Components and OO

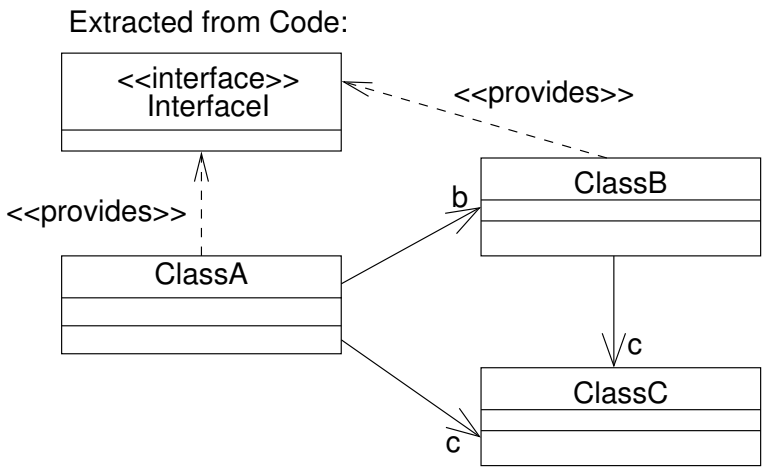
This part describes approaches for structuring object-oriented software systems into (white-box) components. Often, these components cannot be built separately.

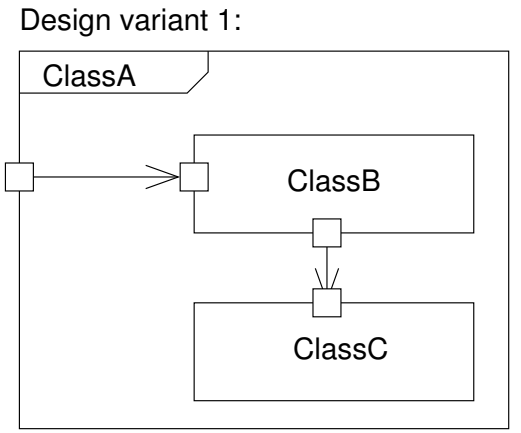
Components and OO I

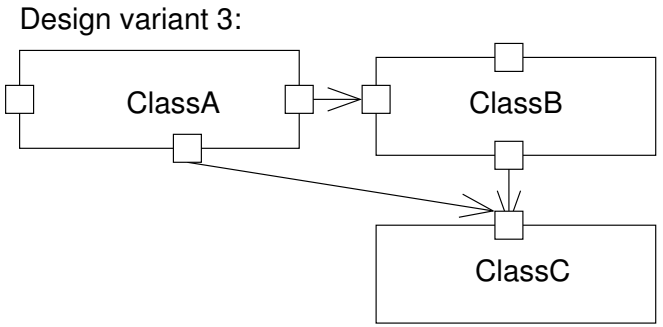
In the source code and in UML models, we usually have associations between classes. These associations can be either references to other components, or the referenced objects are part of one component.

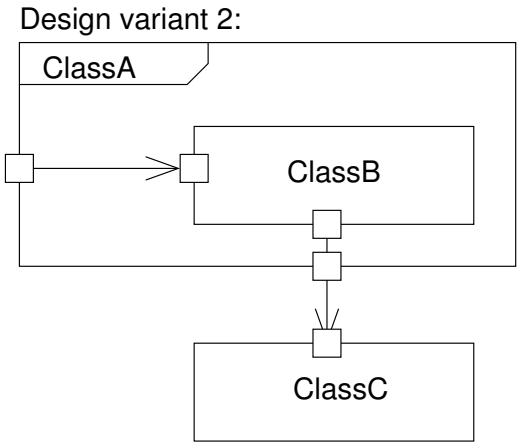
```
 class ClassA implements InterfaceI{
   private ClassB b;
   private ClassC c;
}
class ClassB implements InterfaceI{
   private ClassC c;
}
class ClassC {
   ...
}
```

Components and OO II









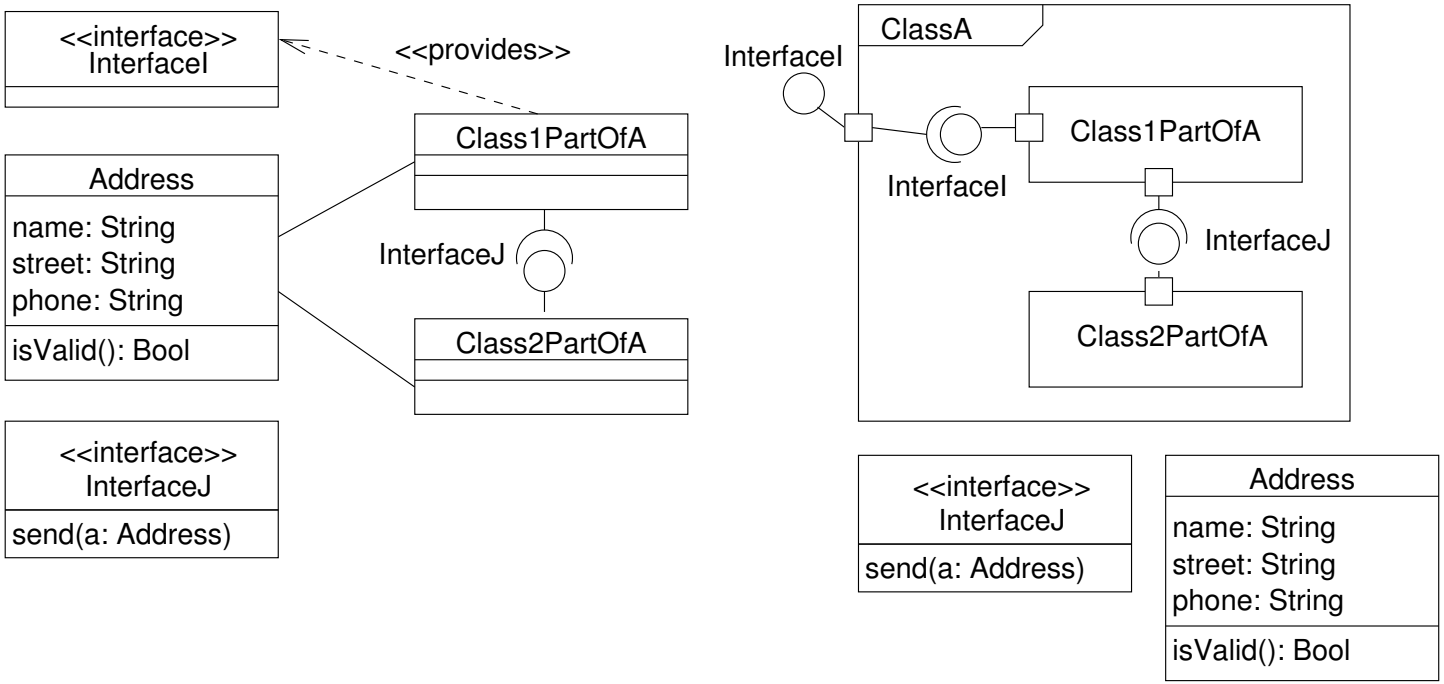
Components and OO III

Be careful

Not all objects can be clearly associated to a certain component: some objects are used to exchange complex data between components and exchanged as parameters, e.g. a user object is created in the user interface component and sent to the application component for further processing.

