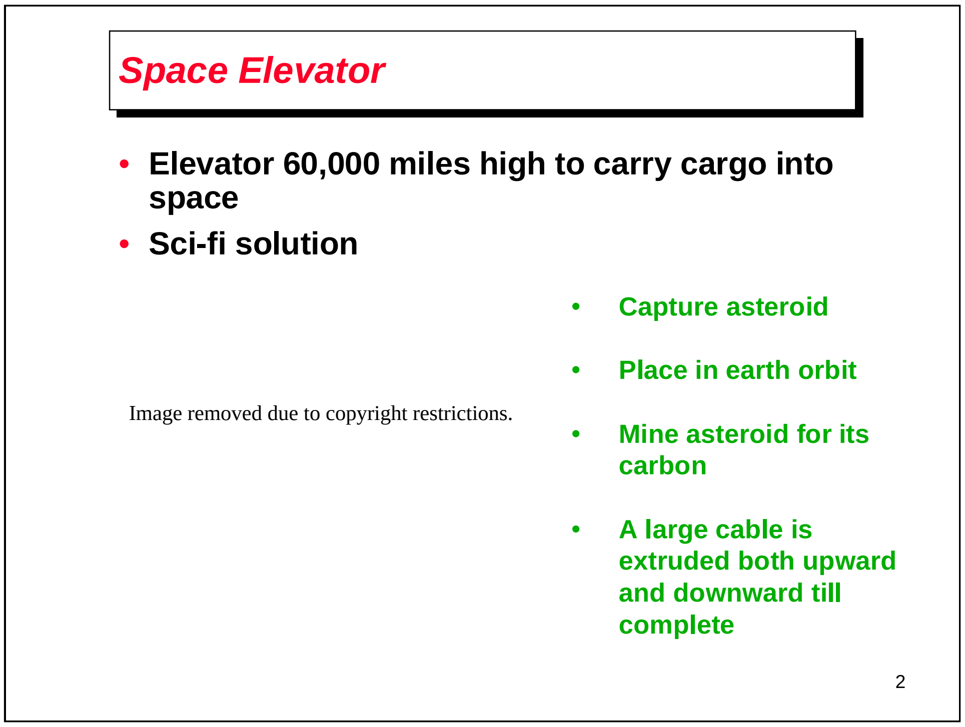
# 6.170 Design Project Experiences: Space Elevator Simulator Part I

Image removed due to copyright restrictions.



## *Space Elevator*

- **Elevator 60,000 miles high to carry cargo into space**
- **Sci-fi solution**

Image removed due to copyright restrictions.

- **Capture asteroid**
- **Place in earth orbit**
- **Mine asteroid for its carbon**
- **A large cable is extruded both upward and downward till complete**

## *Building a Space Elevator*

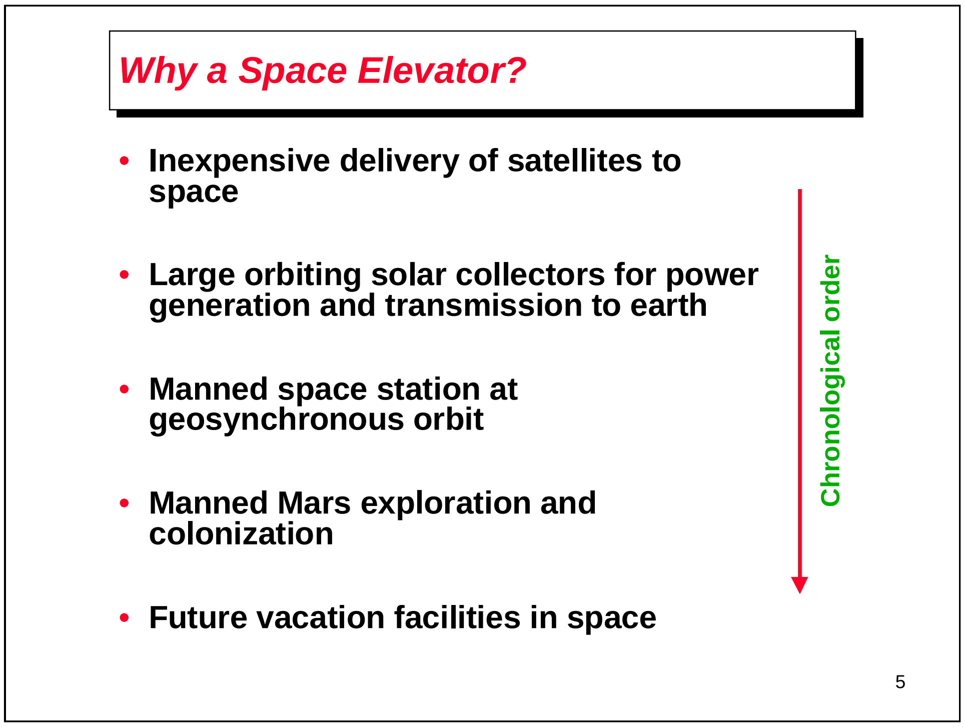
- Arthur C. Clarke, “The Fountains of Paradise”, 1978
  - Invents super strong monofilament carbon fibers and builds a cluster of cables (!)
  - Built a solid tower around the skeleton cables (??)
  - Ran the tower past geosynchronous orbit to a large counterweight
- With the invention of the carbon nanotube (1991), the concept is moving from science fiction to the fringes of reality
- Edwards and Westling, “The Space Elevator”, 2002 -- NASA-funded feasibility study.

3

## *“Realistic” Space Elevator*

Image removed
due to
copyright restrictions.

