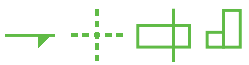
Tensegrity Software
the interface architect



Graph Framework Getting Started

Getting Started

First Steps with the Tensegrity Graph API

Getting Started: First Steps with the Tensegrity Graph API

Copyright © 2004, 2005 Tensegrity-Software GmbH

Abstract

The *Getting Started Tutorial* is composed of a number of smaller tutorials, each of which is designed to show the *Java Application Programmer* how to accomplish one or more specific programming tasks using the Tensegrity Graph Framework. This framework allows you, among other things, to associate "entities" with one another and graphically render all of their interrelationships.

Using, copying, modifying or distributing this document is not permitted without permission by the copyright holder.

Table of Contents

Preface ... vi
1. Introduction ... 1
2. Installation ... 4
2.1. Installation ... 4
2.2. Configuration ... 4
3. Hello Node ... 7
3.1. Tutorial At A Glance ... 7
3.2. Running The Tutorial Example ... 7
3.3. Graph *Creation* ... 8
3.4. Node and VisualNode *Creation* ... 10
3.5. Node and VisualNode *Insertion* ... 11
3.6. Edge and VisualEdge *Creation* ... 11
3.7. Edge and VisualEdge *Insertion* ... 12
3.8. Summary ... 12
4. MVC ... 14
4.1. Tutorial At A Glance ... 14
4.2. Running The Tutorial Example ... 14
4.3. MVC in the *Tensegrity Graph API* ... 15
4.4. Graph Controllers ... 17
4.5. GraphController *Creation* ... 17
4.6. GraphController *Association* ... 17
4.7. Visual Element *Creation* ... 18
4.8. Summary ... 19
5. Repository ... 21
5.1. Tutorial At A Glance ... 21
5.2. Running The Tutorial Example ... 21
5.3. Geometry *Definition* ... 22
5.4. Geometry XML ... 24
5.5. Geometry Pool ... 26
5.6. Geometry *Association* ... 27
5.7. Style *Definition* ... 27
5.8. Style Pool ... 28
5.9. Style *Association* ... 29
5.10. Repository Element *Definition* ... 29
5.11. Element Pool ... 30
5.12. Summary ... 30
6. Modeling Rules ... 31
6.1. Tutorial At A Glance ... 31
6.2. Running The Tutorial Example ... 31
6.3. Graph Rule *Definition* ... 32
6.4. NodeRule *Definition* ... 33
6.5. EdgeRule *Definition* ... 34
6.6. RuleRegistry *Population* ... 35
6.7. Rule *Association* ... 35
6.8. Summary ... 36
7. Layout ... 37
7.1. Tutorial At A Glance ... 37
7.2. Running The Tutorial Example ... 37
7.3. LayoutController ... 38
7.4. Layout Context (Type Configuration) ... 39
7.5. Summary ... 40
8. Event Handling ... 41
8.1. Tutorial At A Glance ... 41

8.2. Running The Tutorial Example 41
8.3. Event *Registration* 42
8.4. Event *Veto* 43
8.5. “Big Brother” *Registration* 45
8.6. Event *Logging* 46
8.7. Summary 46
9. Skeleton 47
9.1. Tutorial At A Glance 47
9.2. Running The Tutorial Example 48
9.3. Application Frame 49
9.4. Commands 51
9.5. Preferences 52
9.6. Launch Tasks 53
9.7. Modeling Our Example 57
9.8. Summary 58
Bibliography 60
Index 61

Preface

In the hectic world of software development, it often seems that nothing ever stays the same. New languages appear, APIs change, code gets rewritten and the techniques for writing most software solutions undergo change all the time.

This tutorial and the code that goes with it is no exception. We expect this framework to change and improve in time. Better abstraction, fewer couplings and more precise naming and documentation are just a few of the changes you should expect.

We read and create a lot of framework and project code. We write and review just as much documentation, including javadocs, user manuals, technical manuals, requirements specifications, and so on.

If something you encounter here or elsewhere makes no sense to you, or you feel it just doesn't belong where you found it, we would appreciate your comments in the form of an email written to:

`<documentation@tensw.com>`.

Chapter 1. Introduction

The *Getting Started Tutorial* is composed of a number of smaller tutorials, each of which is designed to show the *Java Application Programmer* how to accomplish one or more specific programming tasks using the Tensegrity Graph Framework. This framework allows you, among other things, to associate "entities" with one another and graphically render all of their interrelationships.

In order to make the examples more intuitive, applicable and enjoyable, we have chosen to model user and functional rights administration. This we presume is a typical subsystem of any business software application. In this manner, the business workflow should be well-known and intuitive, so that you may concentrate on our framework design and APIs instead of the underlying business case our framework is going to model and graphically render.

In a nutshell, the entities and relationships found in the business case are as follows. They will be randomly found in the subsequent tutorials in this manual.

1. **User**:

   Literally that thing which "uses." A person is usually referred to as a user when he or she has a private password and is assigned at least one role in an application context. A user does not represent any named and persistent functional role but simply describes a person who has the ability to log into and use a particular software application. In a more abstract context, a "user" may refer to any runtime object that "uses" another resource.

2. **Group**:

   A set of users forms a group. In systems that have many users in them, groups are easier to handle when assigning roles to large numbers of equally-capable individuals.

3. **Role**:

   An abstraction which encapsulates a set of application rights. Roles are used to group these rights into named objects which can be subsequently assigned to users and groups of users in an application context.

4. **Right**:

   Sometimes called a "functional right." Rights are persistent objects that encapsulate the capability to do something with a named resource. Rights form the core of authorization services and are typically bundled by roles.

5. **Permission**:

   A permission is not the same thing as a right. A permission is created when a user is assigned to a particular right, thereby granting him/her access to the action defined in that right.

6. **Resource**:

   This can mean anything and everything. Resources usually describe things that live in the application domain. Complete files or parts of them, database records and runtime objects are three of many types of resources that users may interact with in an application context.

