



Creating, Reusing and Executing Components in Evolve

© Andrew McVeigh, 15th November 2010

What is Evolve?

Evolve is a tool for defining components and connecting them together to form new components. It also provides a runtime engine to execute these configurations. In other words, you use it to create applications out of plain vanilla classes, using an interface-centric approach.

You may be wondering: how does this differ from Spring, Guice, Unity or any other run-of-the-mill dependency injection approach? The simple answer is that Evolve is based on a remarkable hierarchical component approach derived from over 20 years of advanced research, that other approaches can only clumsily approximate. This is basically an “electronic circuit board and chip” metaphor for software, where component instances are joined together using connectors to form larger components. The underlying foundation scales well to support large, complex systems, but can seamlessly handle fine-grained components too. It provides powerful facilities for reuse and evolution, allowing multiple variants of an application to be built quickly and easily.

In short, Evolve allows you to create principled and elegant systems that have a direct and powerful link between architecture and implementation. The systems you create using Evolve are naturally extensible and can be easily adapted for new requirements.

And surprisingly, it's a very straight-forward approach that is most likely simpler than what you are currently using despite being far more capable. Curious? Read on and find out how powerful and liberating components can truly be.

Contents

1	Introduction	6
1.1	What is Evolve?	6
1.1.1	Highlights of Evolve	7
1.1.2	A Quick Peek Inside Evolve	8
1.2	What is a Component?	10
1.2.1	A Leaf Component	10
1.2.2	A Composite Component	12
1.3	Compositional Hierarchy	13
1.4	The Backbone Component Language	14
1.5	A Quick Review	15
2	Installing and Using Evolve	16
2.1	Installing Evolve	16
2.2	Setting Up The Environment	17
2.3	Navigating Around the Tutorial Model	19
2.4	Compiling the Tutorial Files	21
2.5	Subjects and Views and the Subject Browser	22
2.6	Running A Backbone Program	24
3	Overview of the Tutorials	25
4	Tutorial A: Creating and Composing Components	28
4.1	Defining the Leaves	29
4.2	The RentalCar Composite	32
4.3	Making the Example Runnable	33
4.3.1	Indexed Connectors	34

4.4	Generating Code	35
4.4.1	Checking the Model	35
4.4.2	Generating Java and Backbone Code	36
4.4.3	Lifecycle Callbacks	39
4.5	Running the Model	39
4.5.1	Running Using the Backbone Interpreter	40
4.5.2	Running Outside of Evolve	40
4.5.3	Calling the Backbone Interpreter from a Program	41
4.5.4	Running Without the Backbone Interpreter	41
4.6	Importing Beans and Refreshing Leaves from Code	42
4.7	Summary	43
5	Tutorial B: Resemblance and Evolution	44
5.1	Component Resemblance	44
5.1.1	Creating Deltas	45
5.1.2	Viewing and Manipulating Deltas	46
5.1.3	Stereotypes and Resemblance	47
5.1.4	Using Resemblance to Define a Leaf	47
5.1.5	Renaming Inherited Constituents	48
5.2	Component Evolution	48
5.2.1	How to Evolve a Component	48
5.2.2	Evolution = Resemblance + Replace	49
5.2.3	Remaking the Compositional Hierarchy	50
5.2.4	Using Evolution to Rename a Component	50
5.3	Interface Resemblance and Evolution	50
5.4	Evolution Without Changing the Implementation Class	51
5.5	Retirement	51
5.6	Strata	51
5.6.1	Organizing a Model	52
5.6.2	Relaxed and Strict Strata	52
5.6.3	Read only Strata	53
5.6.4	Strata Perspective	53
5.6.5	Destructive Strata	55

5.6.6	Nested Strata and Packages	55
5.6.7	Strata as a Unit of Ownership	55
5.7	Primitives	56
5.8	Summary	56
6	Tutorial C: Factories, Hyperports and States	57
6.1	Factories	57
6.1.1	Using the Creator Port	58
6.1.2	Factories Are Isomorphic	60
6.1.3	Factories Can Instantiate Complex Structures	60
6.1.4	Factories Can Be Nested	61
6.1.5	How Do Factories Work?	61
6.2	Hyperports	62
6.2.1	Connecting Through a Hierarchy	62
6.2.2	Hyperports Instead of Singletons	64
6.2.3	Evolution Instead of Aspects	65
6.3	State Machines	66
6.3.1	A Top-Down Overview	67
6.3.2	Running the Machine	69
6.3.3	Creating the State Machine and Leaf States	70
6.3.4	Replacing a Part with One Of a Different Type	72
7	Tutorial D: The GWT / Hibernate Car Rental System	76
7.1	Setting Up and Running the Example	78
7.1.1	Setting Up Eclipse	79
7.1.2	Setting Up the H2 Database	79
7.1.3	Setting Up the Evolve Variables	80
7.2	Creating the Back End: The Rental Car Service	81
7.2.1	Creating the GWT Service Interface	81
7.2.2	Creating a Service Without Persistence	81
7.2.3	Creating a Service With Persistence	83
7.2.4	Mapping the Components Using Hibernate	84
7.2.5	Alternating Between the Persistent and Non-Persistent Variants	85

7.3	Creating the User Interface	86
7.3.1	Importing the GWT Widgets into Evolve	86
7.3.2	Connecting Up Widgets	89
7.3.3	Quick Turnaround	92
8	Documenting Evolve Models	93
8.1	Marking Up Figures and Exporting Them As Images	93
8.2	Linking to Images in a Wordprocessor	94
8.2.1	Using Word	94
8.2.2	Using Lyx	95
8.2.3	Using OpenOffice	95
8.3	A Few Tips	96
9	Advanced Modeling in Evolve: A Teaser	97
9.1	Highly Extensible Systems	97
9.1.1	Evolve as a Way of Avoiding Framework Syndrome	98
9.1.2	Sharing Parts of a Model in a Collaborative Environment	98
9.1.3	Evolve as a Plugin System	99
9.2	The Interaction Between Resemblance and Evolution	99
9.3	Extensible Feature Modeling	101
9.4	Reengineering a Legacy Application	102
9.5	Evolution Instead of Aspects	103
9.6	Product Lines	103
9.7	Controlling Component Complexity Through Visual Locking	103
9.8	Strata as Modules	104
9.9	Extending State Machines	104
10	Product Roadmap and Further Reading	105
10.1	The Evolve Roadmap	105
10.1.1	Team System	105
10.1.2	Support for Other Implementation Languages	105
10.1.3	Concurrency Constructs	105
10.1.4	Web Publishing System	105
10.1.5	Protocol Analysis	106
10.1.6	Internet Distribution System and Licensing Manager	106
10.2	Further Reading	106

Chapter 1

Introduction

1.1 What is Evolve?

In essence, Evolve is a convenient and powerful way to connect together classes to create a program. It also features innovative reuse and evolution facilities, allowing you to quickly customize an application for new requirements, without destroying the old version.

Several years ago as the technical lead for a large Java enterprise application, I started wondering about components. The system we constructed had a number of natural abstractions which could have been loosely termed components but we struggled to separate these out from each other or find their true essence. After thinking deeply about this, I “invented” a theory of hierarchical components which would allow me to break a system down neatly into connected components. Or at least I thought I invented this approach...

Of course I didn't really invent it at all - I rediscovered it. It turned out that hierarchical components had been discovered over 10 years before¹. That work was a large part of the inspiration for Microsoft's COM and Philips also used an embedded version for the software in some of their television sets.

Evolve is a fusion of a UML2 CASE tool with this powerful, hierarchical component technology. It allows you to connect up classes like an electronics designer wires up chips. Components are created in Evolve, using UML2 composite structure diagrams, which can then be executed. At all times, the link between the graphical view and the code is kept intact. Components at the lowest level are simply plain classes, in keeping with JavaBeans² conventions, allowing you to take advantage of your existing libraries and create new ones easily.

The takeaway from the above is that Evolve allows you to define systems where the architecture and implementation are always kept synchronized. This provides all of the advantages of dependency injection approaches, with far more power and none of the disadvantages. The applications

¹By Professors Jeff Magee and Jeff Kramer from Imperial College, London. That system was called Darwin.

²Although JavaBeans is often used to refer to Swing/AWT widgets, we use the term to represent a simple class with getters and setters.

created are naturally extensible, and can be adapted quickly for new features. Reassuringly, this is all backed by a formal specification and sits on the bedrock of over 20 years of advanced research into components.

1.1.1 Highlights of Evolve

Some of Evolve's key features are listed below.

- Components are defined using standard UML2 diagrams.
- JavaBeans can be treated as full components.
- Java code can be generated for new JavaBeans, and existing JavaBeans can be quickly imported.
- Extensive facilities are provided for the reuse and evolution of components, giving the benefits of dependency injection, aspects and much more.
- State machines are fully supported, along with their reuse and evolution.
- A single model can hold and execute many variants of a system.
- Full error checking guarantees all components will connect correctly at runtime.
- Full support for renaming and restructuring parts of the model.
- Sections of a model can be exported and imported for collaborative development.
- Component programs can be executed. Two runtime options are available:
 1. The Backbone runtime engine can be used. This is <350kb, including a full parser. It is fully open-source and has very little performance overhead.
 2. A simple Java program can be created directly from the model, removing any runtime overhead. In this mode, a library of 15kb is used, consisting mainly of interfaces.

1.1.2 A Quick Peek Inside Evolve

The diagram below shows the internal makeup of Evolve. The aim of showing this is to give you some insight into the inner workings of the approach.

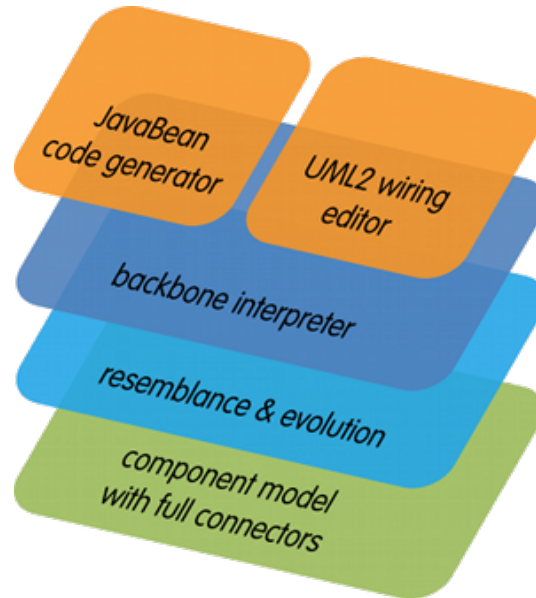


Figure 1.1: The anatomy of Evolve

These parts are explained below, starting from the lower layers and working up.

A hierarchical component model with full connectors

At the heart of Evolve is a hierarchical component model with full connectors. Connectors act like wires between components, making it simple and intuitive to express detailed structures that are difficult or impossible in other approaches such as dependency injection.

Resemblance and evolution

These two constructs provide unprecedented levels of support for component reuse. Resemblance is a form of component inheritance. Evolution builds on this to allow the structure of an existing system to be remodeled, without destroying the original definition. These facilities can be used to create variants of a system, or to switch in test components.

The Backbone interpreter

This executes the programs created in Evolve.

Backbone is a simple textual domain-specific language that allows plain-vanilla JavaBeans to act as full components, and to be wired up according to the component model. The interpreter reads in the text files, and uses the instructions to instantiate and connect up the beans.

JavaBeans have a representation in Backbone and also in Java code. Composite components, formed by wiring together JavaBean instances, only have a Backbone representation however.

This level is analogous to the Spring XML bean configuration, but more powerful. Backbone is fully open source.

UML2 wiring editor

On top of this, Evolve provides a graphical editor for connecting up and evolving components. Industry standard UML2 component diagrams and state charts are used.

JavaBean code generator

Optionally, Evolve can help you define and generate setter / getter code for JavaBeans. Evolve also makes it easy to import and wire together existing beans.

1.2 What is a Component?

A component is a unit of software that can be instantiated, and uses interfaces to describe which services it provides and requires.

In other words, a component is separated from its environment by interfaces. A component instance can then be connected up to other instances which offer compatible interfaces. It really is that simple!

1.2.1 A Leaf Component

With a bit of effort, we can make a Java class fit into this definition. A class can certainly be instantiated, and we can use conventions for provided and required interfaces. We call this a leaf component - it cannot be further broken down into smaller parts.

Look at the component in figure 1.2, which uses UML2 component notation, and consider how we might map this onto Java code. It is a component for performing a spellcheck of a document. It provides the `ISpellCheck` interface through the `check` port, and requires the `IDocument` interface through the `document` port. It also provides the `ISuggest` interface via the same port.

In other words, if you call the `check()` method on the `ISpellCheck` interface, it will extract the document by calling the `retrieveText()` method on `IDocument`. A client can call `suggestCorrection()` on the `ISuggest` interface, and the checker will look for a suitable correction. Finally, the component has an attribute `dictionaryName`, which is the dictionary it uses.

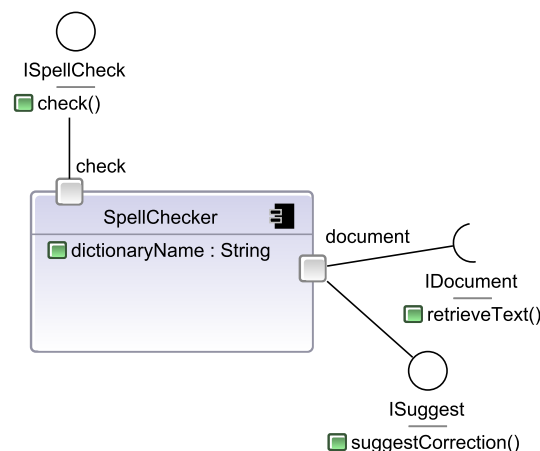


Figure 1.2: A spellchecker leaf component

To map this onto a Java class following bean conventions, we would do something like the following.

```
public class SpellChecker implements ISpellCheck
{
```

```

// the attribute
private String dictionaryName;
public String getDictionaryName() { return dictionaryName; }
public void setDictionaryName(String name) { dictionaryName = name; }

// the required port
private IDocument document;
public void setDocument(IDocument doc) { document = doc; }

// the provided port for document
private ISuggest document_Provided = new ISuggest {
    // methods for ISuggest implementation...
}
public ISuggest getDocument_Provided()
    { return document_Provided; }

// methods for ISpellCheck...
...
}

```

The attribute is mapped neatly enough onto a setter and getter. The provided check port is also mapped easily onto the implements of the class. The required interface IDocument of the port document can be mapped onto the document field, with a setter only. This is an interface that the component requires, so it can call methods on it.

The only really troublesome part is mapping the provided interface of document. For this, we use an anonymous inner class, and provide a getter for it. We use the `_Provided` suffix as a convention for these situations.

As long as we follow these conventions, we can regard each JavaBean as a leaf component. We have introduced the following terms.

Leaf	A component which cannot be further decomposed into others. An atomic unit, implemented by a class.
Provided interface	A service provided by a component. Mapped onto the implemented interfaces of the class or onto an instance of an anonymous inner class.
Required interface	A service required by a component. Mapped onto a field of the class, with a setter only.
Port	A named “gate” insulating the component from its environment. All provided and required interfaces must be via ports.
Attribute	A configurable field of the component.

1.2.2 A Composite Component

A composite component connects up instances of other components to make a new component.

Suppose that we had a `Document` leaf as shown in figure 1.3. Note how the interfaces of the `spell` port of `Document` are the opposites of the interfaces provided and required by the document port of `SpellChecker`.

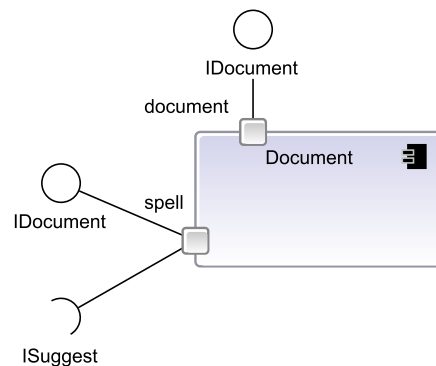


Figure 1.3: A `Document` leaf component

`SpellChecker` provides an `ISuggest` implementation, and requires an `IDocument` implementation. `Document` requires an `ISuggest` implementation and provides an `IDocument` one. As these are complementary, we can connect an instance of each of these together, joining them via a connector `conn` as shown in figure 1.4.

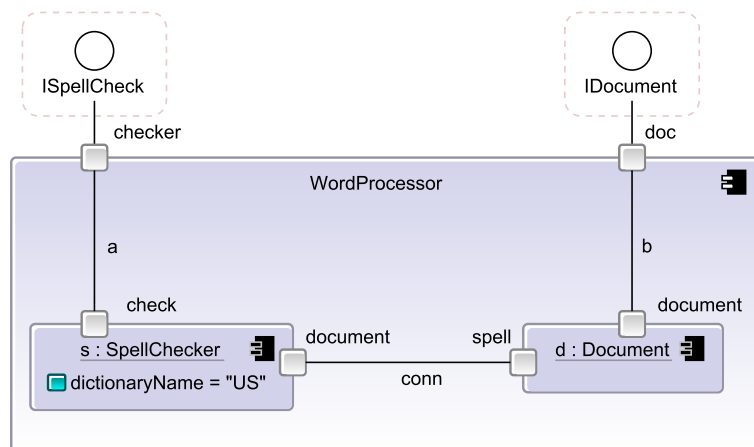


Figure 1.4: A `WordProcessor` composite component

The concept of instances is critical here - we call these “parts”. We have literally defined a new component by connecting together two instances of other components. Note that we did not need to specify the interfaces on the external ports `checker` and `doc`. Eclipse inferred them from the definitions of the internal parts and their connections.

This has effortlessly accomplished something that is often very difficult in Dependency Injection approaches: we have connected two instances (`SpellChecker` and `Document`) to both refer to each other³.



Note that a composite component exists only in Evolve, and it does not need a Java class associated with it. To execute this as a program, the Backbone runtime engine “flattens” all of the composites, removing them completely, and directly connects instances of leaves together to create a running system.

The terms we have introduced are below.

Composite	A component created by wiring together instances of other components (parts) using connectors, and selectively exposing ports of internal parts.
Part	Another name for a component instance.
Connector	A wire joining together two ports of component instances.
Constituent	A general name for a port, part, connector or attribute.

1.3 Compositional Hierarchy

Note that the form of the `WordProcessor` composite, if we ignore the internal parts and connectors, is the same form as that of leaves. It has ports with provided and required interfaces, and can have attributes. In other words, we can now treat the composite like any other component and use instances of it to define new components. Figure 1.5 shows how we might define an office suite composite, as being made up of a `WordProcessor` instance and a `SpreadSheet` instance.

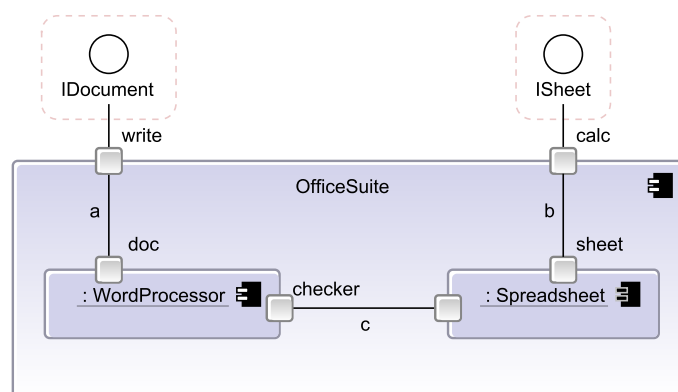


Figure 1.5: An `OfficeSuite` composite, made up of a `wordprocessor` and `spreadsheet` instance

³Without having to pollute the leaf definitions with notions of singleton or prototype.

We are building up a hierarchy of component instances, as shown in figure 1.6. In this case, the top level is `OfficeSuite`, and under that are instances of `WordProcessor` and `Spreadsheet`. `WordProcessor` further decomposes into two other instances which are instances of leaf components.

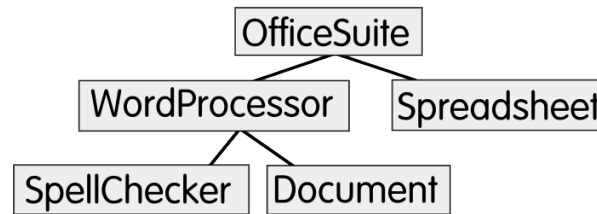


Figure 1.6: The compositional hierarchy of `OfficeSuite`

The ability to form a hierarchy in this way is very similar to how electronics design is done. If we consider that a composite is wiring together chips (parts) using connectors, then the analogy is very close. The key difference though is that once we've created a composite (populated circuit board), we can then instantly reuse it as a part in another composite. It is like being able to shrink an electronic circuit board into a chip on the spot. It's a big advantage, on top of the advantages already proven by the approach in controlling and managing the complexity of digital electronics. The reuse and evolution constructs of *Evolve* work directly on the compositional hierarchy.

1.4 The Backbone Component Language

So what is Backbone and where does it fit into this? Backbone is a component language and a runtime engine. An *Evolve* model is turned directly into a compact Backbone textual definition, which can then be executed.

The Backbone definition for `WordProcessor` is as follows.

```

component WordProcessor
{
  ports:
    checker,
    doc;
  parts:
    s: SpellChecker
      slots:
        dictionaryName = "US",
    d: Document;
  connectors:
    conn joins document@s to port@d,
    a joins check@s to checker,
    b joins document@d to doc;
}
  
```

At runtime, the Backbone engine reads in the definitions, and flattens them into connections between leaf components. It then instantiates the classes corresponding to the leaves, connects them up and passes control over to one of the instances.

A Backbone program can also be executed outside of Evolve. It can also be translated into a simple Java class, allowing any program to be run independently from Evolve or Backbone.

1.5 A Quick Review

That is really all there is to the underlying component approach. We create leaf components in Backbone, and Evolve can help us generate the code for these as simple Java classes. We can import existing JavaBeans into a model as leaf components. We then connect up parts into bigger, composite components which can then be further used as parts in other designs. Composite components do not require any Java code; they exist only within Evolve and Backbone. A model can be executed by the Backbone runtime engine which connects up instances of the leaves directly.

Component hierarchy provides a powerful (and scalable) design approach that has served digital electronics very well for decades - ever since the introduction of the integrated circuit. Evolve allows the same approach for software. All the advanced techniques that we will introduce subsequently in this document are based on these fundamentals.

Unlike in electronics, however, we are not working with physical components. There are advantages to this - the first is that we can immediately “shrink” down a composite and treat it like any other component. Second, we can introduce constructs to manipulate the structure and connectors of a system which evolve and remake it in powerful and convenient ways. Evolve started as a way to produce highly extensible software, and this ability to remake a hierarchy is what ensures that Evolve components can always be extended and customized.

The following chapter will show you how to install and startup the Evolve environment, and how to navigate quickly around a model. We will then walk through an example model, showing how to create and execute component programs.

Chapter 2

Installing and Using Evolve

2.1 Installing Evolve

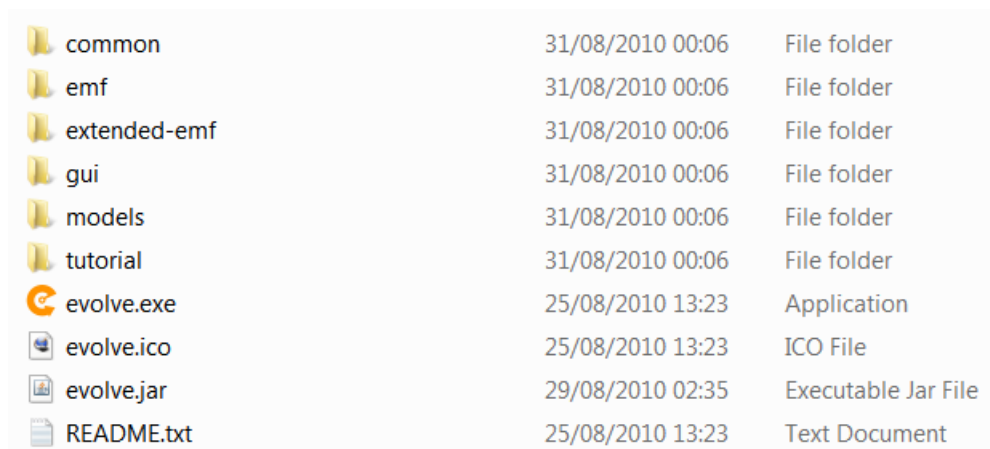
Evolve is written in Java and runs on any system with Java 1.6.

Note that although 1.6 is required to run the graphical part of Evolve, the Backbone runtime engine and any Java code generated from Evolve are compatible with Java 1.5+.

Download Evolve from

<http://www.instrinsarc.com/evolve/download>

Unzip the distribution into a directory and you should see the following set of files.



common	31/08/2010 00:06	File folder
emf	31/08/2010 00:06	File folder
extended-emf	31/08/2010 00:06	File folder
gui	31/08/2010 00:06	File folder
models	31/08/2010 00:06	File folder
tutorial	31/08/2010 00:06	File folder
evolve.exe	25/08/2010 13:23	Application
evolve.ico	25/08/2010 13:23	ICO File
evolve.jar	29/08/2010 02:35	Executable Jar File
README.txt	25/08/2010 13:23	Text Document

Figure 2.1: The top level files in the Evolve distribution

There are several ways to run Evolve. If you are using Windows, you can double-click `evolve.exe` or create a shortcut, drag it to the desktop, and double-click that. If you are using Linux or MacOS, then you may be able to double-click the executable `evolve.jar` file. Alternatively, if you wish to run Evolve from the command line, change to the install directory and type the following.

```
java -jar evolve.jar
```

If you want to increase the memory available, pass it in as a JVM parameter.

```
java -Xmx265m -jar evolve.jar
```

Upon running Evolve, you should see a screen like the following. The startup diagram shows a stratum containing the Backbone definitions. Every model builds on this.

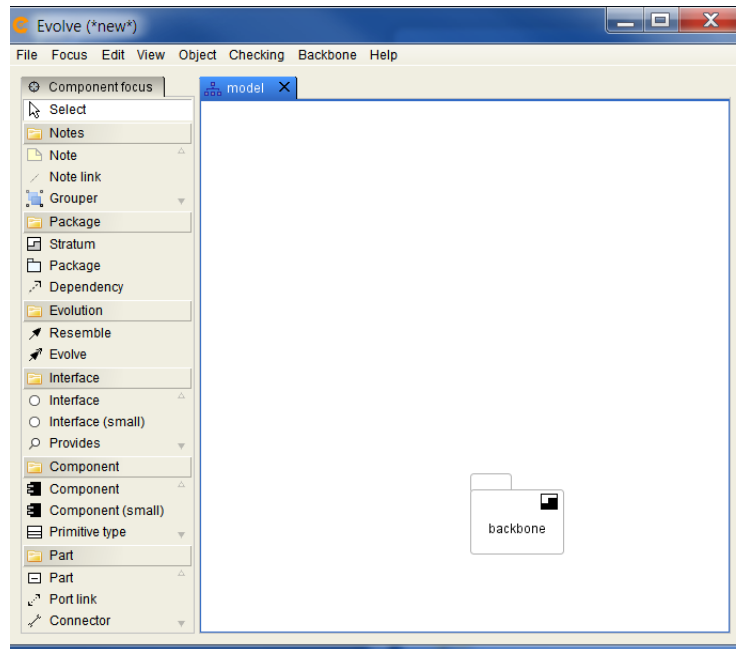


Figure 2.2: Evolve on startup

You may like to change the Swing look and feel used for Evolve. To do this, invoke the `File>Preferences>Edit Environment Preferences` menu item to bring up the preferences dialog, and change to the “Appearance” tab.

2.2 Setting Up The Environment

Before going further, we need to configure a small number of environment variables from within Evolve. Select the `File>Preferences>Edit Environment Preferences` menu item or press `F4` and choose the Variables tab. You should see something like figure 2.3. The `EVOLVE` variable will already be automatically set to the directory that you ran Evolve from. This is overwritten each time you run Evolve. The `MODEL` variable is automatically set to the directory of the model currently loaded.

We first need to add a variable `BB`, which will be where Evolve writes the Backbone files to. Type `BB` into the text box, and then press the “Add” button. Then use the “Folder...” button next to the variable to point it to a folder you have write permission for.

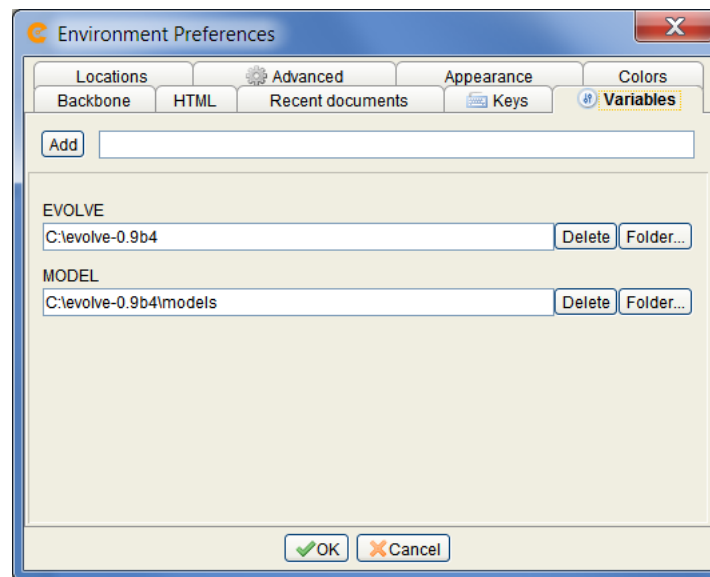


Figure 2.3: Evolve environment variables

Finally, we need to set up some a variable for the tutorials later in this document. Add the variable CARS and point it to the tutorial\CarRental directory of the installation. Alternatively, you can configure CARS using the EVOLVE or MODEL variables. I use the following settings, as per figure 2.4.