- **Spacecraft launched into geosynchronous (35,000km) orbit.**
- **Spacecraft lowers thin ribbon toward ground, and moves outward to keep it from falling, eventually ending up at 100,000 km.**
- **When ribbon reaches earth, it is tied to a base station (floating platform off the coast of Ecuador).**
- **To strengthen and widen the initial ribbon, climbers, powered by lasers from earth stitch on additional ribbons for 2½ years.**
- **Ribbon is 3 feet wide and supports 13 tons in completed elevator.**



## *Why a Space Elevator?*

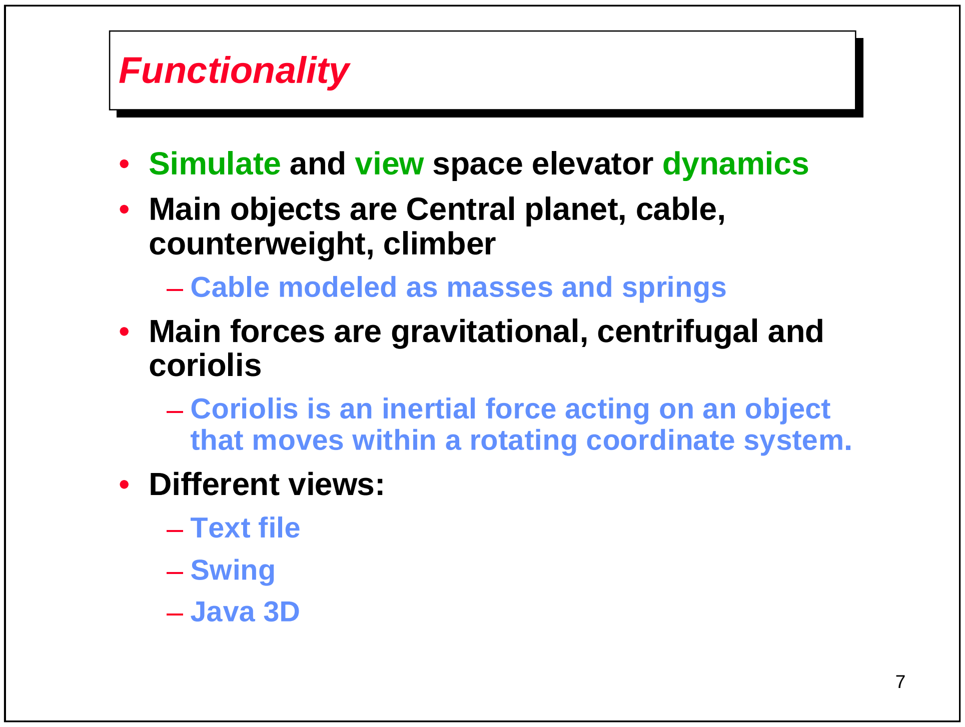
- **Inexpensive delivery of satellites to space**
- **Large orbiting solar collectors for power generation and transmission to earth**
- **Manned space station at geosynchronous orbit**
- **Manned Mars exploration and colonization**
- **Future vacation facilities in space**

**Chronological order**

5

## *Summer Project*

- **Obtain first-hand experience with 6.170-like project, a *space elevator simulator***
- **People & Schedule**
  - **Dwaine Clarke, Srini Devadas, Blaise Gassend, Daihyun Lim**
  - **Six weeks of off-and-on work to produce rudimentary [and buggy] implementation**
  - **Project finished up [and debugged] by Lee Lin and Matt Notowidigdo**



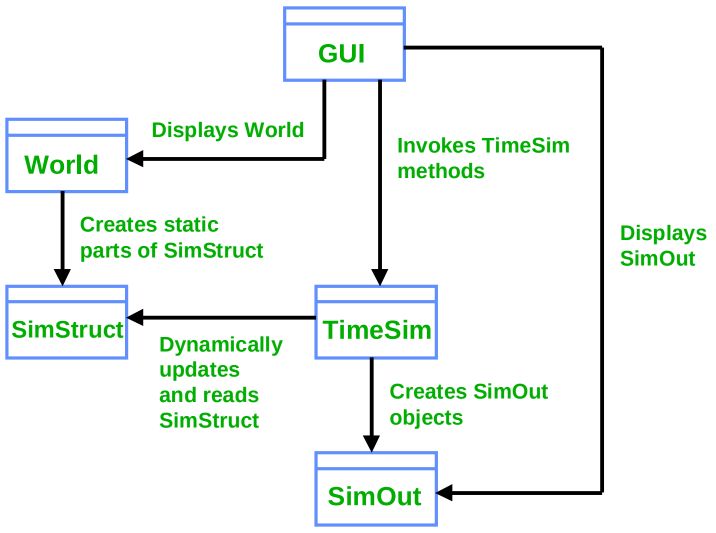
## Functionality

- Simulate and view space elevator dynamics
- Main objects are Central planet, cable, counterweight, climber
  - Cable modeled as masses and springs
- Main forces are gravitational, centrifugal and coriolis
  - Coriolis is an inertial force acting on an object that moves within a rotating coordinate system.
- Different views:
  - Text file
  - Swing
  - Java 3D

7

Because of Coriolis force, the object does not actually deviate from its path, but it appears to do so because of the motion of the coordinate system.