Components and OO IV

Example:



Component coupling levels I

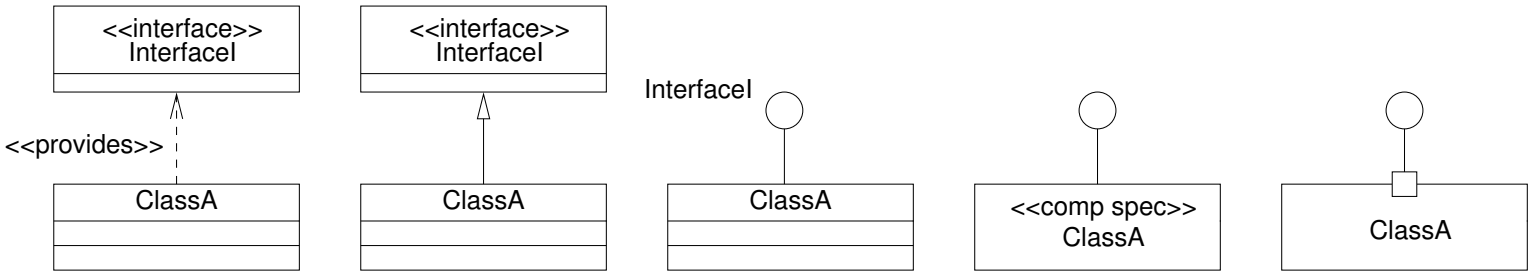
A component may be:

- Just an object of a class
  - May use other objects to provide its functionality
  - The public operations of the class represent the component interface
  - It is not clear which of the objects used to provide the functionality (associated) are part of that component, and which are not.
  - Other objects created by this object can be considered to be part of the component
  - Usually are not built separately

Component coupling levels II

- Object with explicit provided interfaces
  - May use other object to provide its functionality
  - Still not clear which of the objects used to provide the functionality are part of that component
  - Implementation can be better replaced
  - Usually are not built separately

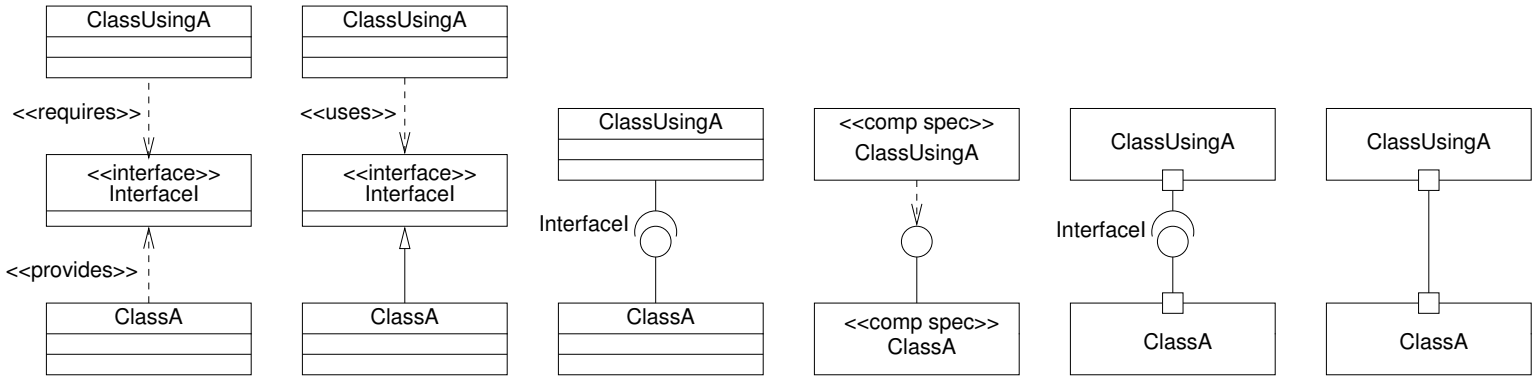
Notation: Class with provided Interface / Lollipop notation / Component according Cheesman and Daniels (2001):



Component coupling levels III

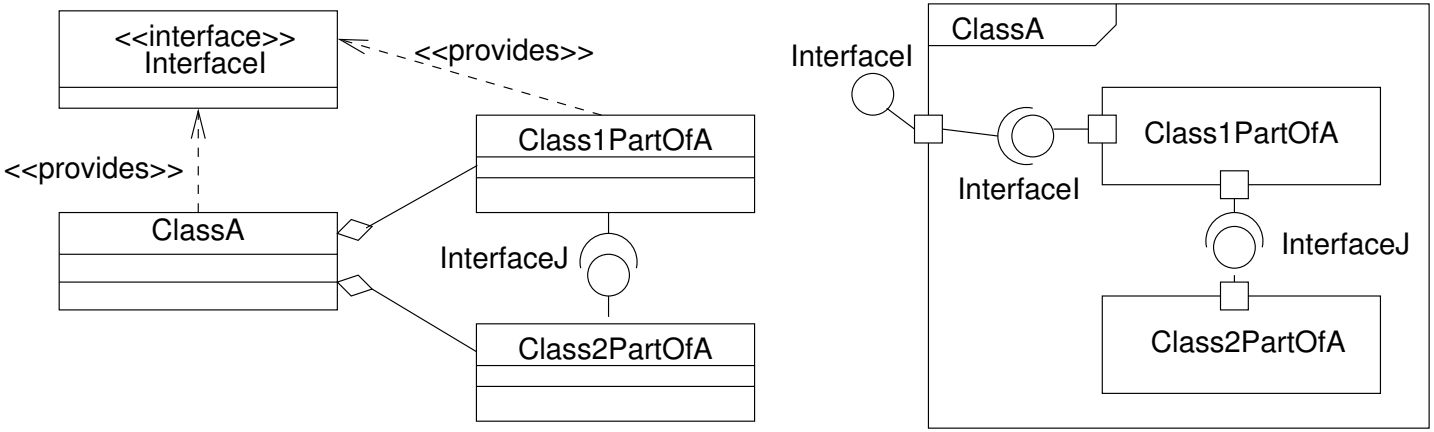
- Object with explicit provided and required interfaces
  - Loose coupling, other components used to provide the functionality are connected during instantiation or initialization
  - Advantage: components can be easily tested separately
  - Other object used to provide the functionality may be created, and they are considered to be part of the component

Notation for 2 connected classes / Lollipop notation / Component according Cheesman and Daniels (2001) / Composite Structure:



Component coupling levels IV

Composition in class diagrams and composite structure diagrams:



Component coupling levels V

- Object with explicit provided and required interfaces that makes use of a component standard (e.g., providing events or messages) to communicate with other components
  - Loose coupling, other components used to provide the functionality are connected at run-time
  - Advantage: components can be easily tested separately
  - Usually, can be built separately

Same notation as for objects with explicit provided and required interfaces

Component coupling levels VI

- All components are separate processes that communicate using events or messages
  - Loose coupling, other components used to provide the functionality are connected at run-time
  - Advantage: components can be easily tested separately
  - Usually, can be built separately

  Same notation as for objects with explicit provided and required interfaces

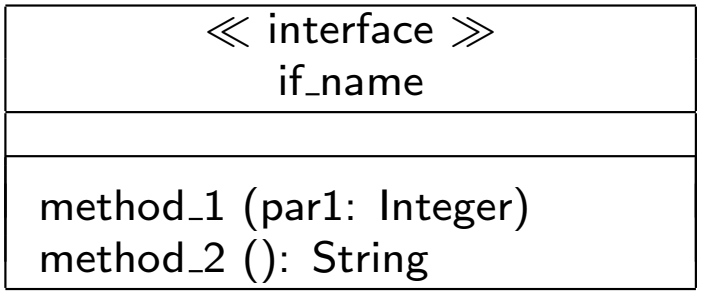
Simple Java Components

This part describes an implementation approach for components with explicit provided and required interfaces.

Implementation of components with explicit provided and required interfaces

- A component has provided and required interfaces that can be connected with other components.
- A component only uses functionality from its required interfaces, from the programming language, and a limited set of operations of the operating system (e.g., tasks, threads, memory allocation, timers, messages, synchronization mechanisms).
- Provided and required interfaces are represented by interface classes.
- Interface operations are called synchronously.
- Advantage: These classes / components can be easily tested separately.

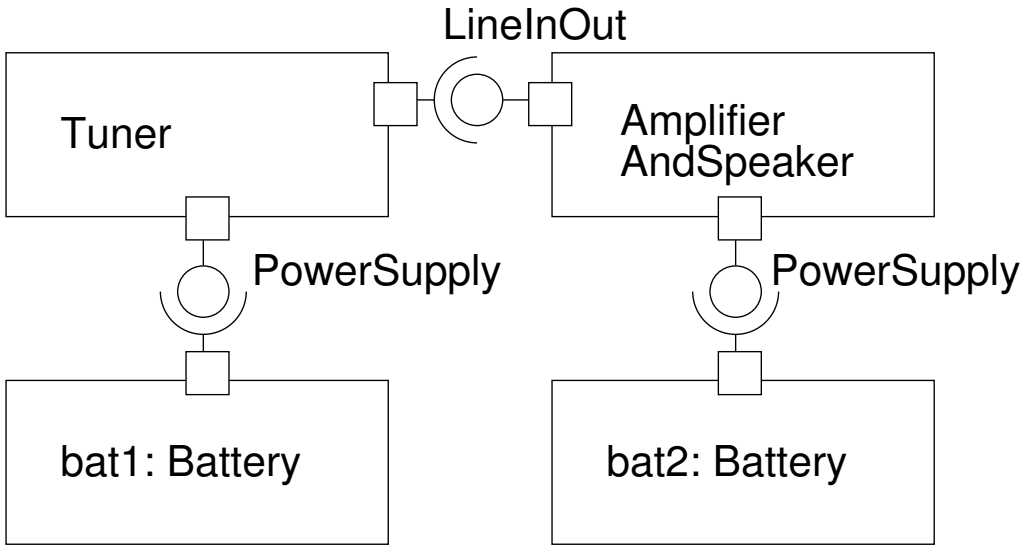
Implementation of interfaces in Java




```
package project_name;
public interface if_name {
    public void method_1 (int par1);
    public String method_2 ();
}
```

The project name should be added as a package. Otherwise additional parameters are necessary to compile the project.
Note: `int` is a simple data type, and `String` is a class.

Implementation of provided interfaces in Java I



Each provided interface is defined as an interface class, e.g.:

```
public interface LineInOut {
   public void transmitMusic();
}

public interface PowerSupply {
   public void powerOn();
}
```

Implementation of provided interfaces in Java II

A component can implement / provide several interfaces, e.g.:

```
public class AmplifierAndSpeaker implements
                              LineInOut, PowerSupply  {
   public AmplifierAndSpeaker (){} //constructor

   public void transmitMusic() { Play;}
   public void powerOn() { Action2;}
}
```

All provided operations must be implemented as methods.

Implementation of required interfaces in Java I

A component can use / require several interfaces, defined as interface classes.

```
public class Tuner implements PowerSupply {
   private LineInOut outputDevice;
   public Tuner(){ outputDevice = NULL; }
   public void connectTo(LineInOut par) {outputDevice = par;}

   public void powerOn() {
     while (true) {
        if (outputDevice!=NULL) outputDevice.transmitMusic();
   }
}
```

Implementation of required interfaces in Java II

- The required interfaces become private attributes (`outputDevice` of type `LineInOut`).
- The component has to provide methods to connect the component to the required components (`connectTo`). In these connect methods, the private attributes are initialized.
- Via these private attributes, the connected components can be used. They should only be used if they are initialized (`if (outputDevice!=NULL) ...`).

Alternatively, it is possible to leave out the method `connectTo` and initialize the connected interface in the constructor.

- The component `Tuner` also provides the interface `PowerSupply` and implements the method `powerOn`.

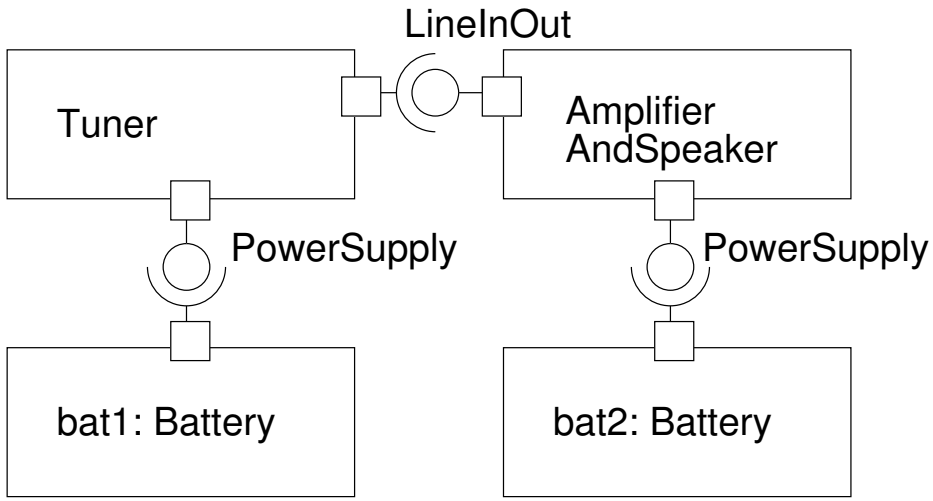
Implementation of required interfaces in Java III


```
public class Battery {
   private PowerSupply suppliedDevice;

   public Battery(){ suppliedDevice=NULL }

   public void connectTo(PowerSupply suppliedDev) {
        suppliedDevice = suppliedDev;
        suppliedDevice.powerOn();
   }
}
```

The component `Battery` powers on the supplied device when connected. It requires the interface `PowerSupply`.