7. **Action**:

   This defines what a user can actually *do* with a resource. An action includes a verb and the resource or class of resources that are involved. It might help to think of actions as particular class methods, such as constructors (create), destructors (delete), getter methods (read), setter methods (write) and

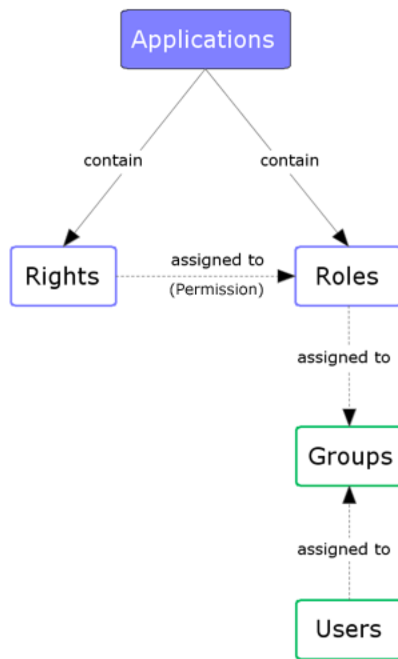
so on. Usually a subset of the classes of an application domain, also known as “course-grained” entities, are used to define actions for use in authorization schemes.

8. **Application**:

   There is never just **one** application in a company. Different applications offer access to different resources and as such provide a unique set of user roles. Sometimes applications are further broken down into subsystems, services or “modules,” which provide fine-grained control in the administrative process.

The following domain diagram illustrates the fundamental entities of the introduced business case.

Figure 1.1. Example Business Case (Conceptual Diagram)



In the conceptual diagram above, you can see how most of the aforementioned domain entities statically relate to each other. In most cases, the identified domain objects represent concrete types that have persistent attributes and/or behavioral responsibilities. For this reason, their names are bounded by rectangular boxes to signify that they are types in the domain model and/or classes in a subsequent design model.

In at least one case, however, an entity from our domain model represents a unique association between two concrete types. For this reason, that entity type (Permissions) is modeled as a line connecting the two entities that are involved in the special relationship.

Note

In the context of the Tensegrity Graph Framework and the business entities above, a *Node* represents a class and an *Edge* represents a relationship from one of those classes to another class. In other domain languages, an *Edge* might be referred to as an arc, a link, an association, a transition or something else altogether. Additionally, graph theory uses the term “vertex” to designate what we call a *Node*.

It is important to remember that both nodes and edges belong to the graph domain and that the Tensegrity Graph Framework allows attributes to be associated with them. These attributes can include names, states and anything else required for the business entities being modeled.

Chapter 2. Installation

This chapter will guide you through setting up the Eclipse platform, a universal IDE with excellent Java development support. It is not absolutely necessary that you use Eclipse to understand this tutorial. It makes using it, however, much more convenient.

2.1. Installation

You are required to copy numerous files to a local directory on your computer. In this and the following section, we show you exactly what you need to copy, where you should copy to and how to configure your development environment to be able to compile and run the tutorial examples. We do this in a step-by-step fashion. If you follow these instructions exactly, we hope that you will avoid any complications whatsoever.

The *First Steps Tutorial* is composed of the following artifacts:

- Tensegrity Framework Jar Files (Java Archives)
- Tensegrity Framework Java Documentation (Javadoc)
- Tensegrity First Steps Tutorial (This document)
- Tensegrity First Steps Source Code (Zipfile)

Note

While working with the Tensegrity Graph Framework you will notice that a special care has been taken to remove from the core functionalities any dependency on the windowing toolkit being used. The Tensegrity Graph Framework currently provides a wide range of AWT/Swing-based GUI components supporting its core functionalities, but also a set of GUI components based upon SWT (Standard Widget Toolkit) for integration with the Eclipse platform. At this point in time all the following tutorials except the “Skeleton”-based one are illustrated by both AWT/Swing-based and SWT-based samples.

2.2. Configuration

2.2.1. Eclipse

This section explains how to install the distributed files so that you are able to compile and run the example tutorials from within Eclipse. We assume that you have successfully installed and started your Eclipse environment at this point in time.

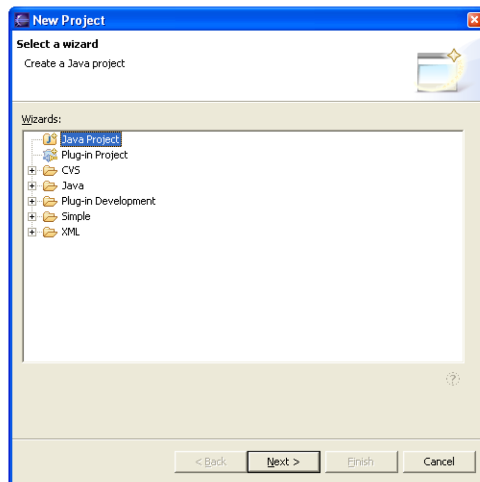
Depending on your version of Eclipse, some of the following steps will be different. In general, however, the steps are as follows:

1. *Install* JAR Files
2. *Install* Javadocs
3. *Create* Eclipse Project

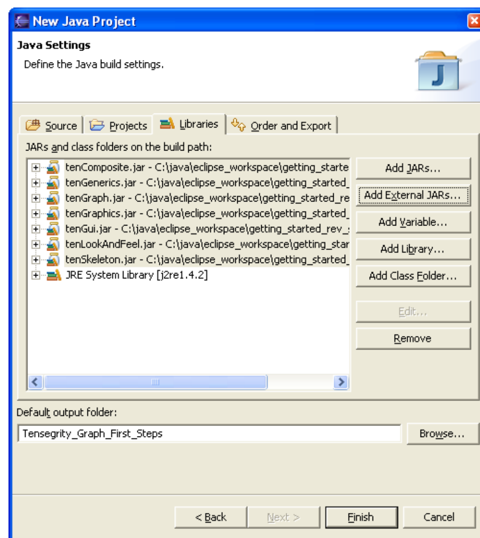
Each tutorial in this manual is located in its own directory that contains an executable file. This will be a java class with an implemented main method.

2.2.1.1. Configuration for Eclipse Version 3.X

1. Select the main menu “File”.
2. Select the child menu item “New”.
3. Select the child menu item “Project...”.
4. A Wizard dialog window appears.