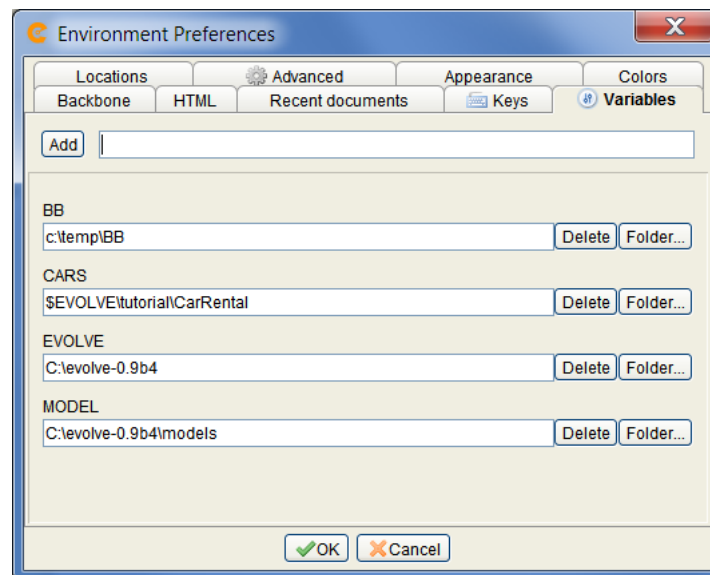


Figure 2.4: Setting up the BB and other variables

Finally, we need to ensure that Evolve has access to the Java executable. Click on the Backbone tab, and verify that the “Java command” describes the command to run Java by clicking “Test”. If not, amend it to point to the correct binary.

2.3 Navigating Around the Tutorial Model

We are now going to load the tutorial model that is used in the following chapters. Select the **File>Open>Existing model** menu item. Navigate to the tutorial directory of the installation, and select the `car-rental.evolve` file. After a couple of seconds of loading, you should see the model shown in figure 2.5.

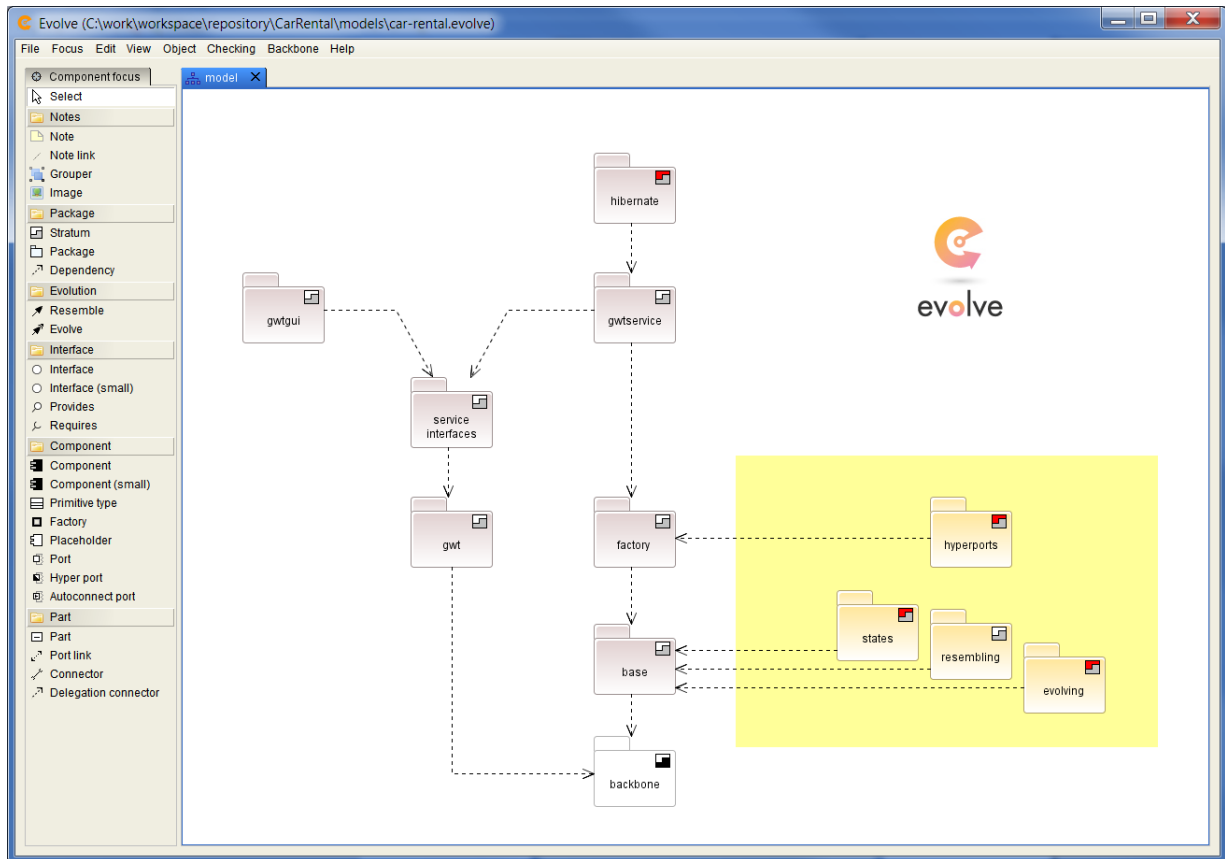


Figure 2.5: The tutorial model



Evolve models are XML files. Files with a `.evolve` extension are uncompressed. To create a compressed model choose the `.evolvez` extension when saving.

Navigating into Strata

The folder-like boxes named `base`, `factory` etc. are strata. These are very similar to UML2 packages. To visit into a stratum, double-click it, or alternatively middle-click it. To exit back to the parent, double-click or middle-click the diagram background. To go backwards or forwards in the diagram history, select the **View>Open previous diagram** or **View>Open next diagram** menu options.

Tabs

Evolve uses a tabbed graphical interface. To open a new diagram tab, select the **View>Open current in new tab** menu option. The tab can then be dragged into a different position as required. To avoid the tab docking, hold the shift key while dragging its titlebar.

The Tool Palette

To the left of the screen is the tool palette. It will be set to “Component focus” by default, showing the tools used to define components. To change the focus, use the **Focus** menu options. The “State focus” allows state machines to be defined. The “Feature focus” and “Profile focus” are advanced features, allowing for requirements feature modeling and the creation of UML2 profiles respectively.

To use a tool from the palette, click on it and then click on the diagram. For repeated application of the tool, simply hold down the shift key for as long as required.

An extremely useful alternative to using the tool palette directly is to press the **space** key over the diagram. This will bring up a mini-palette with tools relevant for the current diagram context. This allows rapid entry of a design. Note that to draw a connecting line between nodes, you must hover over the inside edge of the shape before pressing **space**.



When pressing **space** to bring up the mini-palette, you will only see tools relevant for the current focus. For instance, if we are in the “State focus” mode and we press space, we will not see the component creation tools.

The Size of a Diagram

You will notice that a diagram often does not have scroll bars. These are hidden until required to save diagram real estate. To enlarge the diagram simply drag a figure to the edge of the diagram - the diagram will enlarge and the scroll bars will appear.

Keys

The keyboard shortcuts can be redefined in the “Keys” tab of the **File>Preferences>Edit Environment Preferences** dialog. By default the keys are set to sensible defaults for Windows and Linux. MacOSX users may find several of them inconvenient however and will want to remap these. The labels above the keys are the same as the menu options that the key will invoke.

Pasting into a Wordprocessor

Elements on a diagram can be copied **control C** and pasted **control V** back onto the same, or another, diagram. We can also paste into an external wordprocessor such as Word or OpenOffice Writer. For external programs, the underlying vector graphics format used is the Enhanced Windows Meta-file format (EMF).

When pasting internally, the copy will be inserted at the position of the mouse pointer.

Full Screen Mode

Evolve can be switched to full screen mode by pressing **F11** or invoking the **File>Toggle full screen** menu option. In this mode, the top menus are hidden and can be revealed by moving the mouse to the top of the screen.

To further increase the diagram real estate, drag the titlebar of the tool palette to the dock on the left hand side. The palette will then be hidden and the screen will be almost completely taken up by the diagram space. You will then need to use the **space** key to invoke the mini-palette to create diagrams.

2.4 Compiling the Tutorial Files

The tutorials make use of the Java code in the Evolve `tutorial\CarRental` directory. To compile these classes, use the provided `build.xml` ant script in that directory.

This directory also contains an Eclipse project. To import this, use the Eclipse menu option **File>Import>Existing Projects into Workspace**. In the dialog, select the `tutorial\CarRental` directory, and it will detect the project. Import this, ensuring that you don't use the "copy into workspace" option.

You will also need to define the `EVOLVE` classpath variable so that Eclipse can locate the Backbone jar. Use the "Classpath Variables" section of **Window>Preferences** to point this to the root Evolve installation directory. The project should now compile correctly. For reference, our Eclipse classpath variables are shown in figure 2.6.

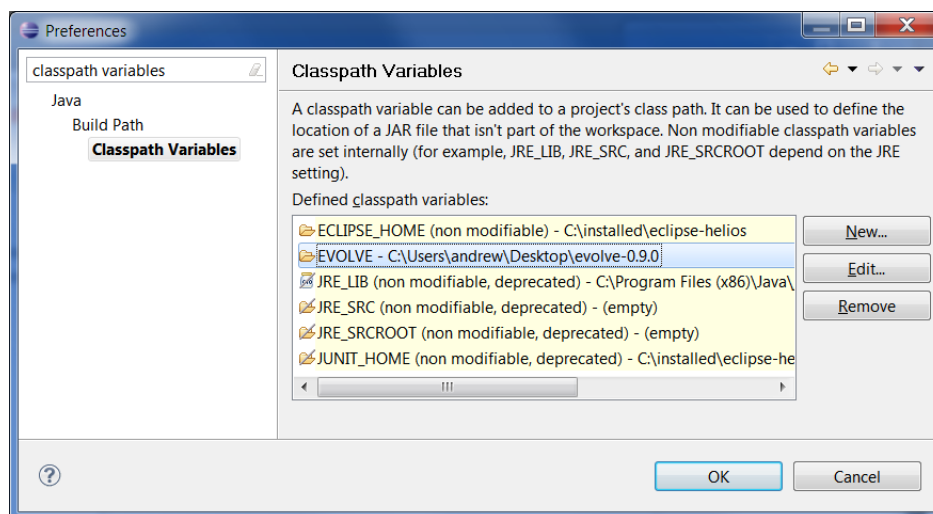


Figure 2.6: Eclipse classpath variables

We created the project in Eclipse Helios but it should work in any recent version.

2.5 Subjects and Views and the Subject Browser

Evolve is a powerful and versatile CASE tool, and adopts common CASE tool conventions. A key convention is the distinction between a subject, and the view of that subject.



A subject is the underlying data of a component or other model element. A view is the graphical presentation of this shown on a diagram. Deleting a view will not delete the subject. Deleting a subject, however, will delete all its views

Consider if we create a component `Test`. We can select the item and copy and paste it several times as shown in figure 2.7. These are all different views of the same subject, the component `Test`. We can delete views by selecting them and invoking `Edit>Delete views only`. To remove a subject and all its views, however, we need to press `delete` or invoke `Edit>Delete`.

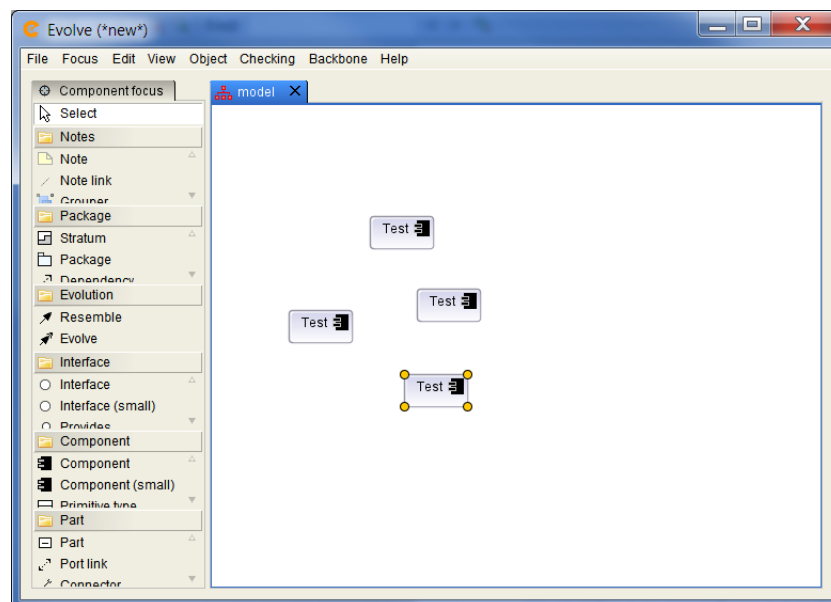


Figure 2.7: Multiple views of the same component subject

Diagrams allow us to see graphical views of subjects. To look at the subjects directly, however, we instead use the subject browser. Select an item on the screen and invoke `Object>Browse element` - this will bring up the browser on the subject of the element selected, as per figure 2.8. The top left tree shows the strata of the model, the bottom left tree shows the elements in the currently selected strata, and the right hand pane shows the details of the selected element.

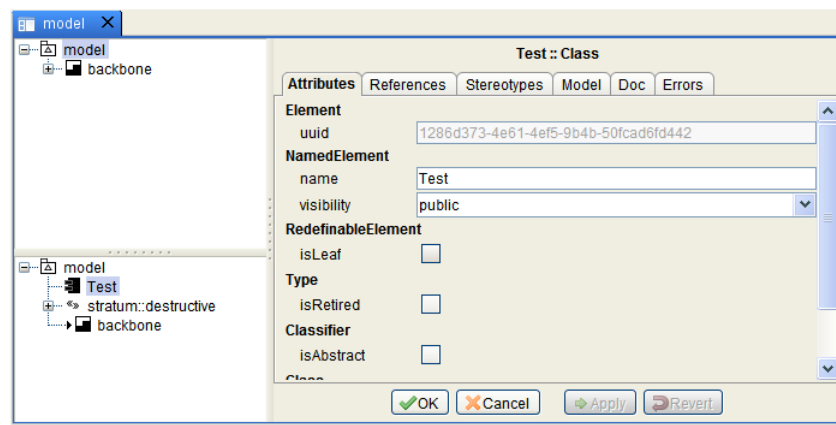


Figure 2.8: Looking at the Test component subject in the browser

Creating a View of an Existing Element

Consider if we have an existing component `SpellChecker`, and an unnamed component. If we want the unnamed component to be a further view of `SpellChecker`, we start typing the name (“Spe...”) and then press the `tab` key as in figure 2.9. A list of possible matches will then appear, and we can click on one. This will result in the view referencing the chosen component.

If we do not press `tab`, and simply type in the whole “`SpellChecker`” name, then we will have created a different component with the same name. Remember to press `tab` when referencing an existing element!

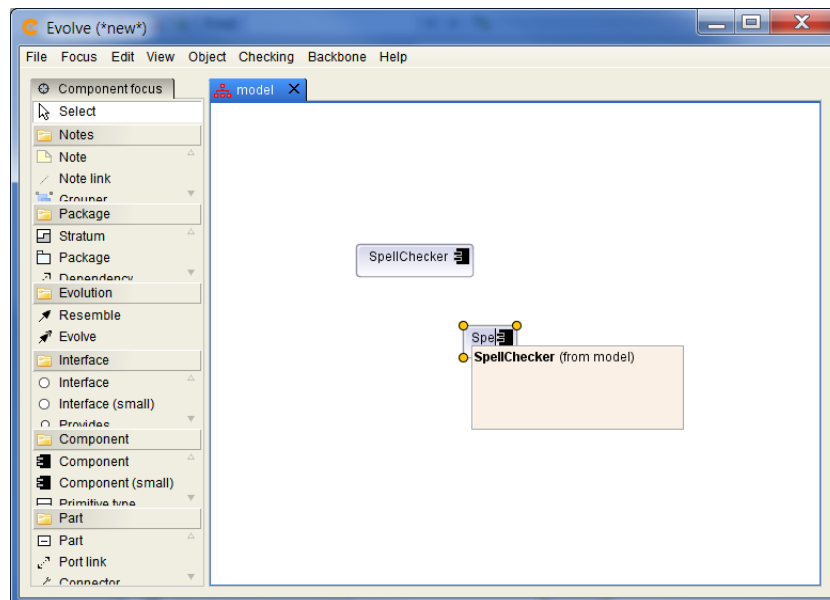


Figure 2.9: Press tab while typing to reference another component

2.6 Running A Backbone Program

As a final check of the installation, we will execute a Backbone program. This will check all of the setup including the environment variables we set earlier.

Load the `car-rental.evolve` model as per section 2.3. Middle-click on the base stratum to navigate into it. First, “tag” the current stratum to indicate that we want to run it, using the `Backbone>Tag current stratum` menu option. Then choose the `Backbone>Run Backbone` menu option. The components in the diagram are instantiated and a runner dialog should show appear as in figure 2.10.

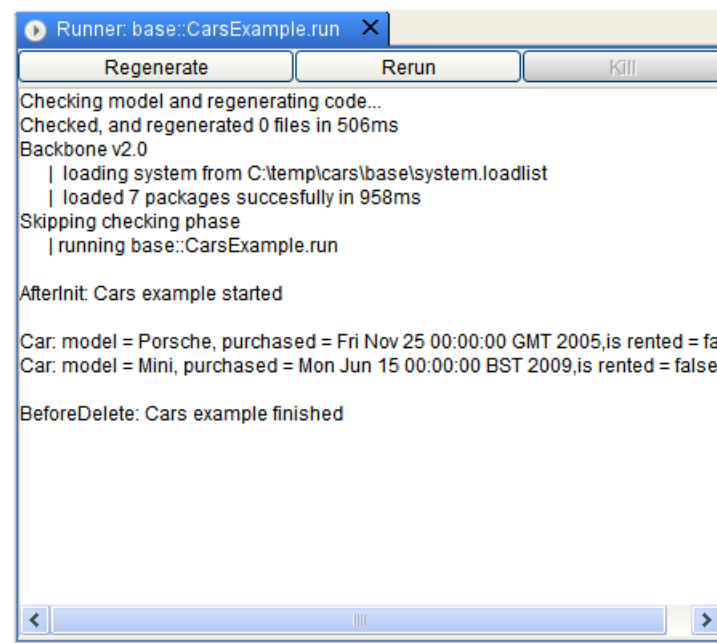


Figure 2.10: The Backbone runner window

If this all works, your installation is correct. Congratulations, you have just run your first Backbone program! This particular program will be described fully in chapter 4.

Chapter 3

Overview of the Tutorials

This chapter looks at an overview of the tutorial examples, contained in the `car-rental.evolve` model. Load the model by following the instructions in section 2.3. You should see the following strata.

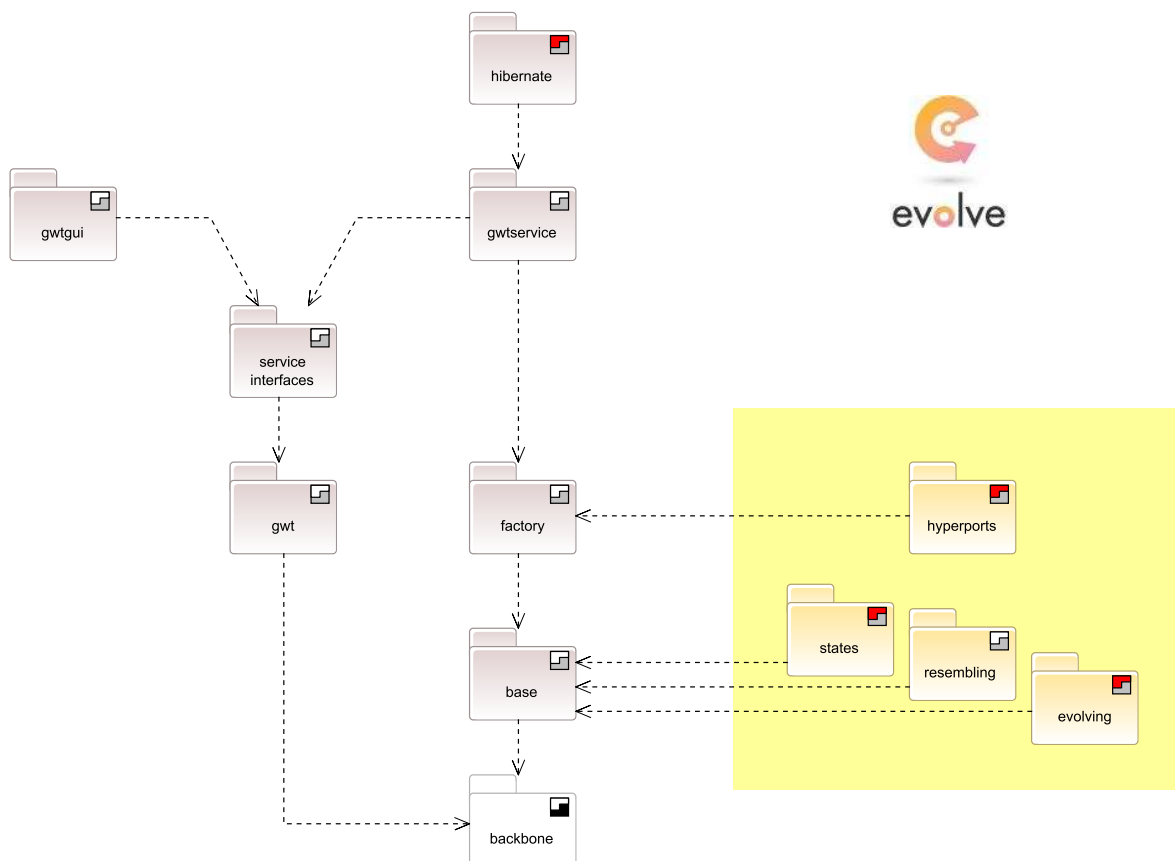


Figure 3.1: The tutorial model

The tutorials, in the rest of this document, use this model to work through the creation of a

system to track rental cars. Each stratum adds features to finally build up to a car rental system with a GWT¹ front-end and a Hibernate² back-end, fully specified as components.

The stratum used for Tutorial A is as follows.

- **base**
This stratum defines the components for representing rental cars. We demonstrate how to generate Java class skeletons from leaf components and also how to roundtrip these so that any code added in an IDE is not lost. We then show how to use the Bean Importer to update the leaves if the structure of the Java classes is modified externally. Finally, we show how to register for lifecycle callbacks, and various ways to execute the model.

The strata in Tutorial B cover using resemblance for reuse, and component evolution.

- **resembling**
This introduces resemblance, which is a form of structural inheritance between components.
- **evolving**
This introduces evolution, which builds on resemblance to allow an existing component to be globally and incrementally altered, without destroying the original definition.

The strata in tutorial C build on the previous strata to demonstrate some advanced techniques.

- **states**
This shows how to define state machines. These components with the visual appearance and behavior of UML2 statecharts. State machines can be resembled and evolved.
- **factory**
This shows how to dynamically instantiate components. Basically, factories are composite components where the internal parts and connectors are created on demand. This is a powerful approach that allows complex structures to be created easily. As they are also components, factories can be resembled and evolved giving access to the full power of these constructs for dynamic structures.
- **hyperports**
Hyperports are ports which cut through the compositional hierarchy. They allow a part, at a high level in the hierarchy, to connect to any number of lower level parts. This gives the convenience of singletons, without the disadvantages.

At this point, we will have covered the basic techniques in full. Tutorial D then applies these to build the GWT front-end and Hibernate back-end, resulting in a complete application for tracking rental cars.

- **gwt**
We use the Bean Importer to import the GWT widget classes into Evolve as leaves.

¹Google Web Toolkit, <http://code.google.com/webtoolkit/>

²The Hibernate object-relational mapper, See <http://www.hibernate.org/>

- `service interfaces`
The service interfaces between the client and server are defined. These interfaces are implemented by the back-end and used by the front-end to retrieve information about the rental cars.
- `gwtservice`
This builds the server as a set of Evolve components. It keeps track of the rental cars and who is renting them.
- `gwtgui`
This builds the client as a set of GWT widgets, expressed as Evolve components. It displays the rental cars, and allows new cars to be added, using ajax techniques.
- `hibernate`
This evolves the `gwtservice` stratum to store the car rental information in a database. We use the evolution construct to update the server to make it transactional.

The GWT and Hibernate integrations are simple because Evolve is basically a JavaBean system and these technologies work well with beans. The use of beans in the Java ecosystem is a lightweight and yet powerful approach to components, and Evolve leverages this fully. It is often the case that integration with other technologies involves little other than importing a bean library into Evolve and then working with the imported component definitions.

Chapter 4

Tutorial A: Creating and Composing Components

In this tutorial, we will create a component to hold information about a car and another for representing renters. We will then connect instances of these into a composite component to represent a rental car, and execute this as a program.

Along the way we will discover:

- How to create leaf components and tie each of them to an implementation class.
- How to generate Java skeleton code for a leaf component, and regenerate without overwriting code added outside of Evolve.
- How to use the Bean Importer to refresh a leaf if its fundamental definition is altered outside of Evolve.
- How to connect up parts to form a composite component.
- Using slots to configure the state of parts.
- How to execute a Backbone program inside Evolve.
- How to run a Backbone program independently from Evolve.
- How to generate a program from a model that doesn't require Backbone.

This is a lot of functionality to get through. Bear with me, however, and you'll soon be creating your own components and composing with ease.

This tutorial assumes you have loaded the example model (see section 2.3) and have navigated into the base stratum by middle-clicking on it. You should see the model shown in figure 4.1 on your screen. Section 2.6 has already described how to run this model and you should have seen the output as per figure 2.10.

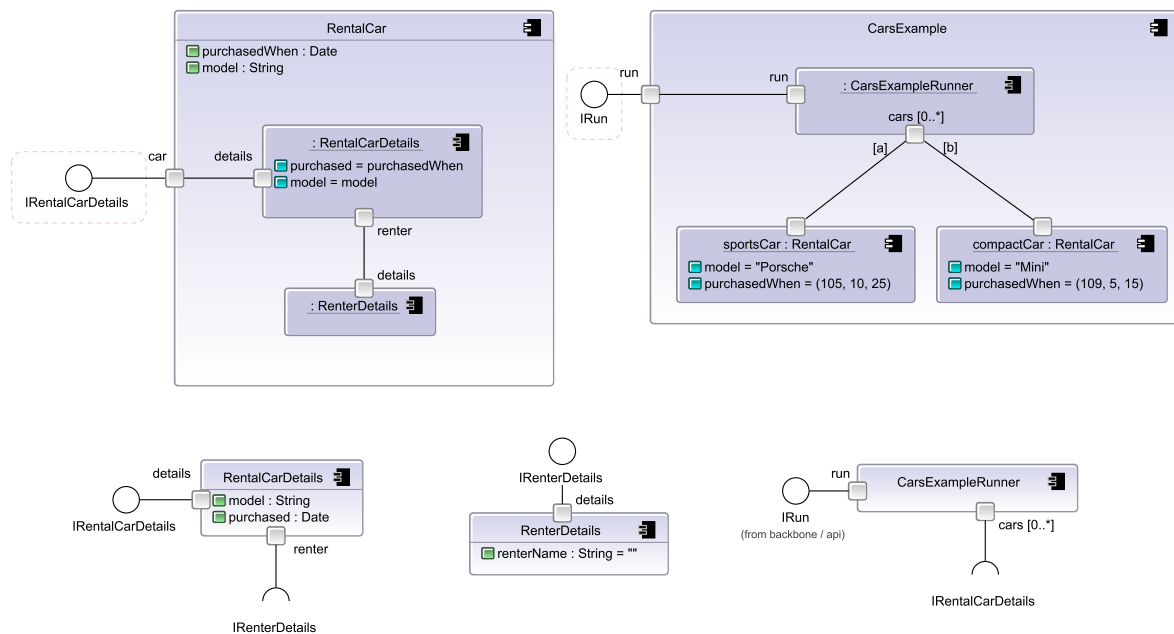


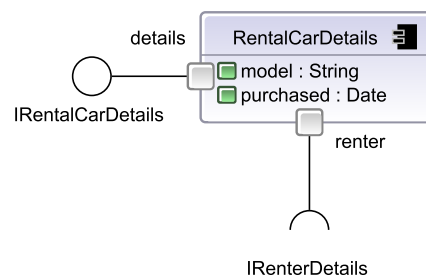
Figure 4.1: The base model, defining rental car components

4.1 Defining the Leaves

In this section we will define the leaf components, which each represent a Java class. Shortly after that we will connect instances of them together to make a composite component which we can execute.

The two leaves we are primarily interested in are `RentalCarDetails` which holds the car details, and `RenterDetails` while holds information on the renter.

Let us examine `RentalCarDetails`, shown in figure 4.2, in more detail. It has two attributes, `model` and `purchased`, which hold the model of the car and the date it was purchased. It also has a port `details` which provides the interface `IRentalCarDetails` allowing the data held to be accessed. The port `renter` requires the `IRenterDetails` interface, and through this the renter details can be accessed.

Figure 4.2: The `RentalCarDetails` leaf



We can see that this component is a leaf because the icon in the top right corner has filled-in legs. A composite component has white legs.

