The FavaBeans Programmer's Guide

by Ihab Awad

The FavaBeans Programmer's Guide

by Ihab Awad

Table of Contents

List of Figures

Chapter 1. Scope and Approach

Introduction

In this chapter, we describe what FavaBeans is and the scope of the problems we are attempting to address by building it.

Object-Oriented User Interfaces

FavaBeans is a framework to help you build an object-oriented user interface (ooui) into your application using Java Swing. An ooui is a user interface (ui) presenting an *end-user* model of "objects" which can carry out certain behaviors and can be directly manipulated by their "views".

The arguments in favor of using an ooui are given in Collins95, and we highly recommend this book to you as a companion to your use of FavaBeans. Further information about oouis is available from Berry98.

Facilities Provided by FavaBeans

ooui implementations often make heavy use of drag and drop (dnd) interactors; FavaBeans supports this by presenting a simplified api that speeds up the process of adding dnd to your application, while still allowing you full access to the underlying Java Swing dnd api should you choose or need to use it.

FavaBeans comes bundled with a rich set of views for browsing collections, analogous to the "icon", "list" and "details" views commonly found in a modern operating system file browser ui. However, the FavaBeans collection views are not limited to file browsing, but are available to any part of your application. This fact dramatically improves the consistency of the your application's ui, and should help speed up your development.

FavaBeans includes a system "registry" which allows you to add specific views and other arbitrary ui functionality for your application objects. However, in the absence of specific knowledge about your objects, FavaBeans will use standard Java Beans introspection to build up simple, generic Bean property sheets and collection views. While these may not be suitable for distribution to your end-users, they are an important development aid, allowing you to concentrate on your application logic while your ui is still under development, and to develop your ui elements incrementally, using the generic views for temporary navigation through your application objects if need be.

Model and View Objects in FavaBeans

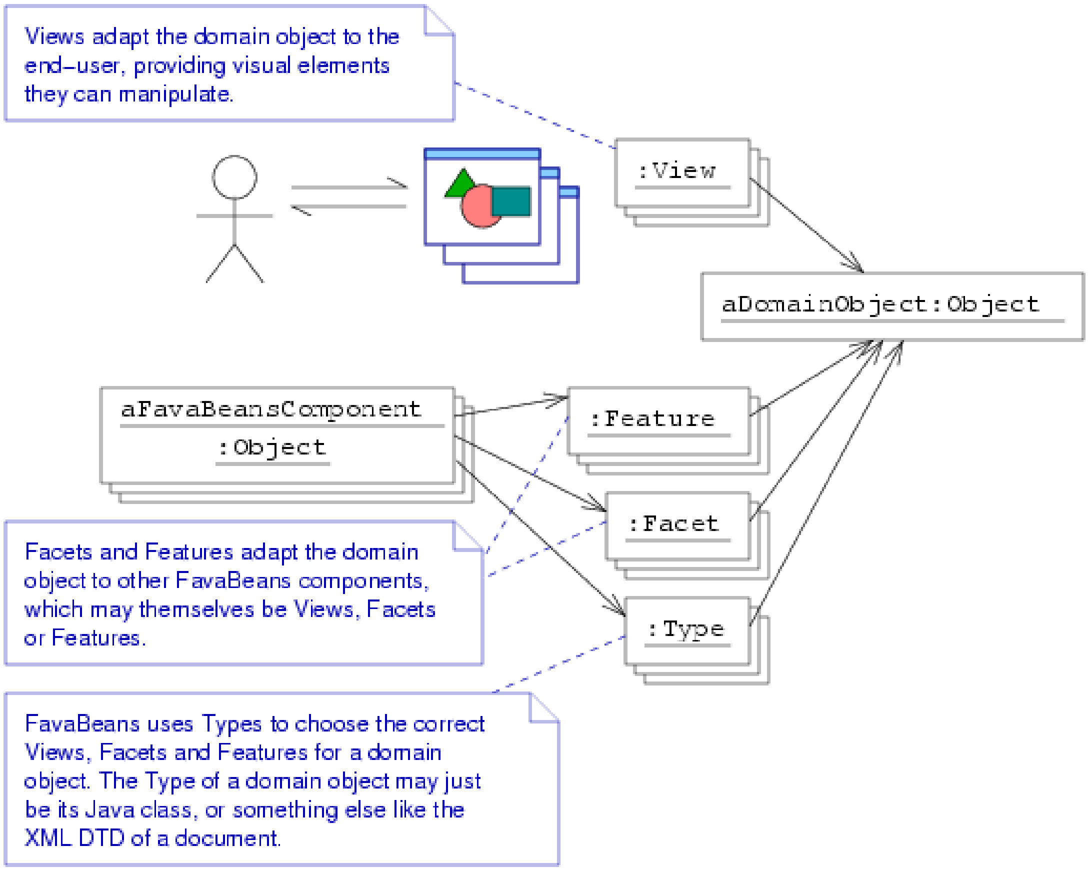
FavaBeans is based on the "Model, View, Controller" design paradigm as introduced in Smalltalk (Goldberg89 and Krasner88). However, we differ from the Smalltalk pattern in choosing to include the functionality of a Controller object in the View. This practice is well-established by Collins95, and its use in Java Swing is documented in Fowler00.

Collins95 suggests a nomenclature whereby the combined View/Controller object is referred to as a *delegate*, thereby calling the modified design pattern the "Model, Delegate" approach. We do *not* adopt this naming scheme; instead, we describe Model and View objects with the assumption that the View objects take on the role of a Controller as well.

High-Level Architecture of the Framework

The basic components of FavaBeans are shown in Figure 1.1.

Figure 1.1. Fundamental components of FavaBeans.


Views adapt the domain object to the end–user, providing visual elements they can manipulate.
:View
aDomainObject:Object
aFavaBeansComponent
:Object
:Feature
:Facet
:Type
Facets and Features adapt the domain object to other FavaBeans components, which may themselves be Views, Facets or Features.
FavaBeans uses Types to choose the correct Views, Facets and Features for a domain object. The Type of a domain object may just be its Java class, or something else like the XML DTD of a document.

We will describe these components in more detail in the ensuing chapters but, briefly:

- A domain object (*i.e.*, some part of the application, like an `Employee` or a `TechSupportCallLog`), is required to know nothing about FavaBeans.

- A `View` is the fundamental visible UI component in FavaBeans. Each `View` visually represents at most one domain object; on the other hand, a domain object may be represented by many `Views`.