5. Select “Java Project” from the Wizard list.
6. Press the “Next” button.
7. Enter “Tensegrity_Graph_First_Steps” without quotes into the “Project name:” text field.
8. Press the “Next” button.
9. Select the “Libraries” tab.
10. Press the “Add External JARs...” button.
11. Navigate to the Tensegrity libraries and select them as well as the SWT library (lib/swt/SWT3_0.jar) while holding down the **STRG** key.



12. Press the “Open” button to confirm and close the “JAR Selection” dialog. If you select the wrong JARs by mistake, you can always delete them later.

13. Append “/bin” without quotes to the “Default output folder:” text field.

14. Press the “Finish” button.

Chapter 3. Hello Node

This tutorial provides *Java Application Programmers* with their first exposure to the Tensegrity Graph Framework. It is designed with simplicity in mind, so that you may quickly learn about the basic features of the framework as well as the most important terms in the software framework domain space.

We therefore recommend that you read this tutorial and run the tutorial example before moving on to any other tutorials in this manual.

3.1. Tutorial At A Glance

The table below gives you some important information about this tutorial.

Table 3.1. Tutorial Aspects

Tutorial Aspect	Tutorial Description
Approximate Duration	20 Minutes
Expected Outcome	An application displaying a view with two nodes connected to each each other. This example gives the *Java Application Programmer* an initial exposure to the following classes: • `GraphModelFactory` • `GraphViewFactory` • `GraphObjectContainer` • `VisualGraphObjectContainer` • `Graph` • `VisualGraphView` • `Node` • `VisualNode` • `Edge` • `VisualEdge`
Source Files	The tutorial examples come with two source packages containing the java and configuration files needed to compile and run them. These are “com.tensegrity.firststeps.hellonode” and “com.tensegrity.firststeps.swt.hellonode” respectively for the AWT/Swing-based and SWT-based examples.
Creating Files	Not applicable in this tutorial
Modifying Files	Not applicable in this tutorial

3.2. Running The Tutorial Example

To run the tutorial example, please do the following in Eclipse:

1. Select package “`com.tensegrity.firststeps.hellonode`” or “`com.tensegrity.firststeps.swt.hellonode`” with the left mouse button.
2. Select the menu “Run”.
3. Select the child menu “run as”.

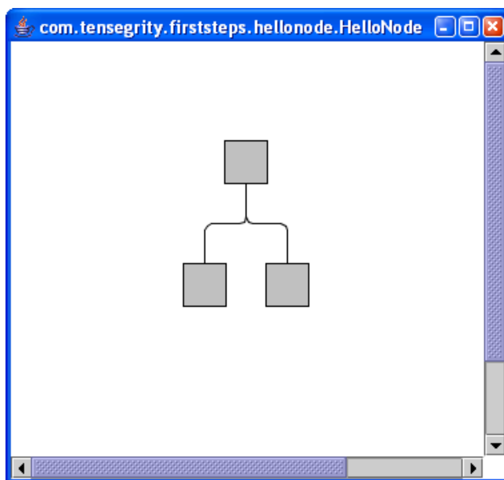
4. Select “Java Application”
5. The main method from the `HelloNode.java` compiled class will be executed.
6. Close the Java Application window when you have finished viewing the tutorial example.

Note

When running the SWT-based example described above, you will most likely encounter an error of this type: java.lang.UnsatisfiedLinkError: no swt-win32-3062 in java.library.path. In order to successfully run the exampe, file `swt-win32-3062.dll` (which can be found beside `SWT3_0.jar` under `lib/swt`) must be located on your classpath. An easy way to achieve this is by doing the following:

1. Select the menu “Run”.
2. Select “Run ...”.
3. Ensure that the correct example is selected on the list on the left of the dialog.
4. Select the “Arguments” tab.
5. Enter the following on the “VW Arguments:” text area: “-Djava.library.path=XXX\workspace\Tensegrity_Graph_First_Steps\lib\swt” where “XXX” is the path to your eclipse installation folder.
6. Press “Run”.
7. Close the Java Application window when you have finished viewing the tutorial example.

Figure 3.1. Screenshot HelloNode



3.3. Graph *Creation*

In this section you will learn how to create an empty Graph and a VisualGraphView instance. Together these classes lay the basic foundations of the Tensegrity Graph Framework.

3.3.1. Graph Concepts

A graph is a container entity consisting of nodes and edges. A node (or vertex) is part of a set of vertices of the graph. Additionally, a graph consists of a set of edges, which describe how the nodes (or vertices) are connected to each other.

A graph is formally defined as:

- A set of vertices (nodes [1]) *V*.
- A set of edges *E*.
- Each edge is a pair (*V1,V2*), where *V1* and *V2* denote distinct vertices that are both elements of *V*.

All elements of *V* and *E* must be unique as required by sets in the mathematical sense.

The Graph class in the *Tensegrity Graph API* represents this mathematical data and is often referred to as the "model." This non-visual component can be used on its own for applying graph algorithms and deriving specialized graph classes. Most notably, we distinguish between directed and undirected graphs. In a directed graph, all edges have a direction or orientation, while an undirected graph has edges with no orientation. Both directed and undirected graphs are represented by classes which implement the Graph interface.

Example 3.1. Creating a Graph instance

```
// get an instance of the factory
graphModelFactory= GraphModelFactory.newInstance();

// instantiate a graph object
graphModel= graphModelFactory.newGraph();
```

In the example above, it is important to note that a new Graph instance is created through a method call to a factory object. This technique decouples clients from code that decides which component class should be instantiated, providing a higher level of indirection, reuse and maintainability. Also note that exception handling has been ignored as in most examples in this tutorial. Under normal programming circumstances, exception handling must be part of your client code.

3.3.2. VisualGraphView Concepts

As the name implies, a VisualGraphView defines a "view" for a particular Graph instance to which it is always connected. In other words, a VisualGraphView is always coupled to a model which contains the connectivity and non-visual attribute data.

The VisualGraphView is also a container (see base class VisualGraphObjectContainer) and consists of the visual counterparts contained in a Graph. As a result, it manages instances of classes VisualNode and VisualEdge, each of which visualizes a Node and an Edge from the Graph model respectively. Although these visual elements exist, it is not always necessary to create them explicitly. Depending on the type of GraphConroller you are using (see MVC [14] chapter for more details), visual elements may be created automatically for you in a view when adding elements to a con-

[1] In the *Tensegrity Graph API*, a vertex is consistently called a "node" and a graph is usually referred to as the "model".

nected `Graph` model.