Implementation of required interfaces in Java IV

The components *bat1*, *bat2*, *myTuner*, and *myAmp* can be connected as follows:




```
AmplifierAndSpeaker myAmp = new AmplifierAndSpeaker();
Tuner myTuner = new Tuner();
Battery bat1 = new Battery();
Battery bat2 = new Battery()
myTuner.connectTo(myAmp);
bat1.connectTo(myTuner);
bat2.connectTo(myAmp);
```

Component specifications

Structural notations for components:

- Composite structure diagrams
- Class diagrams
- Component diagrams

Additionally to the structure, the behavior of the components must be described using

- Pre- and postconditions for all interface operations (design by contract)
- Sequence diagrams
- State machines

What have we learned?

- In object orientation, a component can take different forms. These different forms come along with different coupling levels.
- Advantage of loosely coupled components: they can be built and tested separately.
- Provided and required interfaces of components implemented in Java are represented by interface classes. The component has to provide methods to connect the component to the required component.

JavaBeans

JavaBeans I

JavaBeans

- are **reusable software components** for Java.
- can be manipulated visually in a builder tool (e.g., Sun's NetBeans).
- are classes written in the **Java** programming language.
- **encapsulate many objects** into a single object (the bean).
- conform to the following **convention**: JavaBeans are
  - **serializable**.
  - have a **no-argument constructor**.
  - allow access to **properties** using getter and setter methods.

JavaBeans II

More information:

- Sun's JavaBeans product webpage:
  `http://java.sun.com/javase/technologies/desktop/javabeans/index.jsp`
- Sun's JavaBeans API webpage:
  `http://java.sun.com/javase/technologies/desktop/javabeans/api/index.html`
- Sun's JavaBeans tutorials:
  `http://java.sun.com/docs/books/tutorial/javabeans/`

Events

Events

- are a mechanism for propagating **state change notifications** between a *source* JavaBean and one or more *target* JavaBeans.
- are the basis to plug JavaBeans together in an **application builder**.
- can be **caught and processed** by JavaBeans.
- have many different uses, but a common example is their use in a window system toolkit for delivering notifications of mouse actions, widget updates, keyboard actions, etc.

Event Model I

- **Event notifications** are propagated from sources to listeners by Java method invocations on the target listener objects.
- Each distinct kind of event notification is defined as a distinct Java method. These methods are then grouped in `EventListener` interfaces that inherit from `java.util.EventListener`.
- Event listener classes identify themselves as interested in a particular set of events by implementing some set of `EventListener` interfaces.

Event Model II

- The state associated with an event notification is normally encapsulated in an event state object that inherits from `java.util.EventObject` and which is passed as the sole argument to the event method.
- Event sources identify themselves as sourcing particular events by defining registration methods and accept references to instances of particular `EventListener` interfaces.
- In circumstances where listeners cannot directly implement a particular interface, or when some additional behavior is required, an instance of a custom **adaptor class** may be interposed between a source and one or more listeners in order to establish the relationship or to augment behavior.

Properties

Properties

- are **attributes** of a Java Bean that can affect its appearance or its behavior.
- Example: a GUI button might have a property named "Label" that represents the text displayed in the button.
- can be accessed by other JavaBeans calling their **getter and setter methods**.
- typically are **persistent**, so that their state will be stored away as part of the persistent state of the JavaBean.
- can have arbitrary types, including both built-in Java types such as `int` and class or interfaces types such as `java.awt.Color`.

Accessor Methods

- For **readable properties** there will be a getter method to read the property value.
- For **writable properties** there will be a setter method to allow the property value to be updated.
- For simple properties the accessor type signatures are:

  simple setter `void setFoo(PropertyType value);`
  simple getter `PropertyType getFoo();`

Indexed Properties I

- An **indexed property** supports a range of values. Whenever the property is read or written one specifies an index to identify which value is required.
- Property indexes must be of type `int`.
- For indexed properties the accessor type signatures are:

  indexed setter

  ```
  void setter(int index, PropertyType
  value);
  ```

  indexed getter

  ```
  PropertyType getter(int index);
  ```

Indexed Properties II

array setter

```
void setter(PropertyType values[]);
```

array getter

```
PropertyType[] getter();
```

- The indexed getter and setter methods may throw a `java.lang.ArrayIndexOutOfBoundsException` runtime exception if an index is used that is outside the current array bounds.

Bound Properties I

- Sometimes when a JavaBean property changes then either the JavaBeans container (i.e., a program that uses the JavaBean) or some other JavaBean may wish to be notified of the change.
- A JavaBean can choose to provide a change notification service for some or all of its properties.
- Such properties are commonly known as **bound properties**, as they allow other components to bind special behavior to property changes.
- The `PropertyChangeListener` event listener interface is used to report updates to simple bound properties.

Bound Properties II

- If a JavaBean supports bound properties then it should support a pair of event listener registration methods for `PropertyChangeListener`:

  add listener

```
public void
addPropertyChangeListener
(PropertyChangeListener x);
```

  remove listener

```
public void
removePropertyChangeListener
(PropertyChangeListener x);
```

Bound Properties III

- When a property change occurs on a bound property the JavaBean should call the `PropertyChangeListener.propertyChange` method on all registered listeners, passing a `PropertyChangeEvent` object that encapsulates the name of the property and its old and new values.
- The event source should fire the event after updating its internal state.

Example: JavaBean Person I

The class Person has a property name, that can be changed using setName(). After a change, the JavaBean informs all listeners of this change.

```
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;

public class Person
{
  private String name = "";
  private PropertyChangeSupport changes =
    new PropertyChangeSupport(this);

  public void setName(String name)
  {
    String oldName = this.name;
    this.name = name;
    changes.firePropertyChange("name", oldName, name);
  }
```

Example: JavaBean Person II


```
  public String getName()
  {
    return name;
  }

  public void addPropertyChangeListener(
    PropertyChangeListener pcl)
  {
    changes.addPropertyChangeListener(pcl);
  }

  public void removePropertyChangeListener(
    PropertyChangeListener pcl)
  {
    changes.removePropertyChangeListener(pcl);
  }
}
```

Example: Reaction to Property Change I

Registered `PropertyChangeListener` can react to the `PropertyChangeEvent`.

```
public class ReportChange implements PropertyChangeListener {
  @Override
  public void propertyChange(PropertyChangeEvent e)
  {
    System.out.printf("Property '%s': '%s' -> '%s'%n",
      e.getPropertyName(), e.getOldValue(), e.getNewValue());
  }
}
```

Example: Reaction to Property Change II


```
Person person = new Person();
ReportChange reportChange = new ReportChange();

person.addPropertyChangeListener(reportChange);

person.setName("Ulli");
// expected output: Property 'name': '' -> 'Ulli'
person.setName("Ulli");
// no output expected
person.setName("Chris");
// expected output: Property 'name': 'Ulli' -> 'Chris'
```

Constrained Properties I

- Sometimes when a property change occurs some other bean may wish to validate the change and reject it if it is inappropriate.
- We refer to properties that undergo this kind of checking as **constrained properties**.
- In Java Beans, constrained property setter methods are required to support the `PropertyVetoException`. This documents to the users of the constrained property that attempted updates may be vetoed.