- Various portions of the FavaBeans framework create `Facets` and `Features`; these act as adapters, presenting the capabilities of the domain object in a uniform way to other components in FavaBeans. Each domain object may have many `Facets` and `Features`.

- Programmers use FavaBeans via the following techniques, listed in ascending order of tight integration with the framework and need to know the details of the framework interfaces:

    - Adding information to the `TypeMetadataRegistry` accessible from class `FavaBeans`.
    - Creating new `Facet` and `Feature` implementations, or extending existing ones, which adapt domain objects to the standard elements of the framework, including standard `Views` (such as "icon views" of collections and "property sheets"); and --
    - Building new, application-specific `View` implementations.

- The FavaBeans framework associates a domain object with the correct `Views`, `Facets` and `Features` based on the `Types` that match the domain object. `Types` can be extended to implement various criteria for organizing domain objects, such as:

    - The Java class of the domain object, such as `myapp.datamodel.TechSupportCallLog`.
    - Elements of the Java class of the domain object, such as the fact that it publishes the `PropertyChangePpropertyChange` event set.
    - Other criteria, such as the XML DTD of the document represented by the domain object.

  A domain object may match more than one `Type`. The only restriction we impose is that the `Types` of a domain object may not change at run-time.

Chapter 2. Views

Introduction

FavaBeans exists to support and facilitate views -- *i.e.*, visual UI components -- which display the information in domain objects. In this chapter, we discuss the minimal requirements for a valid FavaBeans view.

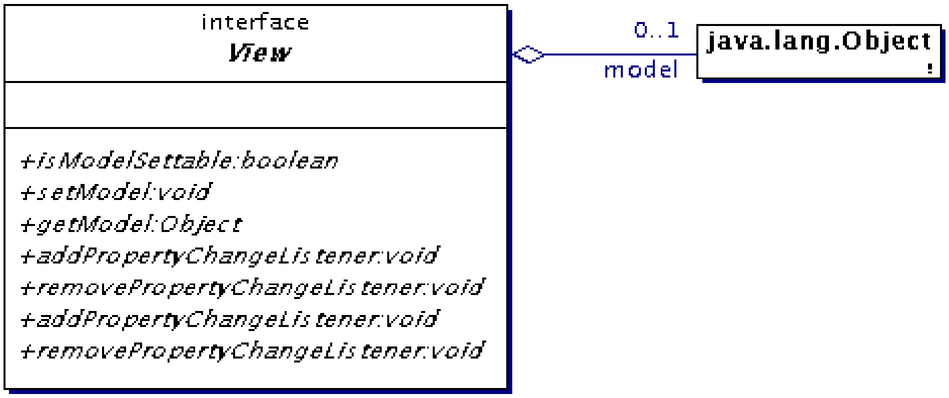
Interface View

A `View` is the fundamental component which displays some object. The actual interface is [perhaps somewhat deceptively] simple (Figure 2.1). In addition, we require that a compliant FavaBeans `View` implementation be a subclass of `Component`, thereby allowing it to be embedded in other UI elements.

Important

We may consider relaxing the requirement that a `View` be a subclass of `Component`; this would support frameworks where a view element is part of an abstract "scene graph" and not a directly paintable AWT/Swing component. We got this idea from Jazz [http://www.cs.umd.edu/hcil/jazz/], which we will probably be using as an additional "look and feel" for FavaBeans in the future.

Figure 2.1. Interface `View`.


interface
View
+isModelSettable:boolean
+setModel:void
+getModel:Object
+addPropertyChangeListener:void
+removePropertyChangeListener:void
+addPropertyChangeListener:void
+removePropertyChangeListener:void
0..1
model
java.lang.Object

The `View.ModelMmodel` property of a `View` permits a FavaBeans implementation to use a configured `View` repeatedly for viewing multiple objects.

Unresolved Issues

How will a `View` interact (if at all) with `java.beans.beancontext` stuff?

How will a composite `View` make this fact known? Through its `Folder` `Facet`? How does that relate to the composition inherent in `Container` and `BeanContext`?

Chapter 3. The Type System

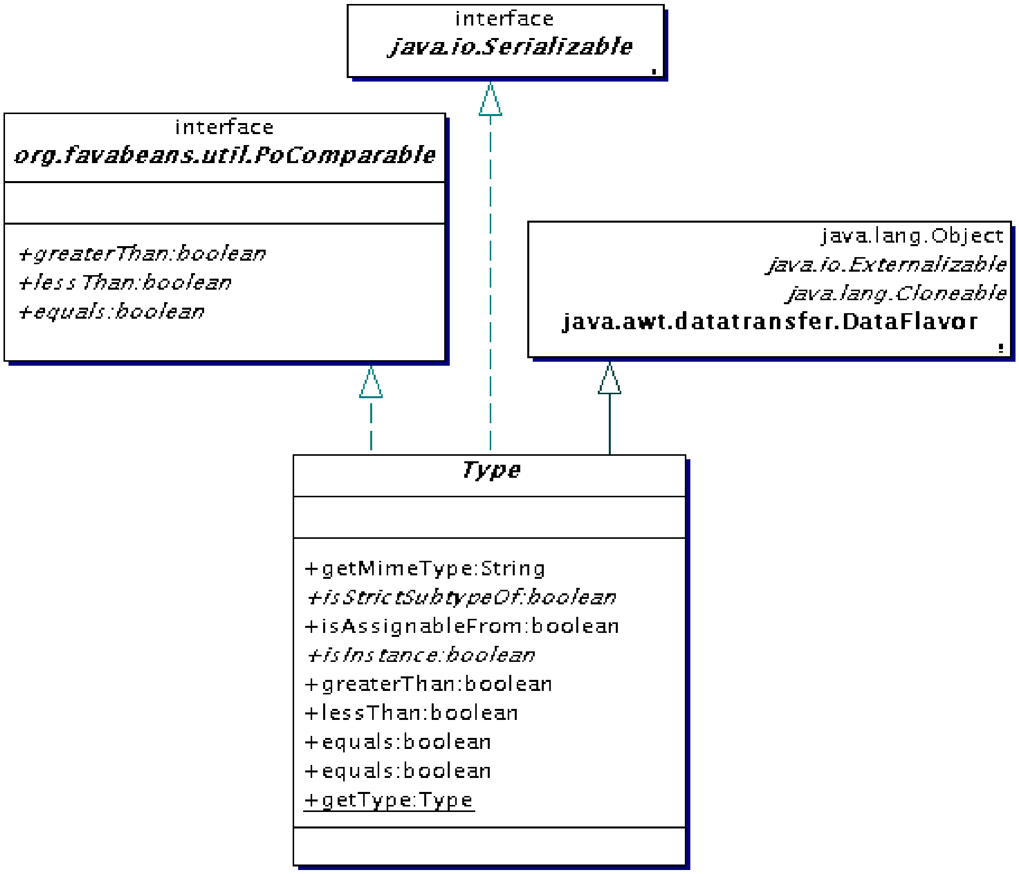
Introduction

In this chapter, we describe the fundamental system of data "types" that underlies the categorization and display of objects in a running instance of FavaBeans.