The `RenterDetails` leaf is defined similarly, in figure 4.3. This provides the `IRenterDetails` interface, which allows it to be connected up to `RentalCarDetails` which requires the same interface.

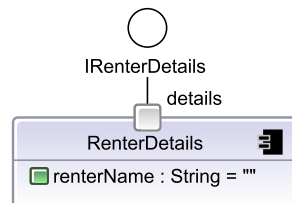


Figure 4.3: The `RenterDetails` leaf

How We Created the `RentalCarDetails` Leaf

The leaf component was created by choosing the “Component” tool from the palette on the left, and applying it to the diagram. An alternative is to simply press the `space` key over the diagram, giving a choice of tools appropriate for the context. The attributes were created by pressing `space` in the body of the component and selecting the attribute option.



Note that it is not necessary to press `tab` to enter the type of an attribute. Simply press `space`, select the attribute option and enter something like “a: int = 10” to create one.

The ports were created by pressing `space` closer to the edge of the component. The interfaces (both provided and required) were created by selecting the interface option and creating the interfaces near the component.

The provided port line was created by pressing `space` on the port. The provided link was then dragged over to the interface. The required port line was created in a similar fashion.

The Implementation Class of a Leaf Component or Interface

The implementation class used for a leaf is the Java package that is set for the stratum where the component is defined, along with the name of the class. This is the Java class that will be generated for the leaf.

To set the package, go back to the top level diagram and select the stratum. You will see an icon in the top left corner. Click this and enter in the desired package name, as shown in figure 4.4.

Now go back into the diagram and select the `RentalCarDetails` leaf. You will see an icon at the top left, as shown in figure 4.5. Click this and it will show you the full class name. This is the Java implementation class that the leaf is associated with. Note that this value is not changed

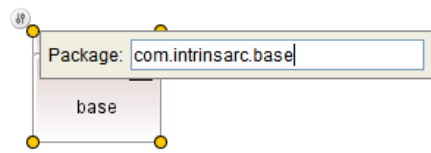


Figure 4.4: Setting the Java package of a stratum

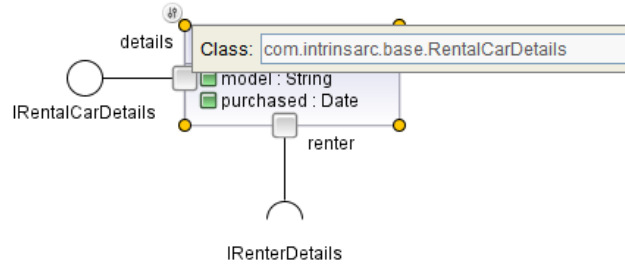


Figure 4.5: Viewing the implementation class of a leaf

directly using the text entry box - instead, to alter it either change the package of the stratum, or the name of the class.

Interfaces use the same approach.

Forcing a Different Implementation Class

It is possible to force a different implementation class for a leaf. To do this, select the leaf and invoke the **Object**▷**Browse element** menu option or press **F2**. You should see a subject browser. Click on the “Stereotypes” tab and you will see a dialog like figure 4.6 below. Enter the full class name in the force-implementation field¹.

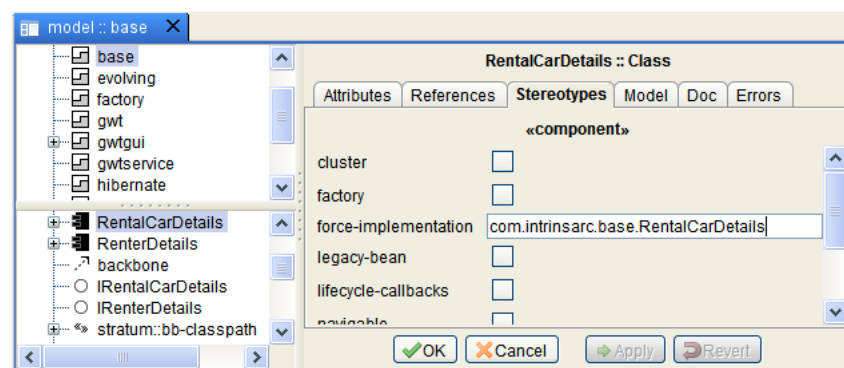


Figure 4.6: Setting the implementation class for a leaf

¹In UML2 terminology, the forced implementation class of a leaf component is held on the stereotype.

Interfaces can the same approach to force the implementation interface.

4.2 The RentalCar Composite

We have the `RentalCarDetails` leaf which holds information about a car, and the `RenterDetails` leaf which holds information about a renter. Together these hold the full set of details for a rental car, but we would like to treat these as a single unit rather than two separate entities. A composite component allows us to do just this.

So, we now create a `RentalCar` composite that contains a `RentalCarDetails` part and a `RenterDetails` part. We connect the two parts together via their compatible interfaces, and add attributes to the composite for the state we wish to set. We then create slots in the `RentalCarDetails` part which bind to the attributes. In this way, we can set the state at the composite level and have it propagate into the parts - in effect we can now ignore the fact that the composite is made up of two parts. This is shown in figure 4.7.

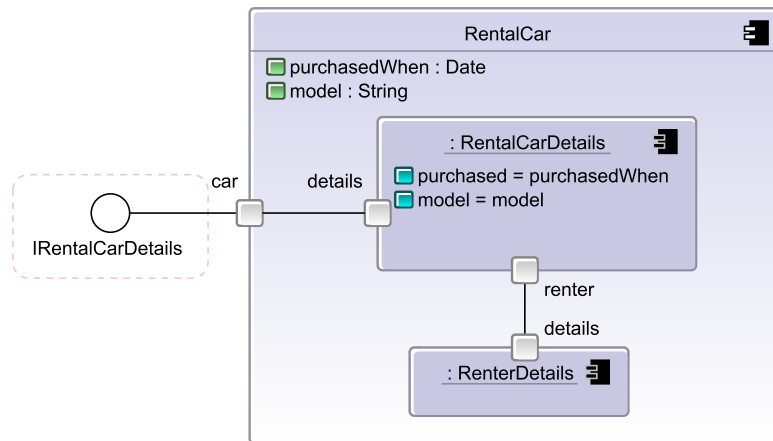


Figure 4.7: The `RentalCar` composite component

How We Created the Composite

The composite component was created by choosing the “Component” tool from the palette on the left, and applying it to the diagram. The ports and attributes were created in the same way as those for the leaf - by pressing `space` on or near the edge of the component, or by explicitly choosing tools from the palette.

The parts were created by resizing the component to be a bit larger, and pressing `space` in the middle of the component. The “Part” menu item was then chosen, creating an empty part inside the component. If you try this you will notice that the text inside the part is simply a colon “:”. Select the part and type `partA: RentalCarDetails` for instance. The first section is the name, followed by a colon and then next comes the type.

Several things should be noted when entering a part. The first is that the colon is very important as it separates the name and type. Also, the name can be empty but the type cannot. Finally, note

that it is not necessary to press **tab** to select the type - Evolve will find the existing component with the entered type name.

To create the connectors, press **space** over the start port, select “Connector” and then click on the end port.

It is never necessary to specify the required and provided interfaces of ports on a composite. These are determined automatically by a powerful inference algorithm that takes into account the internal connections in the component. It also understands interface inheritance. Inferred interfaces have a dotted line around them.



If you want to hide connectors to a port, middle click on that port. Middle click again to re-show the connectors.

4.3 Making the Example Runnable

To make our small example runnable, we require the component equivalent of a `main()` method. This is provided by the `IRun` interface, which has an `void run(String args[])` method. We create a small runner leaf as shown in figure 4.8. Note that this has an indexed port called `cars` with a multiplicity of `[0..*]`. This will turn into `List<IRentalCarDetails>` in the generated bean, with the appropriate `addCar()`, `setCar()` and `getCars()` methods.

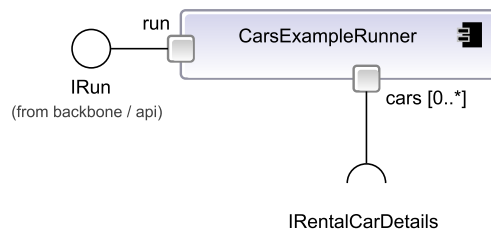


Figure 4.8: The runner component

We can now configure up the full program. We create the `CarsExample` composite (figure 4.9), which connects up a `CarsExampleRunner` part, along with two rental cars: a sports car and a compact car. We use slot values to set the attributes of the parts and these propagate all the way down into the leaf parts.



The types of attributes are known as “primitive types”. These can be created using the “Primitive type” tool in the component focus. When a slot, such as `model` is set in figure 4.9, the literal string will be assigned directly to the attribute. In the case of a set of parameters, such as `(109, 5, 5)` for the `purchased` attribute, a suitable constructor will be found for the primitive type. In this case, the `java.util.Date(int, int, int)` will be called at runtime. Other possible literals are `default` (the default value for the type) or `null`.

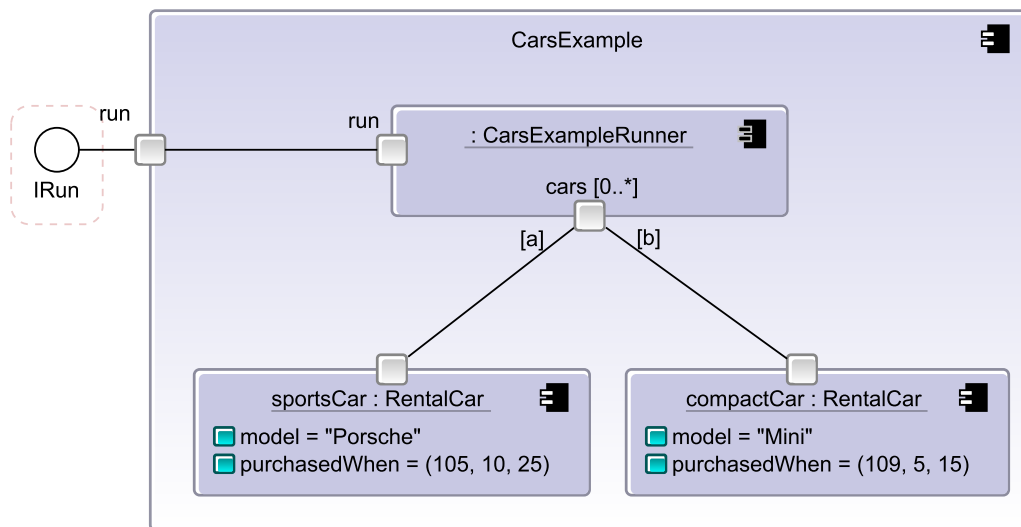


Figure 4.9: The CarsExample composite

4.3.1 Indexed Connectors

In figure 4.9, the connectors used indices of `[a]` and `[b]` to connect to the `cars` indexed port. When connecting to such a port, indices allow us to specify an ordering. The indices can be alphabetical, in which case lexical ordering is used (i.e. `a` before `b`), or alternatively integers can be used.

When using integers, the actual integer index will be used to make the connection. On the other hand, alphabetical indices just order the connections. Alphabetical indices are preferred over integers, as this allows a resembling or evolving definition to always insert an index in between existing indices². We will see this used in the next chapter. In our example, an example of an index that comes between `[a]` and `[b]` is `[ab]`.



If no index is provided, connections happen in any order. It is possible to force the use of indices by marking a port as ordered through the subject browser.

²Evolve and Backbone come from research into highly extensible systems, and alphabetical indices play a large part in this approach.

4.4 Generating Code

Evolve generates Java code for leaves and interfaces, and Backbone definitions for composites. This section discusses how this works and how to set the options.

4.4.1 Checking the Model

Before any code can be generated, the model must be checked for errors. To do this for the current stratum, press **F8** or invoke the **Checking>Check current stratum** menu option. All the elements in the stratum will be checked for correctness according to a large set of rules governing compatible interfaces etc. This is a bit like the component equivalent of the checks that a compiler performs before compiling a program in Java or C++.

If you have an error, such as forgetting to set the implementation class of a leaf, the offending element will be flagged with a small red cross. Hover over this, and you will see the error described in more detail, as in figure 4.10. If you bring up the repository browser on the element in question, you can click on the “Errors” tab to see the location and error description.

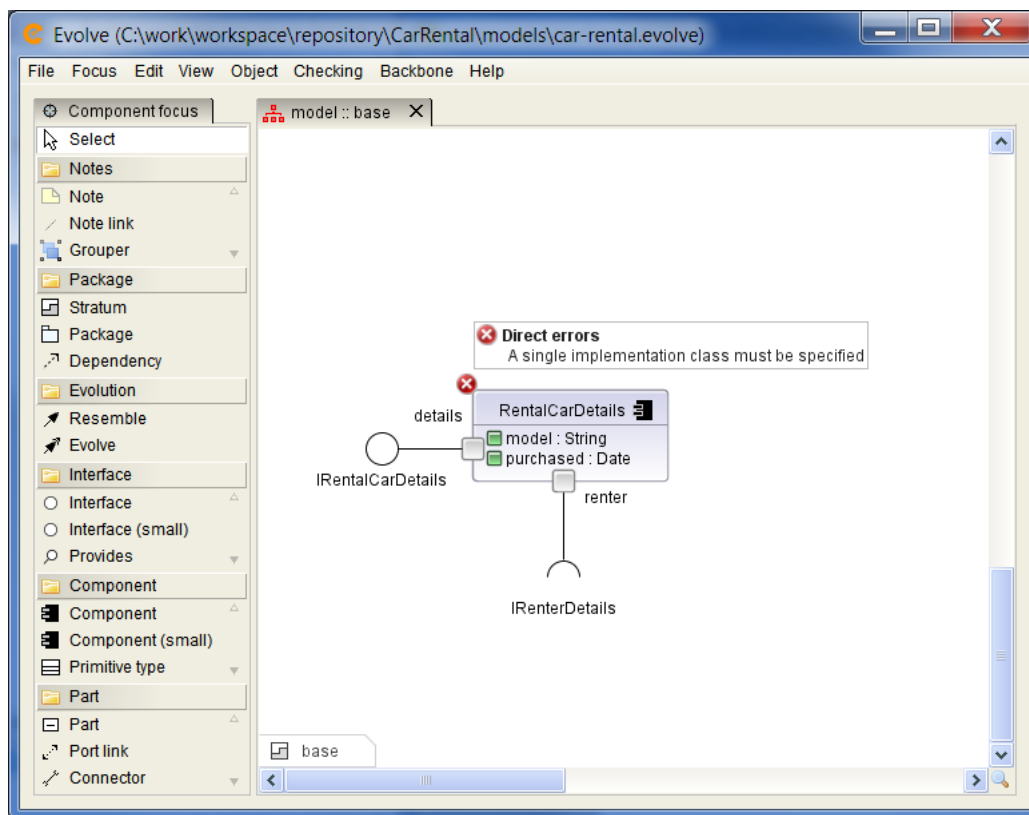


Figure 4.10: Errors are shown on the diagram



The entire system can be checked by pressing `shift F8` or by invoking the `Checking>Check everything` menu option. This is advanced usage which will be discussed in a later tutorial. It checks all possible strata combinations for any errors.

4.4.2 Generating Java and Backbone Code

Assuming the model is error free, we can now generate Java code for the leaves and Backbone definitions for the composite components. First, we must ensure that the stratum code generation parameters are correctly set. Bring up the browser for the base stratum itself. Select the stereotype tab, and you should see something similar to figure 4.11.

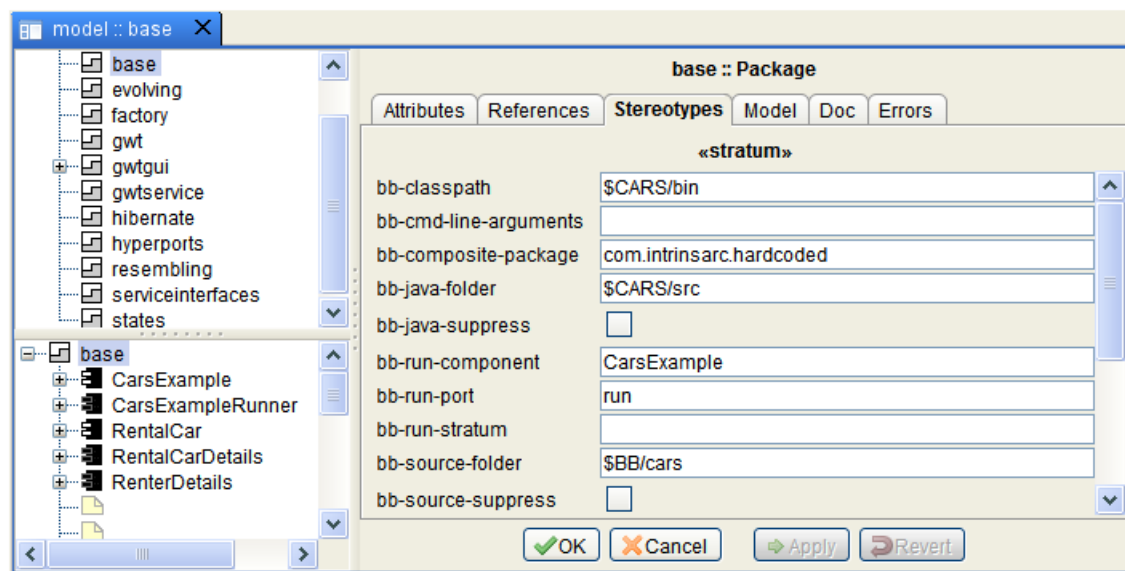


Figure 4.11: The code generation options for the base stratum

The various fields control how code generation and reverse engineering for the stratum will proceed. The relevant fields are as follows.

- **bb-java-folder**
Java code will be written into packages, starting with this as the source folder. Recall that back in section 2.2, we defined the variable `CARS` to point to the tutorial directory.
- **bb-java-suppress**
If this is ticked, no Java code will be generated for the stratum even if the generation phase is invoked.
- **bb-classpath**
This is the classpath used to find class and jar files for importing beans into Evolve. Separate any items by a space, and use quotes for classpath entries with spaces. This should be set to be the corresponding “bin” directory for the “source” directory `bb-java-folder`.

- `bb-source-folder`
This specifies where to write the Backbone definitions for the stratum.
- `bb-source-suppress`
If this is ticked, no Backbone code will be generated.

As these fields are already configured in our example, we can generate the code. First, we must tag the stratum by pressing `control T` or invoking `Backbone▷Tag current stratum`. This only needs to be done once. It simply indicates which stratum will be used for running or generation from now on.

Next, we generate the Java and Backbone code by pressing `control G` or by invoking `Backbone▷Generate Backbone`. Note that a generation always performs a current stratum check beforehand, so you do not need to do this manually.

Inspecting the Backbone Code

The Backbone code will be written to the `$BB/cars/base` location, which will depend on the folder you have set the `BB` variable to. As an alternative to looking at the files directly, you can look at them in Evolve. Go back up to the top level in the model (middle-click the diagram background), right-click on the base stratum and select “Show Backbone code”. The code will look like the following listing.

```
stratum base
  parent model
  is-relaxed
  depends-on backbone
{
  interface IRentalCarDetails implementation-class
    com.intrinsarc.base.IRentalCarDetails { }
  interface IRenterDetails implementation-class
    com.intrinsarc.base.IRenterDetails { }
  component RentalCar
  {
    attributes:
      model: String,
      purchasedWhen: Date;
    ports:
      car;
    parts:
      c6d4caa6-cff6-4bee-b245-ea03d785d2ae: RentalCarDetails
      slots:
        purchased = purchasedWhen
        model = model,
      11f33863-3b39-4b6b-88b4-6089e3930e2b: RenterDetails;
    connectors:
      ... lots of extra lines
  }
}
```

The connection between the graphical component views and the generated Backbone code should be fairly clear. Note that in general Backbone identifies elements according to their UUID³, using element names solely to help with human readability. When listing Backbone code in Evolve, the names are put in where possible. Where they are missing, the UUID is shown. In the files generated to `BB/cars`, the UUID will always be present.

In other words, human-readable names are not used for identity in Backbone, UUIDs are used instead. If the name of a component changes, its UUID will remain constant, meaning that renaming does not present a problem. The only names that really matter are those that affect Java code such as port and attribute names.

Inspecting the Java Code

The Java source code for `RentalCarDetails` will be written to

```
$CARS/src/com/intrinsarc/base/RentalCarDetails.java
```

Looking at this code gives insight into how Evolve forward-engineering works.

```
public class RentalCarDetails
// start generated code
    // main port
    implements com.intrinsarc.base.IRentalCarDetails {
        // attributes
        private java.util.Date purchased;
        private String model;
        // attribute setters and getters
        public java.util.Date getPurchased() { return purchased; }
        public void setPurchased(java.util.Date purchased)
            { this.purchased = purchased; }
        public String getModel() { return model; }
        public void setModel(String model) { this.model = model; }
        // required ports
        private com.intrinsarc.base.IRenterDetails renter;
        // port setters and getters
        public void setRenter(com.intrinsarc.base.IRenterDetails renter)
            { this.renter = renter; }
// end generated code
    ... code omitted
}
```

Evolve looks for the `// start generated code` and `//end generated code` markers and replaces anything in between with its generated code. By placing your own code outside of these

³A globally unique identifier assigned when the element is created.

markers, you can regenerate the leaf code and keep the added code safe. If you wish to avoid generating any code for this class completely, simply remove the markers. If you wish to start generating code for a class again, place the markers back in.



The port corresponding to the interfaces implemented directly by the Java class is called the “main” port. This must be a non-indexed port with at least one provided interface. In the situation where more than one port qualifies, you must explicitly force one main port by ticking the `force-bean-main` stereotype attribute on a port.

Note that Java interfaces are generated with an almost identical marker-based approach.

A Recap of Evolve Implementation Class Generation

Let’s quickly recap the code generation side of Evolve. We define leaf components which are associated with Java classes. Evolve can generate a skeleton for each leaf class, automatically creating the getters and setters to represent the leaf. You can then add methods and fields to the class using an IDE, as long as you write your additions outside of the code generation markers. If you do this, Evolve will preserve your code changes when you regenerate allowing you to move between Evolve and the IDE at will.



For those familiar with dependency injection, note that Evolve uses setter injection exclusively. It does not use constructor injection, as this prevents situations like those shown in figure 1.4 where the two parts refer to each other. Setter injection is more general, and places no restrictions on the type of complex connections that are often made in advanced component models.

4.4.3 Lifecycle Callbacks

To indicate that a leaf should be told that initialization has occurred, and that destruction is about to occur, tick the `lifecycle-callbacks` option in the stereotype properties of the leaf. This will result in it implementing the `ILifecycle` interface, which defines the `afterInit()` and `beforeDelete()` methods. `afterInit()` is called after all leaves have been instantiated and after all connections have been made. `beforeDelete()` is called before any parts or connectors are destroyed.

4.5 Running the Model

Evolve provides several different ways to run a program. The simplest is to use the Backbone interpreter, which reads in the composite definitions and uses this information to create instances of the leaf classes and connect them together. Once they are connected, Backbone is no longer involved and there is no overhead after initialization.

4.5.1 Running Using the Backbone Interpreter

To run this example, we need to tell Backbone which component to instantiate and which port to run. Bringing up the stereotype tab again on the stratum, as shown in figure 4.11, we set `bb-run-component` to `CarsExample` and `bb-run-port` to `run`. We then tag the stratum using `control T` if this hasn't already been done, and run the component by pressing `control R` or invoking the `Backbone>Run Backbone` menu option. Doing this will bring up the runner window of figure 4.12, showing that `CarsExampleRunner`'s `run()` method has iterated over the configured cars and printed them out.

Note that running a program always performs a check of the current stratum (section 4.4.1) and a code generation (section 4.4.2) before execution.

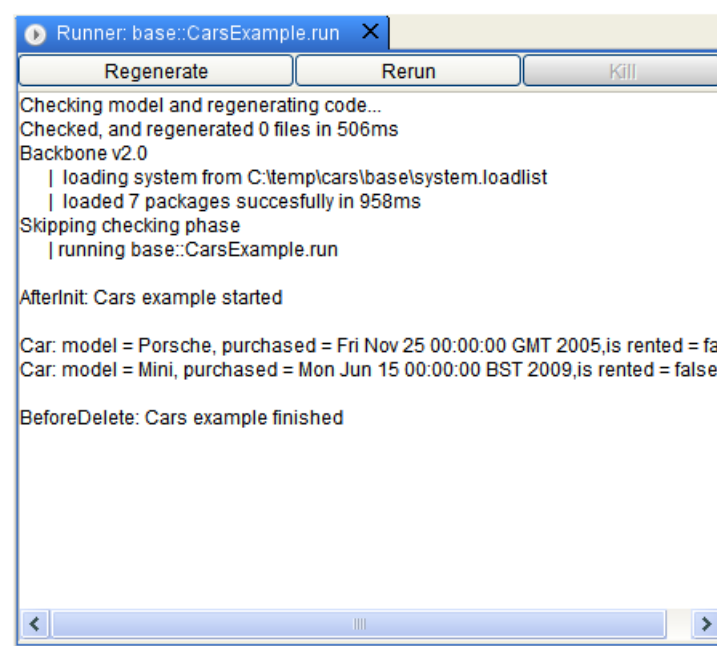


Figure 4.12: Running the car example

4.5.2 Running Outside of Evolve

When we generated the code, it wrote the following file.

```
$BB/cars/base/run-load-list.bat
```

This can be directly executed from a command line, either as a batch file under Windows, or renamed to a shell file under Unix-like systems and executed.

This uses the `system.loadlist` file in the same directory, which specifies which strata to include. Note that the loadlist has two sections: the first defines the variables (from the Evolve model) and the second describes the strata included.

4.5.3 Calling the Backbone Interpreter from a Program

Alternatively, if you wish to call the Backbone interpreter from your program, instantiate the interpreter class and call the `runBackbone()` method. Note that you must reference the `backbone.jar` library.

```
BackboneInterpreter interp = new BackboneInterpreter();
interp.runBackbone(
    loadListFile, stratum, component,
    port, noCheck, args, beforeConnections);
```

Change the parameters to be correct for the program you wish to execute. The `noCheck` parameter controls whether the Backbone model will be checked before execution - set it to true if you do not want the check.

The `beforeConnections` parameter must implement the `IRuntimeCallback` interface. If this parameter is not null, the runtime will call back on the `beforeConnections()` method of the interface, after the parts have been created but before any connections are made.



The `beforeConnections` callback can be used to set any required interfaces of the top level component. Use the `setRequiredPort()` method of the passed in `BBSimpleInstantiatedFactory` instance to do this. This allows the components created to use interfaces provided by your program.

4.5.4 Running Without the Backbone Interpreter

The two previous sections used the Backbone interpreter to execute the Backbone files. The interpreter parses the component definitions, flattens out the composites, instantiates the leaf classes, connects them together and then calls the `IRun::run()` method on the `CarsExampleRunner` instance. The Backbone jar file is around 350kb in size.

Alternatively, Evolve is able generate a small Java program from the model which achieves the same effect. To do this, set the `bb-composite-package` stereotype property on the `stratum` (figure 4.11) to be the Java package where you would like the factory to be generated. In this case we are using the `com.intrinsarc.hardcoded` package. Press **control shift G** or invoke the **Backbone>Generate full implementation** menu option and the `CarsExampleFactory.java` file will be generated. This has a `main()` method on it which will run an identical program to that of the Backbone interpreter. The runtime overhead in this mode is virtually zero, as opposed to the interpreter which has some startup time. Further, it only requires the tiny `backbone-base.jar` library (~15kb) which contains mainly interfaces and some state machine related classes.



All features are available in this mode, including the advanced features and techniques discussed in later chapters.

The ability to generate a Java program representing an Evolve model allows us to use components in all sorts of exotic environments. In chapter 7, we will use this to create a GWT program. GWT

will then translate the program to Javascript and run the Evolve components directly in a web browser.

4.6 Importing Beans and Refreshing Leaves from Code

Evolve contains a bean importer, which can process existing class and jar files and import them as leaf components. This can be used to import JavaBean libraries⁴ into Evolve, or it can be used to update leaf definitions to reflect changes in code⁵.

The importer uses the `bb-classpath` setting on the stratum to know which files to process. To start the importer, choose the `Backbone>Import beans into current package` menu option. You should see a window like in figure 4.13. Expand out the tree on the top left and select the base package.