Note

A *VisualGraphView* contains additional data which is required for its representation in a view, including coordinate data, drawing style and formats. Remember, a *Graph* simply manages the node and edge instances without understanding how these elements should be rendered and positioned in dimensional space.

Example 3.2. Creating a `VisualGraphView` instance

```
// Get a factory instance
GraphViewFactory viewfactory= GraphViewFactory.newInstance();

// Create a new visual graph view using a graph controller
graphView= viewfactory.newVisualGraphView(controller);
graphView.enableUndoRedo();

// Add the view to managed list of views
controller.addVisualGraph(graphView);
```

In the example above, it is important to note that a new `VisualGraphView` instance is created through a method call to a factory object. This technique is used when creating most if not all of the graph elements (model, view and controller) in this framework.

Also, it is important to note that a new `VisualGraphView` instance requires a reference to a `GraphController` which must be advised to manage the newly created view after instantiation. `GraphController` classes are discussed in more detail in the chapter entitled MVC [14].

3.4. Node and VisualNode *Creation*

In this section, you will learn how to create two `Node` instances and two `VisualNode` instances. Together these classes lay more foundations of the Tensegrity Graph Framework.

3.4.1. Node *Creation*

We've already explained what nodes are. In the *Tensegrity Graph API*, the `Node` interface represents the functionality of any potential vertex in a graph. We call this vertex a “node” because the former tends to confuse fewer non-technical people who come into contact with graph theory and our framework.

You will eventually want to view the javadocs for more detailed information about creating nodes. There are several overloaded methods which allow more precise control over the creational process. Below is one of the utility methods that allows you to delegate some creational information to the factory.

Example 3.3. Creating two `Node` instances

```
// create three nodes with one port each
GraphModelFactory modelFactory = GraphModelFactory.newInstance();
nodeA= GraphModelFactory.makeDefaultNode(modelFactory, "a");
nodeB= GraphModelFactory.makeDefaultNode(modelFactory, "b");
nodeC= GraphModelFactory.makeDefaultNode(modelFactory, "c");
```

In the example above, we repeatedly show that a new Node instance is created through a method call to a factory object. Also note that exception handling has been ignored in this and other examples in this tutorial. Exception handling is, of course, coded into the executables we have provided you with.

3.4.2. VisualNode *Creation*

A VisualNode is the visual representation of a Node. There can be more than one VisualNode instance which references a unique Node instance. Each of these VisualNode instances, however, must belong to a unique instance of class VisualGraphView.

VisualNode instances do not have to be created explicitly. If you are using the ModelBasedGraphController class, visual nodes are automatically created and inserted into a VisualGraphView the moment you create Node instances in the model.

3.5. Node and VisualNode *Insertion*

In this section, you will learn how to add two Node instances to a Graph instance and two VisualNode instances to a VisualGraphView instance.

3.5.1. Node *Insertion*

Example 3.4. Adding two Node instances to a Graph

```
// add nodes to graph
graphModel.addNode(nodeA);
graphModel.addNode(nodeB);
graphModel.addNode(nodeC);
```

The example above is quite trivial. All you do is tell the graph to add nodes using the addNode(Node) method, which is part of the more abstract interface GraphObjectContainer.

3.5.2. VisualNode *Insertion*

If you are using the ModelBasedGraphController class with your Graph model, VisualNode objects are created and inserted into your view the moment they are created in the model. For this reason, you are not required to create them explicitly, unless of course you are using another type of controller which does not support this behavior. Please read the MVC [14] chapter for more details about other GraphController implementations which require you to explicitly insert VisualNode objects into your view.

3.6. Edge and VisualEdge *Creation*

In this section, you will learn how to create an Edge instance and a VisualEdge instance.

3.6.1. Edge *Creation*

The word Edge comes from graph theory and is used to describe the association between two nodes.

Link, connection, arc, transition and other domain-specific words are used to describe this association object as well.

In the true mathematical meaning, an edge must be connected to two nodes at all times. Isolated edges - those with one or no connected nodes - may exist in a visual graph only and are denoted as such. This means that isolated edges are not part of any graph model.

Example 3.5. Creating an `Edge` instance

```
// create edges and add them to the graph
edgeAB= modelFactory.newEdge(nodeA, nodeB);
edgeAC= modelFactory.newEdge(nodeA, nodeC);
```

In the example above, creating a new `Edge` instance is trivial yet consistent with all other creational methods in this framework - via factory.

3.6.2. VisualEdge *Creation*

`VisualEdge` instances do not have to be created explicitly. If you are using the `ModelBasedGraphController` class, visual edges are automatically created and inserted into a `VisualGraphView` the moment you create `Edge` instances in the model.

3.7. Edge and VisualEdge *Insertion*

In this section, you will learn how to add two `Edge` instances to a `Graph` instance and two `VisualEdge` instances to a `VisualGraph` instance.

3.7.1. Edge *Insertion*

A `Graph` instance understands the `add` method because it is derived from the `GraphObjectContainer` interface.

Example 3.6. Inserting an `Edge` instance

```
graphModel.addEdge(edgeAB);
graphModel.addEdge(edgeAC);
```

3.7.2. VisualEdge *Insertion*

If you are using the `ModelBasedGraphController` class with your `Graph` model, `VisualEdge` objects are created and inserted into your view the moment they are created in the model. For this reason, you are not required to create them explicitly, unless of course you are using another type of graph controller which does not support this behavior. Please read the MVC [14] chapter for more details about other `GraphController` implementations which require you to explicitly insert `VisualEdge` objects into your view.

3.8. Summary

In this chapter you learned about the most important classes and interfaces in the *Tensegrity Graph API*. These entities provide the building blocks for creating applications that model and visualize graphs.

In order to simplify this chapter, we purposefully left out details about how to explicitly create some visual elements, in particular instances of classes `VisualNode` and `VisualEdge`. The *Tensegrity Graph API* allows you to delegate those instructions to a special `GraphController` object that communicates with your `Graph` model and views. In the next chapter (MVC [14]), you will learn more about how these controllers work.