Class Type

A `Type` object is an analogue of a Java class; it allows us to categorize objects, but provides for more fine-grained control over this categorization than would be possible were we to categorize objects based on their Java class alone. See Figure 3.1 for more details; the additional classes shown in this class diagram are discussed below.

Figure 3.1. Class `Type`.



Class `Type` implements `PoComparable`, allowing us to use it in partially ordered data structures.

We have made the simplifying design decision that the Type of an object does not change at run-time. Without this simplification, the state machine for the objects in our framework would be complicated by having to handle "type changed" events.

Important

We may relax the requirement that the Type of an object not change at run-time later on.

Notably absent from the contract of class Type is any notion of uniqueness of the of the Type of a given object. On the other hand, any subsystem responsible for returning the Type of an object can and should return at least the appropriate JavaType (see below); effectively, this is a guarantee that there exists a minimum of one Type for any Object.

A developer could, if they so wished, define subclasses of Type which recognize information that is not necessarily representable by an Object; the most obvious example would be a Type associated with a Java primitive data type such as integer. We neither recommend nor discourage this approach, but we note that such a Type may not be very useful, as it would be unable to recognize instances of itself via the Type.isInstance(Object) method.

Subclasses of class Type

The FavaBeans framework includes two pre-defined subclasses of Type; both are depicted in Figure 3.1.

Class JavaType is a simple extension of Type which describes a single Java class or interface. Accordingly, it is constructed with one Class argument.

A DataSourceType represents a type of MIME-typed data stream. It is a bridge to the type system defined by the Java Activation Framework (JAF), and especially to class DataSource. As such, given:

```
String mimetype = "text/html"; // for example . . .
Type[] supers = new DataSourceType(mimeType).getSupertypes();
```

the following is always true:

```
(supers.length == 1) &&
supers[0].equals(new JavaType(javax.activation.DataSource));
```

Associating Objects with Values: The TypeMetadataRegistry

A TypeMetadataRegistry assigns an object to a value via the best matching Type of the object. It is one of the main ways whereby FavaBeans requires no intrusion into the domain object model: we can bind all the view-related information that we might need to domain objects solely by defining the appropriate Types and adding information to TypeMetadataRegistrys in the UI.

As a practical and simple example, let's assume that we have the following classes:

```
Class Person { /* ... */ }
class Employee extends Person { /* ... */ }
class Doctor extends Employee { /* ... */ }
```

and we choose some icons to be used for representing instances of some of these classes, and add them to some centrally available TypeMetadataRegistry:

```
Icon personIcon = /* ... */;
Icon employeeIcon = /* ... */;

TypeMetadataRegistry tmr = /* ... */;

tmr.put(new JavaType(Person.class), "icon", personIcon);
tmr.put(new JavaType(Employee.class), "icon", employeeIcon);
```

A UI element could then use this TypeMetadataRegistry to display the icons for a number of objects:

```
TypeMetadataRegistry tmr = /* ... */;

Person[] people = new Person[] {
    new Person("Pat Okoye"),
    new Employee("Joy Albright"),
    new Doctor("Peace Freeman"),
};

for (int i = 0; i < people.length; i++) {
    Icon theIcon = (Icon)tmr.getForObject(people[i], "icon");
    /* display the object using 'theIcon' */
}
```

and, in this way, use our preferred icons for the Person and Employee objects. Furthermore, the UI element would automatically represent the Doctor object via the best matching icon which, in our example, happens to be that which we associated with class Employee.

Object Facets

A Facet is an object which provides an alternative representation of another object. Examples of a Facet could be:

- An object that represents an XML file in a filesystem as a parse tree of XML node objects.
- A "persistence" facet of an object, to which the responsibilities for making the object persistent in some storage medium are delegated.
- An object which represents an Employee object, from the domain model of some application, as a Drawable with the capability to be drawn on a Canvas.

The FavaBeans Facet objects owe their lineage in part to the following previous work:

- The GoF *Decorator* design pattern (Gamma95).
- The Facet interface in ObjectSpace Voyager (ObjectSpaceVoyager).
- The definition of "interfaces" by delegation, as used in Microsoft's [D]COM (MicrosoftCOM).

Creating Facets: Faceted and the FacetRegistry

An object implements Faceted in order to be tightly integrated into the Facet system. An object returns a Facet of itself of a requested Type via its Faceted.getFacet(Type) method. Any lifecycle management of Facets is the responsibility of the Faceted object.

More typically -- and more powerfully -- we can create a Facet for an object that is not aware of the Facet system; we do this with the help of a FacetRegistry. Some "startup" component initializes a FacetRegistry (typically that available via class FavaBeans) with information about the Types of objects for which Facets may be created, and some FacetFactory objects to which the FacetRegistry may delegate the responsibility for actually constructing Facets. A FacetRegistry is responsible for maintaining a list of already constructed Facets for an object.