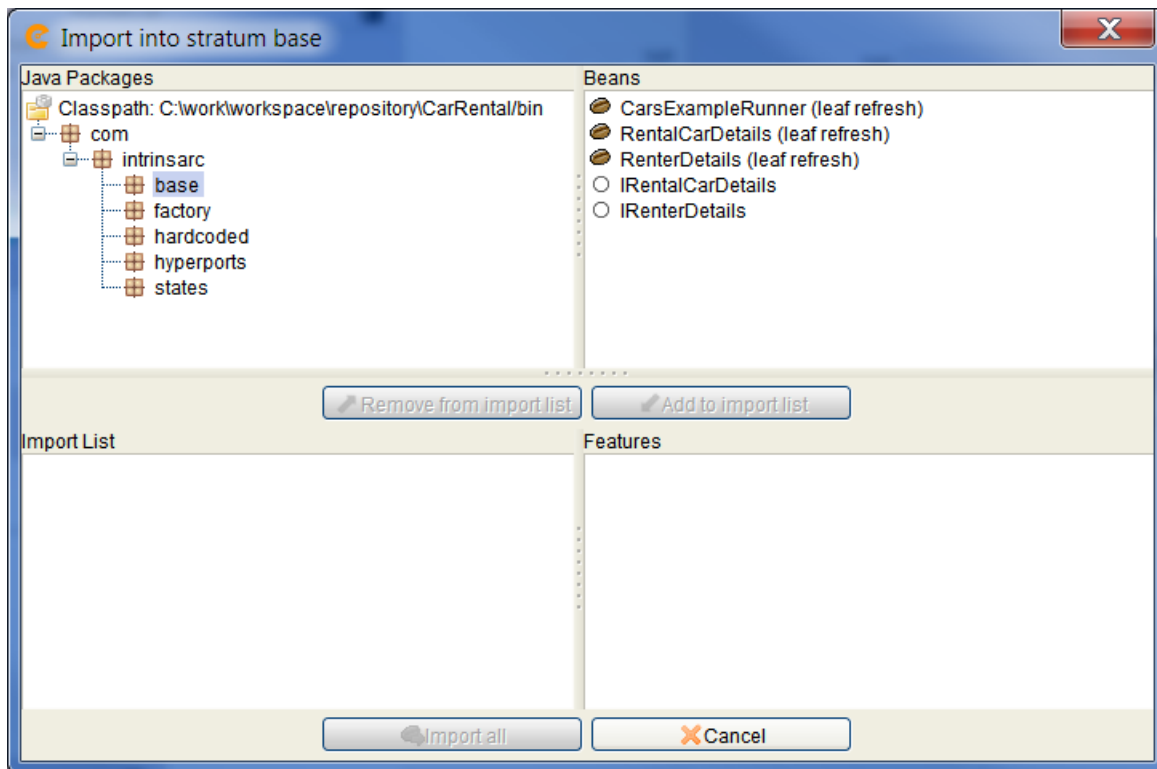


Figure 4.13: The Bean Importer

The top right list shows the beans and interfaces found in the selected package. In this case, it has found our three beans and two interfaces and indicates that the current Evolve definitions will be refreshed if the beans are imported. Selecting the beans and interfaces and clicking on “Add

⁴Many programmers will already have a collection of beans from using other Dependency Injection approaches such as Spring and Guice. The importer allows these to be used in Evolve.

⁵The importer is very fast, and can analyze approximately 3000 complex beans per second on my machine. It analyzes the entire Java rt.jar library in around 5 seconds.

to import list” sees them appear in the “Import list” pane in the bottom left. Clicking on an entry will then show on the bottom right pane what it will look like as an Evolve leaf. This is shown in figure 4.14.

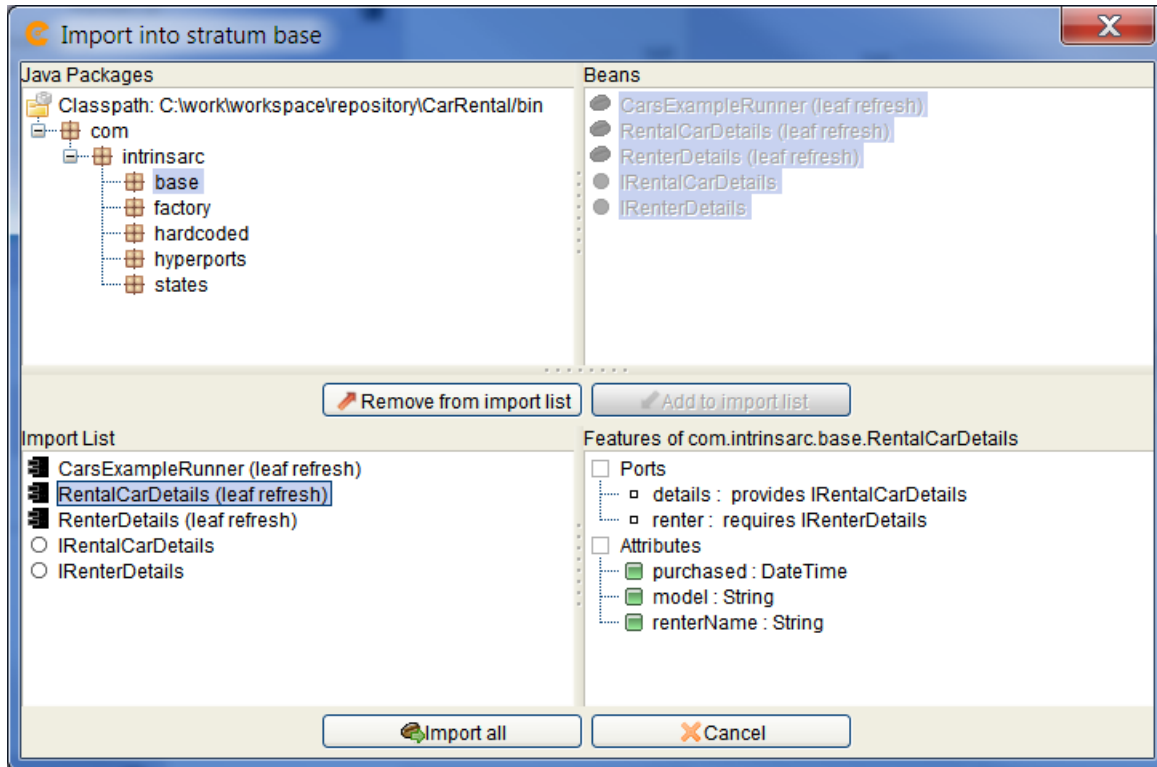


Figure 4.14: The importer showing the Evolve representation of a bean

After importing the beans, the model is updated. Note that if you import a bean library, then the subject definitions will go into the model, but will not be visible on a diagram until they are explicitly added. You must add a new component to a diagram and then use the `tab` key to reference an imported one, as shown in section 2.5. It can then be expanded to show the ports and attributes required. We explain this in more detail in section 7.3.1.

Note that the importer works off class and jar files, and not source files. If you make changes that you want to see in the importer, compile your code before you import.

4.7 Summary

We’ve covered a lot in this section - you now know how to create leaves and composites, and how to import and reuse existing bean libraries. In the next chapter, we will show how to resemble and evolve components, allowing us to quickly customize and build on an existing application without destroying its original definition.

Chapter 5

Tutorial B: Resemblance and Evolution

This chapter describes how to reuse and evolve components using the resemblance and evolution constructs.

Resemblance is structural inheritance between components. It allows us to define a new component by inheriting and adjusting the structure of one or more existing components.

Evolution replaces an existing component definition with another throughout a model. Since it builds on resemblance, we can use it to inherit the structure of, and incrementally adjust, an existing component.

When we modify inherited structure in this way, Evolve records deltas rather than copying and directly modifying. This is crucial, because it means that any later changes to resembled components will flow through to the resembling or evolving component. This also allows us to use strata to represent branches of an application, and to use evolution to merge and rectify any conflicts between them.

If you are now thinking “Evolve has moved some of the concepts of version control into a component system” then you have understood the concepts.

5.1 Component Resemblance

Navigate into the `resembling` stratum in the `car-rental.evolve` model. In this stratum, a new `ResemblingExample` composite is defined using resemblance from `CarsExample`, as shown in figure 5.1.

Which part of the structure inherited from `CarsExample` have we changed? Well, very little in this case - we’ve simply replaced the connectors in order to switch around the connector indices. Bear with me, I’ll show more complex examples later. Press the **F6** key or invoke the `View>Toggle delta view` menu option to see the delta marks. If you hover over the marks they will indicate which elements have been replaced. Even though Evolve works internally with deltas, it will always show you the fully expanded structure of each component.

Resemblance is a powerful facility for reuse. We can inherit from any number of components and adjust the structure to suit.

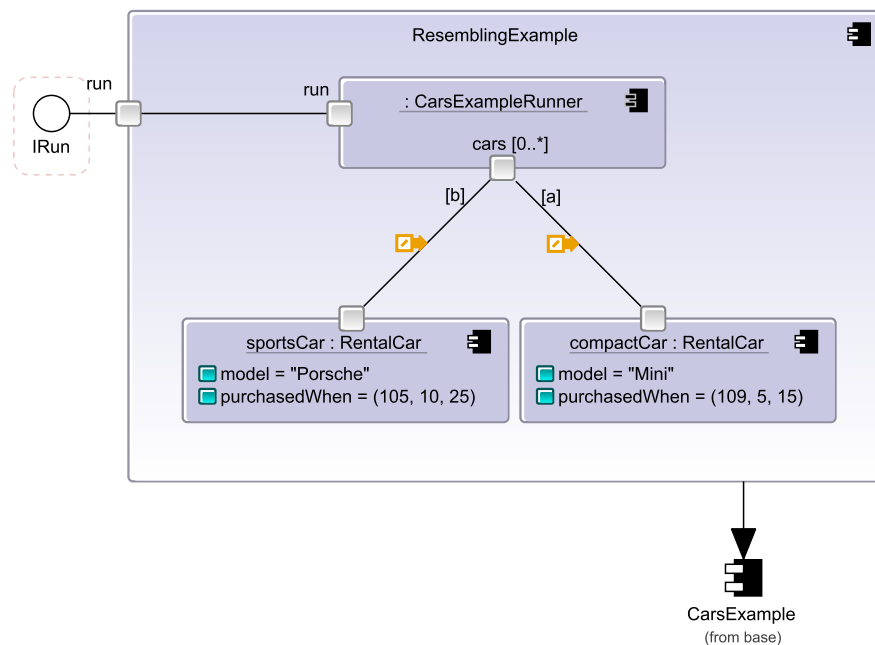


Figure 5.1: Using resemblance to define a new component



Why did we call this construct resemblance rather than inheritance? Resemblance is different as it allows for deletions and non-compatible replacements (overrides). In Evolve, resemblance completely replaces inheritance.



Multiple resemblance is supported. Note that if we have diamond inheritance (i.e. A resembles B and C, B and C resemble D) then only a single copy of D's inherited constituents will be present in A. The UUIDs of the inherited elements identify them uniquely, saving us from the perils of multiply inherited base constituents.

5.1.1 Creating Deltas

Inherited parts, ports, connectors and attributes can be replaced or deleted. To delete an inherited item, press the **Delete** key or use the **Edit>Delete** menu option. This creates a delete delta, shown on the diagram as a red delta icon against the component. To replace an inherited item, right-click on the item and choose "Replace". This creates a replace delta with an orange icon. Any added elements will result in a green delta icon.

If we run the `resembling` stratum, we will see that the cars are listed in the opposite order to before, showing that we have successfully reversed the order of the connections. This is shown in figure 5.2.

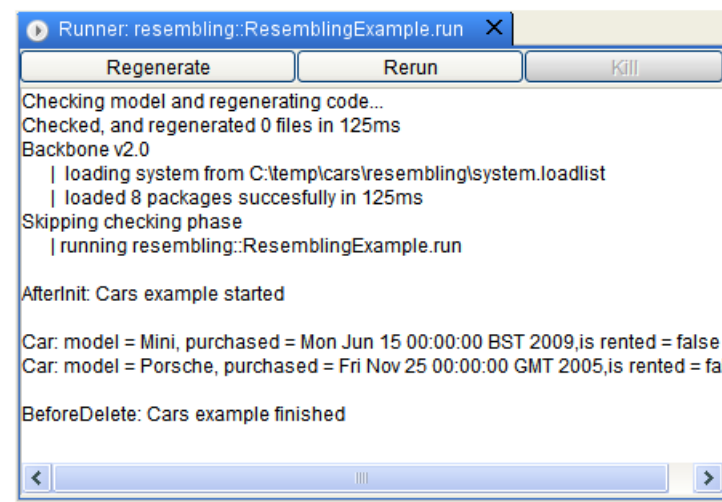


Figure 5.2: Running with the connectors reversed

5.1.2 Viewing and Manipulating Deltas

We can use the subject browser to view deltas directly. Bring up the subject browser on the `ResemblingExample` component, and we see that it is empty apart from two connector replacements (figure 5.3). The rest of the structure is inherited.

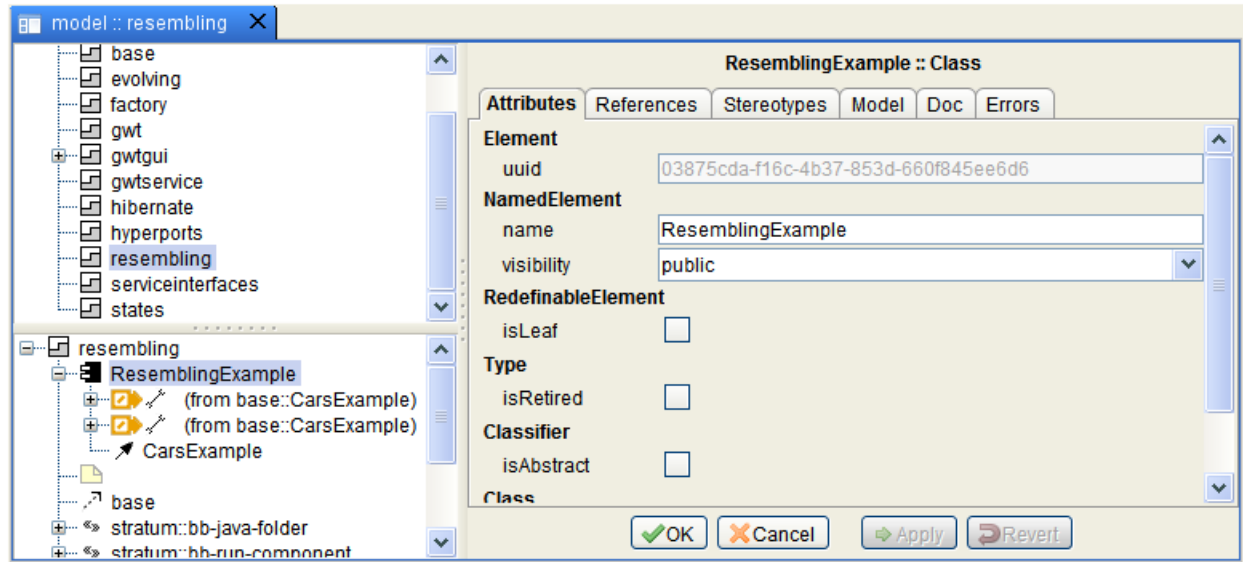


Figure 5.3: Viewing the deltas in the browser

An add or replace delta can be removed from a model by invoking `Edit>Delete` on the diagram, but a delete delta can only be removed via the browser. To delete a delete delta, right-click on it in the bottom left pane, and choose “Delete”.

If we right-click on the stratum and select “Show Backbone code” we will see that the textual

representation records only the deltas also.

```

component ResemblingExample
  resembles base::CarsExample
{
  replace-connectors:
    8daadec4-6075-4d41-8211-03e24dfb9a27 becomes
    eae5ef56-6075-4f6e-b7e8-50316ff159fc
    joins car@sportsCar to
    cars[b]@9f2e80c4-4127-4e65-ac8e-6b99495fe2c3,
    1662d77f-35ff-4116-967e-fbc95309760f becomes
    6f81c572-30a1-489f-aea8-05b17b329866
    joins car@compactCar to
    cars[a]@9f2e80c4-4127-4e65-ac8e-6b99495fe2c3;
}

```

5.1.3 Stereotypes and Resemblance

Every newly created component in Backbone has a «component» stereotype. This is where properties such as `implementation-class` are held.

When we use resemblance to define a new component, however, the stereotype of the resembling class is deleted. In other words, the ressembler also inherits its stereotype. If we use multiple resemblance, we must replace the multiple inherited stereotypes with one of our own¹. Do this by right-clicking the component in the bottom left pane of the subject browser and choosing `Replace with «»▷component`.

5.1.4 Using Resemblance to Define a Leaf

Leaves may also be defined using resemblance. In that case, the resembled leaf will be used as the superclass when we generate code for the ressembler. This mapping onto implementation inheritance can be turned off via the `no-inheritance` stereotype property of the component.

(Note that if the component we are resembling has forced the implementation class using the `force-implementation` stereotype property, our new component will also have this as it will inherit the stereotype. If we wish to alter this we must replace the inherited stereotype in our new component and alter the property setting).

With a bit of care, a leaf can be defined using resemblance from a composite (if we delete all the parts and connectors) and a composite may be defined via resemblance from a leaf (if we add parts and connectors).

¹It is an error for a component or interface to have more than one stereotype.

5.1.5 Renaming Inherited Constituents

When we replace bits of the structure we have inherited, we can safely rename them without affecting their identity. This is because Evolve always uses the UUID of an element for identity. In our small example, we could assign new names to the connectors.

5.2 Component Evolution

We now show how to evolve a component in order to change its definition. We evolve the `CarsExample` composite to add an extra car and change an existing car.

Navigate into the evolving stratum from the example model, and ensure that delta marks are showing. You should see a composite as in figure 5.4. Look carefully at the prime (') character at the end of the component name and the text underneath the name. This is an evolution of the `CarsExample` composite from the base stratum. We have replaced one of the parts in order to change its slots values, and added the `midSizeCar` part.

Note also that we have added the new connector at index `[ab]`, to place it between the two existing parts.

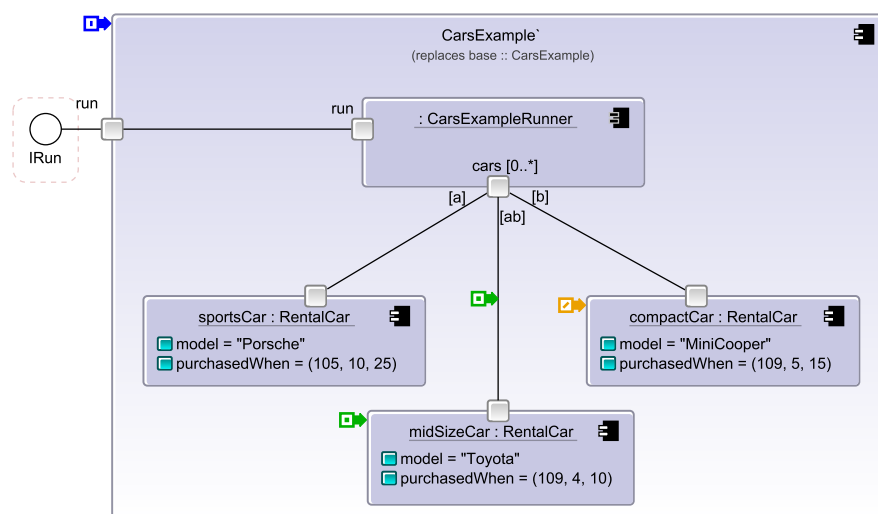


Figure 5.4: Evolving the `CarsExample` composite component

5.2.1 How to Evolve a Component

To create the evolution, we first placed a new component on the diagram. We then typed in the first few characters of `CarsExample` and pressed `tab` and selected `CarsExample` from the list. We now had a view of the existing `CarsExample` component from the base stratum. Finally, we right-clicked on the component and chose “Evolve” from the menu, creating the evolution in figure 5.4. We added a new part and connector and replaced the existing part.

Running this small example gives us the output of figure 5.5, which demonstrates that the new part has been connected between the existing ones, and that the replaced part details have taken effect.

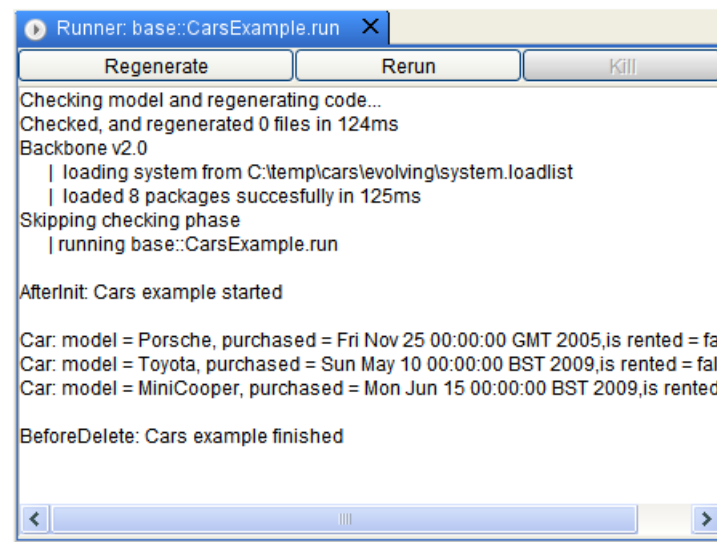


Figure 5.5: An evolution of the CarsExample composite

5.2.2 Evolution = Resemblance + Replace

Evolution and resemblance at first glance seem very similar. They both inherit the structure of one or more components, and allow this to be adjusted via deltas. What is different?

In our resemblance example, we defined a *new* component. In our evolution example, we adjusted the definition of an *existing* component. Evolution is resemblance and then a replacement of an old definition with the new definition. As the deltas are kept separate from the original definition, the original is not destroyed or modified directly.

In other words, resemblance reuses existing structure to make a new component. Evolution adjusts the structure of an existing component.

Evolution can affect the model in many places, in deep ways. Consider, for instance, if we have a compositional hierarchy as shown in figure 5.6, from the perspective of stratum X. At the top is component A, which is made up of two parts of type B and C, and each of those components in turn are made up of D and E.

In another stratum Y, which depends on stratum X, we evolve D to be D' - we turn D from a leaf into a composite of F and G parts. If we look at the system from the perspective of stratum Y, the compositional hierarchy will have been changed to that of figure 5.7. If we look from the perspective of stratum X, we still see the original hierarchy in figure 5.6.

In contrast, if we wanted to reuse the structure of D and adjust it without affecting the hierarchy of A, we would simply define a new component using resemblance.

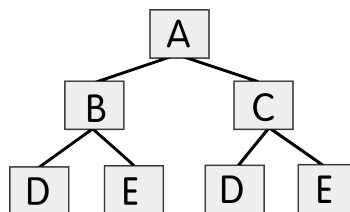


Figure 5.6: Original composition hierarchy of component A

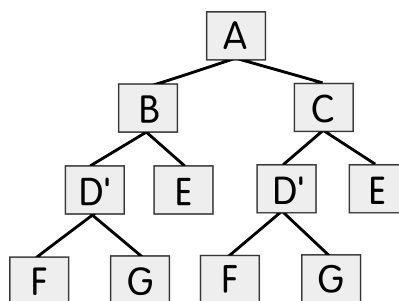


Figure 5.7: After evolving D to D', now composed of F and G

5.2.3 Remaking the Compositional Hierarchy

Evolution allows us to reach deep into the compositional hierarchy of a system and globally change certain areas, possibly in multiple places. We can expand out the hierarchy by evolving leaves into composites, and compress the hierarchy by doing the reverse. We can literally remake an existing system in any way we require, to customize it for new requirements.

These far reaching changes, however, are only applied from the perspective of the stratum with the evolution in it (Y). If we look from the perspective of X, we see the original system.

This allows us to define multiple variants of a system in the same model. Consider that in our example, we have the base stratum, which defines `CarExample`. We also have an evolved variant of `CarExample` in the evolving stratum. In practice, it is not uncommon for a single `Evolve` model to contain 10 or more variants of a system.

5.2.4 Using Evolution to Rename a Component

When we evolve a component (or interface) we can also change its name if desired..

Recall that names do not control identity in `Evolve`. As such, unless a name relates directly to Java code in a leaf or interface, it can be changed without affecting the system.

5.3 Interface Resemblance and Evolution

Interfaces can also be defined using resemblance and evolution. If an interface resembles another, it will be generated in Java code to “extend” that interface. In other words, resemblance between two interfaces creates a sub/super-interface relationship.

5.4 Evolution Without Changing the Implementation Class

It is possible to evolve both a leaf and an interface but keep the same implementation class as before.

Why would you want to do this? The answer lies in the fact that the implementation class of an evolved definition will always be placed before the original element in the Backbone classpath. In other words, this uses the JVM trick of shadowing an older implementation or interface with a different one, if we generate the new implementation with the same package and name, but into a different classpath folder.

This particular technique will not work if using Backbone with OSGi or other module systems which control the classpath more strictly than standard Java.

5.5 Retirement

An existing component or interface can be retired in a stratum. Create a view of an existing component in a new stratum, select it, and invoke `Edit▷Delete`. The element will appear with a large red cross over it, indicating it has been retired, as in figure 5.8. To enforce the retirement, it is an error to refer to the retired element in strata that depends on the stratum with the retirement. In effect, retirement is the delta counterpart of subject deletion.

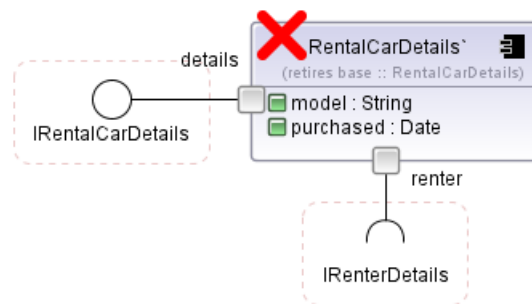


Figure 5.8: Retiring a component

Behind the scenes, retirement is implemented using evolution. The element is evolved, and the retirement flag is replaced and set to true.

5.6 Strata

We have been working with strata for a while - this section explains them in more detail along with the rationale for their existence.

Strata are folder-like entities that own components and interfaces. They also act like modules in other systems - they can selectively export nested strata and import other strata.

5.6.1 Organizing a Model

The first purpose of strata is to allow a model to be organized. As in our example car rental model, we can partition a single problem space into multiple areas, and express how they depend on one another. We have already seen dependencies between strata - for instance, the evolving stratum depends on the base stratum. A dependency is shown as a dotted line with an arrow to the stratum that is depended on.

It is an error to have mutually dependent strata, or any cycles in the strata dependency graph at all. This forces a layering effect where we can analyze a system by looking at each stratum from the base upwards.

Locating Elements

Each component or interface is owned by a single stratum, called its home stratum. When we show a view of a component in a foreign stratum, it will show its home stratum under the name. Note that a component can only be evolved in a foreign stratum that can see the component's home stratum.

We can change the home stratum of a component or interface by locating it in a different stratum. To do this, click on the view of the element in the foreign stratum, and press the **control L** key or invoke the **Object▷Locate elements here** menu option. If you bring up the subject browser before doing this, you can watch the element relocating into the new stratum.

5.6.2 Relaxed and Strict Strata

Elements can only refer to other elements in their home stratum, or in strata that their home depends upon. The error checker will flag any violations to this rule.

When first created, a stratum is "relaxed". Any strata that depend on a relaxed stratum Y, for example, will also have access to the strata that Y depends on.

However, it is possible to prevent this by marking a stratum as "strict". Do this by unticking the stratum's "Relaxed" menu item. If Y is strict, then any strata that depend on Y will not automatically have access to the strata underneath.

Figure 5.9 shows this more clearly. Stratum X is strict (note the different icon) and stratum Y is relaxed. Both depend on Z. Stratum A, which depends on X, can only see definitions in X and will not be allowed access to definitions in Z. Stratum B, which depends on Y, can see both Y and Z's definitions.

Note that would still be possible for A to access Z's definitions by adding another dependency from A directly to Z.

Strict and Relaxed strata are useful as they allow us to model architectural layers in a system. In layering, a decision is often made to restrict access to lower-down layers. The equivalent layer diagram corresponding to figure 5.9 is shown in figure 5.10.

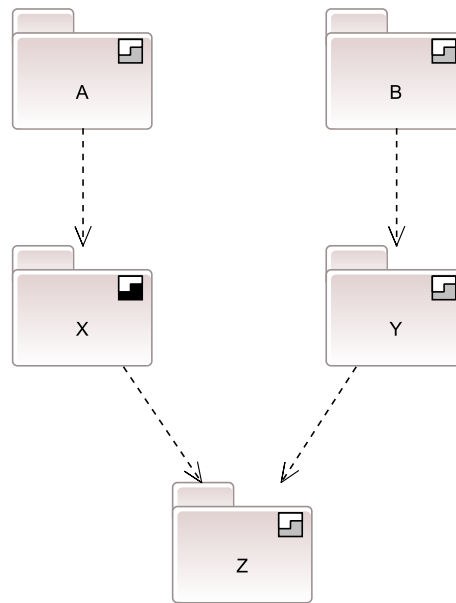


Figure 5.9: X is a strict stratum, preventing A from seeing Z

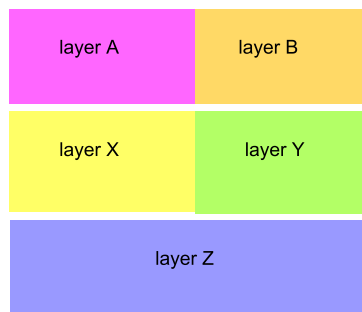


Figure 5.10: A equivalent layer diagram of the strata

5.6.3 Read only Strata

A stratum can be marked as read only by right-clicking on the stratum and ticking the “Read only” option. This will prevent any modifications to elements owned by that stratum.

Read only strata can also speed up checking of a model when evolution is used. Chapter 7 describes how this can be helpful when a model becomes large and complex.

5.6.4 Strata Perspective

As we look upwards in the strata dependency chain, starting in our example with `base`, then moving onto `factory` and `evolving`, and then onto `gwtService` and so on, we see that more and more components and interfaces are added to the system. In addition, if evolutions are present, then adding in strata can change existing components and interfaces.

We can view each stratum, going up this dependency chain, as a system in its own right. If we just consider the system from the perspective of base, we have the components and interfaces in base. If we look from the perspective of *evolving*, we have the definitions from base, with *CarsExample* as evolved by base. As we change perspective, we change how the system looks.

Another way to view this is shown in figure 5.11. As we move upwards and add another stratum to the system, that stratum can reuse existing definitions in lower strata using resemblance. It can use evolution to replace definitions of existing elements in lower strata. It can also add definitions.