Chapter 4. MVC

This tutorial provides *Java Application Programmers* exposure to some design concepts in the Tensegrity Graph Framework. Specifically, we now give you important information about the Model-View-Controller (MVC) Pattern and how this pattern implementation supports the relationship between a `Graph` and a `VisualGraph`.

4.1. Tutorial At A Glance

The table below gives you some important information about this tutorial.

Table 4.1. Tutorial Aspects

Tutorial Aspect	Tutorial Description
Approximate Duration	15 Minutes
Expected Outcome	An understanding of the relationship between a `Graph` and a `VisualGraph`. This example gives the *Java Application Programmer* an initial exposure to a controller and how it coordinates and decouples the communication between the aforementioned classes.
Source Files	The tutorial examples come with two source packages containing the java and configuration files needed to compile and run them. These are “com.tensegrity.firststeps.mvc” and “com.tensegrity.firststeps.swt.mvc” respectively for the AWT/Swing-based and SWT-based examples.
Creating Files	Not applicable in this tutorial
Modifying Files	Not applicable in this tutorial

4.2. Running The Tutorial Example

To run the tutorial example, please do the following in Eclipse:

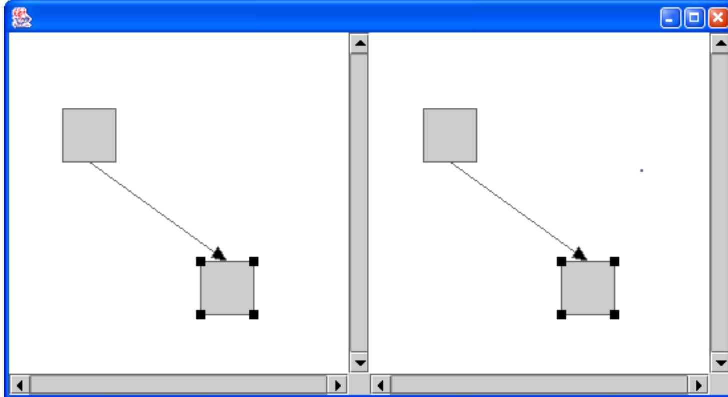
1. Select package “`com.tensegrity.firststeps.mvc`” or “`com.tensegrity.firststeps.swt.mvc`” with the left mouse button.
2. Select the menu “Run”.
3. Select the child menu “run as”.
4. Select “Java Application”
5. The main method from the “MVCExample.java” compiled class will be executed.
6. Select elements in either of the two views and acknowledge that the other view remains synchronized.

Note

Running the SWT-based example as described above you will probably encounter an error of this type: “java.lang.UnsatisfiedLinkError: no swt-win32-3062 in java.library.path”.

Please refer to the note in section "Running The Tutorial Example" of chapter "Hello Node" in order to see how to overcome this issue.

Figure 4.1. Screenshot MVC Example



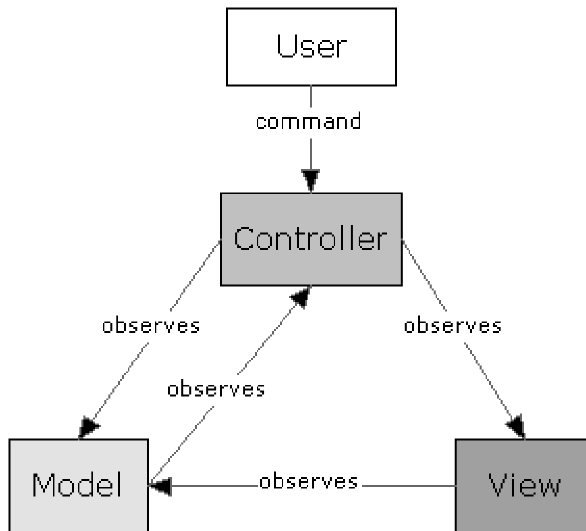
Two views attached to a single model.

4.3. MVC in the *Tensegrity Graph API*

MVC (Model-View-Controller) is perhaps the oldest and most widely used software pattern and dates back to the late 1970s. It's goal has been to prevent data (models) from explicitly knowing about all the data visualizations that depend on it and also to decouple the actions that are needed to work with and modify the data located in the model from the model itself. By centralizing data in one source, redundancy had been eliminated. A *Controller* object was introduced to coordinate the modifications that are necessary for the data located in the model.

Although there are many deviations, the MVC pattern basically consists of three collaborating roles. The first is called the model, which is responsible for storing data. The second role is the view and takes care of the visualization of any data located in the model. The third role is the controller. Its responsibility is to facilitate manipulation of the data in the model by propagating user events to it and to propagate model changes to all dependent views. These three roles logically participate in the Observer pattern as well: view objects observe a model, model objects observe a controller and controller objects observe both model and view.

Figure 4.2. MVC Roles

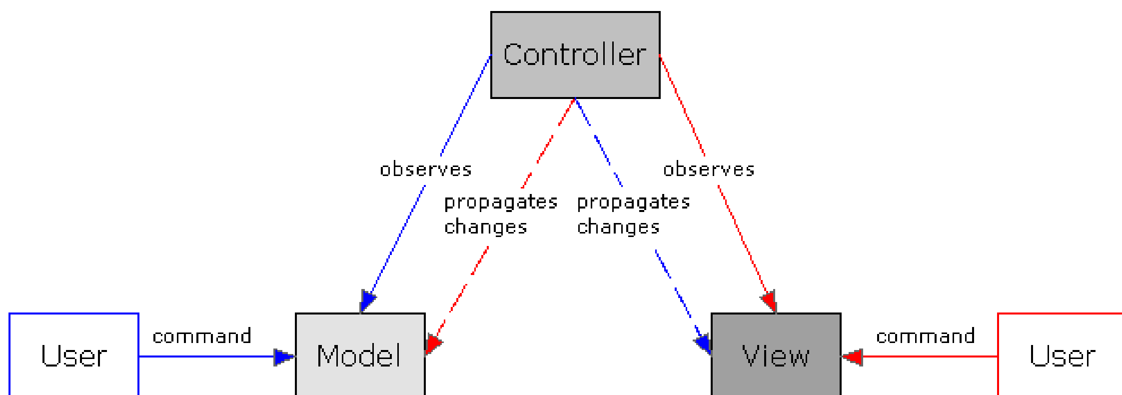


The *Tensegrity Graph API* implements the MVC pattern by supporting multiple views for a single `Graph` model. The controller serves an administrative role by notifying all `VisualGraph` views about any model changes which have occurred. Normally, model changes lead to a situation where one or more views might not reflect the current state of their model. The MVC pattern decouples our `Graph` models from the views by having a `GraphController` act as a mediator. This controller notifies all views when the model changes and these views subsequently refresh themselves, if necessary, by requesting data from their associated `Graph` model.