A typical interaction with a FacetRegistry starts by adding some FacetFactory objects. Given:

```
interface WebAccessible { /* ... */ }
interface HtmlAccessible extends WebAccessible { /* ... */ }

interface Vehicle { /* ... */ }

Vehicle theVehicle; /* assuming already exists */
FacetFactory fac0, fac1; /* assuming these already exist */

FacetRegistry theFacets = /* ... */;
```

We could add:

```
theFacets.addFactory(new JavaType(Vehicle.class),
                     new JavaType(WebAccessible.class),
                     fac0);

theFacets.addFactory(new JavaType(Vehicle.class),
                     new JavaType(HtmlAccessible.class),
                     fac1);
```

We could then use the FacetRegistry to obtain an HtmlAccessible Facet of the Vehicle object:

```
HtmlAccessible h0 = (HtmlAccessible)
  theFacets.getFacet(theVehicle, new JavaType(HtmlAccessible.class));
```

The FacetRegistry would cache this Facet so that:

```
theFacets.getFacet(theVehicle, new JavaType(HtmlAccessible.class)) == h0;
```

and would return the same Facet for compatible supertypes of that for which it was originally constructed:

```
theFacets.getFacet(theVehicle, new JavaType(WebAccessible.class)) == h0;
```

until the FacetRegistry.clearFacets(Object) method is called:

```
theFacets.clearFacets(theVehicle);
```

after which Facets for this object are created anew:

```
HtmlAccessible h1 = (HtmlAccessible)
  theFacets.getFacet(theVehicle, new JavaType(HtmlAccessible.class));
h1 != h0;
```

Important

We need to address lifecycle issues for Facet objects. This will apply to all things like views, *etc.*, and will be generally useful. This is all complicated by the messy strong references in Java AWT and Swing for event listeners, which may prevent some components from getting garbage collected correctly. This will need to be addressed at some point.

Summary

Type objects are the analogue of Java classes and behave similarly to Class objects. FavaBeans provides class TypeMetadataRegistry for binding arbitrary information to instances of a Type.

Facets are an alternative representation of another object, which we refer to as the *primary object* of the Facet. FavaBeans provides class FacetRegistry whereby Facets can be constructed for a primary object based on the Types of the primary object and the desired Facet. The work of actually constructing a Facet is delegated to a FacetFactory.

Chapter 4. The Standard Facets

Introduction

In this chapter, we list the standard sub-interfaces of `Facet` which a FavaBeans implementation should provide. These are the fundamental manner in which domain objects, developed independently of FavaBeans, are integrated into the framework and given the standard behaviors that elements of the framework are expected to have.

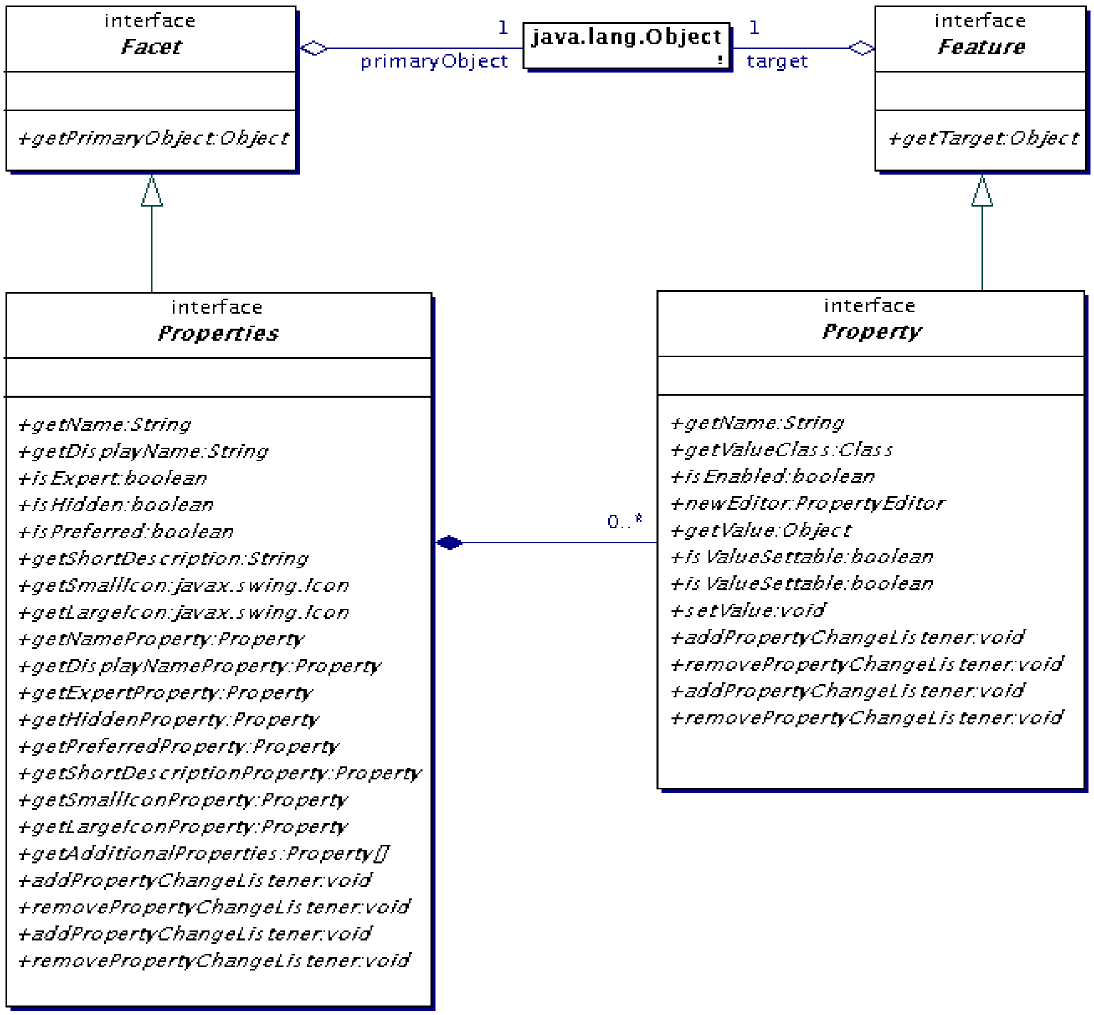
Interface Feature

Before we say any more about `Facets`, we should first introduce another interface: `Feature`. A `Feature` has a *target* (see `Feature.TargetTtarget`) some information about which it provides; there is not, however, necessarily a unique `Feature` of a given `Feature` for a given object. This distinctino will become clearer as we present examples of actual usage.

The Properties Facet

The `Properties` of an object (Figure 4.1) provides the basic functionality to display the primary object in the UI.