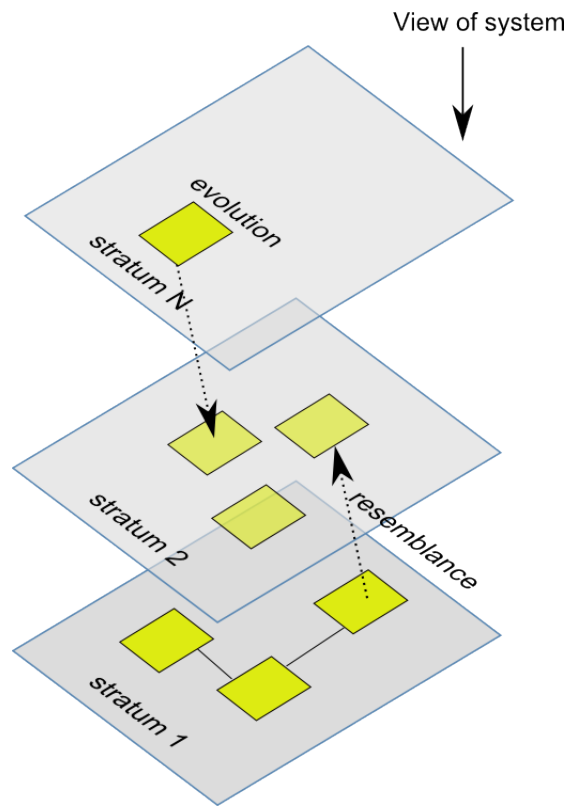


Figure 5.11: A conceptual view of stratum perspective

Each stratum therefore introduces a new perspective and this perspective can represent a different system variant. To error check all these variants and all possible permutations, invoke the `Checking>Check everything` menu item.

Evolutions combinatorially increase the amount of checking required for a model. In our example model, to check from the *resembling* perspective we can simply check base, and then check *resembling*. To check, however, from the *evolving* perspective, we must check base, then check base with the evolutions from *evolving*, and then finally check *evolving*.

Evolve uses a number of graph-based techniques to prune the amount of checking down to a manageable level.



If we know that definitions in a given stratum will not be further evolved, and the stratum contains a large number of complex components, checking performance can be dramatically improved by marking the stratum as read only, and ticking the `check-once-if-read-only` stratum property. This will prune the graph of combinations even further.

5.6.5 Destructive Strata

A stratum containing evolutions is known as destructive², as applying the deltas can radically change an underlying model. To give a visual indication to the designer, destructive strata have a slightly different icon with red in the upper half.

To mark a stratum in this way, right-click on it and tick the “Destructive” option. In our tutorial model, the `states`, `evolving` and `hibernate` strata are destructive.

5.6.6 Nested Strata and Packages

Strata can be nested. A nested stratum can access any strata that its parent depends on, and a parent can reference any definitions in its child strata³. The visibility of nested packages can be controlled, allowing a stratum to act like a module. A full discussion of nested strata is beyond the scope of this document, and will be covered in an advanced techniques manual.

A package can also be nested within a stratum and used to further organize a model. A package is more lightweight than a stratum (particularly for error checking purposes), and elements from a package are actually owned by the stratum that owns the package. An example of a stratum with nested packages is `gwtgui`.

5.6.7 Strata as a Unit of Ownership

A stratum is a unit of ownership, and is generally controlled by a single developer or development team who are the only ones with write privileges to it.

In a collaborative situation, it is possible for multiple developers to share a common base model, and then build on top of this using strata they own. The common base is treated as read only by most of the developers, and they use evolution instead to make any alterations. Strata can be exported using the `File>Export selected strata` menu option, and imported using `File>Inspect exported file`.

Although beyond the scope of this document, this import/export facility can function as a robust and powerful alternative to plugin architectures. This will be described in an advanced technique manual.

Both collaboration and plugins are also touched on in chapter 9.

²The use of the word destructive is historical, it simply means that the stratum has evolutions.

³These rules are different to UML1.3+. We have carefully constructed Evolve's rules to fit into the extensibility approach.



You must be in a foreign stratum to evolve a component. Evolving it in its home stratum doesn't make sense - if you own its home stratum in order to make an evolution there, you can simply directly modify the component rather than evolve it.

5.7 Primitives

As discussed earlier, a primitive is the type of a component attribute such as `int` or `java.util.Date`. To create a primitive, use the “Primitive type” tool in the component focus. Set `implementation-class` to the full class of the primitive type.

You may have guessed already - primitive types can be resembled and evolved.

5.8 Summary

Resemblance is a form of structural inheritance, allowing a new component to inherit and modify the structure of existing components using deltas. This provides a fast and powerful way to reuse structure. In Evolve, resemblance completely replaces inheritance.

Evolution builds on resemblance to allow a stratum to make incremental alterations to existing components (and interfaces) in lower down strata. This allows a new stratum to completely remake the compositional hierarchy of an existing system, and to extend and customize it in arbitrary ways.

Neither resemblance nor evolution directly modify existing components. Instead they hold deltas and apply these to the structure of these components. This adds many of the features of a version control system directly into the component language, allowing strata to be used to represent branching and subsequent merging and conflict resolution. Conflicts are resolved using further strata to evolve and correct errors.

Finally, it is worth noting that resemblance and evolution interact in a very clever way to allow the resemblance structure of an existing system to be remade in a new stratum. This is explained further in chapter 9, where we give an overview of how this works.

Chapter 6

Tutorial C: Factories, Hyperports and States

This chapter looks in depth at the factory, hyperports and states strata in the tutorial model.

Factories are a way to dynamically instantiate components. They are components whose “insides” are created on demand. The insides can be instantiated as many times as required, and can be later destroyed on demand.

Hyperports offer a convenient way to cut through the compositional hierarchy. Consider if we wanted to connect directly from A down to both parts of type G in figure 5.7. Hyperports offer a simple way to do this without the need for lots of connectors. In electronics terms, this is almost the equivalent of jumper wires between circuit boards. Interestingly, hyperports give all of the advantages of the singleton design pattern, without any of its disadvantages.

Finally, we describe how Evolve supports state machines. State machines are an incredibly useful way to describe how a system changes state, and our approach allows them to be implemented as easily as other components. In fact, Evolve state machines are components with the visual look of UML2 state machines, and they can be resembled and evolved. They also support nesting and chaining.

So, without further ado, we present these useful features. They are built on top of the component foundations discussed in earlier tutorials, and we will be making use of resemblance and evolution in our examples.

6.1 Factories

A factory is a way to dynamically instantiate the insides (parts, connectors, attributes) of a component. In contrast, the previous component structures we have looked at have all been created fully at startup.

Let's create a rental car using this approach. Look inside the factory stratum, and you will see a CarFactory composite as in figure 6.1.

This was created using the “Factory” tool in the tool palette.

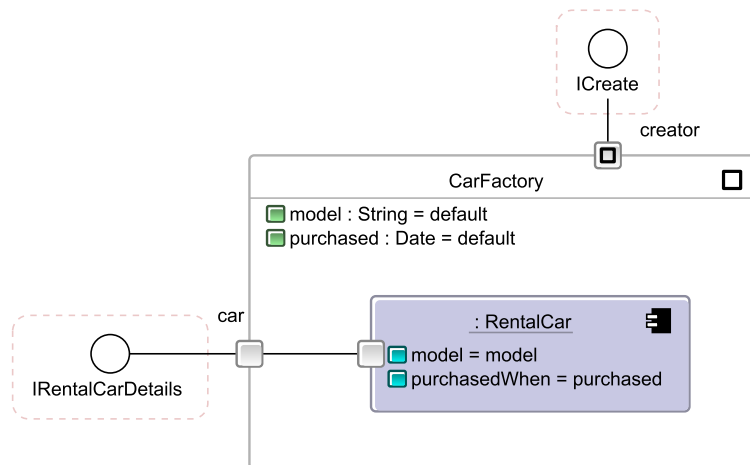


Figure 6.1: The CarFactory composite component

A factory is a composite component, with a creator port automatically added. This port provides the ICreate interface with the following methods, which allow the insides to be created or destroyed.

```
public interface ICreate
{
    public Object create(Map<String, Object> suppliedParameters);
    public void destroy(Objectemento);
}
```

The create() method instantiates the insides, returning a memento object which can then be passed into the destroy() method to reverse the process. create() also accepts a set of parameters which, if provided, will override the slot settings. For instance, we could put a ("model", "Ferrari") pair into the map and pass it in.

6.1.1 Using the Creator Port

To use the port, we must connect it up to another component that has some logic - in this case we make a small runner component called FactoryExampleRunner. This is shown in figure 6.2. We can then connect everything up into the FactoryExample composite of figure 6.3.

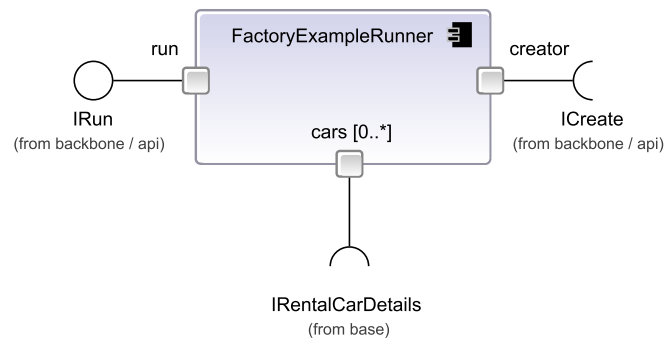


Figure 6.2: A runner component which can use the creator factory port

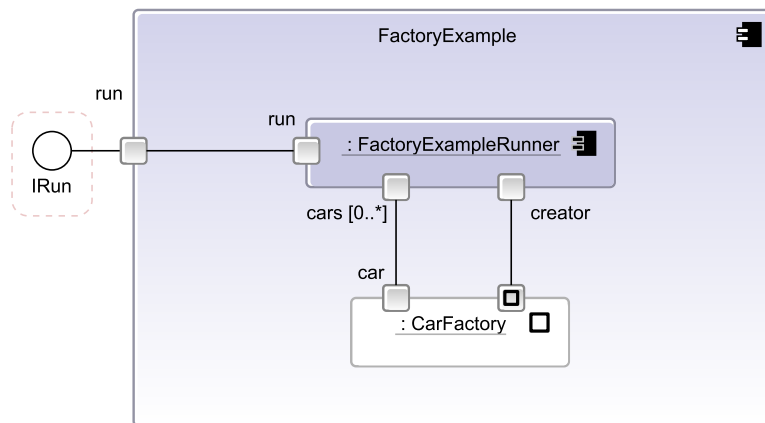


Figure 6.3: The FactoryExample composite component

To execute the example, tag and run the stratum and the output should look as below. Our runner component dynamically adds a Volkswagen, then a Saab, then destroys the first car and then the second.

```
After making first car:
  Volkswagen, purchased = Mon Nov 10 00:00:00 GMT 2008
After making second car:
  Volkswagen, purchased = Mon Nov 10 00:00:00 GMT 2008
  Saab, purchased = Fri Jun 05 00:00:00 BST 2009
After deleting first car:
  Saab, purchased = Fri Jun 05 00:00:00 BST 2009
After deleting second car:
```

The Java code in `FactoryExampleRunner.java` to accomplish this is as follows.

```
Map<String, Object> params;
```

```

params = new HashMap<String, Object>();
params.put("model", "Volkswagen");
params.put("purchased", new Date(108, 10, 10));
Object memento1 = creator.create(params);
printCars("After making first car:");
params = new HashMap<String, Object>();
params.put("model", "Saab");
params.put("purchased", new Date(109, 5, 5));
Object memento2 = creator.create(params);
printCars("After making second car:");
creator.destroy(memento1);
printCars("After deleting first car:");
creator.destroy(memento2);
printCars("After deleting second car:");

```

6.1.2 Factories Are Isomorphic

The technical term for this approach is actually “Isomorphic Factories”¹. In other words, a factory component has the “same shape” as a normal component, and the connectors are connected up in the same way. Factories have all the advantages of composite components.

Isomorphic factories makes working with dynamic structures easy, but crucially also allow a designer to see the full connections that can be made at runtime by looking at static diagrams of the model. Component languages² have this as a primary concern - the ability to fully describe all of the possible connections between dynamically created components from a static description. Although this removes some of the flexibility of being able to create object structures in an ad-hoc fashion, we gain a large advantage in that we can now analyze the entire system and know all of the runtime possibilities.

6.1.3 Factories Can Instantiate Complex Structures

Our factory is instantiating an instance of `RentalCar`. However, we could have put a much more complex set of parts inside our factory, with a lot more connectors and ports. Also consider that `RentalCar` is actually made up of two leaf instances which are connected. Factories can handle this type of complexity neatly³.

Note that we have used a single factory to make multiple instances, and that the cars `[0..*]` port will get another connection each time an instance is made.

¹The Darwin language introduced the concept, but did not allow direct programmatic instantiation.

²These languages are known as Architecture Description Languages (ADLs) in research and academia. They deal with defining the structure of a system as interconnected components.

³Compare and contrast this with how you instantiate complex structures in Spring. If you want to dynamically create multiple instances, a bean must be marked as prototype. However, this limits complex connections between bean instances, forcing you to mark them all as singletons. Ouch.

6.1.4 Factories Can Be Nested

A factory can contain instances of other factories. There is no limit to this nesting, use it as needed.

To see the connections that will result from nesting, right-click on the `FactoryExample` composite and choose the `Show flattened structure` menu option. This will show something like figure 6.4. This is the flattened structure of the components. We first touched on this in section 1.2.2 - flattening is the way that Evolve removes composites in order to make connections directly between the leaf instances.

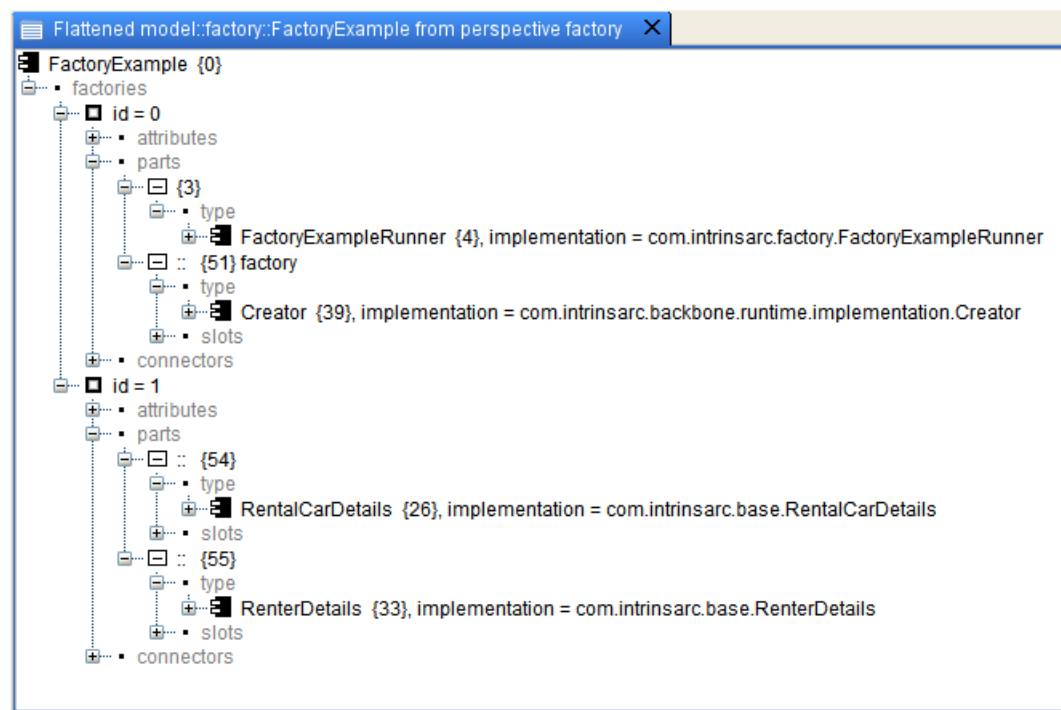


Figure 6.4: The flattened form of `FactoryExample`

If you look carefully, you will see the first surprise – the `FactoryExampleRunner` part is also in a factory (`id=0`). Evolve must instantiate the parts to run the example, and it uses a factory to do this also!

The second factory (`id=1`) is the flattened `RentalCar` component that will be created when we instantiate `CarFactory` programmatically. If you expand out the connectors of this, you can see how it will be joined to the parts in the first factory.

If we added a factory part inside `CarFactory`, it would turn into another factory (`id=2`) at the same level as the others.

6.1.5 How Do Factories Work?

Look carefully at figure 6.4 and you will see that the first factory (`id=0`) contains a part of `Creator` type. This is automatically inserted by the Backbone component expansion process,

based on knowing that the component is a factory. This part contains the logic to instantiate the insides of the factory.

This part is connected up to the creator port of the factory, before being “pushed up” into the parent factory. This is why the create port automatically provides the ICreate interface. When a factory is created it is set to resemble `FactoryBase`, and the creator port is inherited from this.

6.2 Hyperports

A hyperport is a port that cuts through the compositional hierarchy. A hyperport of a part automatically connects to any compatible, but unconnected ports underneath it in the hierarchy.

This provides a principled and far more flexible approach to the Singleton design pattern from the original Design Patterns book⁴. The original pattern is inflexible, prevents unit testing and ties a programmer into a single instance unnecessarily. Let’s now look at how hyperports remove these limitations.

For this section, navigate into the `hyperport` stratum of the example model.

6.2.1 Connecting Through a Hierarchy

Suppose that we wanted to amend the factory example of figure 6.3, so that we printed out a warning to a printer every time the purchased field of `RentalCarDetails` was accessed, along with details of the previous access. We want all cars to use the same printer. How can we do this?

We will want to first create the printer component. We define a very basic leaf, as shown in figure 6.5. Look carefully at the icon on the port - it is a hyperport, created using the “Hyper port” tool. The `IPrinter` interface contains a single `print(String warning)` method, and writes any received warnings to the console.

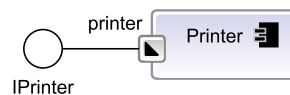


Figure 6.5: A printer component with a hyperport

We can now place an instance of this inside an evolution of `FactoryExample` (from the previous section). Figure 6.6 shows this.

⁴I hate the OO form of the Singleton pattern. I guiltily admit though to sometimes using it in conventional Java programs. There is no excuse in a component system though.

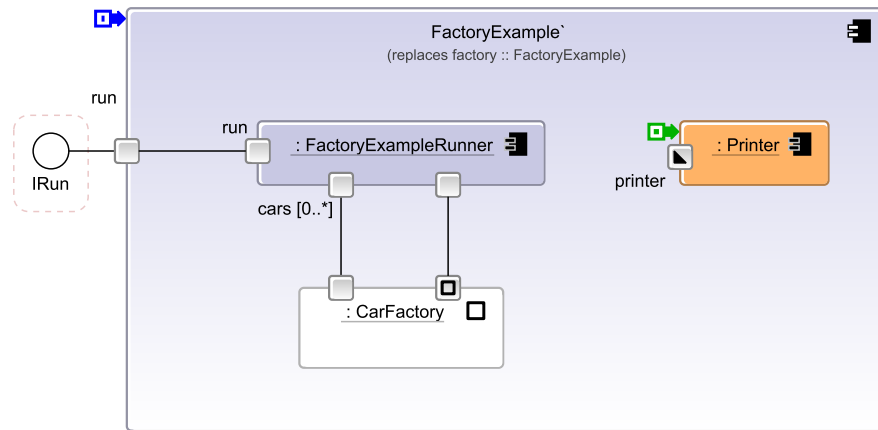


Figure 6.6: Evolving FactoryExample to add a printer

We also need to adjust RentalCarDetails to take advantage of the printer. We evolve it and add the printer port, as shown in figure 6.7.

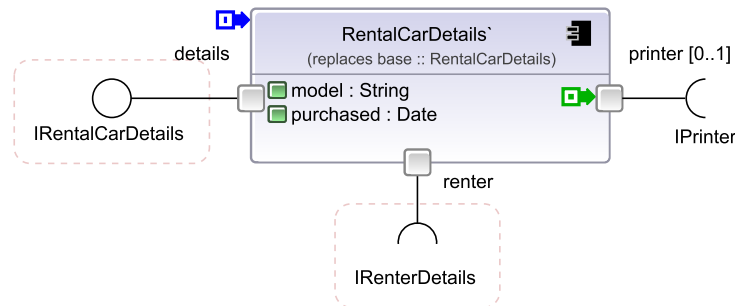


Figure 6.7: The evolved RentalCarDetail component uses the printer

When evolving this leaf, we changed the implementation class name to `PrintingRentalCarDetails` which generated a class which inherits from the original `RentalCarDetails`. If you examine `PrintingRentalCarDetails.java`, you will see that it inherits all of the fields and ports from `RentalCarDetails`, and then overrides the `getPurchased()` method to the following logic:

```
private Date lastAccess = null;
@Override
public Date getPurchased()
{
    if (printer != null)
    {
        String access = ">> purchased field accessed for car "
            + getModel();
        if (lastAccess == null)
            access += ", not previously accessed";
        else
```



```

        access += ", previous access was " + lastAccess;
        printer.print(access);
        lastAccess = new Date();
    }
    return super.getPurchased();
}

```

Running the model in this stratum produces the following output. Compare and contrast this with the output from section 6.1.1.

```

After making first car:
Printer: >> purchased field accessed for car Volkswagen,
           not previously accessed
           Volkswagen, purchased = Mon Nov 10 00:00:00 GMT 2008
After making second car:
Printer: >> purchased field accessed for car Volkswagen,
           previous access was Thu Sep 02 17:16:04 BST 2010
           Volkswagen, purchased = Mon Nov 10 00:00:00 GMT 2008
Printer: >> purchased field accessed for car Saab,
           not previously accessed
           Saab, purchased = Fri Jun 05 00:00:00 BST 2009
After deleting first car:
Printer: >> purchased field accessed for car Saab,
           previous access was Thu Sep 02 17:16:04 BST 2010
           Saab, purchased = Fri Jun 05 00:00:00 BST 2009
After deleting second car:

```

6.2.2 Hyperports Instead of Singletons

Consider the compositional hierarchy changes that the evolutions have introduced, as shown in figure 6.8. The top shows the hierarchy from the perspective of the factory stratum. Applying the evolutions from the hyperport stratum results in the hierarchy below this. The single `Printer` instance is also connected across two levels of the hierarchy and will be used by each (dynamically allocated) instance of `RentalCarDetails`.

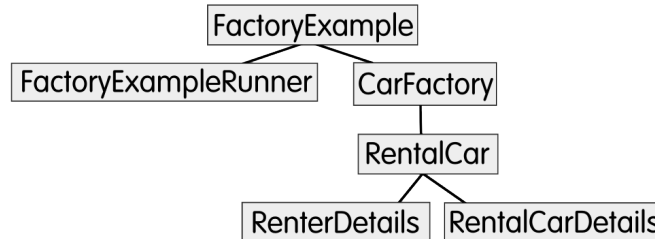
This is a form of singleton, whose scope is constrained by the hierarchy. However, this arrangement is flexible. If we were to evolve `RentalCar` to include its own printer, that would be used in preference and we would have a `Printer` instance per car. For unit testing, we can evolve to replace the real printer part with a mocked out version.

Also, a hyperport will only connect to an unconnected port. If we evolve `RentalCarDetails` to connect to a local printer, then the hyperport above it will not be used.



A hyperport has to allow many connections, as it may be connected to many other ports. A hyperport with just provided interfaces is suitable, as is an indexed hyperport with some required interfaces.

Before evolution:



After evolution:

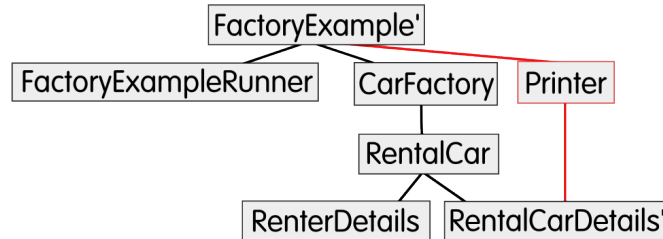


Figure 6.8: Before and after the evolution to add the printer

6.2.3 Evolution Instead of Aspects

Aspects were invented to allow cross-cutting behavior to be added to a program and encapsulated in a single place. As powerful as they are, they suffer from the use of lexical pattern matching which is fragile, no ability for one aspect to directly refer to another, ordering issues, and a trade-off between the amount of context available to an aspect and its level of reuse/applicability. Aspects also are defined separately to the objects they act on, making it difficult to see which ones will be applied at runtime.

Evolution can achieve many of the same effects of aspects, and much more, without the disadvantages. By using evolution to make changes to a system instead of aspects we find that:

- The change is recorded as deltas using UUIDs for identity.
This is not lexically fragile and can be combined and merged with other evolutions.
- We can build on this evolution, and directly refer to the changed elements.
A further stratum, building on `hyperport`, can refer to the evolved components and add its own changes.
- We can see at a glance which changes are applied.
Viewing the compositional hierarchy of the components from the perspective of `hyperport` will show us the full structure, without us needing to work out in our heads what the structure will be like at runtime.
- Evolution has no conflict between a primary and secondary axis.
Aspects are often criticized for being the (weaker) secondary axis to the primary axis of the programming language. Although systems like Hyper-J have partially dealt with this, AOP systems in industrial usage suffer from this limitation.

Evolution can also be used to add in parts between existing connections, much like an explicit form of the Spring proxy-based AOP approach. Consider figure 6.9 which shows an evolution of `RentalCar` that places a proxy part in between the two existing parts. To accomplish this, the evolution simply added the proxy part and connector, and re-routed an existing connector.

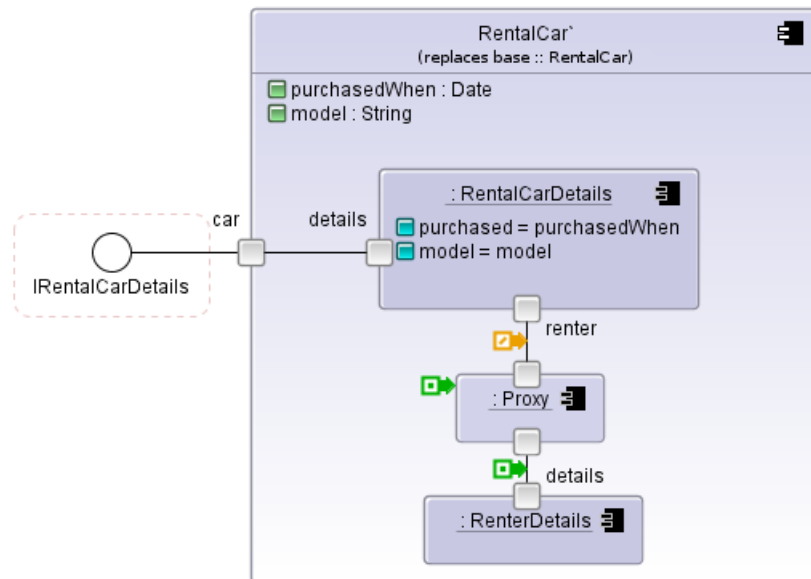


Figure 6.9: Using evolution to add a proxy

The quantification side of aspects (e.g. `set*`) can be partially achieved by evolving a base component in a resemblance tree.

These comments may be upsetting to AOP advocates. It is true that the above discussion is only cursory, and that the printer example could have also been achieved using AOP. However, the advantages of the evolution approach start to accrue when the examples become more involved than this short tutorial will allow. I will address this topic in a later manual on advanced techniques.

Section 9.5 discusses aspects further.

6.3 State Machines

State machines are a technique for describing how a system's state undergoes transitions in response to events. They are useful precisely because they force the programmer to be explicit about which events are handled and what these will do to the system.

Evolve makes the creation, reuse and evolution of state machines as simple as using components. In fact, Evolve state machines are really just components “under the covers”, with the visual appearance of UML2 statecharts. They can be combined with ordinary components in powerful ways. Further, Evolve state machines are executable.

If you have yet to experience the power of states, now is the time! Navigate into the states stratum for this part of the tutorial.

6.3.1 A Top-Down Overview

Our example is going to replace the `RenterDetails` part with a state machine, in the `CarRental` definition shown in figure 4.7. This state machine will feature two states: `available` and `rented`. When someone rents the car, we will send in a `rent()` event and move to the `rented` state. When the renter returns the car we will send in a `returnRental()` event and move back to the `available` state.

We are going to describe this in a top down fashion. Figure 6.18 shows the full state machine, with the transitions between the states.

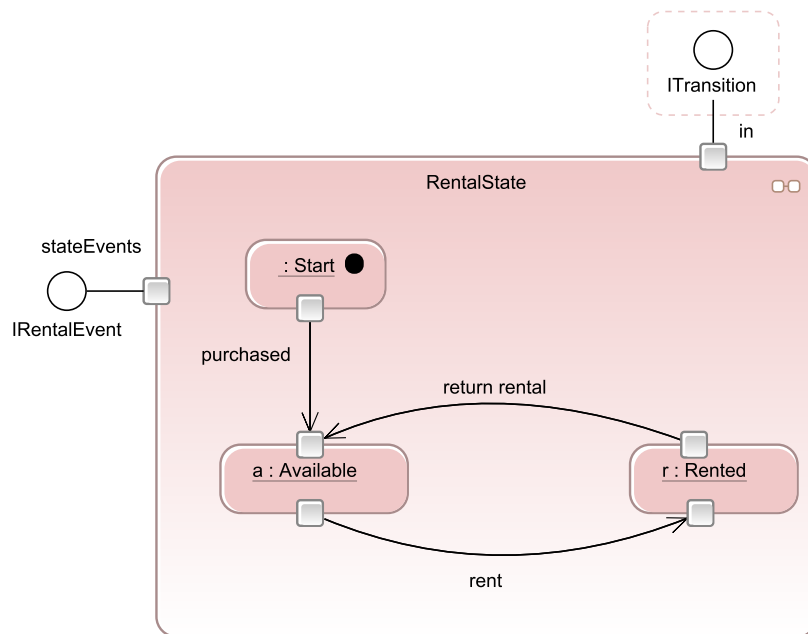


Figure 6.10: The state machine

As the state machine is really just a composite component, it has only a Backbone representation and no associated Java code. The transitions, shown with arrows, are just connectors wiring from the out port of one state to in ports of other states.