This is not the end of the story, however. Within the *Tensegrity Graph API*, a controller cannot be used to exclusively receive and propagate user events directly to a model only. This is because a `GraphView` is a manipulatable, persistable object in its own right. In such a MVC scenario, where a view contains unique attributes not found it its model, the view essentially "acts" like model itself. For this reason, both models and views must be capable of directly receiving user events.

The following figure illustrates these two scenarios better: Manipulations that are made to a model are propagated to a view (blue edges) and manipulations that are made to a view are propagated to its model (red edges). Notice that user events (commands) may be directed towards the model or towards the view.

Figure 4.3. MVC in the Tensegrity Framework



4.4. Graph Controllers

Graph controllers manage the views that are active for a given `Graph`. They do this by notifying these views about model and view changes. Different controllers are available in the *Tensegrity Graph API* that vary the way in which model and view exchange messages. In the following sections we discuss the different `GraphController` implementations that are accessible through the `GraphController-Factory`.

4.4.1. ClientServerGraphController

This controller propagates user commands directly to a view.

The class `ClientServerGraphController` manages a typical (simple) client and server scenario, where there is one privileged master view (server) and one to many client views. The client views are read-only and simply reflect the current state of the master view. These client views are often distributed on different computers and need to synchronize their output with the output of the master view.

4.4.2. ModelBasedGraphController

This controller propagates user commands directly to a model.

Class `ModelBasedGraphController` manages a typical model-based scenario in which changes to the model are automatically reflected by specific predefined visual operations. This controller is also responsible for automatically inserting visual elements into a view whenever model elements are created and inserted into a `Graph`.

4.5. GraphController *Creation*

In this section, you will learn how to create and use a specific `GraphController`.

A `GraphController` instance is created by requesting a particular controller “type” from the factory, as the code below illustrates.

Example 4.1. GraphController Creation

```
// Create client-server graph controller
GraphControllerFactory cFactory=
    GraphControllerFactory.newInstance();
graphController=
    cFactory.newClientServerGraphController(graphModel);
```

In the method above, you should note that a `GraphController` instance is always coupled to a model `Graph`. This is not true for graph views, however, as we will see next.

4.6. GraphController *Association*

In this section, you will learn how to associate a `GraphController` to a `VisualGraphView`.

Example 4.2. VisualGraphView Creation

```
// Get a factory instance
GraphViewFactory viewfactory= GraphViewFactory.newInstance();

// Create a new visual graph view using a graph controller
graphView= viewfactory.newVisualGraphView(controller);
graphView.enableUndoRedo();

// Add the view to managed list of views
controller.addVisualGraph(graphView);
```

In the method above, you can see that a `GraphController` instance is required to create a new `VisualGraphView`. Additionally, you are required to manually add the view to the controller once is has been successfully created.

4.7. Visual Element *Creation*

In this section, you will learn how to programmatically create visual elements and insert them into a view when using a `GraphController` implementation *which does not handle this for you automatically*.

Note

Class `ModelBasedGraphController` is the only `GraphController` implementation which provides automatic creation and insertion of visual graph elements at this time. All other implementations do not automatically create visual elements when model elements are created.

Example 4.3. Creating a `VisualNode` programmatically

```
// Exception handling has been ignored to shorten the example

// Retrieve a factory instance which can create visual elements
GraphViewFactory viewFactory = GraphViewFactory.newInstance();

// Retrieve the geometry definition using the input string.
GeometryDescriptor gDesc = GeometryPool.get(geometryDesc);

// Specify the geometry description while building a new visual node
VisualNode visualNode = viewFactory.newVisualNode(modelNode, gDesc);

// Retrieve and style descriptor using the input string
StyleDescriptor sDesc = StylePool.get(styleDesc);

// Apply the style descriptor to the new visual node
visualNode.getBaseComposite().applyStyle(sDesc);
```

The method above shows you how to manually create a new `VisualNode`. Please notice that you are required to specify a `GeometryDescriptor` when requesting a new object from the factory. Similarly, a `StyleDescriptor` is also required for the created `VisualNode`. Both geometries and styles are discussed in the Repository [21] chapter inside this manual.

Example 4.4. Adding `VisualNode` objects to a `VisualGraphView`

```
// Assuming you have two visual nodes and a visual graph object
graphView.addVisualNode(visualNodeA);
graphView.addVisualNode(visualNodeB);
```

Once you have created a new `VisualNode`, it must be manually added to a view. The `addVisualNode` method shown in the example above is part of the `VisualGraphObjectContainer` interface, from which `VisualGraphView` is derived.

Example 4.5. Creating a `VisualEdge` instance

```
// Exception handling has been ignored to shorten the example.

// Get an instance of the graph view factory
GraphViewFactory viewFactory = GraphViewFactory.newInstance();

// Create a new visual edge via factory method
VisualEdge visualEdge =
    viewFactory.newVisualEdge(modelEdge, source, target);

// Retrieve the geometry definition using the input string
GeometryDescriptor gDesc = GeometryPool.get(geometryDesc);

// Apply the geometry descriptor to the new visual edge
visualEdge.getBaseComposite().applyGeometry(gDesc);

// Retrieve and style descriptor using the input string
StyleDescriptor sDesc = StylePool.get(styleDesc);

// Apply the style descriptor to the new visual edge
visualEdge.getBaseComposite().applyStyle(sDesc);
```

The example above is quite straightforward. Via factory method you can request a new `VisualEdge` instance. Subsequently, you set the geometry and style descriptors, which define the visual edge's graphical appearance. Both geometries and styles are discussed in the Repository [21] chapter inside this manual.