Figure 4.1. The `Properties` `Facet`.


interface
Facet
+getPrimaryObject:Object
1
primaryObject
java.lang.Object
1
target
interface
Feature
+getTarget:Object
interface
Properties
+getName:String
+getDisplayName:String
+isExpert:boolean
+isHidden:boolean
+isPreferred:boolean
+getShortDescription:String
+getSmallIcon:javax.swing.Icon
+getLargeIcon:javax.swing.Icon
+getNameProperty:Property
+getDisplayNameProperty:Property
+getExpertProperty:Property
+getHiddenProperty:Property
+getPreferredProperty:Property
+getShortDescriptionProperty:Property
+getSmallIconProperty:Property
+getLargeIconProperty:Property
+getAdditionalProperties:Property[]
+addPropertyChangeListener:void
+removePropertyChangeListener:void
+addPropertyChangeListener:void
+removePropertyChangeListener:void
0..*
interface
Property
+getName:String
+getValueClass:Class
+isEnabled:boolean
+newEditor:PropertyEditor
+getValue:Object
+isValueSettable:boolean
+isValueSettable:boolean
+setValue:void
+addPropertyChangeListener:void
+removePropertyChangeListener:void
+addPropertyChangeListener:void
+removePropertyChangeListener:void

Parts of `Properties` are very similar in purpose -- and, therefore, deliberatey similar in syntax -- to a `FeatureDescriptor`. Specifically the "standard properties" are the information obtained by methods such as `Properties.DisplayNameDdisplayName` and `Properties.SmallIconSsmallIcon` is intended for UI elements to be able to display a minimal iconic representation of the primary object.

The standard properties are also obtainable as full-fledged `Property` objects. A `Property` is a that encapsulates some sort of information about its target. Methods to obtain these are, for example, `Properties.DisplayNamePropertyDdisplayNameProperty` and `Properties.SmallIconPropertySsmallIconProperty`.

Finally, zero or more additional `Property` objects, built depending on the application at hand, are available via the `Properties.AdditionalPropertiesAadditionalProperties` method.

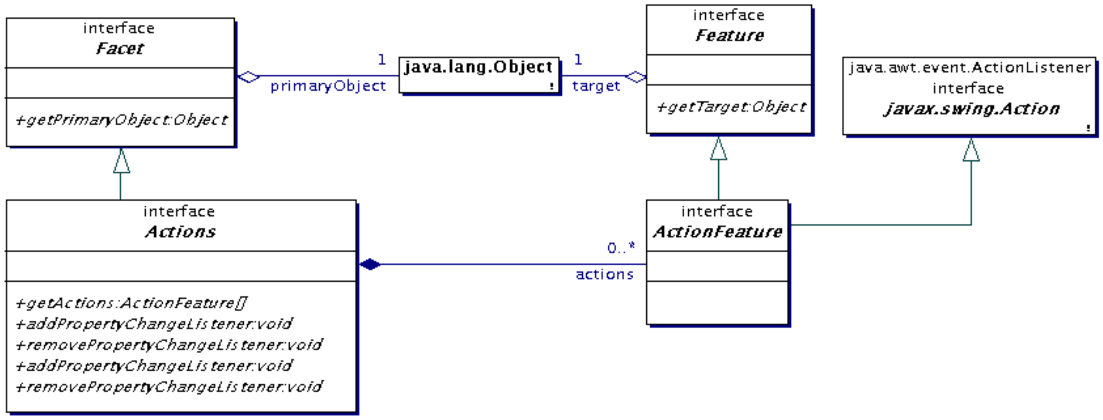
Important

The number of additional properties is allowed to change at run-time; views that depend on this fact are expected to listen for `PropertyChangePpropertyChange` events, with a key of *additionalProperties*, from the `Properties` object they are using.

The Actions Facet

An `Actions Facet` (Figure 4.2) provides a list of `ActionFeature` objects representing commands that the end-user may execute its primary object. These may be plaed in a menu or on a toolbar, or otherwise presented, at the discretion of relevant UI elements.

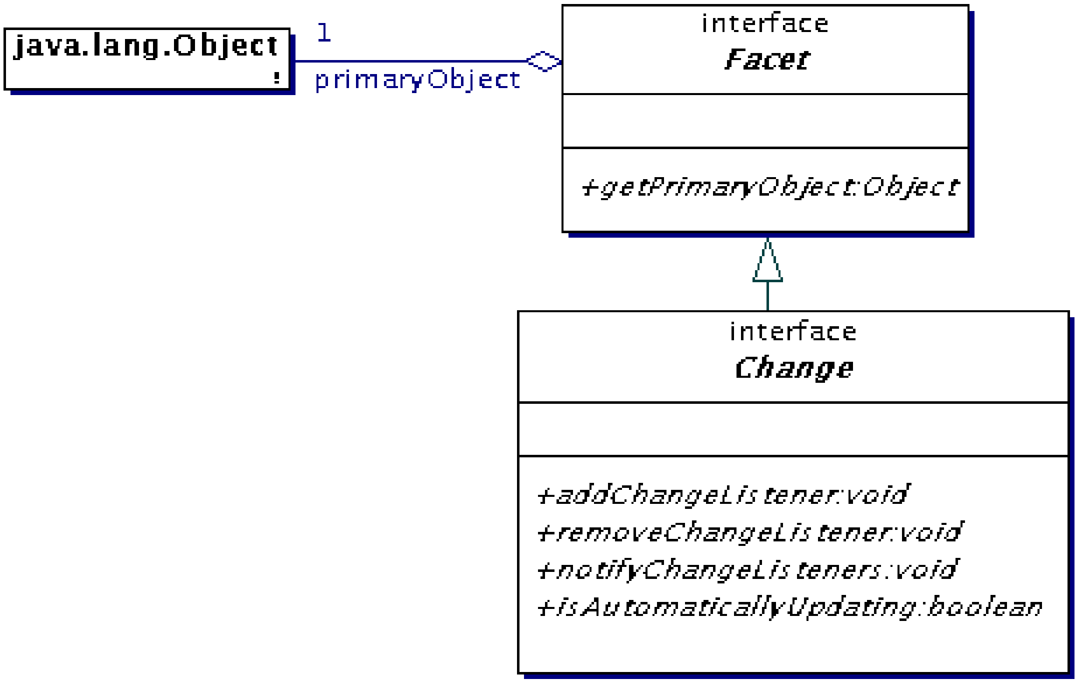
Figure 4.2. The `Actions Facet`.


interface
Facet
+getPrimaryObject:Object
1
primaryObject
java.lang.Object
1
target
interface
Feature
+getTarget:Object
java.awt.event.ActionListener
interface
javax.swing.Action
interface
Actions
+getActions:ActionFeature[]
+addPropertyChangeListener:void
+removePropertyChangeListener:void
+addPropertyChangeListener:void
+removePropertyChangeListener:void
0..*
actions
interface
ActionFeature

The Change Facet

The `Change Facet` of an object (Figure 4.3) publishes the `ChangeCchange` Bean event set and fires an event whenever the primary object has changed in a manner that will affect observers.