## Our First MDD



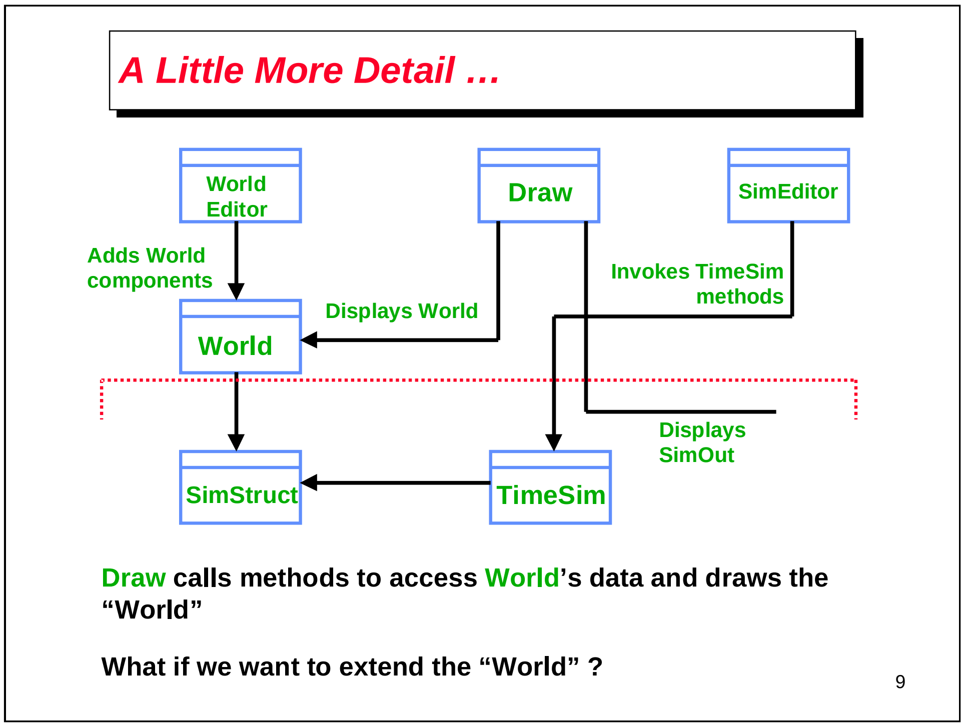
8

The World consists of all of the elements in the Space Elevator and space. This includes the Central Planet (earth), the cable, the climber and the counterweight, at the minimum.

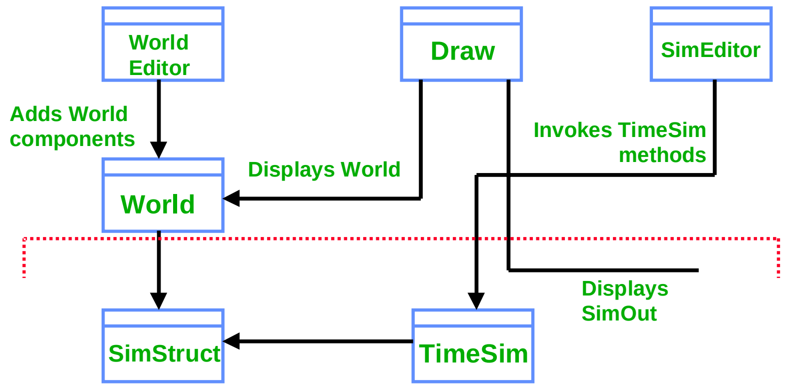
You can add other planets and also describe the cable as a collection of cables.

Simstruct is the basic simulation structure that is created by World and TimeSim. World objects turn into SimStruct objects. For example, a cable may be turned into a collection of masses and springs.

TimeSim incorporates methods to simulate World elements and includes the equations that model the physical forces. TimSim continually updates SimStruct during simulation. The output of TimeSim, that is, the statistics of the simulation is SimOut. Note that the positions and velocities of World elements are part of SimStruct. SimOut corresponds to statistics gathered during simulation, for example, the maximum force, minimum force, etc.



## A Little More Detail …

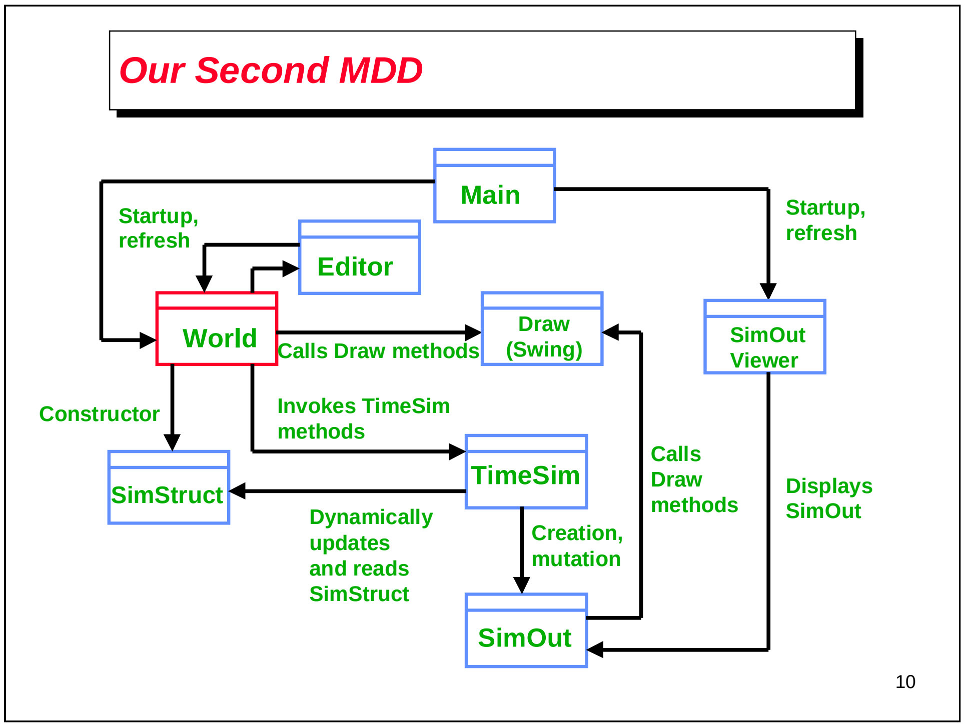


**Draw calls methods to access World's data and draws the "World"**

**What if we want to extend the "World" ?**

9

The problem is that it is not very extensible if we want to add different components to World, say the Moon, in addition to Central Planet, cable, etc.



