnecessary Thread code using our pattern
COMMAND OBJECTS VS. THREADS

- Threads can use command objects
- Command objects can be used in non-thread contexts
  - Undo/redo
  - Processing command interpreter commands
If we want multiple animations then we must create our own threads.
Threads use command objects.
Command objects represent method calls.
Animation Pattern

- Begin with the object to be animated
- Write one or more animating object with one or more looping methods to animate the animated object that take animation controls as arguments
- Write one or more implementations of Runnable, each of which takes the animation controls and the above two objects as parameters and calls a looping method of the animating objects
- In the main program or some model method, create one or more Thread instance, passing to the Thread constructor an instance of the Runnable command object.
- Execute the start() method on each Thread instance.