Figure 4.3. The `Change Facet`.



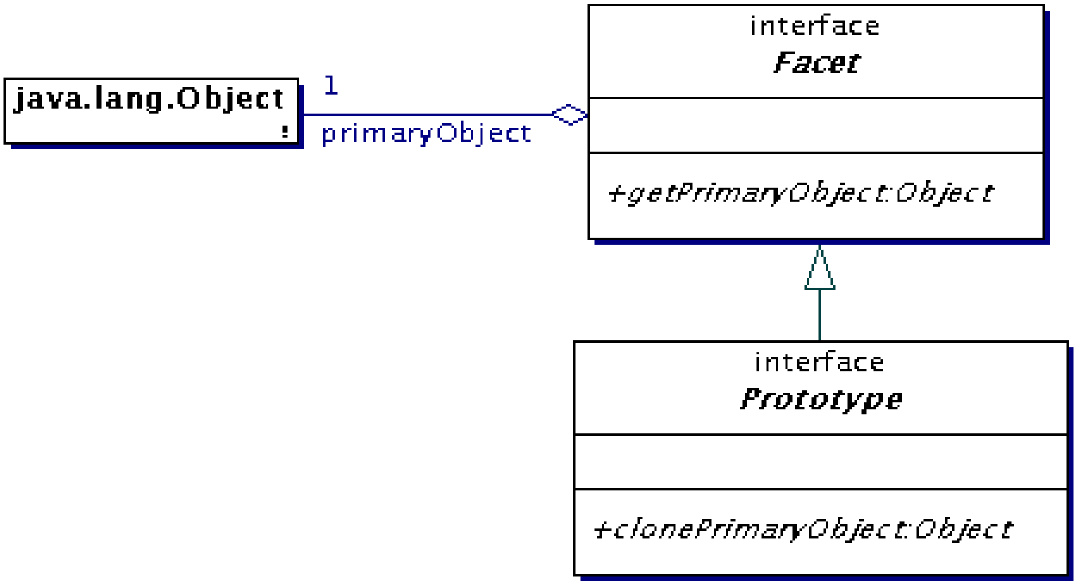
Other objects (including other `Facet`s of the primary object) may utilize the `Change Facet` as a way to isolate themselves from the details whereby an object indicates a state change.

The `Change` interface is designed to accommodate -- and shield other components from the distinctions between -- cases where the primary object informs clients of changes to itself and cases where the end-user must manually trigger an "update" of all views. The `Change.AutomaticallyUpdatingAautomaticallyUpdating` property indicates whether the associated `Change Facet` is listening for events from its primary object. If this property is `true`, then calls to its `Change.fireChange()` method should be no-ops. However, if this property is `false`, each call to `Change.fireChange()` will force the broadcast of a `ChangeEvent`.

The Prototype Facet

The `Prototype Facet` of an object (Figure 4.4) represents its capability to clone itself. This is an operation that is invokeable by the end-user, and is intended to be a higher-level construct than that provided by `Object.clone()` (but with similar semantics).

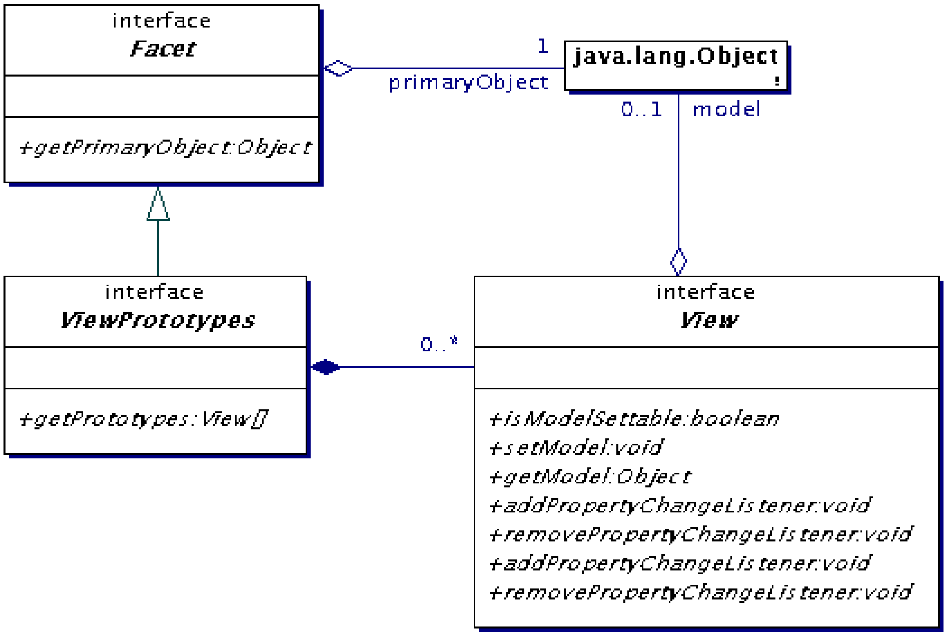
Figure 4.4. The `Prototype Facet`.



The ViewPrototypes Facet

Each object has a `ViewPrototypes Facet` (Figure 4.5) which presents a set of prototypical `View` objects.

Figure 4.5. The `ViewPrototypes Facet`.



Given the settting:

```
Object employee = /* ... */;
FacetRegistry fr = /* ... */;
```