Constrained Properties II

The following operations in the setter method for the constrained property must be implemented in this order:

1. Save the old value in case the change is vetoed.
2. Notify listeners of the new proposed value, allowing them to veto the change.
3. If no listener vetoes the change (no exception is thrown), set the property to the new value.

Constrained Properties III

- A simple constrained property might look like:

```
PropertyType getFoo();
void setFoo(PropertyType value)
  throws PropertyVetoException;
```

- In the body of a setter method, the `fireVetoableChange` method is invoked on the `VetoableChangeSupport` attribute of the Java Bean before the `firePropertyChange` method is invoked on the property that is changed.

Constrained Properties IV

- A simple setter method for a constrained property might look like:

```
public void setFoo(boolean foo)
  throws PropertyVetoException{
    boolean oldValue = this.foo;
    vetos.fireVetoableChange("foo",
      oldValue, foo);
    this.foo = foo;
    changes.firePropertyChange("foo",
      oldValue, foo);
}
```

Constrained Properties V

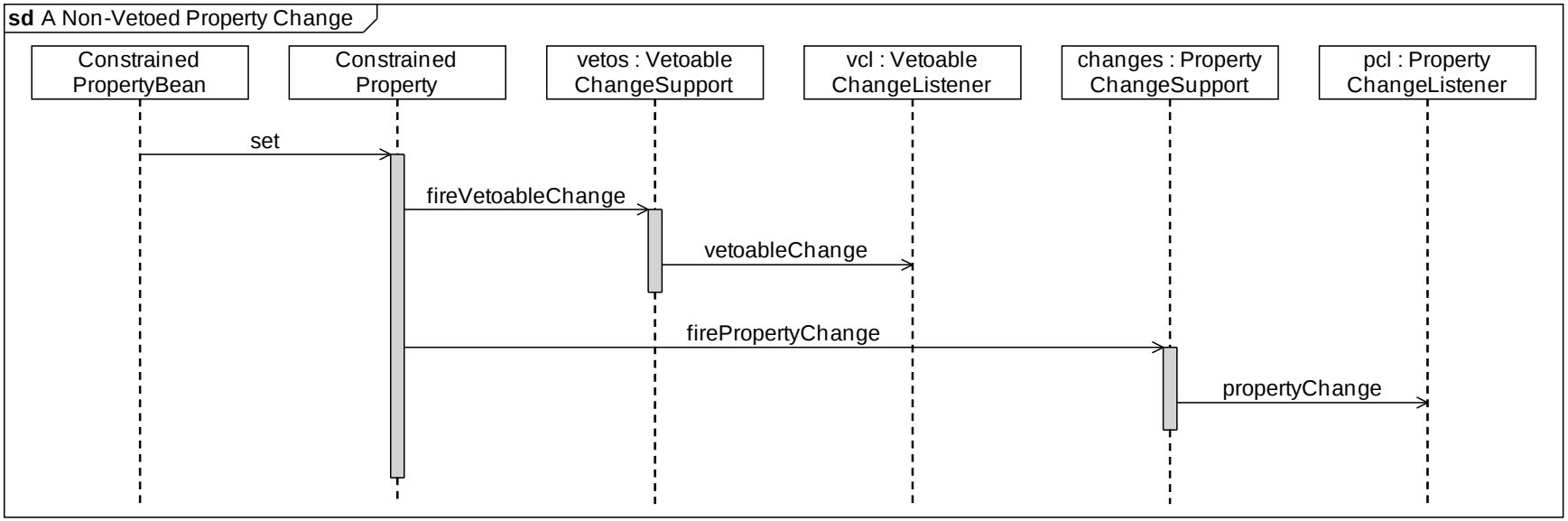
- The `VetoableChangeListener` event listener interface is used to report updates to constrained properties. If a bean supports constrained properties then it should support a pair of event listener registration methods for `VetoableChangeListeners`:

```
public void addVetoableChangeListener
  (VetoableChangeListener x);
public void removeVetoableChangeListener
(VetoableChangeListener x);
```

Constrained Properties VI

- When a property change occurs on a constrained property the bean should call the `VetoableChangeListener.vetoableChange` method on all registered listeners, passing a `PropertyChangeEvent` object that encapsulates the name of the property and its old and new values.

Constrained Properties VII



Constrained Properties VIII

- If the event recipient does not wish the requested edit to be performed it may throw a `PropertyVetoException`. It is the source bean's responsibility to catch this exception, revert to the old value, and issue a new `VetoableChangeListener.vetoableChange` event to report the reversion.
- The initial `VetoableChangeListener.vetoableChange` event may have been relayed to a number of recipients before one vetoes the new value.

Constrained Properties IX

- If one of the recipients vetoes, then one has to make sure that all the other recipients are informed (fire another `VetoableChangeListener.vetoableChange` event) that the old value is restored. The source may choose to ignore vetoes when reverting to the old value.
- The event source should fire the event before updating its internal state.