## Our Second MDD

Let's have objects in the World call Draw methods and draw themselves. This makes for a more extensible world because we can add modules and add their draw methods.

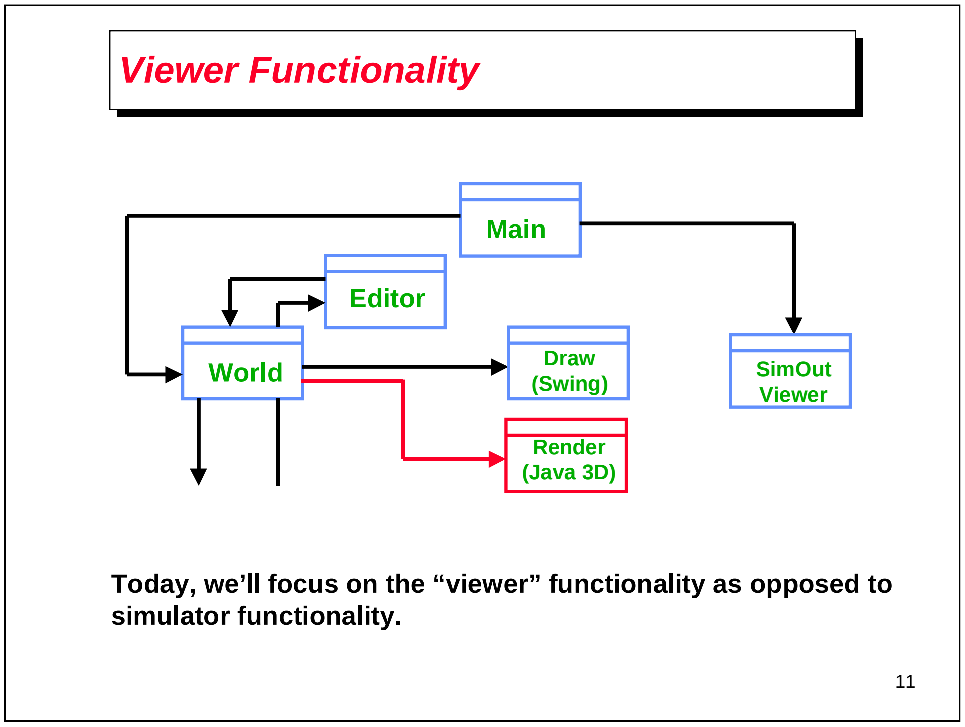
In general, if you have a module that is required to continually change within the lifetime of the software application, then it is a good idea to reduce the dependency ON that module.

You can make the extensible module depend on other less frequently changing modules.

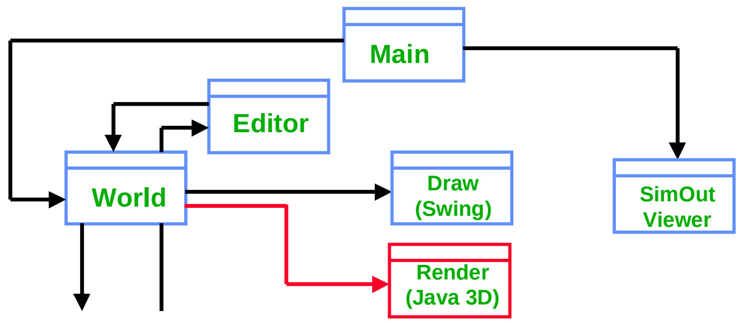
Similarly, we will have objects in World call the simulator methods to simulate their dynamics.

In this MDD it is clear that World is the central module/ADT. World and Editor are tightly coupled since the World will need to invoke Editor methods to edit its objects, and the editor will need to invoke World methods to update the objects with new ones, or update object positions.

World objects call methods in Draw and TimeSim to draw or simulate themselves.



## Viewer Functionality

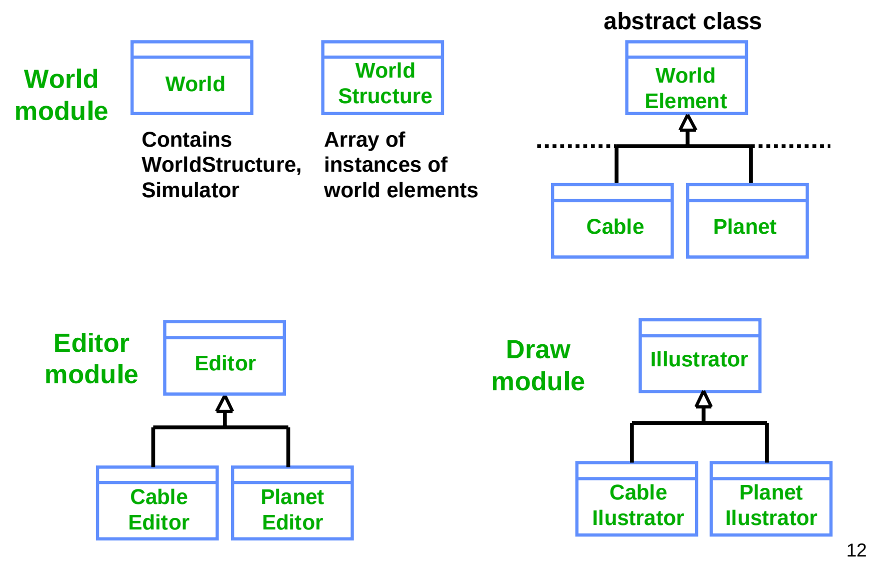


**Today, we'll focus on the "viewer" functionality as opposed to simulator functionality.**

11

We'll focus on the Viewer for this lecture. Note that I have added the Java 3D viewer here. We will not be talking about Java 3D in this lecture, but I added it to show that one can have multiple views of the same information.