Example 4.6. Adding a `VisualEdge` to a `VisualGraphView`

```
//Assuming you have a visual edge and a visual graph object
graphView.addVisualEdge(visualEdgeAB);
```

Once you have created a new VisualEdge, it must be manually added to a view. The add method shown in the example above is part of the `VisualGraphObjectContainer` interface, from which `VisualGraphView` is derived.

4.8. Summary

In this chapter you learned about an important pattern implementation in the *Tensegrity Graph API*, namely MVC or Model-View-Controller. Our pattern implementation decouples graphs from views and introduces a `GraphController` component to coordinate the communication paths between them.

Since there is more than one `GraphController` implementation, you are able to choose one that most closely matches your requirements.

In the next chapter (Repository [21]), we talk about how to create and modify the visual element templates which facilitate the manual creation of visual graph documents.

Chapter 5. Repository

This tutorial provides *Java Application Programmers* exposure to additional features of the Tensegrity Graph Framework. Specifically, we provide you important information about creating a repository of graph elements from which users may interactively select and drag into their view documents.

When designing the visual elements of a repository, you specify them in xml and load them programmatically into several runtime "pool" objects (singletons) which cache this information. From then on, you may easily assign geometries and styles to each visual element in your repository or visual graphs.

Note

A Repository in the *Tensegrity Graph Framework* is not the same thing as the word "Repository" used in other domain vocabularies, such as in RDMS contexts, where the words "database, store, persistence and repository" are quite often used interchangeably. To avoid confusion, you might think of our repository as a "Library" of predefined, domain-specific graph elements that may be copied or "instantiated" into your working design documents.

5.1. Tutorial At A Glance

The table below gives you some important information about this tutorial.

Table 5.1. Tutorial Aspects

Tutorial Aspect	Tutorial Description
Approximate Duration	35 Minutes
Expected Outcome	An application showing a graph document view containing selectable and dragable nodes and edges. This example gives the *Java Application Programmer* an initial exposure to geometries, styles and rules, which are specified in XML and loaded at application start-up.
Source Files	The tutorial examples come with two source packages containing the java and configuration files needed to compile and run them. These are "com.tensegrity.firststeps.repository" and "com.tensegrity.firststeps.swt.repository" respectively for the AWT/Swing-based and SWT-based examples.
Creating Files	`geometry.xml`, `styles.xml`, `rules.xml`
Modifying Files	Not applicable in this tutorial

5.2. Running The Tutorial Example

To run the tutorial example, please do the following in Eclipse:

1. Select package "`com.tensegrity.firststeps.repository`" or "`com.tensegrity.firststeps.swt.repository`" with the left mouse button.

2. Select the menu "Run".

3. Select the child menu “run as”.

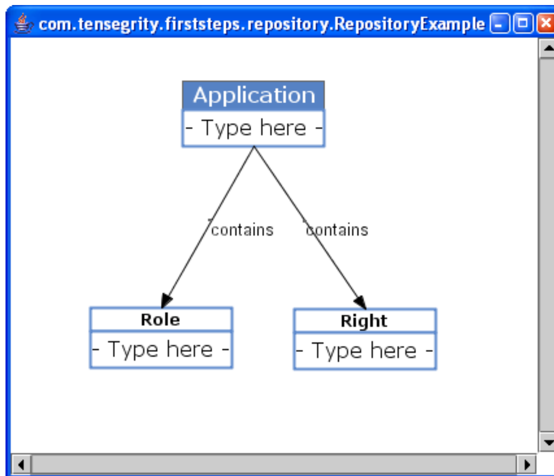
4. Select “Java Application”

5. The main method from the “RespositoryExample.java” compiled class will be executed.

Note

Running the SWT-based example as described above you will probably encounter an error of this type: “java.lang.UnsatisfiedLinkError: no swt-win32-3062 in java.library.path”. Please refer to the note in section “Running The Tutorial Example” of chapter “Hello Node” in order to see how to overcome this issue.

Figure 5.1. Repository Elements in a Document



The repository elements shown above are defined in various xml files discussed in the remainder of this chapter.

5.3. Geometry *Definition*

Geometries are vector-based descriptions and composite groupings that allow you to define a subset of the visual attributes of your repository elements. These particular attributes define the geometric shapes that are rendered whenever a visual element is drawn in a repository or document view.

The following xml fragment is part of the file named `geometry.xml`. Here you should quickly glance over its contents and ponder the composite geometry named “AppLabelGeometryDescriptor”. A composite geometry, referred to as a `GeometryDescriptor`, combines one or more individually-defined `GeometryItem` elements previously declared in the same `geometry.xml` file.

In this and the following sections, we will generally describe the most important structural aspects of all of these xml elements and point you to additional resources that thoroughly discuss the configuration possibilities available to you.

Example 5.1. Geometry Definition Fragment

```
...

<list name="GeometryItem">
  <attribute name="Name" value="AppTitleLabel"/>
  <attribute name="Type" value="Label"/>
  <set name="Attributes">
    <attribute name="Text" value="Application"/>
    <attribute name="FontUnit" value="Point" />
    <attribute name="HorizontalAlignment" value="Center" />
    <attribute name="VerticalAlignment" value="Center" />
    <attribute name="SizeAdjustment" value="LineCountAndLength"/>
    <attribute name="Clipping" type="Boolean" value="true"/>
    <attribute name="LineShortening" type="Boolean" value="false"/>
    <attribute name="WordWrap" type="Boolean" value="false"/>
    <attribute name="FirstWordCharacterWrap" type="Boolean" value="false"/>
    <attribute name="Editable" type="Boolean" value="false"/>
  </set>
</list>

...

<list name="GeometryDescriptor">
  <attribute name="Name" value="AppLabelGeometryDescriptor" />
  <attribute name="Type" value="Composite" />
  <list name="CoordinateSystem">
    <attribute name="ScaleX" value="0mm, 10mm"/>
    <attribute name="ScaleY" value="0mm, 10mm"/>
  </list>
  <set name="Attributes">
      <attribute name="MinimumSize" type="Size" value="10mm, 10mm" />
  </set>
  <list name="DescriptorItems">
    <list name="DescriptorItem">
      <attribute name="Name" value="TitleArea" />
      <attribute name="GeometryItemName" value="AppTitleLabel" />
      <list name="Coordinates">
        <attribute name="Coordinate" value="0.0mm, 0.0mm" />
        <attribute name="Coordinate" value="10.0mm, 3.0mm" />
      </list>
      ...
    </list>
    <list name="DescriptorItem">
      <attribute name="Name" value="TextArea" />
      <attribute name="GeometryItemName" value="AppCLabel" />
      <list name="Coordinates">
        <attribute name="Coordinate" value="0.0mm, 3.0mm" />
        <attribute name="Coordinate" value="10.0mm, 10.0mm" />
      </list>
      ...
    </list>
    ...
  </list>
</list>

...
```

At first glance, the xml-fragment listed above might look a bit confusing. The same tags appear again and again and they sound rather generic.

