JFace Data Binding I

Introduction.
Data binding with SWT controls

Adrian Stan

Software developer

Up-to-date presentation

- git clone https://github.com/adrstan1/databinding_workshop.git

Data Binding

- A mechanism allowing automatic *validation*, *synchronization* and *conversion* of *values* between objects.
- It is used mainly for *synchronization between user interface components and model properties*.
- Support exists for *SWT*, *JFace*, *Swing*, *EMF*, *GWT*, etc.
- Mostly used in Eclipse applications/plugins

Common use-case

- You need a simple way to *communicate* between your application's *user interface* and the *domain model object* represented by the user interface.
- *Communicate*:

  1) If the user changes the UI I want my model object to be update accordingly.

  2) If the model object updates it's state indirectly, I want the user interface to react accordingly.

  *Note: you may not need 2), but you want 1)

Common use-case

- It certainly can be done without the *data binding framework*, but:

  - I want to minimize the quantity of *boilerplate* code.

  - I don't want to code yet another *Observable*/*Observer* mechanism from scratch.

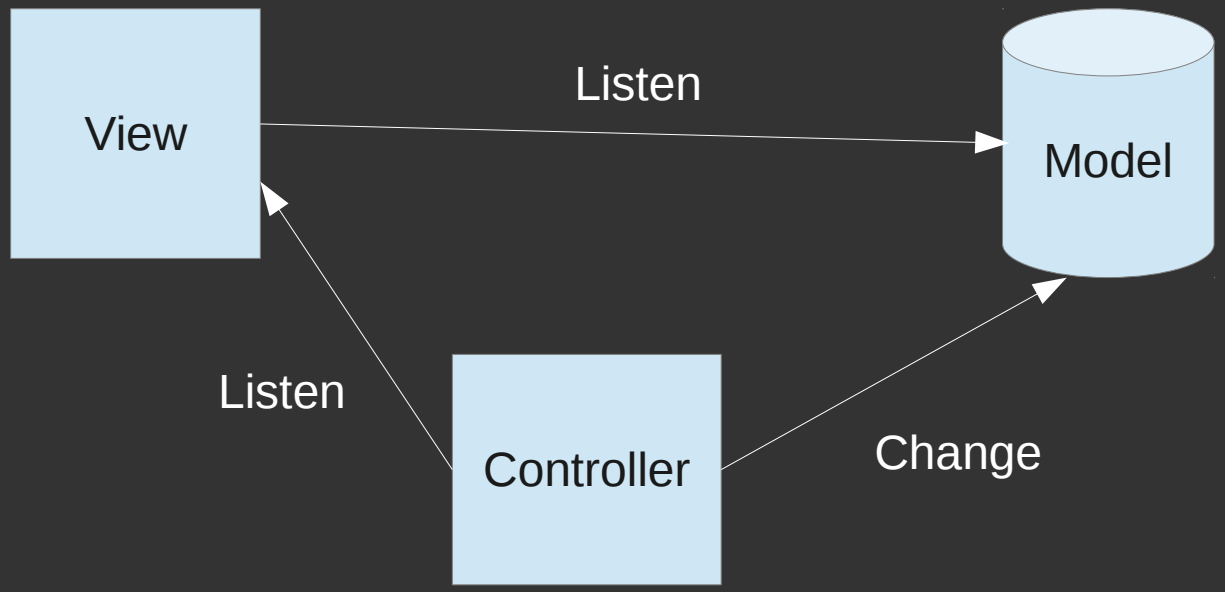
  - I don't like writing *repetitive* code.

  - I want to be *flexible*. For example, I want to be able to easily add *validation* and *conversion* mechanisms.

Common use-case

Let's analyze an *MVC* base solution ...

MVC Review



MVC Review

- 1. The *user* interacts with the UI.
- 2. The *controller* is registered to the *view*, listening for change events.
- 3. The *controller* reacts to the UI changes and updates the *model*.
- 4. The *view* is registered to the *model*, listening for model properties changes.
- 5. The *view* gets notified when the *model* has updated its state and re-renders the *model*.

MVC Review

- Possible drawbacks of an *MVC* base solution:

  1. The *view* needs how to extract information from the *model*, in order to be able to *render* the *model*.

  2. The *controller* code tends to be intrusive: the *controller* needs to have knowledge of the *view* and it needs to know how to update the *model*.