## Class Hierarchy for Viewer



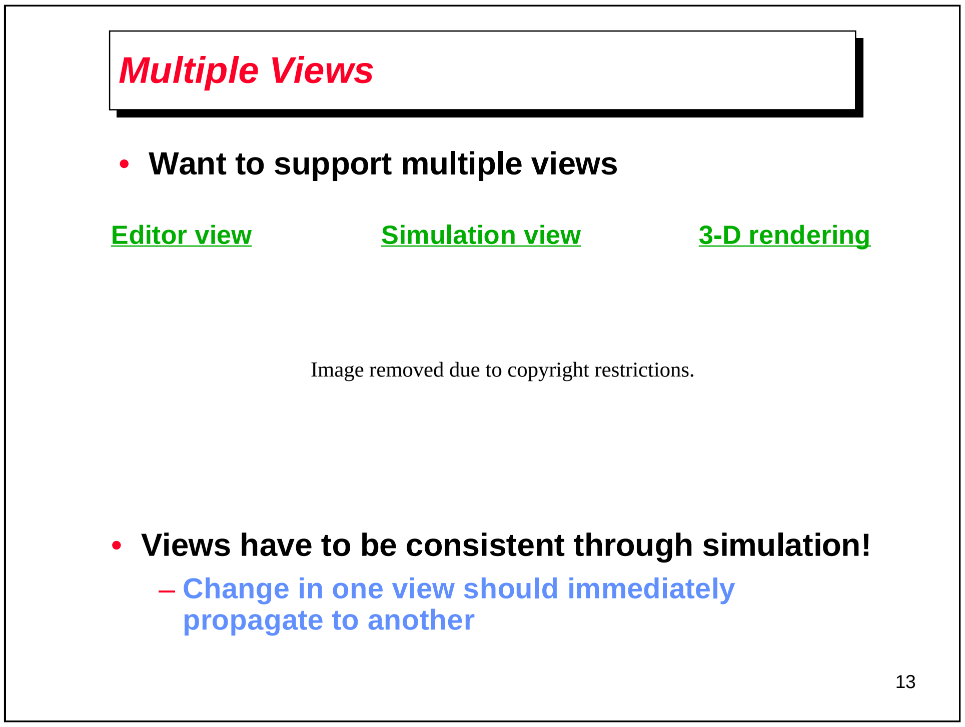
Let’s look at the class hierarchy for the viewer part of the Space Elevator (SE) simulator.

We have a World class, a WorldStructure class and an abstract WorldElement class.  A World object contains a WorldStructure object and a Simulator object.

WorldElements can be instantiated as planets, cables, masses, and connectors (springs).

We have an Editor class that is subclassed by the editors for various WorldElements, so we will have a PlanetEditor, CableEditor, etc.

Similarly, we have an WorldIllustrator (Drawer) class that is subclassed by the various WorldElement illustrators.  So we will a PlanetIllustrator, a CableIllustrator, etc.



## Multiple Views

- Want to support multiple views

Editor view | Simulation view | 3-D rendering

Image removed due to copyright restrictions.

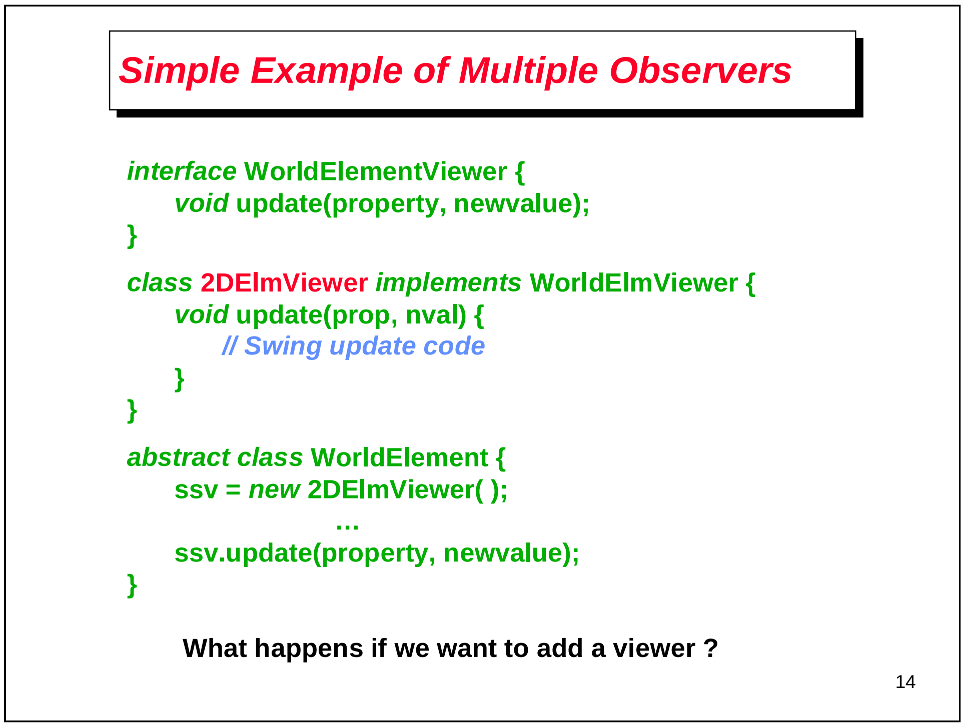
- Views have to be consistent through simulation!
  - Change in one view should immediately propagate to another

13

Editor view has static parameters for WorldElements such as cable and planet. These static parameters are the material constants of the cable, and initial starting position, etc. If there is a change in the parameters during simulation, for example, because of a break in the cable, then the simulator should immediately use the changed parameters.