The `Start` part was inherited when we created the composite, as were the `in` and `event` ports (along with an `out` port and `End` part which were delta deleted). All composite states are automatically created resembling the `CompositeState` component, from which they inherit these constituents.

The `Available` and `Rented` states are shown in figure 6.11. They are both really just leaf components, and hence must be associated with a Java implementation class each: `Available.java` and `Rented.java`. Note that we have added an attribute to the `Rented` state for holding the renter's name.

We also need to define an event interface, in order to trigger the transitions. This interface must resemble `IEvent`. We further make it resemble `IRenterDetails` from the base stratum, in order to allow it to be connected up to `RentalCarDetails` easily. The interface is shown in figure 6.12.

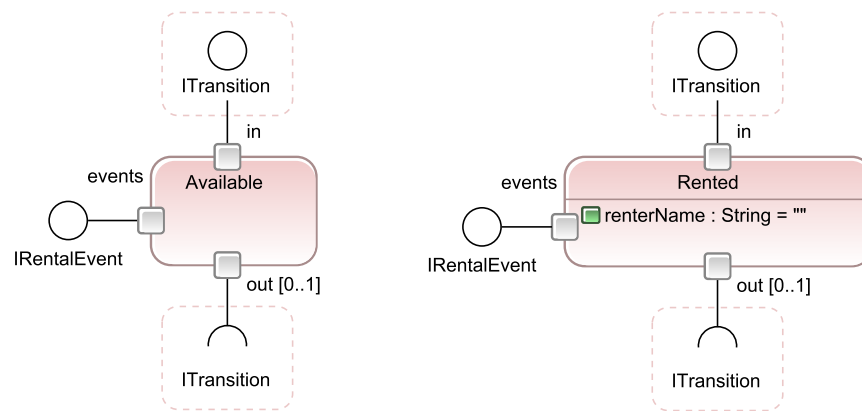


Figure 6.11: The Available and Rented states

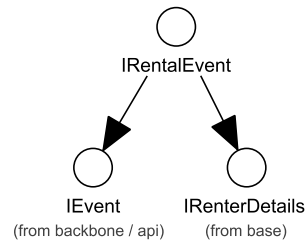


Figure 6.12: The event interface for our state machine

Finally we evolve `RentalCar` in order to replace the `RenterDetails` part. This is shown in figure 6.13. Note that we have also chosen to export the state port, so that we can drive the state machine externally.

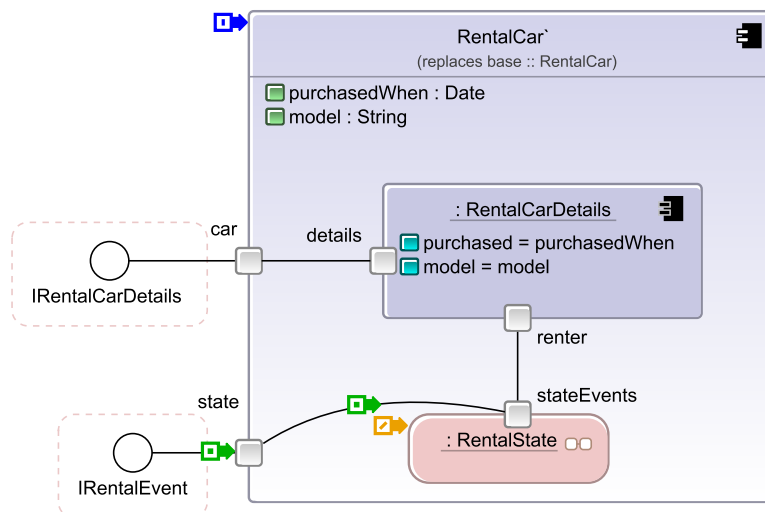


Figure 6.13: Evolving RentalCar to add in the state part

Finally, we create a simple runner and a composite to bring it all together. This is shown in figure 6.14.

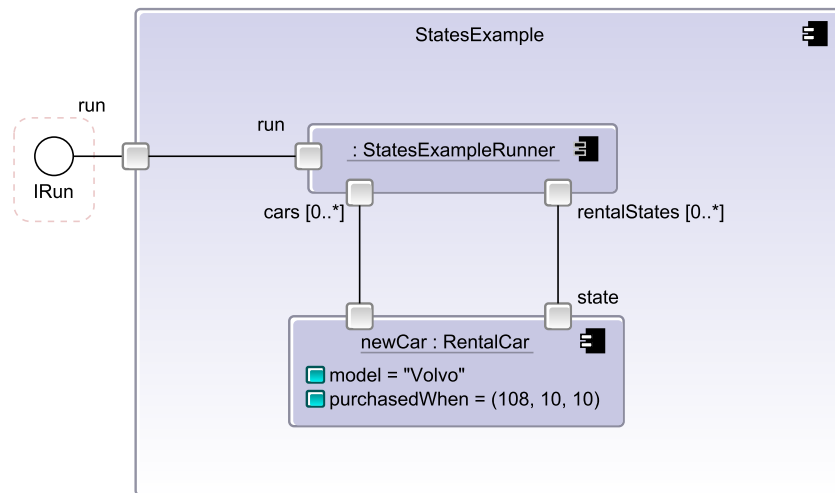


Figure 6.14: The runnable configuration

The compositional hierarchy of the StatesExample component is shown in figure 6.15. It shows that the evolutions have successfully replaced the RenterDetails part with a state machine consisting of Start, Available and Rented states.

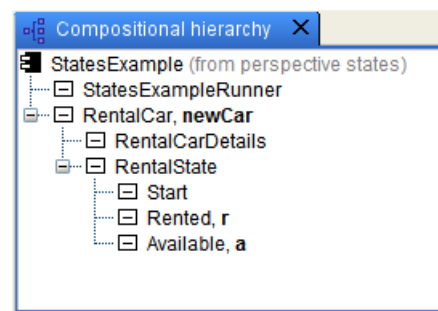


Figure 6.15: The compositional hierarchy of StatesExample

6.3.2 Running the Machine

Let's run the example. Tag the stratum, and the following output should appear.

```
Car: model = Volvo, purchased = Mon Nov 10 00:00:00 GMT 2008,
    is rented = false
: State dispatcher: current state = available
--> Sending event: rent
: State dispatcher: current state = rented
```

```
--> setting the renter now
Car: model = Volvo, purchased = Mon Nov 10 00:00:00 GMT 2008,
    is rented = true
    : State dispatcher: current state = rented
--> Sending event: return rental
Car: model = Volvo, purchased = Mon Nov 10 00:00:00 GMT 2008,
    is rented = false
    : State dispatcher: current state = available
```

Examine the code in the `StatesExampleRunner.java` class to look at the events sent to the machine.

```
state.rent();
...
state.setRenterName("Andrew");
...
state.returnRental();
```

The state machine will automatically transition from the `Start` state into the `Available` state upon instantiation. The `rent()` event then pushes it into the `Rented` state where setting the renter name will store the details in the `renterName` attribute. Finally, sending in the `returnRental()` event causes a transition back to `Available`.

6.3.3 Creating the State Machine and Leaf States

We created each of the leaf state machines by changing to the “State focus” by invoking the `Focus▷State focus` menu option. We then used the “State” tool from the palette. Creating a leaf state in this way produced the component shown in figure 6.16. It inherits the events, in and out ports from the State component it automatically resembles.

We next performed a bit of surgery on this to create the `Rented` state. We added the `renterName` attribute. We also replaced the events port to make it provide `IRentalEvent` which is the event interface we are using. This is shown in figure 6.17. `Available` was created in a similar way.

Finally we created the composite state machine by using the “Composite state” tool. Upon creating an empty composite, it inherits the three ports and start and end parts from `CompositeState`. We delta deleted the end state, as our example state machine never terminates, and we replaced the events port so that it provides our event interface. Finally, we added the state parts (using the “State part” tool) and wired them up using the “Transition” tool. The result was our state machine, as shown in figure 6.18.

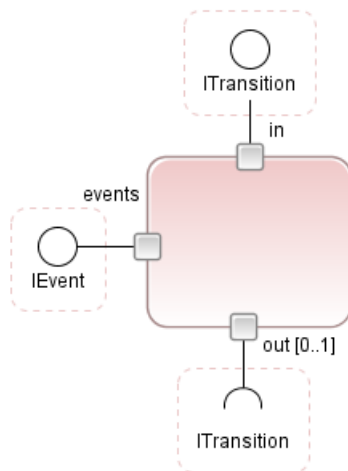


Figure 6.16: An empty leaf state

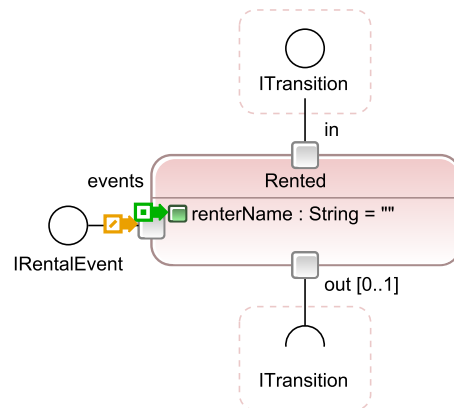


Figure 6.17: The Rented state with delta marks on

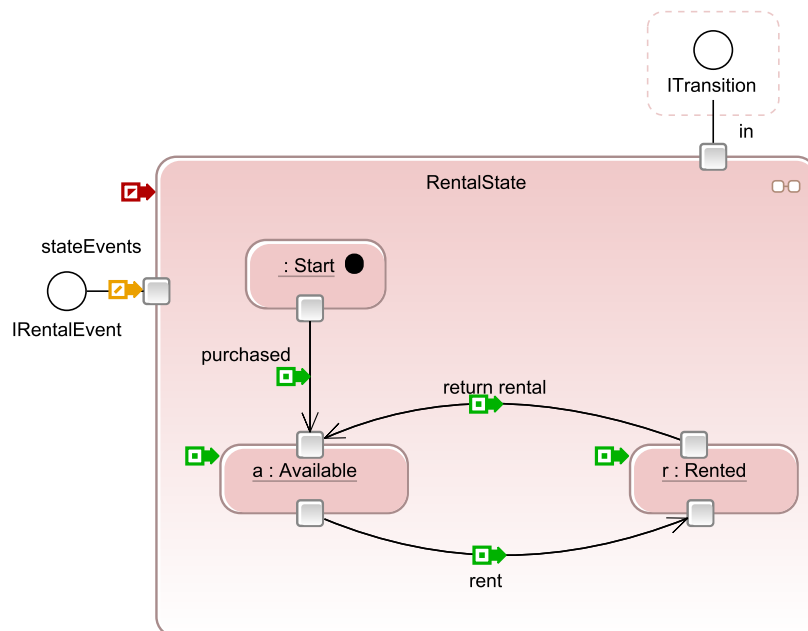


Figure 6.18: The state machine with delta marks on

6.3.4 Replacing a Part with One Of a Different Type

When evolving a component, the simplest way to replace a part is to right-click on it and select “Replace” from the menu. This will give us a replacement part of exactly the same type. This is how we replaced the part in figure 5.4.

However, when we evolved RentalCar in figure 6.13, we instead replaced the RenterDetails part with a RentalState part. In other words, we changed the component that the part is an instance of.

To achieve this we must first place a view of RentalCar on the diagram, and right-click and choose “Evolve” to make an evolution. Then, add a RentalState part and move its ports to be roughly in the same proximity to the part as the ports that we want to map to on the RenterDetails part. You can also delete any unwanted ports. Then, select the RentalState part *but* right-click over the existing RenterDetails part and select “Replace (with existing part)”. This is shown in figure 6.19.

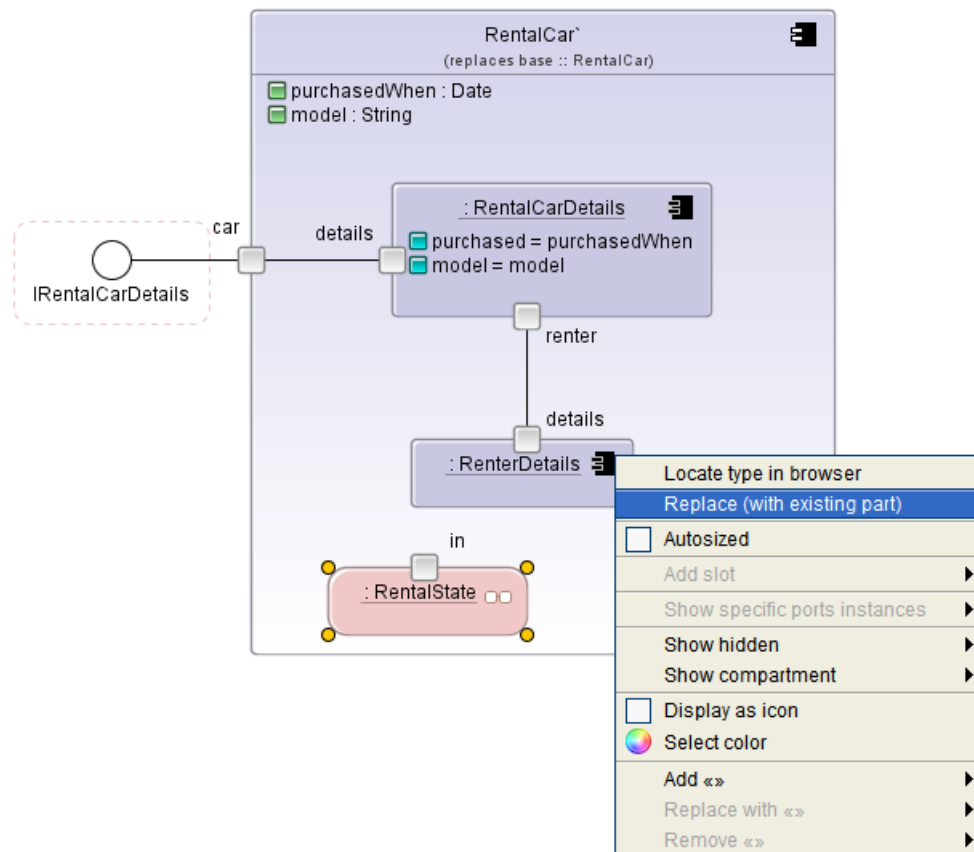


Figure 6.19: Replacing an existing part with a new one of different type

This will replace the part with the new one, and preserve all existing connections.

Implementing the Leaf States

The Java implementation of each leaf state must follow a simple protocol to work correctly - look at `Available.java` for a guide. Note that nearly all of the logic is already inherited from the `State` superclass, including the implementation of the `in` port.

When a state wants to transition to another state, it calls the `enter()` method on the next state via the `out` port and then tests the return value to decide if the transition has been successful. When the machine starts, the start state will call `enter()` on our `Available` part which will make that the current state.

If we want to transition to another state, we use the `enter()` method on the `out` port, as per below. This is the implementation of the `rent()` event for `Available`.

```
public void rent()
{
    current = !out.enter();
}
```

If the next state (`Rented`) accepts the transition, the `Available` state will then cease to be current.

If you want to change the inherited implementation of the `in` port, which by default accepts any transition, replace it and call it a different name. This allows transition acceptance logic to be customized for the state.

Nested State Machines

State machines can be treated as state parts in another machine, in an analogous way to how composite components can be treated as parts in another component. This allows us to use the powerful approach of nested state machines, first explained by David Harel in his groundbreaking work on statecharts⁵.

Multiple Out Ports

It is possible to create multiple `out` ports for a state thereby allowing different transitions outwards on different conditions. There is no special handling for the existing `out` port - it is just provided as a convenience.

In a similar way, a single state machine may instantiate many end terminals. Only one start terminal is allowed, however.

⁵If you can get hold of it, I thoroughly recommend the book "Modeling Reactive Systems with Statecharts" by David Harel and Michal Politi. This describes the STATEMATE tool, but most of the techniques can also be used in Evolve.

Combining States and Components

Combining state machines and components is simple - just include the state parts in the component, or the component parts in the state machine. We have already done the former in figure 6.13. The latter is useful for making component services available to individual states. Figure 6.20 shows how we might adjust our machine to directly propagate a service to the boundary of the composite, or how the Rented part might make use of an embedded AddressBook part.

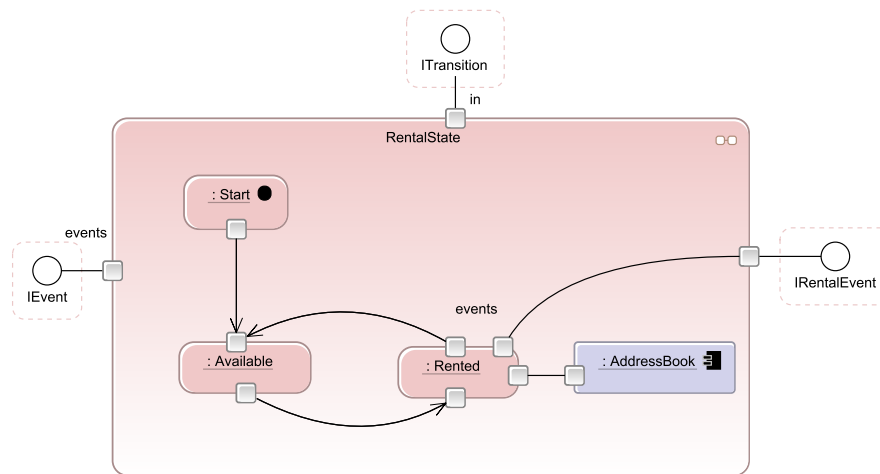


Figure 6.20: Combining state and component parts

How Do State Machines Work?

We've already discussed how state machines are simply composite components with a different visual presentation. However, there is something else clearly going on. For instance, what is behind the `stateEvents` port of figure 6.18? Something must be handling the events!

The same expansion mechanism used for factories (section 6.1.5) is also used for state charts. In this case, a `StateDispatcher` part is automatically inserted which is connected up to the `startEvents` port and all the states. Figure 6.21 shows what the expanded state machine really looks like after removing all the visual niceties. The complexity of this expanded model is the reason why we decided to use a different visual presentation.

The dispatcher acts as a switch. When a method is called on the event interface, the dispatcher routes it to the current state. If a state wishes to transition to another state, it calls via one of its out transition connectors and passes the notion of "current" to it. The dispatcher then calls `isCurrent()` on all the states to work out which is now the current state. Voila, we have a fully functioning state machine.

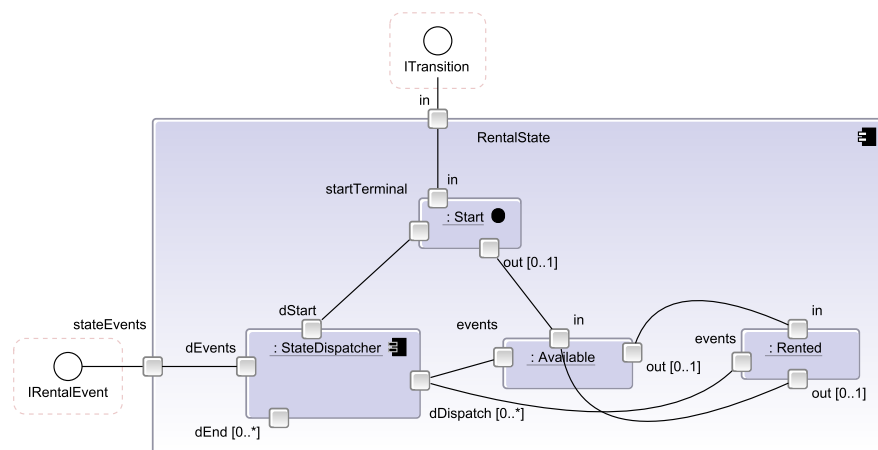


Figure 6.21: The real state machine, as components

Evolve State Machine Limitations

Evolve state machines implement a subset of UML2 state machines. Orthogonal state partitions are not currently supported, nor are history or deep history states⁶. Choice states are not supported explicitly, but can easily be implemented by a standard state.

Summary

Evolve state machines allow you to use states and explicit transitions to model your application, whilst also retaining all the power of the component-based approach. They tie directly into implementation, so your state machines are executable once you have defined the leaves.

These state machines have been designed as part of research into highly extensible systems. You can resemble a state machine and customize states and transitions, or evolve an existing state machine.

Finally and thankfully, the approach also avoids the horrors of the State design pattern from the Gang of Four Design Patterns book⁷.

⁶These facilities are planned for a future release. Orthogonal states will be particularly useful as they will allow for state-based concurrency.

⁷Don't get me started. The issues with the state design pattern are numerous and include limited context available to each state, having transitions fully in code making them hard to see and change, and a lack of support for inheritance and reuse. I'll stop now as I'm getting worked up about it again...

Chapter 7

Tutorial D: The GWT / Hibernate Car Rental System

In this tutorial we will build a GWT ajax front-end for the car rental system, and a back-end service built on Hibernate and the H2 SQL database. To whet your appetite, figure 7.1 shows a picture of the GUI where one person has rented a car, and a further three cars are still available. Figure 7.2 shows the corresponding entries in the database.

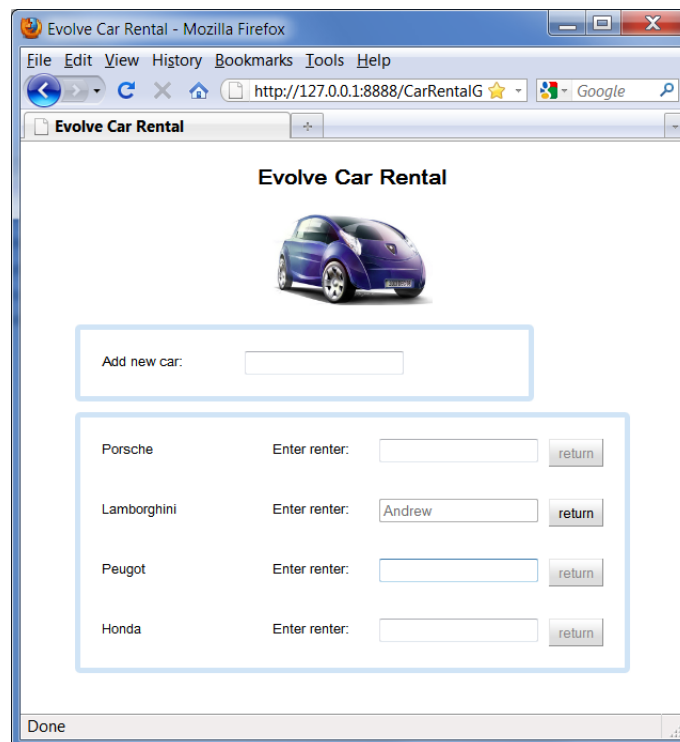


Figure 7.1: The Car Rental GWT front end

Although we have deliberately kept this example small, it demonstrates convincingly that Evolve's component approach can integrate with some fairly exotic technologies. In the case of GWT, our

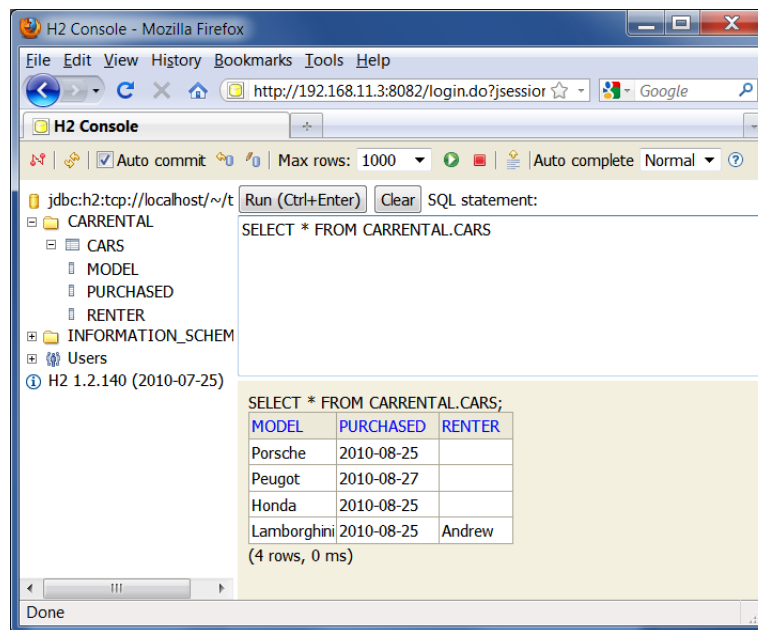


Figure 7.2: The car rental database rows

components will actually be running in the browser as GWT-translated Javascript. In the case of Hibernate, the RentalCar parts will be persisted and recreated from a SQL database. This can all be achieved without any changes to our approach and the full complement of techniques including components, state machines, evolution and resemblance are available.

This tutorial covers the `gwt`, `service interfaces`, `gwtgui`, `gwtservice` and `hibernate` strata from the example model. For reference, we repeat a picture of these strata in figure 7.3. Note that the only part of this tutorial that builds on the previous tutorials is the back end - the `gwtservice` and `hibernate` strata depend on the definitions of rental cars and their factories, as they will create, manipulate and persist instances of cars. The front end does not use the car components.

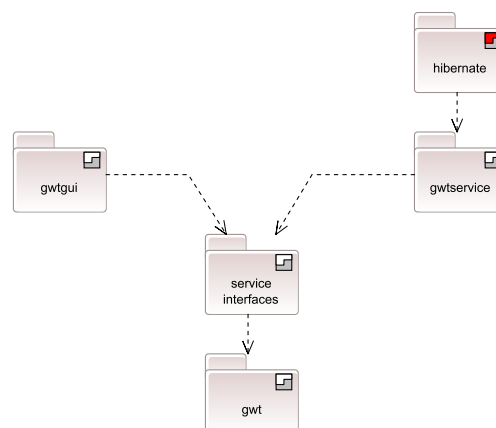


Figure 7.3: The strata used in tutorial D

7.1 Setting Up and Running the Example



Disclaimer: This chapter contains very Eclipse-centric build and run instructions. We created the example in Eclipse using the Google GWT plugin. We apologize for the exclusive focus on a single IDE but we trust that users of other environments will be able to tweak the example to get it running also. We will look at rectifying this situation in the future to at least include an ant build script.

Although the strata for this example are present in the `car-rental.evolve` model, the Java files and libraries are not included with the standard Evolve distribution and must be downloaded separately. Visit the following site.

<http://intrinsarc.com/evolve/downloads>

Download the `tutorial-d-files.zip` file. This contains the `CarRentalGUI` folder at the top level. Unzip this into the `tutorial` directory under the Evolve installation area. The files in the `tutorial` area should look like those in figure 7.4.

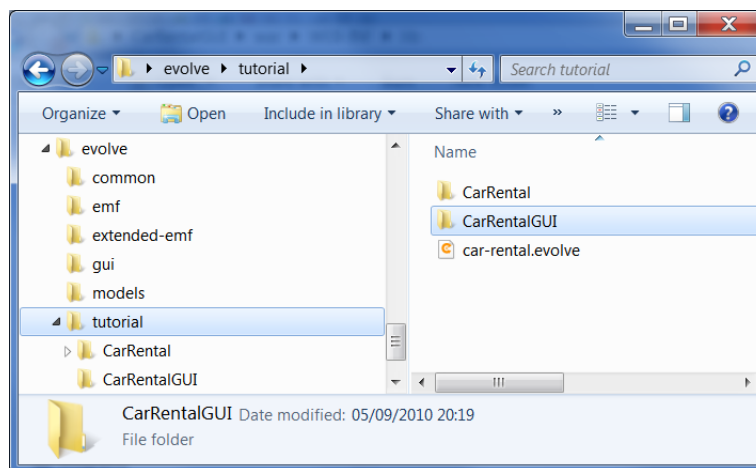


Figure 7.4: Unzip the `CarRentalGUI` folder in the Evolve tutorial directory

To build and run this example you will need:

- Eclipse from <http://www.eclipse.org>
We used Eclipse Helios, version 3.6.
- The GWT plugin for Eclipse from <http://code.google.com/eclipse>
We used the GWT plugin for Eclipse 3.6. Install this by adding the update site to Eclipse using the instructions at the GWT plugin site. This will also download the GWT toolkit. We used version 2.0.4.

- The H2 database from <http://www.h2database.com>
You can use any SQL database really, but H2 is our personal favorite.

We also used Hibernate, but we included the jars in the example so that you do not have to download it.

7.1.1 Setting Up Eclipse

Install the GWT plugin for Eclipse by following the quick start instructions at

http://code.google.com/eclipse/docs/getting_started.html

Ensure that the EVOLVE classpath variable is set in Eclipse to point to Evolve's installation directory. You should have this set up already for the previous tutorials, as described in section 2.4.

Import the project into Eclipse using the **File>Import>Existing Projects into Workspace** menu option. Use the dialog to select the `CarRentalGUI` folder and import the project. Ensure that you don't use the "copy into workspace" option or else the files loaded into Eclipse and the files that Evolve generates will be in different directories.

Note that sometimes Eclipse requires a restart to pick up the correct GWT_SDK reference in the libraries.

7.1.2 Setting Up the H2 Database

Install the correct H2 distribution for your platform.