MVC Review

- *Data binding* tries to simplify things. Most of the time:

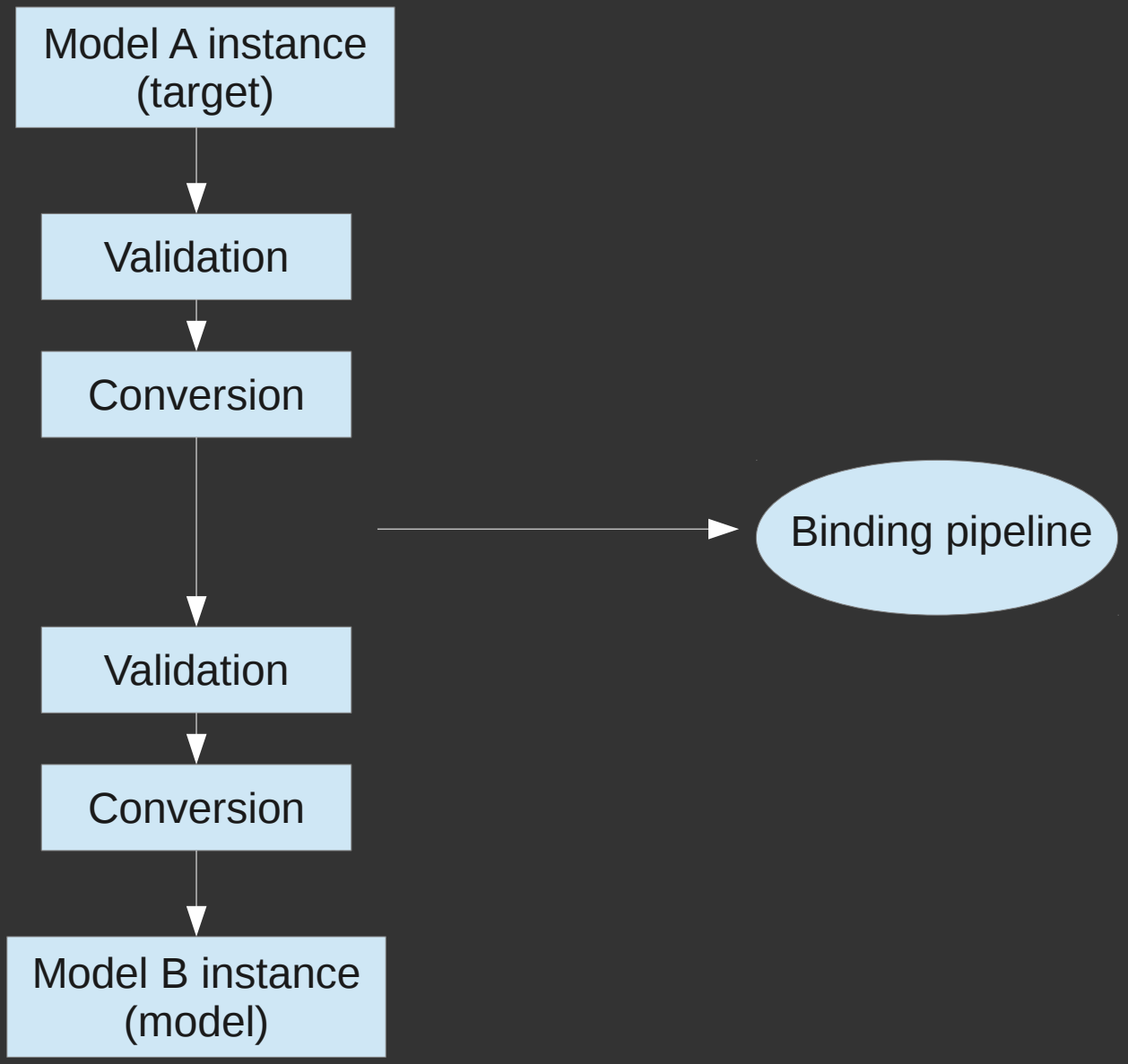
  - *text widgets* needs to bind to single properties of model objects

  - *radio groups* bind to lists/sets of properties and the selected radio button is bound to one of the element in the list/set

  - ...

- This allows building a more easier to use framework.

Binding. The general ideea



Core concepts

- *Observables*
- *Bindings*

Observables

- Allows observing changes in objects.
- Are implementations of the *Observer* pattern.
- *Interfaces*:

  - *IObservable* (super interface for all observables).

  - *IObservableValue* (allows listening to value change events, provides getter and setter for values).

  - Many others: *IVetoableValue*, *IObservableCollection*, *IObservableList*, *IObservableSet*, *IObservableMap*.

Bindings

- Used for *synchronization* between two *observables*.
- The *synchronization* process allows performing *validation* and *conversion* in a flexible manner: validation after get, validation after convert, validation before set.

Value Bindings

- The most used binding.
- Allows binding between *IObservable* instances.

List Bindings

- Allows bindings between *IObservableList* instances.
- *IObservableList* allows listening for incremental list change events.