The simulator view shows the changing positions and velocities of the WorldElements during simulation.

The numbers shown the simulation view are rendered using Java 3D in the 3D view.



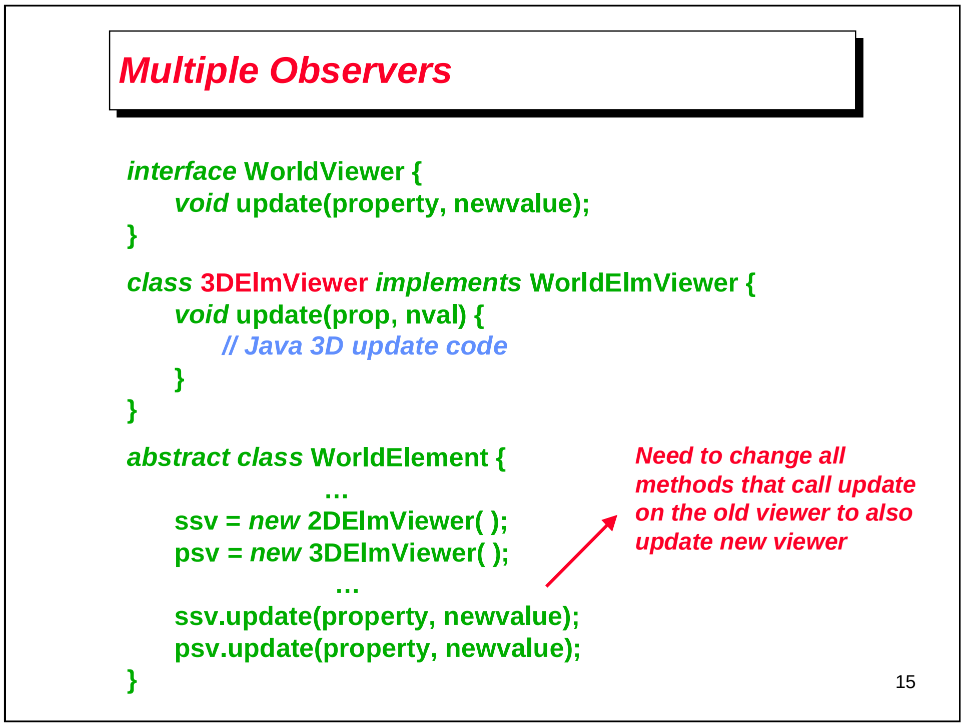
## Simple Example of Multiple Observers

```
interface WorldElementViewer {
      void update(property, newvalue);
}

class 2DElmViewer implements WorldElmViewer {
      void update(prop, nval) {
            // Swing update code
      }
}

abstract class WorldElement {
      ssv = new 2DElmViewer( );
                    …
      ssv.update(property, newvalue);
}
```

**What happens if we want to add a viewer ?**

14

Answer on next slide!

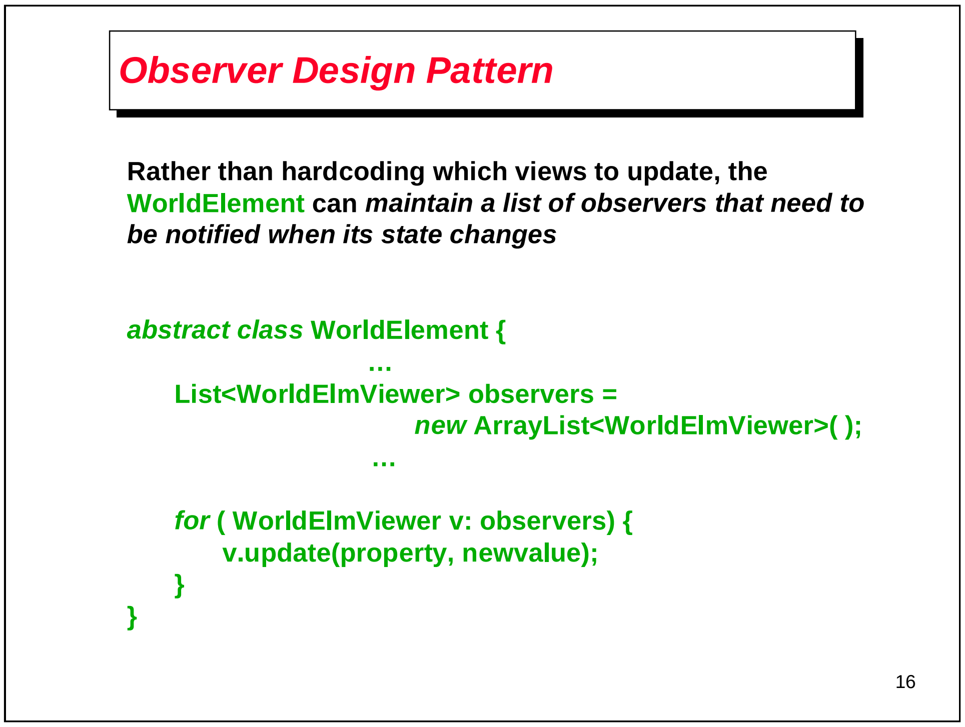


## Multiple Observers

```
interface WorldViewer {
    void update(property, newvalue);
}

class 3DElmViewer implements WorldElmViewer {
    void update(prop, nval) {
        // Java 3D update code
    }
}

abstract class WorldElement {
                 …
    ssv = new 2DElmViewer( );
    psv = new 3DElmViewer( );
                 …
    ssv.update(property, newvalue);
    psv.update(property, newvalue);
}
```