Example: Reaction to Property Change I

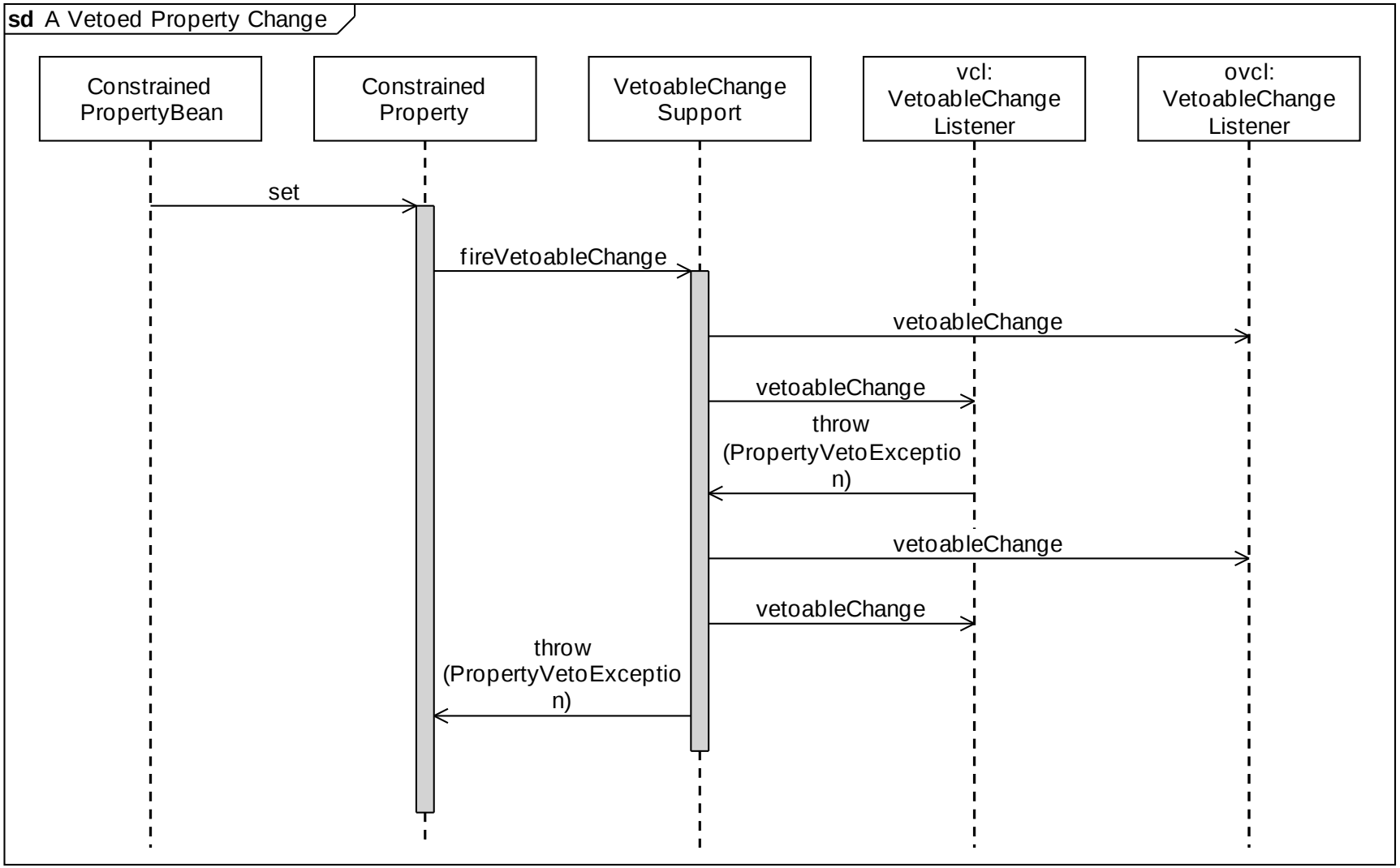
Registered `PropertyChangeListener` can react to the `PropertyChangeEvent`.

```
public class ReportChangeVeto implements VetoableChangeListener {
  @Override
  public void vetoableChange(PropertyChangeEvent e)
    throws PropertyVetoException
  {
    if ( "Name".equals( e.getPropertyName() ) )
      if ( "Ulli".equal.( e.getNewValue()) )
        throw new PropertyVetoException( "Not with me", e );
  }
}
```

Example: Reaction to Property Change II


```
Person person = new Person();
ReportChangeVeto reportChangeVeto = new ReportChangeVeto();

person.addVetoableChangeListener(reportChangeVeto);

try
{
  person.setName("Ulli");
}
catch ( PropertyVetoException e )
{
  // expected output: java.beans.
  //    PropertyVetoException: Not with me
  e.printStackTrace();
}
```

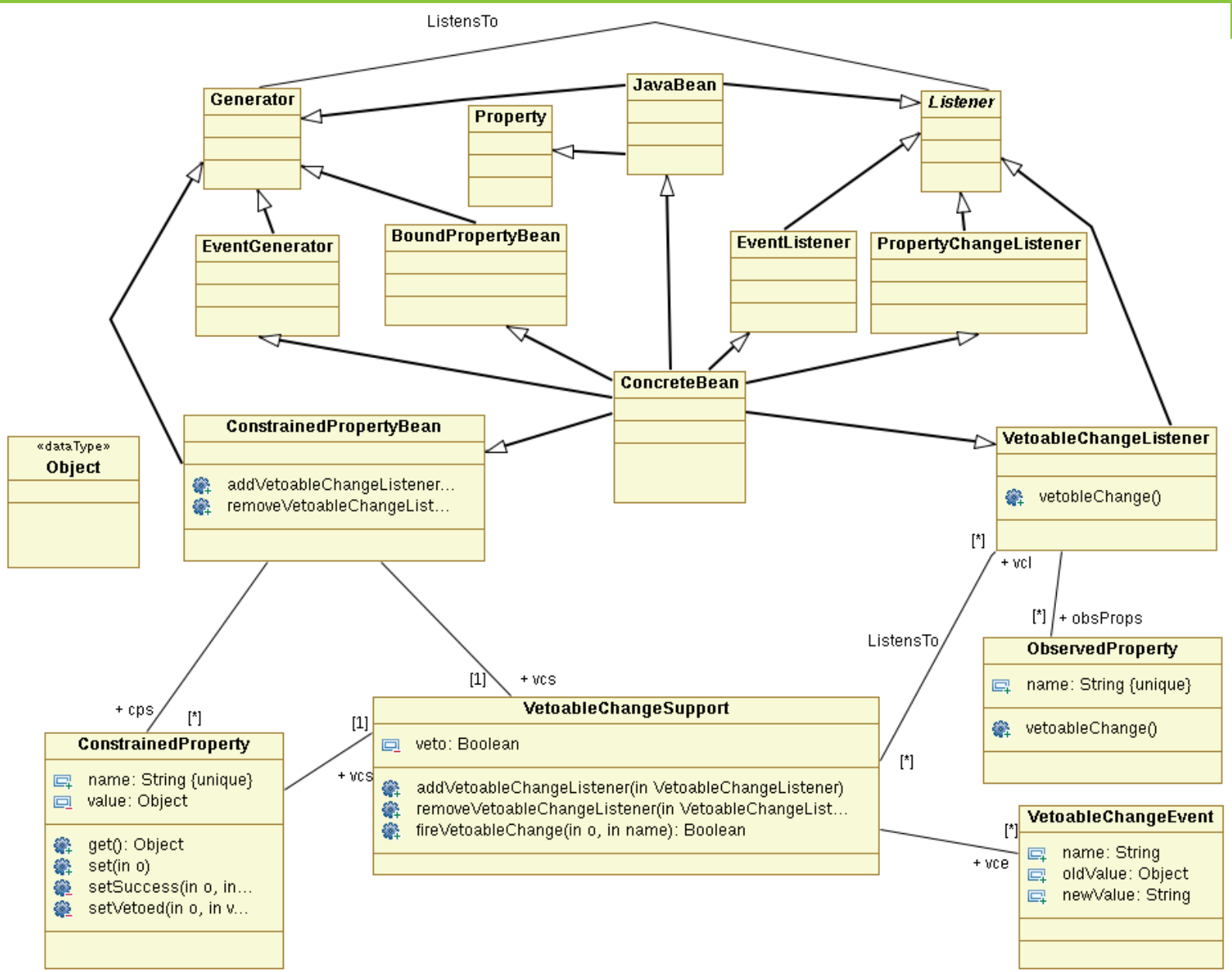
Example: Reaction to Property Change III