Set bindings

- Allows bindings between *IObservableSet* instances.
- *IObservableSet* allows listening for incremental set change events.

Map bindings

- Allows bindings between *IObservableMap* instances.
- *IObservableMap* allows listening for incremental map change events.

Factories

- *Factories* are used for creating *observable* objects.

  - *SWTObservables* - used for creating properties for SWT widgets .

  - *ViewerProperties* - used for creating properties for JFace viewers.

  - *PojoProperties* - used for creating properties for *POJO*s (*Plain Old Java Objects*).

  - *BeanProperties* - used for creating properties for *Java Bean* objects.

  - Other factories are available, see the Eclipse documentation.

Data binding contexts

- The *context* provides support for performing binding between observables.

Review

- *Observables* are created by *factories*.
- *Observables* are bound by using data binding *contexts*.

POJOs

- Java classes that provides *getters* and *setters* but do not propagate *property change* events on change.
- *Example*:

  A POJO model class property is bound to an SWT control. When the SWT control changes its state, it updates the model using the setters (deduced via reflection).

  If the model state changes independently from the UI, the UI is not notified of the change.

  The synchronization happens one-way, from the UI to the model.

Java Beans

- Java classes implementing property change support via the *PropertyChangeSupport* class.
- Allows listeners to register to objects and receive notifications on change.
- The data binding can registers itself to receive change notifications from the Java Bean object.
- Allows the UI to react to indirect changes to model objects. Thus, we can have bidirectional synchronization between UI and model objects.

Property Change Support

- Need to implement the following methods in your Java Bean class.

- *addPropertyChangeListener*(PropertyChangeListener listener)
- *removePropertyChangeListener*(PropertyChangeListener listener)
- *addPropertyChangeListener*(String propertyName, PropertyChangeListener listener)
- *removePropertyChangeListener*(String propertyName, PropertyChangeListener listener)
- *firePropertyChange*(String propertyName, Object oldValue, Object newValue)

Binding - Basic Usage

- bindingContext = new DataBindingContext();
- IObservableValue widgetValue, modelValue;
- widgetValue = WidgetProperties.text(SWT.Modify).observe(firstNameTxt);
- modelValue = BeanProperties.value(PojoCustomer.class, "firstName").observe(customer);
- bindingContext.bindValue(widgetValue, modelValue);

- Bind the content of the *firstNameTxt* text control with the *firstName* property of the customer object, of type *PojoCustomer*.
- When the user modifies the content of the text control, the *firstName* property of the customer object gets updated.

Validators

- *Validators* allows validating data before passing it.
- A validator must implement the *org.eclipse.core.databinding.validation.IValidator* interface.
- A validator is passed to an object of type *UpdateValueStrategy*
- The strategy object is used in the binding process.

Validators - Basic Usage

- UpdateValueStrategy strategy = new UpdateValueStrategy();
- IValidator validator = new AgeValidator();
- strategy.setBeforeSetValidator(validator);
- widgetValue = WidgetProperties.text(SWT.Modify).observe(ageTxt);
- modelValue = BeanProperties.value(BeanCustomer.class, "age").
  observeDetail(customer);
- Binding bindValue = bindingContext.bindValue(widgetValue, modelValue, strategy, null);

- Set a *validator* for the age property of a BeanCustomer object.

Converters

- Allows conversion between the data types.
- A converter must implement the *IConverter* interface.
- A converter is passed to an *UpdateValueStrategy* object.

Control decorators

- Are icons in the user interface used to reflect the status of the validation.
- Control decorators are created using the *ControlDecorationSupport* class.
-
- Binding bindValue = bindingContext.bindValue(widgetValue, modelValue, strategy, null);
- ControlDecorationSupport.create(bindValue, SWT.TOP | SWT.LEFT);

Writable Values

- Are objects that contains references to other objects.
- One can use the writable value object in a bind, instead of the contained object.
- One can set a new reference in the writable value object and the data binding will use the new set reference.
- This way, you can create the data binding once and be able to bind to different objects.
- Writable values are represented by the *WritableValue* class.
- To set a reference in a writable value object, the *setValue()* method is used.
- To bind writable values, the *observeDetail()* method should be used.

Writable Values - Basic example

- WritableValue customerValue = new WritableValue();
- IObservableValue widgetValue = WidgetProperties.text(SWT.Modify).
- observe(firstNameTxt);
- IObservableValue modelValue = BeanProperties.value(BeanCustomer.class, "firstName").
- observeDetail(customerValue);
- bindingContext.bindValue(widgetValue, modelValue);
- customerValue.setValue(customer);
-
- Bind the content of the *firstNameTxt* text widget to the *firstName* property of a *BeanCustomer* object via a *writable value*.