*Need to change all methods that call update on the old viewer to also update new viewer*

15



## Observer Design Pattern

**Rather than hardcoding which views to update, the WorldElement can *maintain a list of observers that need to be notified when its state changes***

```
abstract class WorldElement {
                         …
    List<WorldElmViewer> observers =
                    new ArrayList<WorldElmViewer>( );
                         …

    for ( WorldElmViewer v: observers) {
        v.update(property, newvalue);
    }
}
```

16

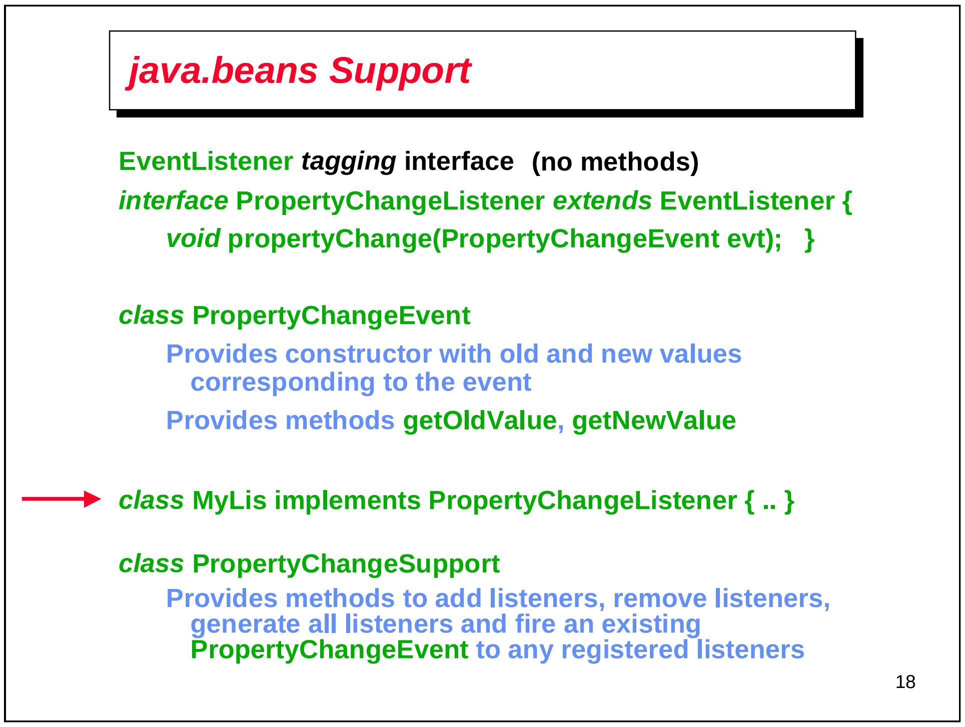
The Observer design pattern is a behavioral pattern that is described in the Design Pattern Lecture 18 notes.

## Observer Design Pattern (contd.)

**In order to initialize the observers the WorldElement class should provide at least two additional methods**

```
abstract class WorldElement {
                        …
   void register(WorldElmViewer viewer) {
      observers.add(viewer);
   }

   boolean remove(WorldElmViewer viewer) {
      return observers.remove(viewer);
   }                    …
}
```

**Glossed over details, e.g., how update's are actually generated**



## *java.beans Support*

EventListener ***tagging*** **interface (no methods)**

***interface*** **PropertyChangeListener** ***extends*** **EventListener {**

  ***void*** **propertyChange(PropertyChangeEvent evt); }**

***class*** **PropertyChangeEvent**

  **Provides constructor with old and new values corresponding to the event**

  **Provides methods getOldValue, getNewValue**

***class*** **MyLis implements PropertyChangeListener { .. }**

***class*** **PropertyChangeSupport**

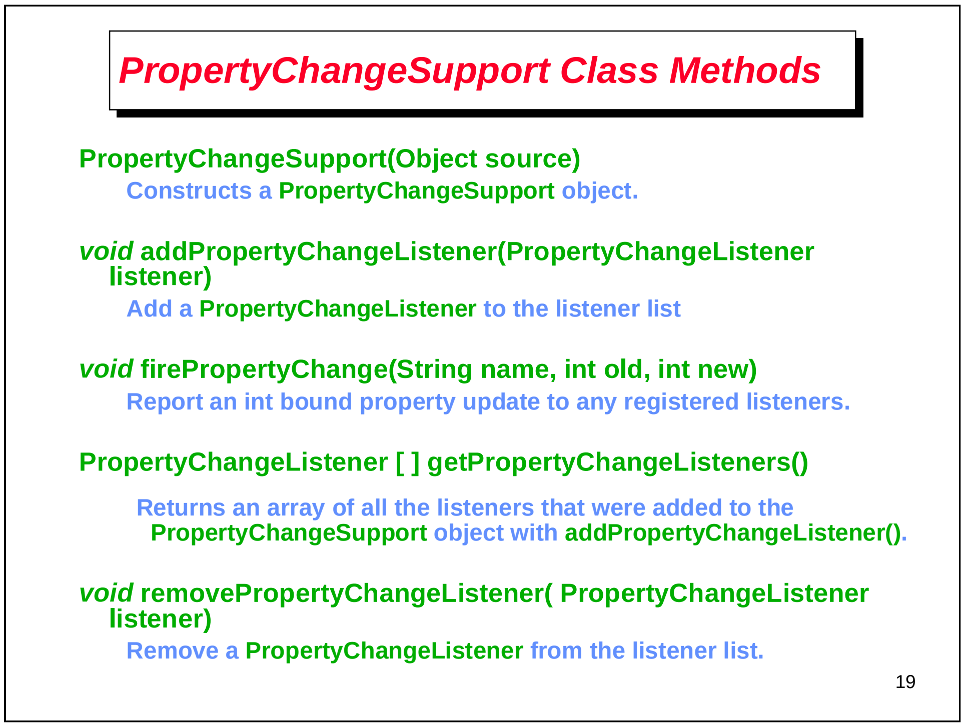
  **Provides methods to add listeners, remove listeners, generate all listeners and fire an existing PropertyChangeEvent to any registered listeners**

18

A tagging interface has no methods; its only purpose is to allow the use of instanceof in a type inquiry:

if (obj instanceof EventListener) . . .