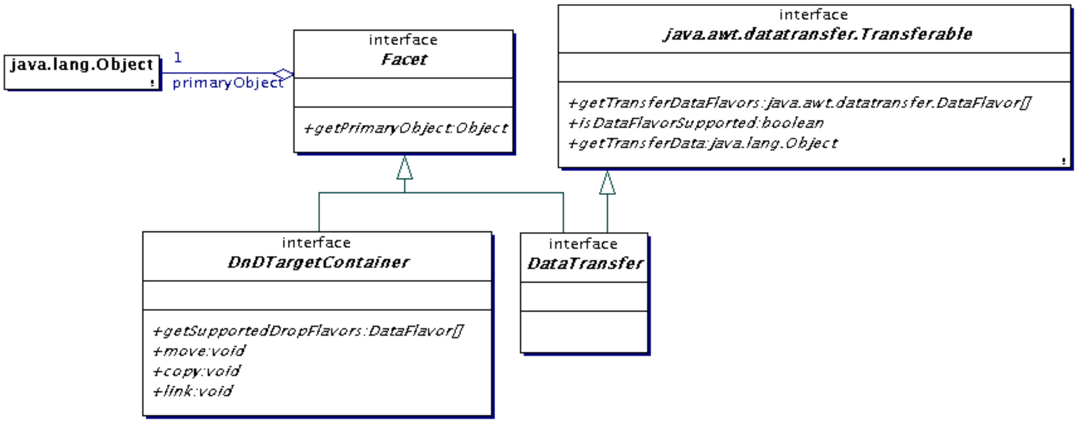
the canonical manner in which a new `View` may be created is:

```
// Get the ViewPrototypes Facet of the domain object

ViewPrototypes fvs = (ViewPrototypes)
  fr.getFacet(employee, new JavaType(ViewPrototypes.class));

// Choose one of the Views available

View someProtoView = fvs.getViewPrototypes()[0]; // say

// Get the Prototype Facet of the chosen View, and create
// a new View using its methods

Prototype s = (Prototype)
  fr.getFacet(someProtoView, new JavaType(Prototype.class));
View newView = (View)s.clonePrimaryObject();

// Tell the newly created View to display the original
// domain object

newView.setModel(employee);
```

Drag and Drop

There exist two `Facets`, shown in Figure 4.6, which support Drag and Drop (DnD). They simplify the work of developers by shielding them from the complexities of the standard Java DnD code (`java.awt.dnd`) while still allowing them to expose DnD functionality to UI elements that are DnD-capable.

Figure 4.6. The Drag and Drop (DnD) `Facets`.


java.lang.Object
1
primaryObject
interface
Facet
+getPrimaryObject:Object
interface
java.awt.datatransfer.Transferable
+getTransferDataFlavors:java.awt.datatransfer.DataFlavor[]
+isDataFlavorSupported:boolean
+getTransferData:java.lang.Object
interface
DnDTargetContainer
+getSupportedDropFlavors:DataFlavor[]
+move:void
+copy:void
+link:void
interface
DataTransfer

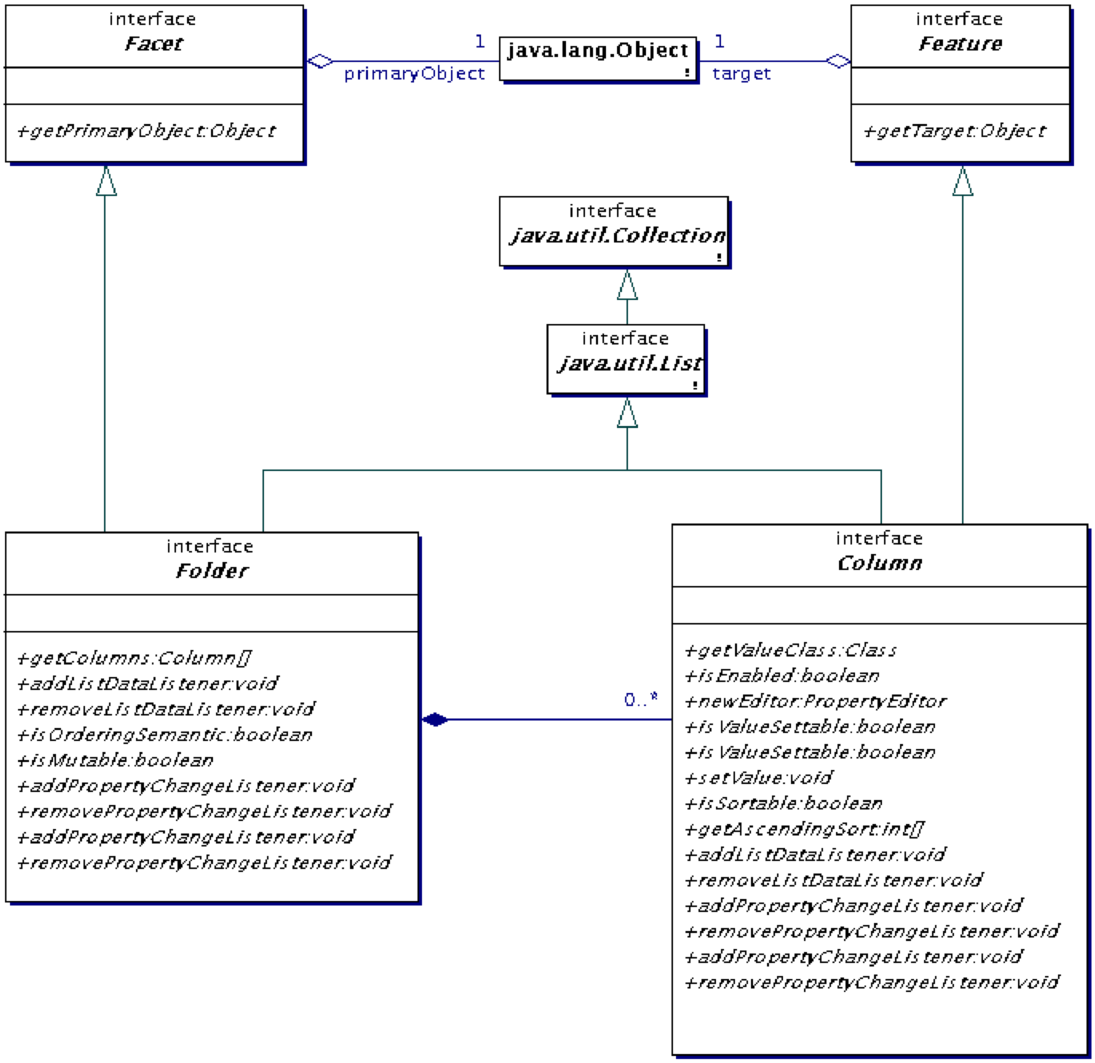
The `DnDTargetContainer Facet` provides the behavior implemented when an object is displayed in some sort of iconic form, and something is dropped onto it. While custom views may provide more sophisticated DnDbehavior, simple actions, like dropping a document onto a printer icon (to print it), can be handled by this `Facet`.

The `DataTransfer Facet` of an object provides a way for an object to publish the `Types` into which it can convert itself. This functionality is used in two main cases: (a) by the DnD subsystem, when the object is the source of a drag operation; and (b) by "clipboard" operations like "cut", "copy" and "paste".