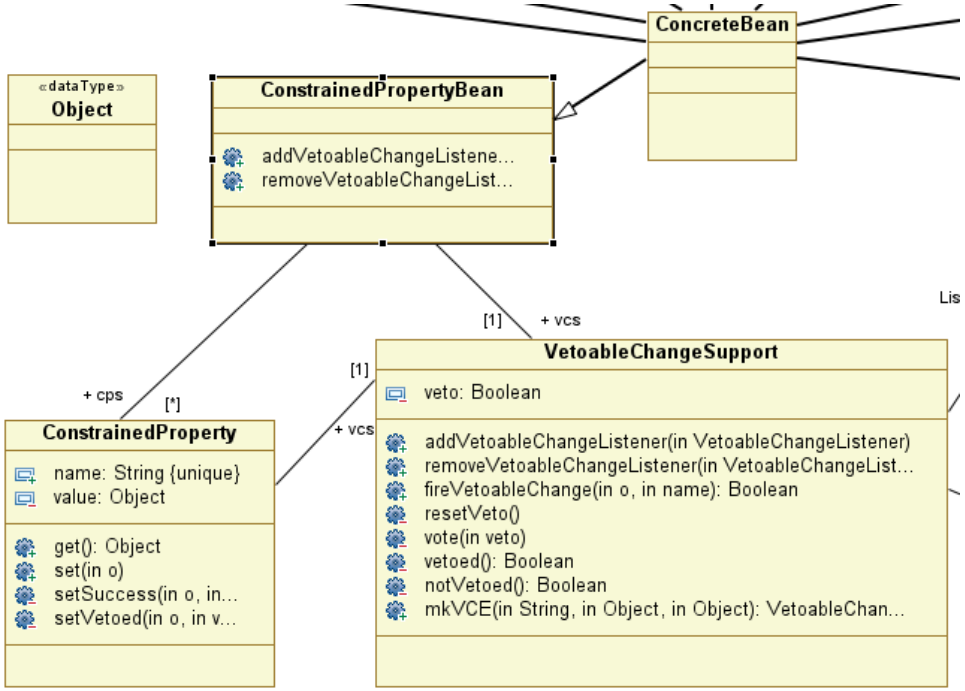
JavaBeans Component Model

- **Component model** to specify the characteristics of a JavaBean according to this model.
- Based on a formalization of Sun's JavaBeans model by Heisel et al. (2002).
- Described as a metamodel using a UML class diagram and OCL constraints.
- Instances of this metamodel constitute concrete JavaBean specifications.

JavaBeans Metamodel I
Class Model



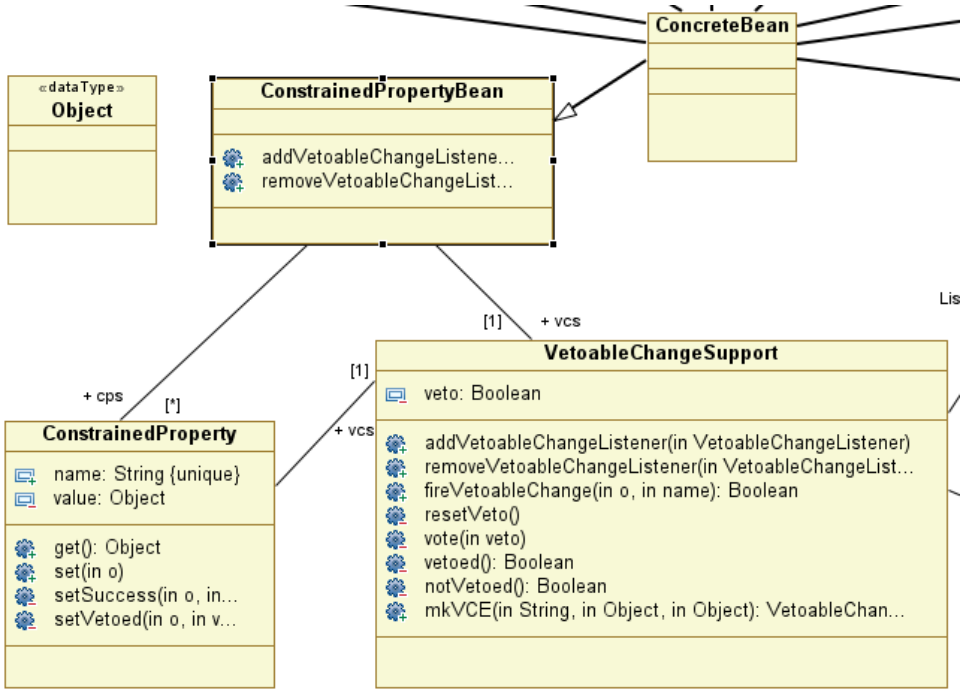
JavaBeans Metamodel II
OCL Constraints



The same `vcs` object must be used by all objects belonging to the set `cps`.

```
context ConstrainedPropertyBean
inv:  self.cps->forAll(
  cp:ConstrainedProperty|cp.vcs=self.vcs)
```

JavaBeans Metamodel III
OCL Constraints



The content of `value` is returned.

```
context ConstrainedProperty.get
post:  result=self.value
```

Creating a Simple JavaBean I

Write the `SimpleBean` code. Put it in a file named `SimpleBean.java`.

```
import java.awt.Color;
import java.beans.XMLDecoder;
import javax.swing.JLabel;
import java.io.Serializable;

public class SimpleBean extends JLabel
 implements Serializable {
   public SimpleBean() {
      setText( "Hello world!" );
      setOpaque( true );
      setBackground( Color.RED );
      setForeground( Color.YELLOW );
```

Creating a Simple JavaBean II


```
        setVerticalAlignment( CENTER );
        setHorizontalAlignment( CENTER );
    }
}
```

- `SimpleBean` extends the `javax.swing.JLabel` graphic component and inherits its properties, which makes the `SimpleBean` a visual component.
- `SimpleBean` also implements the `java.io.Serializable interface`.