A bean is a component based on the JavaBeans architecture.



## *PropertyChangeSupport Class Methods*

**PropertyChangeSupport(Object source)**
Constructs a **PropertyChangeSupport** object.

***void*** **addPropertyChangeListener(PropertyChangeListener listener)**
Add a **PropertyChangeListener** to the listener list

***void*** **firePropertyChange(String name, int old, int new)**
Report an int bound property update to any registered listeners.

**PropertyChangeListener [ ] getPropertyChangeListeners()**
Returns an array of all the listeners that were added to the **PropertyChangeSupport** object with **addPropertyChangeListener()**.

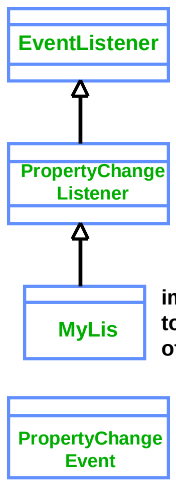
***void*** **removePropertyChangeListener( PropertyChangeListener listener)**
Remove a **PropertyChangeListener** from the listener list.

19

PropertyChangeSupport is a class in the java.beans package. It provides support to update listeners when a particular property changes.

## Pictorially …

**MDD**



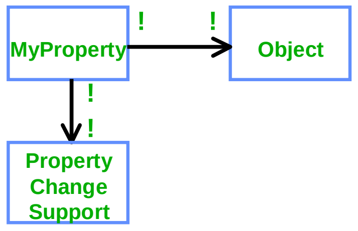
**Tagging interface (no methods)**

**Void propertyChange( PropertyChangeEvent evt)**

**implements propertyChange() to update internal state of listener**

**Constructor**
**PropertyChangeEvent(…)**
**getOldValue(), getNewValue()**

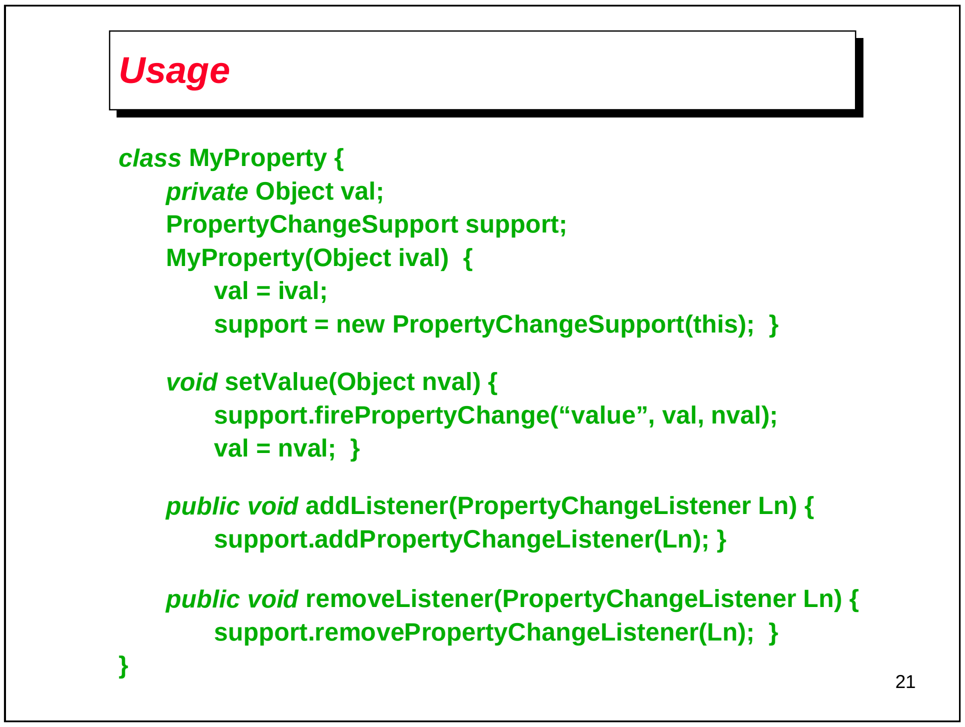
**Object Model**



**addPropertyChangeListener(…)**

**removePropertyChangeListener(…)**

**firePropertyChange(…) {**
**Construct PropertyChangeEvent object evt**
**On all registered listeners, e.g., MyLis, call MyLis.propertyChange(evt)**



## Usage

```
class MyProperty {
      private Object val;
      PropertyChangeSupport support;
      MyProperty(Object ival)  {
            val = ival;
            support = new PropertyChangeSupport(this);  }

      void setValue(Object nval) {
            support.firePropertyChange(“value”, val, nval);
            val = nval;  }

      public void addListener(PropertyChangeListener Ln) {
            support.addPropertyChangeListener(Ln); }

      public void removeListener(PropertyChangeListener Ln) {
            support.removePropertyChangeListener(Ln);  }
}
```

21

Note the use of composition and forwarding in MyProperty.  You do not want to extend PropertyChangeSupport, since you are implementing a property which is not a true subtype of PropertyChangeSupport.

## Demonstration

- Today, a simple demonstration of editing and viewing functionality (with multiple views)

- Tomorrow, more implementation details, Java 3D, and a demo of all existing functionality

22