The connection details used from our GWT back end are kept in `CarRentalGUI\src\hibernate.cfg.xml`. These are shown below.

```
<property name="hibernate.connection.url">
    jdbc:h2:tcp://localhost/~ /carrental
</property>
<property name="hibernate.connection.username">
    sa
</property>
<property name="hibernate.default_schema">
    CarRental
</property>
```

To create this database, log in to the H2 console using the parameters shown in figure 7.5.

Then use the H2 console to create the schema in file `CarRentalGUI\src\schema.ddl`. This is shown in figure 7.6.

We used version 1.2.140 of the H2 client jar. If you use a different version of H2 ensure that you copy the client jar over to replace the one at `CarRentalGUI\war\WEB-INF\lib` and that you adjust the library references in the Eclipse project.

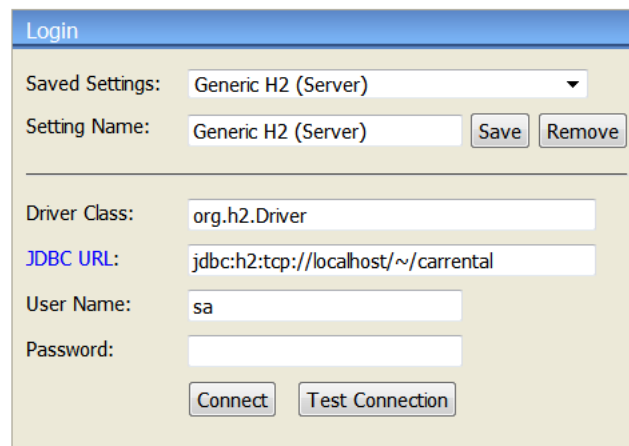


Figure 7.5: Logging into the H2 console

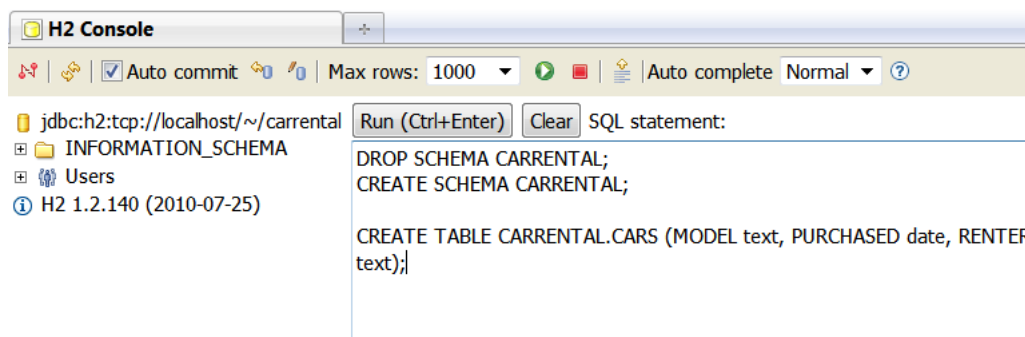


Figure 7.6: Creating the database schema

7.1.3 Setting Up the Evolve Variables

Invoke the **Window>Preferences** menu option in Eclipse, choose the “Variables” tab and add the following three variables.

- **CARSGUI**
Point this at the tutorial\CarRentalGUI directory of the Evolve installation.
- **GWT**
Point this at the directory containing the gwt-user.jar file that was installed by the GWT plugin. You will need to search for this in your Eclipse plugin area.
- **JRTLIB**
Point this at a directory containing the JRE rt.jar library.

Phew, that’s it! You should now be able to test your installation by running the project. Right-click the CarRentalGUI project in Eclipse and invoke the **Run as>Web application** menu option provided by the GWT plugin. If you then point a web browser at the URL listed in the GWT console, you should see something similar to figure 7.1.

7.2 Creating the Back End: The Rental Car Service

In this section we explain how to create the Hibernate/H2 back end service for persisting, manipulating and retrieving rental car components. We first build a service which doesn't use persistence. We then evolve this to add Hibernate support and transactions.

7.2.1 Creating the GWT Service Interface

Navigate into the `service interfaces` stratum. You will see the `IRentalServiceAsync` and `IRentalService` interfaces, which describe the service interface(s) for our car rental system. These interfaces were actually created by the project wizard supplied by the Google GWT plugin when we made the project in Eclipse. We imported the interfaces into Evolve using the Bean Importer so that we could model with them.

As per GWT conventions, `IRentalServiceAsync` is the asynchronous interface to be used by the ajax front end. `IRentalService` is the interface that our server must support.

The methods on `IRentalService` show the calls implemented by the server.

```
public interface IRentalService extends RemoteService
{
    void createRentalCar(String model);
    void rent(int car, String renter);
    void returnRental(int car);
    String[] getCars();
}
```

The last call returns a string array where each line contains the car model and the name of the renter separated by a delimiter.

7.2.2 Creating a Service Without Persistence

Navigate into the `gwt-service` stratum. Here we have defined a server which uses the existing `CarFactory` (section 6.1) factory component to dynamically create cars when the `createRentalCar()` method is called. The `RentalServiceLogic` leaf implements the logic of the server as shown in figure 7.7.

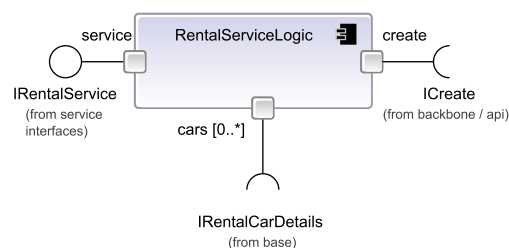


Figure 7.7: The `RentalServiceLogic` leaf implements the logic of the server

The code for this bean is in `RentalServiceLogic.java` and has been deliberately kept simple. Below is the implementation of the `createRentalCar()` method. Of particular interest is the use of parameters to set the car model and purchase date.

```
public void createRentalCar(String model)
{
    Map<String, Object> params = new HashMap<String, Object>();
    params.put("model", model);
    params.put("purchased", new Date());
    create.create(params);
}
```

We then connect this up to the factory part via a composite, as in figure 7.8.

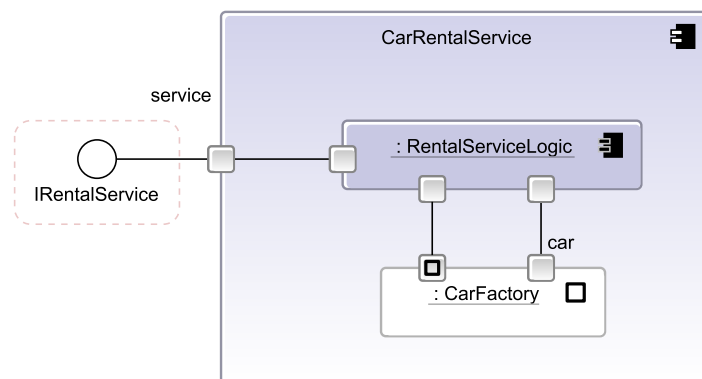


Figure 7.8: Connecting up to a car factory

As the GWT environment cannot use the Backbone interpreter¹, we generated a full implementation (`Backbone>Generate full implementation`) of the `CarRentalService` component to allow our server to be run in the GWT environment.

This created the `CarRentalServiceFactory.java` file.

We now need to “plumb” this into the GWT generated skeleton for the server. When we created the Eclipse project, the GWT project wizard created the `RentalServiceImpl.java` class. We modify it as follows to instantiate the `CarRentalService` component and delegate to it.

```
public class RentalServiceImpl extends RemoteServiceServlet
    implements IRentalService
{
    private IRentalService service =
        new CarRentalServiceFactory().initialize(null, null).
            getService_Provided();
}
```

¹The interpreter uses standard Java reflection which is not allowed in Java GWT code.

```

    public void createRentalCar(String model)
    {
        service.createRentalCar(model);
    }
    ...
}

```

That's it - we now have a rental service created in Evolve that we can run in GWT. Note that the cars created by this service are not persisted - they reside only in memory and will be destroyed when the application terminates.

7.2.3 Creating a Service With Persistence

Navigate into the hibernate stratum. We now amend our server to use Hibernate to persist the cars into a SQL database.

To accomplish this, we simply evolve the RentalServiceLogic leaf, as shown in figure 7.9. We have renamed the leaf to HibernateRentalServiceLogic as part of the evolution which has also altered the implementation class to HibernateRentalServiceLogic.java. We have turned off implementation inheritance, and enabled lifecycle callbacks.

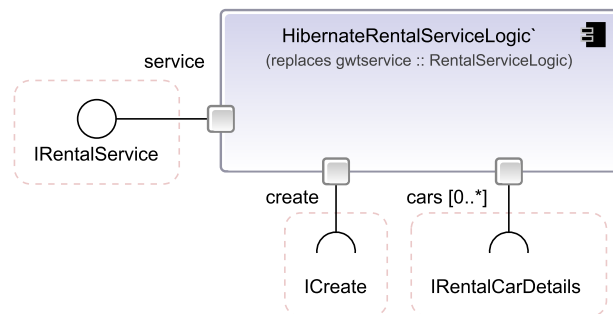


Figure 7.9: Evolving CarRental to add Hibernate support

Examining the HibernateRentalServiceLogic.java class we see the following code used for initialization.

```

private Session session;
public void afterInit()
{
    SessionFactory sessionFactory =
        new Configuration().configure().buildSessionFactory();
    session = sessionFactory.openSession();
    // get all the existing rental cars
    Transaction t = session.beginTransaction();
    cars.addAll(
        session.createQuery("from RentalCarDetails").list());
}

```

```

        System.out.println("Found " + cars.size() +
            " existing car(s) in database");
        t.commit();
    }

```

When the service starts up, the `afterInit()` method is called for the lifecycle callback. This method establishes a Hibernate session (using the connection settings in `hibernate.cfg.xml`) and uses it to find all existing cars in the database. It then adds these to its cars port. In other words, this restores the full state of the server from the database at startup time, and recreates the set of `RentalCar` instances including their connections. Any further created cars will also be added to this port.

The logic for creating a new car is a variation on the non-persistent version.

```

public void createRentalCar(String model)
{
    Transaction t = session.beginTransaction();
    Map<String, Object> params = new HashMap<String, Object>();
    params.put("model", model);
    params.put("purchased", new Date());
    create.create(params);
    session.save(cars.get(cars.size() - 1));
    t.commit();
}

```

It's as simple as that. Note that if we look at the `CarRentalService` composite from this perspective, we can see that the evolution of `CarRental` has automatically affected this as we would expect, as shown in figure 7.10.

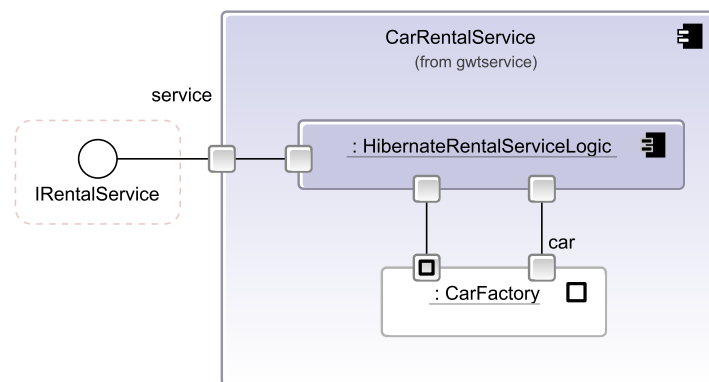


Figure 7.10: `CarRentalService` from the hibernate perspective

7.2.4 Mapping the Components Using Hibernate

We are persisting `RentalCar` parts, each of which are made up of a `RentalCarDetails` instance connected to a `RenterDetails` instance. This connection can be seen by looking at the

RentalCar definition (figure 4.7). As these are instances of beans, persisting them in Hibernate is trivial.

The `carrental.hbm.xml` Hibernate mapping file is shown below. We have used field-level access in order to avoid the need for extra setters, and the renter's name from `RenterDetails` is included in the same table as the car details.

```
<hibernate-mapping default-access="field">
<class name="com.intrinsarc.base.RentalCarDetails" table="CARS">
  <id name="model" type="java.lang.String" column="MODEL" />
  <property name="purchased" type="java.util.Date" column="PURCHASED" />
  <component name="renter" class="com.intrinsarc.base.RenterDetails">
    <property name="renterName" type="java.lang.String" column="RENTER" />
  </component>
</class>
</hibernate-mapping>
```

The DDL for creating the database is in `schema.ddl` and is similarly obvious.

```
CREATE TABLE CARRENTAL.CARS (MODEL text, PURCHASED date, RENTER text);
```

If you are running the example, don't forget to create the table in H2 first.

7.2.5 Alternating Between the Persistent and Non-Persistent Variants

Since the `hibernate stratum` evolves `RentalServiceLogic`, when we generate the full implementation we will overwrite the same `CarRentalServiceFactory.java` file that the `gwt service stratum` also writes to. The stratum we use last to generate a full implementation will be the one used when we run our GWT example.

This allows us to quickly switch between the persistent and non-persistent implementations when testing.

7.3 Creating the User Interface

In this section, we show how to import the GWT widgets as Evolve leaves, and then use these to construct our user interface.

7.3.1 Importing the GWT Widgets into Evolve

Before using GWT to create our interface, we must make the GWT widgets available for use within Evolve. As per Java conventions, they are packaged as JavaBean-like classes and we can use the Bean Importer tool to import these into Evolve as leaves.

We now describe the steps we took to import the GWT widgets into the tutorial model.

First, we created the `gwt` stratum to hold the definitions. We set the `bb-classpath` stereotype property to point to both the `gwt-user.jar` and the `JDK rt.jar` (needed for some primitives) and we started up the importer. We selected the GWT widget package in the top left tree of the importer (`com.google.gwt.user.client.ui`), selected all the beans that were found, and then clicked “Add to import list”. The result is figure 7.11.

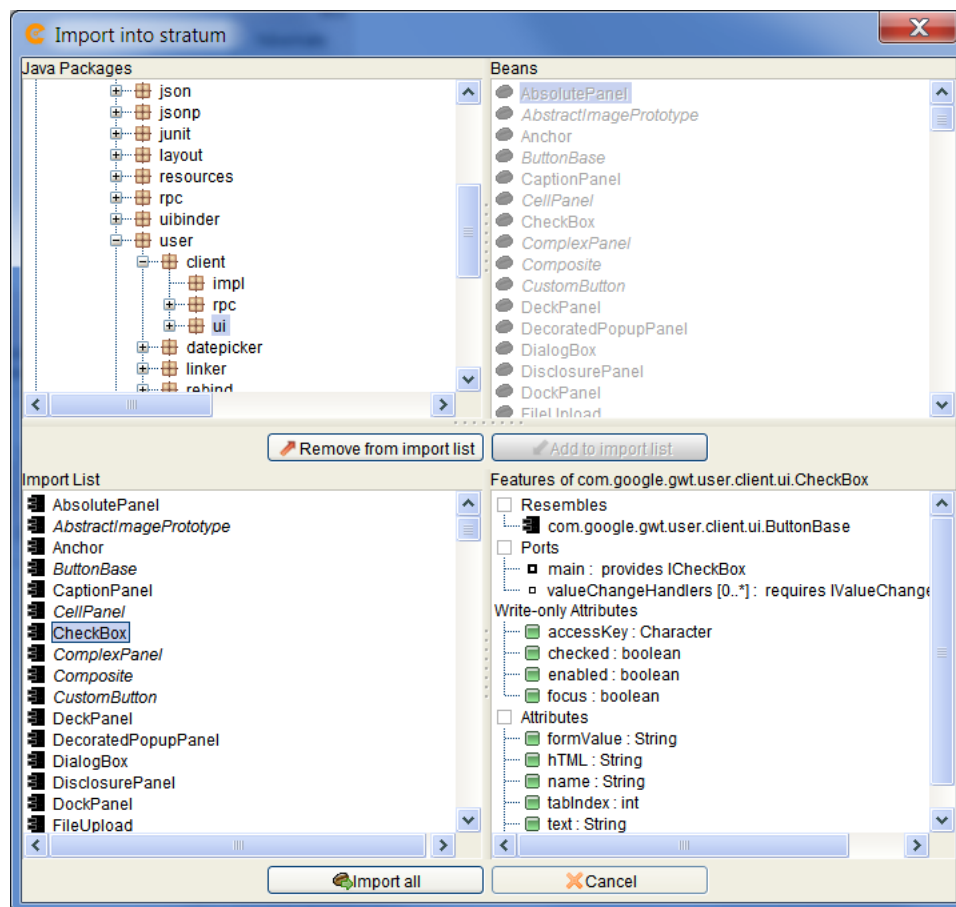


Figure 7.11: Importing the GWT widgets



The importer sometimes confuses a bean for a primitive and vice versa. In the former situation, this generally happens because the bean has no explicit set or get methods. To correct this, right-click on the item in the browser's lower left tree and select **Toggle bean/primitive**.



The importer will use the Java package, of the stratum it is importing into, to decide if it needs to force the implementation class of an element. If the package combined with the element name is correct, **force-implementation** will not be set for the leaf or interface.

In the lower right pane of the importer, we can see how the **CheckBox** bean translated into a leaf component. After noting that there were no errors, we then clicked "Import all" and imported the beans into the stratum. In figure 7.12 we can see the subject browser showing the imported components.

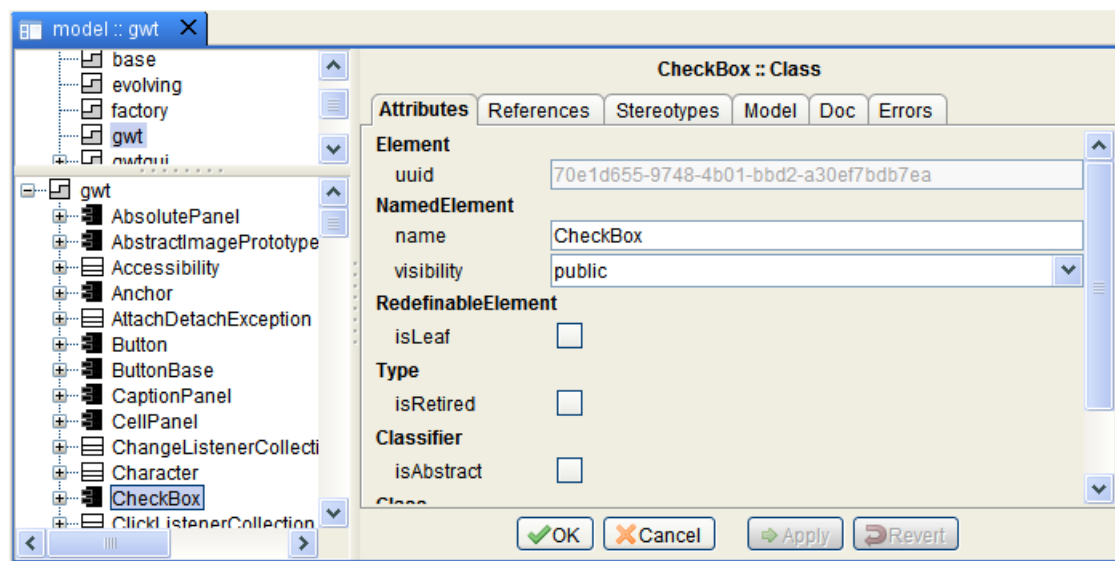


Figure 7.12: The subject browser showing the imported components



If you are going to try the above import on the existing gwt stratum, ensure that you make that stratum writable first. The example model has marked this stratum read only. Right click on the stratum and uncheck the **Read only** option. Notice also that because we have already imported the GWT components into this stratum, you will be presented with refresh options. If you want to start from the beginning, make another stratum, make it depend on the backbone stratum, and copy over the **bb-classpath** property from the gwt stratum. You might also want to do this in a new model.

Creating Views of the Imported Components

We then navigated into the gwt stratum. Because we hadn't yet made any views of the components, the diagram was empty.

To create views, we used the referencing technique first shown in figure 2.5. We created a new component on the diagram, typed the first few characters of `CheckBox`, pressed `tab` and then selected `CheckBox` from the list. We now had a view of the imported component. Note that no ports or attributes were showing at this point.

Because imported components can be very complex with literally hundreds of ports and attributes, by default the first ever view of a component will show none of these. To make them visible, right click on the component and choose `Show specific attributes` or `Show specific ports`. If you want to see all ports and attributes, choose the `Show hidden▷Attributes` or the `Show hidden▷Ports` options.

Legacy versus Non-Legacy Beans

When a bean is imported, if it does not already exist in the model then it is imported as a “legacy” bean. To see what this means, consider that most existing JavaBeans do not implement an interface, but instead rely on direct access to the class. The listing below shows the form of most beans in existing libraries.

```
public class Bean
{
    private int attribute;
    ...
}
public class NextBean
{
    private Bean required;
    ...
}
```

This typical bean is not in the interface-centric style that Evolve uses for components. In contrast to the above, leaves generated from Evolve will always implement interfaces and no leaf will reference another leaf class directly.

To fit existing beans into the Evolve approach, when a bean is imported it is marked as legacy and a synthetic interface is created. The implementation class of this interface is set to the same class as the bean itself. This lets us pretend that the bean is a conventional leaf. Figure 7.13 shows what the beans from the listing above would look like after import.

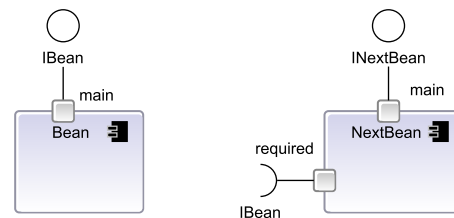


Figure 7.13: Legacy beans have a synthetic interface to mimic real leaves

All of the GWT widgets will be organized in this way on import.

7.3.2 Connecting Up Widgets

We can now use the GWT widgets to construct our GUI. Navigate into the `gwtgui` stratum. You will see three packages. Navigate into the `addcar` widget package and you will see the component shown in figure 7.14 below. This defines the “Add new car:” widget shown at the top of figure 7.1. It allows a new rental car to be entered into the system.

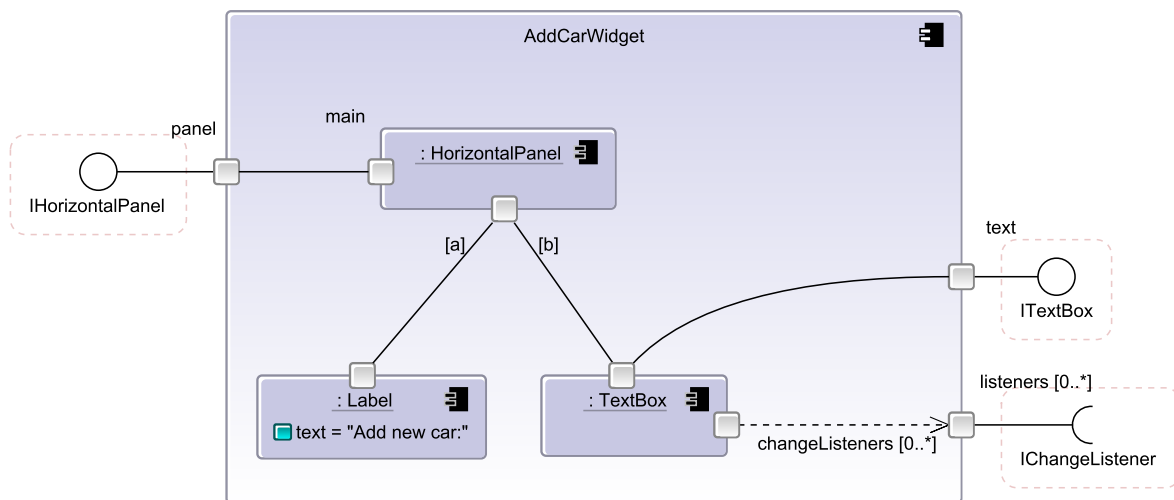


Figure 7.14: The AddCarWidget composite

The widget itself is pretty simple. It uses a horizontal panel to bind together a label and a textbox for entering the car’s information into.



Interestingly, we have managed to define `AddCarWidget` without any GWT-specific support for composite widgets. We have simply used normal Evolve practices. This widget can now be used as a part in a larger design, as we shall soon see.

Now consider building a widget showing a single car and its rental status, with logic to transition between rented and available. Navigate into the `rentallogic & widget` package. Here we have

defined another composite widget as shown in figure 7.15. Multiple of these are shown in the lower part of figure 7.1, one per car.

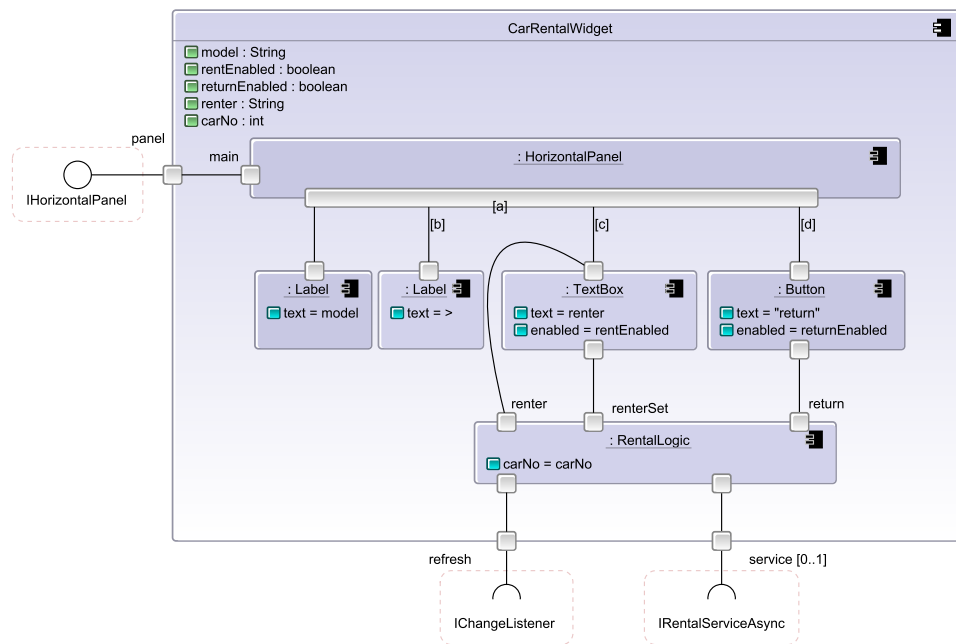


Figure 7.15: The widget for viewing and renting a car

The CarRentalWidget is defined in a similar way to AddCarWidget. In this case a horizontal panel is used to structure two labels, a textbox and a button. We have also added a RentalLogic part which adds some logic to call out to the server to rent the car when the textbox is entered and return it when the button is clicked.

Note the “>” in the value of the slot in the second label. This indicates that the actual value should be taken from the actual-slot-value stereotype property of the slot, and is quite useful when values are long. This property is set to “Enter renter:”.

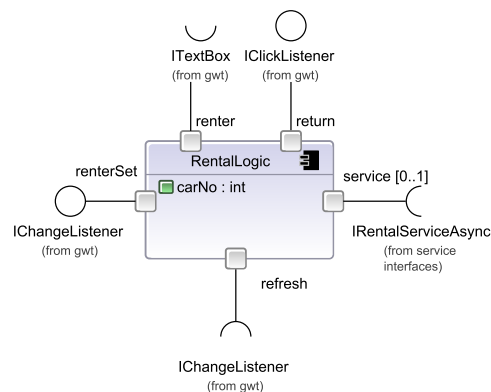


Figure 7.16: The component for handling the rental logic

The RentalLogic leaf, shown in figure 7.16, handles the logic for when a car is rented out and when it is returned from rental. In both cases, a callback from a GWT widget results in a call to the IRentalServicesAsync GWT service interface.



As the widgets we are building are simply components, we can evolve and reuse them to good effect. For instance, we might want to keep the GUI structure (the widgets) separate from the business logic. In that case, we could have defined the CarRentalWidget without the RentalLogic part and used evolution or resemblance to add in the logic parts later.

Putting the User Interface Together

Navigate to the car rental widget package. Here we connect up the widgets to create the final GUI as shown in figure 7.17. Note that we use a CarRentalWidgetFactory part so that a separate CarRentalWidget part can be created per car. Finally, we connect up a GUILogic part which has a small amount of logic to control the population of these widgets.

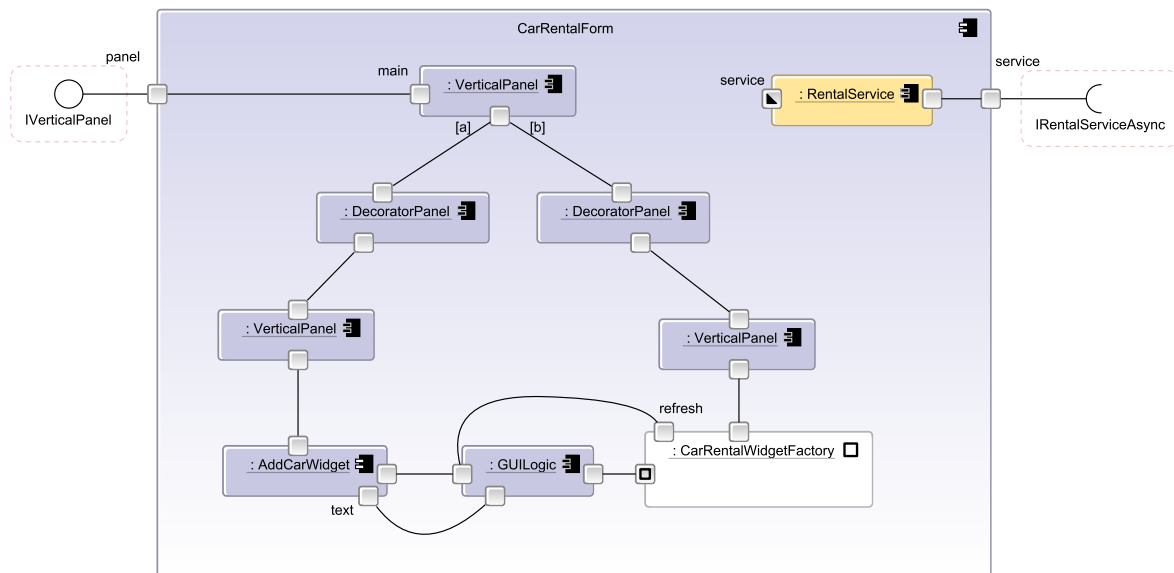


Figure 7.17: Bringing the full UI together

Making the Service Available

The RentalService leaf is simply a shell component with a hyperport, designed to make the IRentalServiceAsync GWT-provided service available to the entire GUI structure. Look at RentalService.java to see how trivial this is.