The Folder and Column Facets

An object which is representable as a collection of other objects may have a `Folder Facet` (Figure 4.7) which enables it to be displayed in a generic "folder" UI element.

Figure 4.7. The `Folder` and `Column Facets`.


interface
Facet
+getPrimaryObject:Object
1
primaryObject
java.lang.Object
1
target
interface
Feature
+getTarget:Object
interface
java.util.Collection
interface
java.util.List
interface
Folder
+getColumns:Column[]
+addListDataListener:void
+removeListDataListener:void
+isOrderingSemantic:boolean
+isMutable:boolean
+addPropertyChangeListener:void
+removePropertyChangeListener:void
+addPropertyChangeListener:void
+removePropertyChangeListener:void
0..*
interface
Column
+getValueClass:Class
+isEnabled:boolean
+newEditor:PropertyEditor
+isValueSettable:boolean
+isValueSettable:boolean
+setValue:void
+isSortable:boolean
+getAscendingSort:int[]
+addListDataListener:void
+removeListDataListener:void
+addPropertyChangeListener:void
+removePropertyChangeListener:void
+addPropertyChangeListener:void
+removePropertyChangeListener:void

A `Folder` provides a list of `Column` objects, each of which represents some property of all the elements of the `Folder`. The `Columns` can be used by UI elements to provide a tabular view of the contents of the `Folder`, and to sort the contents of the `Folder` based on the property represented by the `Column`.

Important

The `FolderFacet` should provide certain standard columns, exactly equivalent to the "standard properties" of `Propertiess`. For example, we would have methods `getDisplayName(int)` and `getDisplayNameColumn()`.

Default Facet Implementations

A newly constructed FacetRegistry provides default FacetFactory implementations bound to the JavaType of class Object. These provide instances of the following Facets:

- Description. Displays a generic icon and selects reasonable defaults for various properties.
- Change. Fires events based on one of the following techniques:

  1. If the primary object implements Observable, we use it; or --
  2. If the primary object publishes the ChangeCchange event set, we listen for it.

  Otherwise, sets the Change.AutomaticallyUpdatingAautomaticallyUpdating property to false and fires events only when Change.fireChange() is called.
- Actions. Provides access to all public methods of the object. Also provides an "update views" action if the Change.AutomaticallyUpdatingAautomaticallyUpdating of the primary object's Change Facet is false; this action invokes the Change.fireChange().
- Properties. Uses standard Java Bean introspection to present the Bean properties of the object as a list of Property objects.
- Prototype. Attempts to use the standard Object.clone() method on the primary object to create and return a new instance of the primary object; throws a FactoryException if this operation throws a CloneNotSupportedException or if some other error happens.
- DnDTargetContainer. A no-op implementation that never accepts any drop operation.
- ViewPrototypes. Contains a single View that displays a "property sheet" for the primary object using its Properties Facet.

In addition, FacetFactory implementations bound to the JavaType of interface Collection provide instances of the following Facets:

- Folder. Provides a list of the contents of the primary object, ordering them as presented by whatever Iterator is provided by the Collection.iterator() method of the primary object. Computes the most general Java superclass and super-interfaces of the contents, and presents each Java Bean property of these as a Column.
- ViewPrototypes. Contains one or more Views that display various "icon views" and tabular "details views" of the primary object using the information provided by its Folder Facet and the associated Columns.

Facets of Views

A View is a first-level object in FavaBeans and can itself be viewed, dragged and dropped, and otherwise manipulated. These behaviors are, in turn, mediated by the View's standard Facets.

Summary

Chapter 4. The Standard Facets

The interaction of UI elements with domain objects in FavaBeans is mediated by a variety of `Facet` objects. These are, briefly:

- `Change`. A uniform way to be notified of changes to the state of the primary object.
- `Actions`. Commands the end-user may execute, usually shown on a toolbar or menu.
- `Properties` and `Property`. Information, often editable, about the primary object; usually shown in a property sheet, and used to display the object as an icon.
- `Prototype`. Behavior that allows an end-user to create a clone of the primary object.
- `DnTargetContainer` and `DnDTarget`. Behavior used to implement responses to DnD operations.
- `Folder` and `Column`. Representation of the primary object as a collection, and columns of information permitting a tabular view of the items in the collection.
- `ViewPrototypes`. Presents a list of `View` objects which can be cloned (via their `Prototype Facets`) to construct new views on the original primary object.

A newly constructed `FacetRegistry` contains convenient default `FacetFactory` objects for the following `Facets`: `Change`; `Actions`; `Properties`; `Prototype`; `DnDTargetContainer`; `Folder`; and `ViewPrototypes`.

Bibliography

[Goldberg89] *Smalltalk 80 The Language*. Adele Goldberg and David Robson. Addison-Wesley. 1989. ISBN 0-201-13688-0.

[Krasner88] *Journal of Object-Oriented Programming* . Aug/Sep 1988. *A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80* . Glenn Krasner and Stephen Pope. 29-49. .

[Collins95] *Designing Object-Oriented User Interfaces* . Benjamin/Cummings. 1995. Dave Collins. [amazon.com] [http://www.amazon.com/exec/obidos/ASIN/080535350X/o/qid=971446149/sr=8-1/ref=aps_sr_b_1_3/104-7333383-6311 .

[Fowler00] Amy Fowler. *A Swing Architecture Overview*. The Inside Story on JFC Component Design. 2000. [html] [http://java.sun.com/products/jfc/tsc/articles/architecture/] .

[Berry98] *OVID: An Overview*. Richard Berry, Scott Isensee, John Mullaly, and Dave Roberts. 1997,1998. IBM Corporation. [html] [http://www-3.ibm.com/ibm/easy/eou_ext.nsf/Publish/104] .

[AppleOD] *Mac OS 8 and 9 Developer Documentation: OpenDoc*. Apple Corporation. [html] [http://devworld.apple.com/techpubs/macos8/Legacy/OpenDoc/opendoc.html] .

[ObjectSpaceVoyager] *ObjectSpace Voyager*. ObjectSpace Incorporated. [html] [http://www.objectspace.com/products/voyager/] .

[MicrosoftCOM] *Microsoft COM Technologies - Information and Resources for the Component Object Model-based technologies* . Microsoft Corporation. [html] [http://www.microsoft.com/com/] .

[Gamma95] *Design Patterns, Elements of Reusable Object-Oriented Software* . Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Addison-Wesley. 1995. ISBN 0-201-63361-2.