Compiling the JavaBean and Generating a Java Archive (JAR) File I

- Create a manifest, the JAR file, and the class file `SimpleBean.class`.
- Use the Apache Ant (`http://ant.apache.org/`) tool to create these files.
- Apache Ant is a Java-based build tool that enables one to generate XML-based configurations files as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project default="build">

 <dirname property="basedir" file="${ant.file}"/>

 <property name="beanname" value="SimpleBean"/>
 <property name="jarfile"
  value="${basedir}/${beanname}.jar"/>
```

Compiling the JavaBean and Generating a Java Archive (JAR) File II


```
<target name="build" depends="compile">
    <jar destfile="${jarfile}"
     basedir="${basedir}" includes="*.class">
 <manifest>
     <section name="${beanname}.class">
 <attribute name="Java-Bean" value="true"/>
     </section>
 </manifest>
    </jar>
</target>
```

Compiling the JavaBean and Generating a Java Archive (JAR) File III


```
<target name="compile">
    <javac destdir="${basedir}">
 <src location="${basedir}"/>
    </javac>
</target>

<target name="clean">
    <delete file="${jarfile}">
 <fileset dir="${basedir}" includes="*.class"/>
    </delete>
</target>
</project>
```

Loading the JavaBean into the GUI builder of the NetBeans IDE I

- It is recommended to save an XML script in the `build.xml` file, because Ant recognizes this file name automatically.
- Load the JAR file. Use the NetBeans IDE GUI Builder to load the jar file.

1. Start NetBeans.
2. From the file menu select “New Project” to create a new application for the bean. You can use “Open Project” to add the bean to an existing application.
3. Create a new application using the “New Project Wizard”.

Loading the JavaBean into the GUI builder of the NetBeans IDE II

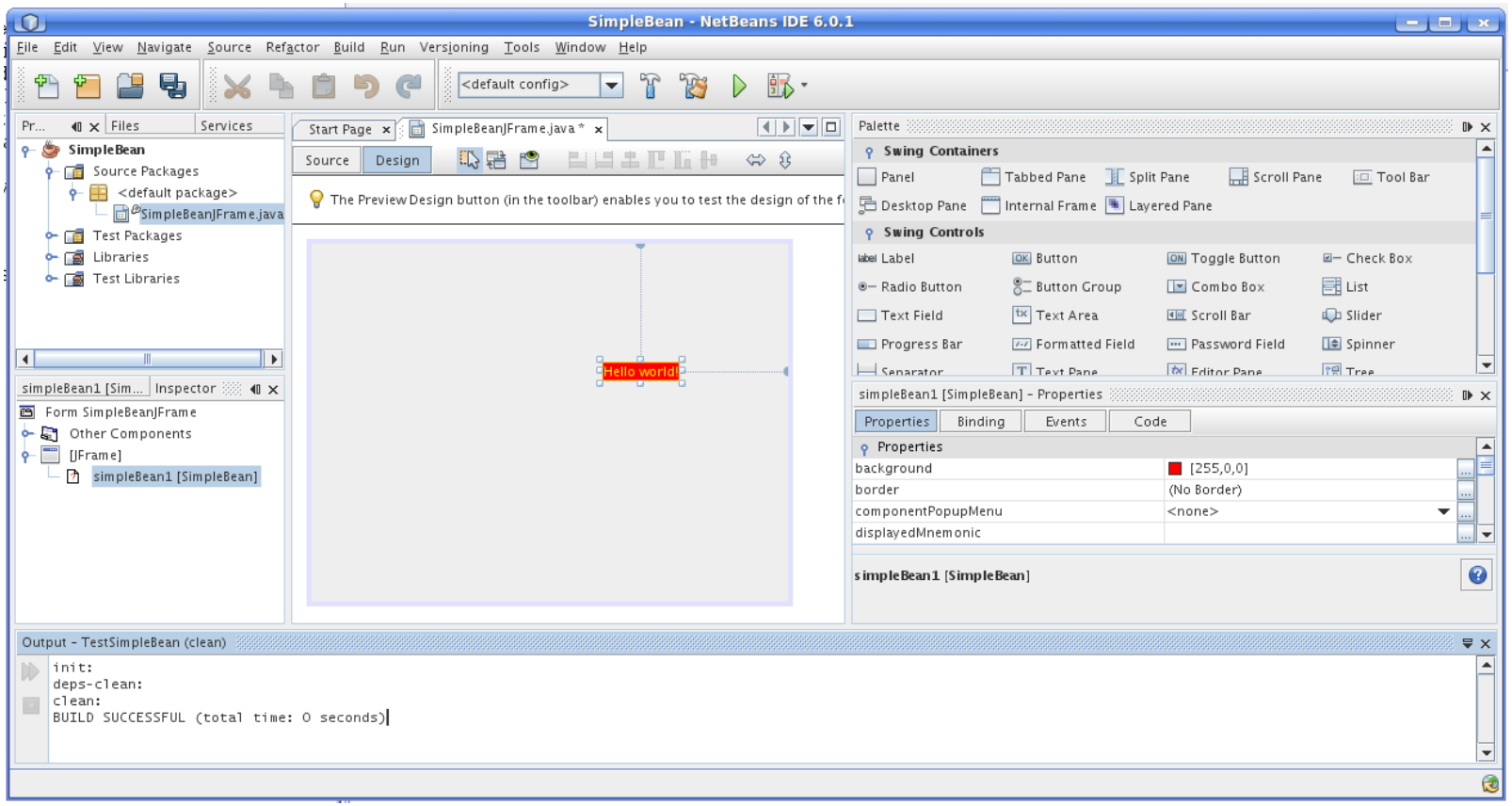
4. Select a newly created project in the list of projects, expand the “Source Packages” node, and select the “Default Package” element.
5. Click the right mouse button and select “New - JFrameForm” from the pop-up menu.
6. Select the newly created form node in the project tree. A blank form opens in the GUI builder view of an editor tab.
7. Open the palette manager for “Swing/AWT components” by selecting “Palette Manager” in the “Tools” menu.
8. In the “Palette Manager” window select the beans components in the palette tree and press the “Add from JAR” button.

Loading the JavaBean into the GUI builder of the NetBeans IDE III

9. Specify a location for the `SimpleBean` JAR file and follow the "Add from JAR Wizard" instructions.
10. Select the palette and properties options from the "Windows" menu.
11. Expand the beans group in the palette window. The `SimpleBean` object appears. Drag the `SimpleBean` object to the GUI builder panel.

Loading the JavaBean into the GUI builder of the NetBeans IDE IV

The following figure represents the `SimpleBean` object loaded in the GUI builder panel:



Inspecting the JavaBean's Properties and Events

- The `SimpleBean` properties will appear in the Properties window.
- For example, one can change a background property by selecting another color.
- To preview the form, use the "Preview Design" button of the GUI builder toolbar.
- To inspect events associated with the `SimpleBean` object, switch to the events tab of the "Properties" window.

What have we learned?

- JavaBeans are (mainly) visual components according to a component model by Sun.
- Consequently, they can be used to build graphical user interfaces using builder tools such as NetBeans.
- Formalization of the JavaBeans component model using UML class diagram, sequence diagrams, and OCL.
- Construction of a simple JavaBean.