Generating the Full Implementation

We cannot run the GWT system inside Evolve - it must be run in hosted mode as provided by the GWT toolkit. To allow this, we use `Backbone>Generate full implementation` to generate the `CarRentalFormFactory.java` class. We then add the instantiation of our factory to the GWT-created module class `CarRentalGUI`, as below. Note the `setService()` line where we explicitly set the required service interface of the form to be that of the GWT server.

```
public class CarRentalGUI implements EntryPoint
{
    public void onModuleLoad()
    {
        CarRentalFormFactory form = new CarRentalFormFactory();
        form.setService((IRentalServiceAsync)
            GWT.create(IRentalService.class));
        form.initialize(null, null);

        RootPanel.get("application").add(form.getPanel_Provided());
    }
}
```

That completes the integration with GWT for the front-end.



We have used Evolve to model every part of our GUI, including the fine-grained widgets. Sometimes, however, it is more pragmatic to use a combination of Evolve components in conjunction with Java classes for certain domains. To use this approach, create less fine-grained components and implement the internal logic using conventional Java classes.



To use state machines in a GWT front-end, set the stratum's `generation-profile` stereotype property to `gwtclient`. This will use a generator to replace the standard state dispatcher component that uses reflection. GWT doesn't like reflection.

7.3.3 Quick Turnaround

Because GWT translates the generated client-side Java code directly into Javascript without requiring it to be compiled, turnarounds to changes in the GUI are extremely fast. i.e. make a change to the GUI components, regenerate the full implementation, and press `F5` in the web browser to refresh the view.

Using this approach we find that the turnaround time for a change is often under 5 seconds.

Chapter 8

Documenting Evolve Models

This chapter describes how to document Evolve models using a wordprocessor.

At a basic level, Evolve supports copying figures and pasting them into Word or OpenOffice. This uses EMF vector graphics, a compact and scalable format. The limitation of this approach, however, is that the copied images will not be automatically updated if the model changes. Because of this, using copy & paste is not recommended for creating substantial documents with many figures - each changed figure has to be manually updated in turn.

Evolve provides another approach, which is far superior. It allows figures to be marked up and exported in bulk from a model. We can then link to these images from a wordprocessor document and update them all when the model changes. This facility is how this document was produced, and also how we wrote a several-hundred page research document with hundreds of figures. Evolve makes creating documentation in this way a relatively painless affair¹.

8.1 Marking Up Figures and Exporting Them As Images

Evolve provides a way to mark up figures to be exported from a model.

To create an export, choose the “Grouper” tool and place a grouper on the diagram. Select it and type “export imagename.extension”, where “extension” can be any of: “eps” (Postscript), “jpg”, “png”, “gif” or “rtf” (Rich Text Format with embedded EMF). If you want to export with delta marks on, use “export **del**tas imagename.extension”. A model can have any number of groupers.

Then, drag a set of figures into the grouper. An example of this is shown in figure 8.1.

¹We collectively shudder when we think of the pain we have suffered through trying to use Rational SoDA for documentation. Evolve provides a far better way.

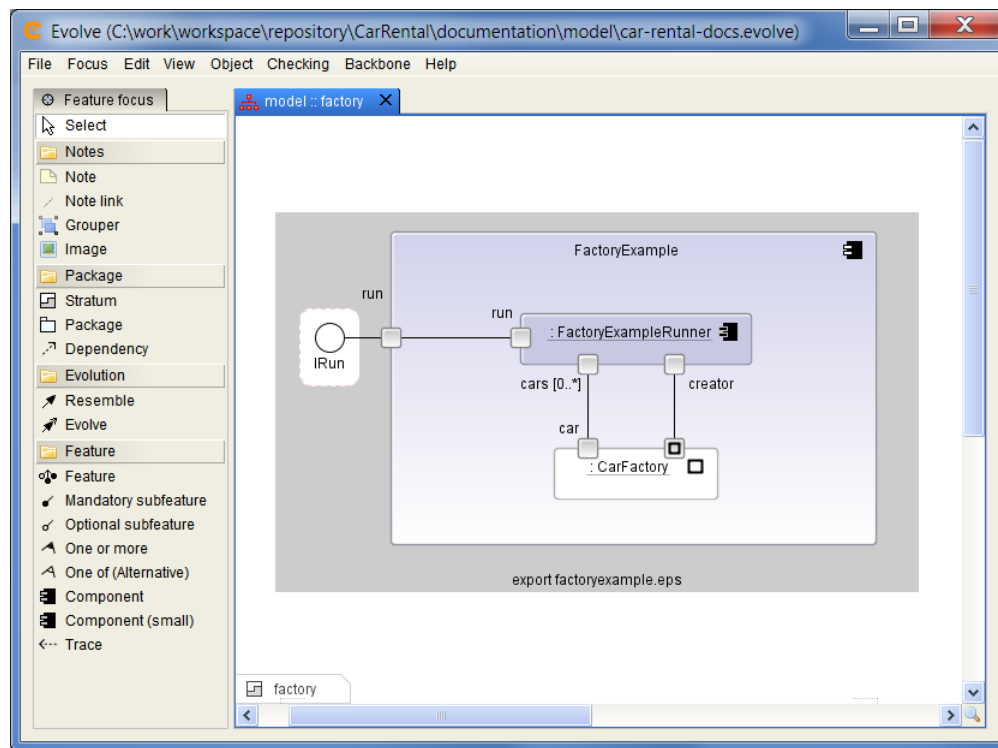


Figure 8.1: A grouper which will export a figure

(Note that we have colored the background of the grouper to make it obvious. Normally the background is transparent.)

To export all the images for the entire model, invoke the **File>Export grouped images** menu option and choose a directory to save into. Every image is saved into the chosen directory, so ensure the image names in the groupers are unique.

That's all there is to it. We will now discuss how to use these images using three different wordprocessors.

8.2 Linking to Images in a Wordprocessor

Any wordprocessor that allows the insertion of linked files can utilize this approach. We have had fantastic results with Microsoft Word and Lyx. OpenOffice Writer can also be used but its images are sometimes cropped and the process of linking is cumbersome.

8.2.1 Using Word

(We describe below how to link the images using Word 2007. It's the version we use, although we are fairly sure older versions will work also.)

The best approach when using Word is to export RTF images (using the “rtf” extension). These contain EMF vector graphics inside, which scale properly when resized.

Suppose that we have exported an image to `c:\images\figure.rtf`. To add this to a Word document, choose the “Insert” tab on the ribbon and invoke the **Object>Text from file...** option. Select the RTF file using the dialog and then change the insert mode to “Insert as link”. The linking part is crucial - if we do not do this we will simply insert a static picture into the document.

Updating Linked Images

To update the links after figures have been re-exported from Evolve, invoke the **Prepare>Edit Links to Files...** menu option. This can be accessed from the large Office icon in the top left of the ribbon. Select all the links and click on “Update Now” and the images will be updated, bringing the document in line with the model.

8.2.2 Using Lyx

Lyx (<http://www.lyx.org>) is an advanced wordprocessor that uses L^AT_EX under the covers. Don’t be put off by the L^AT_EX part - Lyx shields the user from this, and the documents produced look fantastic. I’d encourage you to try Lyx at least once, as it is a superb tool for creating documentation. It is our favorite by a long way.

For Lyx, make sure you export EPS “eps” images. These are postscript, and scale properly when resized. To insert an image into Lyx once it has been exported, choose the **Insert>Graphics...** menu option. Browse for and select the EPS file and it will be inserted as a linked image by default.

Image updating happens automatically in Lyx, there is no need to manually invoke this. This makes for a very pleasant documenting experience - if you want a change to a figure while working on a chapter, switch over to Evolve and hit **control E** and voila, the document magically updates to reflect the model.

8.2.3 Using OpenOffice

We have had less success with OpenOffice. Although this technique can still be used, we would recommend using either Word or Lyx. Our example was tested with OpenOffice 3.2.

To insert a linked RTF image in a Writer document, invoke the **Insert>Object>OLE Object...** menu option. This will bring up a dialog allowing you to choose an Object type. Click on the “Create from file” option at the top of the dialog and then click the “Link to file” option. The last part is crucial or the image will not be updated later. Finally, choose the file using the “Search...” button and the linked image will be inserted.

Updating Linked Images

To update the images in the entire document after an image export, select the **Tools>Update>Update All** menu option.

8.3 A Few Tips

Although any exported images can always be regenerated from a model, it still makes sense to keep these under version control, in addition to the Evolve model that is being documented. This way other developers can also collaborate on the documentation.

If you want to send around a document without links, simply export it in PDF or XPS format.

Finally, a quick word about copy/pasting figures from Evolve into a presentation tool such as PowerPoint. For reasons we cannot fathom, this does not work directly. However, there is an easy workaround - copy the figures into Word/OpenOffice first and then select these and copy/paste from the wordprocessor into PowerPoint/Impress.

Chapter 9

Advanced Modeling in Evolve: A Teaser

As an introduction to Evolve, this document is not the right place to dive too deeply into advanced techniques. This chapter outlines some of these techniques briefly, but in enough detail for the adventurous reader to explore. A future advanced techniques manual will be more forthcoming on these subjects.

The Evolve approach and these techniques are described in great detail in my PhD thesis. My work builds on the influential Darwin model, augmenting it to allow the construction of highly extensible applications. In this chapter I will occasionally refer to this thesis.

Also described in the thesis is a formal logic specification of the underlying component approach. This states precisely how the resemblance and evolution constructs operate. The Backbone interpreter is a refinement of this specification.

The thesis is titled “A Rigorous, Architectural Approach to Extensible Applications”. It can be downloaded at the following location.

<http://intrinsarc.com/research>

(As an aside, my research work commenced after I had accumulated nearly 20 years of industrial software engineering experience in the financial, telecommunications and broadcasting domains. What I guess I’m trying to say is that the work describes a highly pragmatic marriage of the best parts of the industrial and scientific approaches).

9.1 Highly Extensible Systems

I have researched hierarchical component models and their applicability to highly extensible systems in some depth. In fact as you now know, my entire thesis was on this subject. Let me explain a bit about extensibility.

An extensible system is one that can be changed to meet new requirements, without requiring that the implementation code for that system be changed (or even seen).

The upshot is that a system created fully using Backbone components has an amazing property - they can always be extended to add new requirements, without destroying the original application, by utilizing resemblance and evolution. The original design can be amended using deltas, which are then applied to produce the customized system. A designer can easily produce and maintain many variants of an application in this way.

In other words, as you design a system in Evolve, extensibility is naturally woven in without any conscious effort on the part of the programmer. This is a great benefit over existing design approaches which often require huge amounts of pre-planning and forethought to accommodate possible future changes. In Evolve, decisions regarding future features can be delayed until very late, preventing much work and subsequent rework. You can think of Evolve as providing an agile approach to design - you do not need to think too hard about what is coming next, and can take comfort in the knowledge that you will be able to restructure a system to accommodate changes when they are actually needed.

Section 1.3 of the thesis outlines the requirements for an extensibility approach and shows how Backbone satisfies these. The limitations of this approach relate to the granularity of components which govern the possible work required in the worst case scenario if a component must be fully replaced in a system. Put simply: extensibility is always possible, but the amount of work to replace a given component will be dictated by how coarse-grained that component is. This is described in section 8.2 of the thesis. Thankfully, the hierarchical model supports fine-grained components well.

9.1.1 Evolve as a Way of Avoiding Framework Syndrome

A framework is a large-scale, reusable piece of software, which provides “plug-points” or “hooks” that a developer can register with to get callbacks. The motto of a framework is known as the Hollywood Principle: “Don’t call us, we’ll call you”.

Frameworks often require a huge amount of predictive development, and much rework to get the structure correct for reuse. A phrase often used is: “It’s not a framework until it has been reused three times or more”. I call the upfront work required for a framework, and the analysis paralysis that it causes, the “framework syndrome”.

The Evolve approach reduces much of this type of work. A library can be built using Evolve in confidence, knowing that it will be naturally extensible and users can easily customize it without further effort by the framework developers. The framework can then be evolved by further stratum, allowing both version 1.0 and 1.1 etc to coexist.

9.1.2 Sharing Parts of a Model in a Collaborative Environment

Evolve can be used by multiple parties to work on a single model in a collaborative but disconnected way. By “disconnected” I mean where the parties do not all use a common version control system. The version control-like nature of resemblance and evolution allow this sharing and collaboration.

This is explained in section 5.2.12 of the thesis. Basically, a set of common strata are distributed in read only form to people who then extend the system by building their own “extension” strata on top of this. Each extension stratum has a single owner who has write permission. The extensions can then be combined by importing them all into a single model.

When bringing these strata into a single model (effectively a merge), we can get conflicts due to overlapping evolutions of components in the common strata. This can be dealt with using a further stratum and subsequent evolution. In other words, the branches created by the extension strata can be merged using evolution without destroying the extension strata definitions. This is shown in 3.3.7 of the thesis.

9.1.3 Evolve as a Plugin System

You may have been reading the above description of disconnected collaboration and thought: “That sounds like it would make a powerful alternative to a plugin architecture”.

In section 7.1 of the thesis, I compare and contrast Backbone to a plugin approach. i.e. Can Backbone be used in place of a plugin architecture? What are the advantages and disadvantages? The upshot is that Backbone can replace plugin systems and it ameliorates many of their deficiencies¹.

In section 7.1.2 of the thesis I provide a critique of the Eclipse plugin model and show how in certain circumstances extending a plugin-based system can go very badly wrong. Plugin architectures generally require far too much forethought in planning for extension, which Backbone avoids through the provision of the evolution construct.

9.2 The Interaction Between Resemblance and Evolution

In section 5.8 I mentioned that resemblance and evolution interact. This is described formally in section 4.2.3 of the thesis. Thankfully, it is very intuitive so it can also be described easily here.

Consider if we have the system shown in figure 9.1. We have stratum Z at the base, defining component C which resembles B which resembles A. Strata X and Y build on Z and both evolve B. Stratum W depends on Y and also evolves B. Finally stratum V depends on X and W, but does not define an evolution. You might have already noticed that this parallel evolution scenario is really a bit like two branches in a version control system. V is like a merge of the branches which must rectify any conflicts.

So, the question we really want answered is:

“What does the resemblance graph look like from each stratum perspective?”

This is a simple for the Z perspective. For any other perspective, we construct what is called an “expanded resemblance” graph. This factors evolutions into the resemblance graph, allowing us to see exactly how it interacts with resemblance.

¹Unfortunately, Backbone does not currently provide a convenient way to distribute application subsets to end users. This will be rectified in a future version.

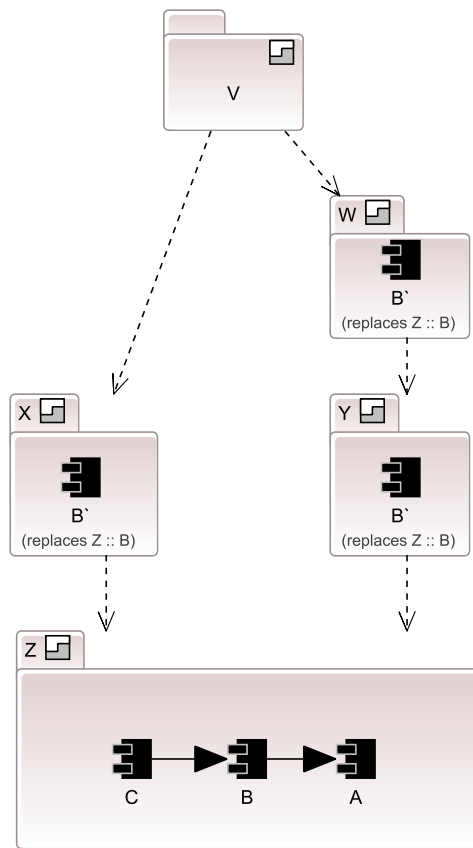


Figure 9.1: Branched evolution and a subsequent merge

Figure 9.2 shows the graph from these five perspectives. You can see that an evolution is always inserted just after the component it evolves. For instance, from perspective Y, we have C resembles B' (from Y) which resembles B which resembles A. Stratum W can already see Y's evolution through its dependencies, so its evolution is inserted after this.

From the perspective of V, we see that the two stratum branches have resulted in C multiply resembling the two divergent evolutions of B! In other words, C, from V's perspective, resembles the merged result of the evolutions.

Consider that the two branches may have made incompatible changes. One side may have deleted a part that the other side depends on, for instance. When these are merged together (in the same way that multiple resemblance is handled normally), the end result can be in error. The rule is that if a merge results in an error, a subsequent evolution in the stratum causing the error is required to correct this. This can always be achieved using the delete/add/replace deltas available with evolution. In our example, stratum V merges the two branches, and would be the place where any errors would need to be corrected.

So, in summary, evolution and resemblance combine to form an expanded resemblance graph which can change from each perspective. Evolution can result in fundamental changes to this graph, inserting definitions in between existing elements defined much lower down. This is what gives evolution much of its power - the ability to affect existing elements and customize their resemblance

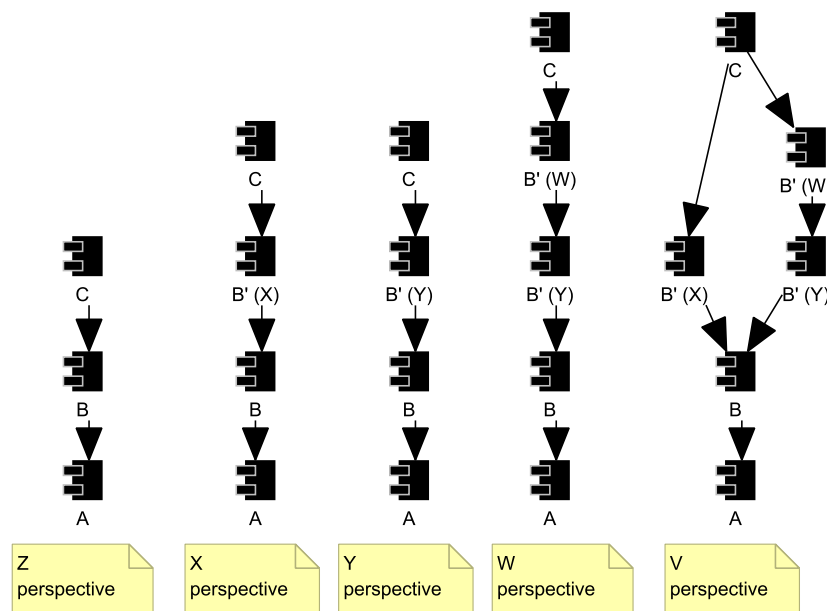


Figure 9.2: The expanded resemblance graph from each perspective

hierarchies. Knowledge of the expanded resemblance graph concepts allows a designer to effect profound changes to an application.

In section 3.3 of the thesis, an example is presented that works through the above rules. It shows how deep changes to a model can result from evolutions, and how merge conflicts can be corrected even when the errors are deep within a compositional hierarchy.

9.3 Extensible Feature Modeling

As you used Evolve, you may have also been thinking: “This is great for design and implementation, but how do I model requirements?”. The short answer is that you model them using requirement features.

A feature is just that - a feature of an application. Features can be broken down into mandatory and optional subfeatures, and certain features can exclude other features. This is a powerful way to model requirements. For more reading on the subject, have a look at the following link.

http://en.wikipedia.org/wiki/Feature_model

Jeff De Luca and Peter Coad (of Togethersoftware fame) were some of the first people to build a development process around features. Features form the unit of estimation and iterations are planned around feature lists. Feature driven development is described in some detail in the following link.

http://en.wikipedia.org/wiki/Feature_Driven_Development

Evolve fully supports these hierarchical feature models, but with an extensibility twist. Yes, you've guessed it - we can use resemblance and evolution to restructure the feature set of an application.

To try out the feature modeling facilities of Evolve, change to the feature focus using the **Focus>Feature focus** menu option. Break a feature down into subfeatures using the “mandatory”, “optional”, “one or more” or “one of” links. To indicate that a component implements a particular feature, draw a trace link from the component to that feature. This is shown in figure 9.3.

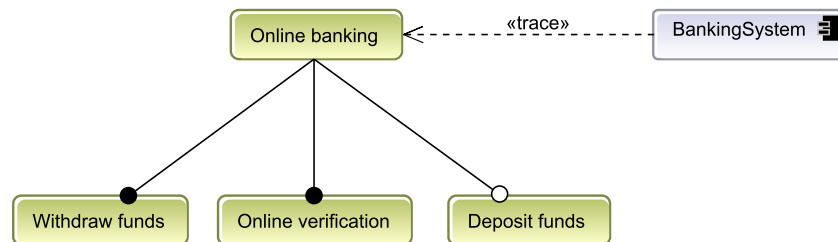


Figure 9.3: Feature modeling in Evolve

To see what features are supported by components, open the feature hierarchy browser by right-clicking on a feature and choose the **Show feature hierarchy** option. This is shown in figure 9.4.

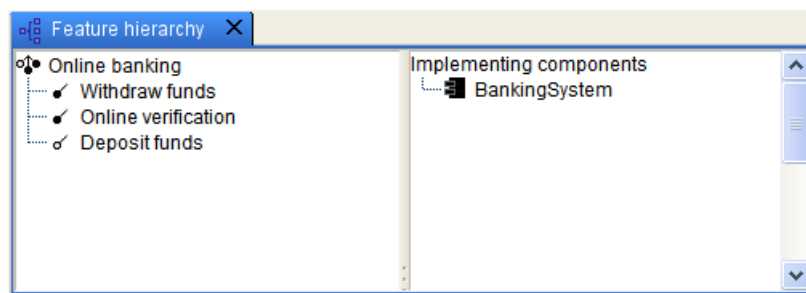


Figure 9.4: The feature hierarchy browser

When evolving a feature, the subfeature links can be replaced or deleted. When evolving a component, the trace links can also be replaced or deleted.

9.4 Reengineering a Legacy Application

Evolve can be applied incrementally to a legacy application, giving the benefits of the approach in an iterative fashion. The overall architecture of a system can be moved quickly to an initial coarse-grained, interface-centric design, and connected up as a composite in Evolve. Over time these coarse components can be refined organically, as and when the system needs changing or extension in specific areas.

In section 7.4 of the thesis I describe how I incrementally reengineered a very mature system with over 200 classes (16kloc), so that it could be extended using Evolve.

9.5 Evolution Instead of Aspects

I have already touched on this subject in section 6.2.3. Evolution can be used to replace aspects, with very few of their disadvantages such as lexical fragility and the tension between reuse and method context of an aspect.

For the readers who would like to dig deeper, I would suggest the following literature.

1. The book “Aspect-Oriented Programming With the e Verification Language” by David Robinson.
The e language uses a form of evolution to model several aspect-like situations, and shows how the approach can offer advantages over aspects.
2. The academic paper “Aspect-Oriented Programming is Quantification and Obliviousness” by Robert Filman and Dan Friedman.
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.28.287>
This paper shows that there is a tension in aspects between the quantification side and making aspects know too much about the context of the underlying code they will apply to. Have you ever wondered why most (all?) AOP examples add logging and transactions to a system? Read this paper for insight as to why aspects tend not to address business logic.

Evolution is perhaps not as straightforward as aspects for pure cross-cutting concerns that do not require an application-level context, such as tracing or logging. For extending an application, and incorporation of new logic however, I contend that evolution is far more powerful and robust.

In the Advanced Techniques manual I will outline a much larger example to show this in detail.

9.6 Product Lines

Much research has gone into the idea of product lines, which are basically a way of producing multiple variants of an application with slightly different feature sets. Evolve compares well against such approaches, providing a number of advantages, and I review it against one such system in section 7.2 of the thesis.

Evolve allows multiple variants of an application to be created in a sensible manner, just like a product line system. It is not uncommon to find a single Evolve model with ten or more permutations of an application. Furthermore, Evolve can use feature modeling which came from research into product lines.

9.7 Controlling Component Complexity Through Visual Locking

When I reengineered a legacy system, I came across one component with around 30 parts (!) which was very difficult to work with graphically. It became apparent that sometimes hierarchy is not enough to fully control visual complexity. It is also sometimes necessary to be able to work

on multiple “fixed” views of a single component, where each view shows a subset of the insides. Visual locking provides this.

To try this, create a new component called “Component”. Create another view of it using the technique described in section 2.5. Then visually lock each view by selecting each in turn, right-clicking and ticking the “Visual lock” menu option. Each is now locked and only constituents added directly via that view will be shown. Normally each view will show every part, port, attribute and connector available to that component unless explicitly hidden. Locking prevents this and allows us to present a component from multiple viewpoints.

Figure 9.5 contains two visually locked views of the same component showing different constituents. The ellipsis at the bottom corner indicates that there are more constituents that are not being shown.



Figure 9.5: Two visually locked views of the same component

9.8 Strata as Modules

A stratum can be used as a module. Through nested strata, a stratum X can control which of its components and interfaces can be accessed by those depending on X. In other words, it can selectively export definitions in nested strata to the wider world.

The rules around nested strata and visibility are described in section 4.2.2 of the thesis. If you are feeling brave, also have a look at the formal description in section 4.2.1.

9.9 Extending State Machines

State machines can be extended using resemblance and evolution. The state machine apparatus in Evolve has been designed to allow extension along a number of dimensions: states, transitions, and events. This is described more fully in section 6.3.3 of the thesis. That section also discusses chaining and nesting of state machines, and offers a more detailed critique of the State design pattern.

Chapter 10

Product Roadmap and Further Reading

In this brief chapter, we outline the Evolve roadmap we are planning for the next year or so. We also present further reading for curious developers who want to know more about the background of hierarchical components and how they relate to dependency injection and other approaches.

10.1 The Evolve Roadmap

10.1.1 Team System

The team version of Evolve is currently under development. This allows multiple developers to collaborate on a single model, avoiding the overhead of exporting and importing stratum.

10.1.2 Support for Other Implementation Languages

We plan to support C# fully. We are also looking into the feasibility of providing C++ support.

10.1.3 Concurrency Constructs

Concurrency in hierarchical component models has a long and distinguished academic research history.

A set of constructs will be added to Evolve and Backbone, allowing concurrency to be described and controlled by properties in the component design. Orthogonal/concurrent states will also be supported.

10.1.4 Web Publishing System

A full web publishing system is being developed for Evolve to allow a model to be dynamically available via a HTTP server. This will be available sometime early 2011, and will build on the team system.

10.1.5 Protocol Analysis

Work has started on adding protocol and behavioral analysis to Evolve. A method-level sequence diagram (with UML2 operators) can be constructed for each port of a leaf, and high level goals can be defined. The analyzer can then determine if the goals can be achieved using the existing protocols. The protocols of any composite component port are determined automatically.

This is useful because it allows you to see how your system will respond to certain situations. You can verify its behaviour by constructing inputs and looking at the output sequence diagrams.

10.1.6 Internet Distribution System and Licensing Manager

Although Evolve has many features which allow it to operate as an effective plugin system, it currently provides no convenient way to distribute strata and associated jars to end users. An internet-based distribution system will be added, along with a license manager to control enabling of various features. This will tie into the feature modeling facility already in Evolve.

10.2 Further Reading

For the interested reader, I would like to recommend the following books and articles.

Software Architectures for Product Families

Mehdi Jazayeri, Alexander Ran, Frank van der Linden

This book was published in 2000 and was how I first found out about Darwin and Koala. The Darwin article by Jeff Magee and Jeff Kramer is excellent and very clear.

The Rich Engineering Heritage Behind Dependency Injection

Andrew McVeigh

<http://www.javalobby.org/articles/di-heritage/>

In this online article, I look at how component systems offer a richer form of dependency injection.

Real-Time Object-Oriented Modeling

Bran Selic, Garth Gullekson, Paul T. Ward

This book describes the approach behind the amazing ObjecTime system, which eventually became RationalRose Realtime. It was literally 15 years ahead of its time. This was the first system to offer structural inheritance, and the effects of the concepts introduced are still being felt today. Concurrency via actors was fully supported. Bran was one of the leads on the UML2 specification and many of the component and state machine concepts introduced into UML came from here.

Aspect-Oriented Programming with the e Verification Language

David Robinson

This book looks at the e language, which has a similar evolution construct to Evolve, but working at a method rather than structural level. It looks at how this construct makes AOP-like crosscutting possible.

Aspect-Oriented Programming is Quantification and Obliviousness

Robert E. Filman , Daniel P. Friedman

<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.28.287>

This paper describes what makes up AOP, and gives insight into why it seems inordinately focused on simple and non-domain-based cross-cutting concerns such as logging and tracing.

Design Principles and Design Patterns

Robert “Uncle Bob” Martin

http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

Where would we be without Uncle Bob to tell us how to structure our object-oriented systems all those years ago? Timeless principles that still apply.