Please do not worry about this. The *Tensegrity Graph API* must deal with many dynamic properties and therefore uses `Attribute` objects at runtime to pass around named values that aren't tied to any partic-

ular class.

Specification files such as `geometry.xml`, as well as others found in our framework, use an attribute paradigm when defining repository elements. Named tags, such as <SizeAdjustment> or <RoundEdges>, do not exist. Rather, this data is held inside different <attribute> elements where “SizeAdjustment” and “RoundEdges” are named xml-attribute values. The tags <list> and <set> enforce cardinality rules for child <attribute> tags, specifically the string found in the value part of the xml-attribute called *name*.

What this means for you is a generic approach at the cost of slightly decreased readability.

5.4. Geometry XML

GeometryItems, *GeometryDescriptors* and *GeometryDescriptorItems*, such as those listed in the previous section's xml fragment, will be loaded into runtime java objects whose classes share the same names. These objects provide the *what and where to draw* data that is needed during runtime rendering.

The classes `Primitive` and `Composite`, on the other hand, provide the runtime *how to draw* functionality. Of course, objects of these classes utilize all of the geometry data you have configured. We only mention these classes because you are allowed direct access to them and sometimes need their instances when requesting new visual elements from a factory.

We should mention that a `Primitive` in the *Tensegrity Graph API* is not the same thing as the word “primitive” used in other programming contexts. Primitives are those parts of a `Composite` container that are responsible for the atomic pieces of a drawing, such as lines, rectangles, polygons and so forth. A `Primitive` basically takes information configured in a geometry file and translates it into objects that are needed for rendering inside a specific device or drawing context. A `Composite` will hold one or more of these `Primitive` objects and draw them in its own `CoordinateSystem`. In short, you provide the information about what to draw and our framework classes take care of the rest.

Although we cover these topics in much greater detail in the *Elements* chapter inside the **Framework Manual**, we shall provide you with some of the key geometry concepts here as well.

5.4.1. GeometryItem

A `GeometryItem` defines a basic, atomic geometric shape, such as a line, rectangle, ellipse, polygon or label. These items are created in a `geometry.xml` or similarly named file and customized with attribute parameters. These attributes define specific visual aspects of the geometry item and vary depending on the item's basic type.

Once a `GeometryItem` has been defined, it can be referenced by any number of composites, which group items into new geometry entities of type `GeometryDescriptor`. This separation of items and composites allows you to easily attain a consistent look and feel for all of your repository elements by centralizing and reusing one or more `GeometryItem` definitions.

Every `GeometryItem` consists of a name, a type and a list of optional attributes. A minimal `GeometryItem` looks like this:

Example 5.2. Minimal GeometryItem

```
<!-- A Minimal Item -->
<list name="GeometryItem">
 <attribute name="Name" value="GEOMETRY_ITEM_NAME"/>
 <attribute name="Type" value="GEOMETRY_ITEM_TYPE"/>
</list>
```

The GEOMETRY_ITEM_NAME text in the example above may be any unique string you wish. The GEOMETRY_ITEM_TYPE text, however, must be one of several predefined values that are described in the *Elements* chapter inside the **Framework Manual**, which is distributed separately. Please read that chapter when you are finished reading this section. There you will discover which additional attributes are required for each geometry item type listed.

5.4.2. GeometryDescriptorItem

A GeometryDescriptorItem represents the first level of GeometryItem (aka “basic item”) re-use. It exists because basic items will be referenced by one or more composites (which we will discuss next). Additionally, basic items do not specify any coordinate data for the objects that draw them. This makes perfect sense. A composite will need to position basic items relative to one another. In most cases, drawing individual items in the context of a composite will not always begin in the upper left-hand corner of a composite's coordinate system.

To make things perfectly clear, a GeometryDescriptorItem represents an association between a single GeometryItem (atomic geometry, basic item) and a single GeometryDescriptor (composite geometry, to be discussed next). A GeometryDescriptorItem element is embedded inside a <DescriptorItem> tag, which you can see in the XML fragment in the previous section.

Every GeometryDescriptorItem, embedded inside a parent GeometryDescriptor element, consists of its unique name, the name of the referenced GeometryItem as well the drawing coordinates within the composite. A minimal GeometryDescriptorItem therefore looks like this:

Example 5.3. Minimal GeometryDescriptorItem

```
<!-- A minimal geometry descriptor item -->
<list name="DescriptorItem">
 <attribute name="Name" value="GEOMETRY_DESCRIPTOR_ITEM_NAME"/>
 <attribute name="GeometryItemName"
            value="REFERENCED_GEOMETRY_ITEM_NAME"/>

 <list name="Coordinates">
  <attribute name="Coordinate" value="X_COMPONENT, Y_COMPONENT" />
  <attribute name="Coordinate" value="X_COMPONENT, Y_COMPONENT" />
  [...]
 </list>

</list>
```

The *Coordinate* attributes listed above define two points for the GeometryItem being referenced. Depending on the item type, you may require more data than this. Please refer to the *Elements* chapter inside the **Framework Manual** for more detailed information about specifying coordinate attributes for the various geometry types available in our framework.

5.4.3. GeometryDescriptor

A GeometryDescriptor represents a grouping of one or more basic items of type GeometryItem. Each item referenced the GeometryDescriptor file definition is represented as a uniquely named GeometryDescriptorItem.

Each GeometryDescriptor definition consists of a name, a type, a coordinate system and, of course, a list of GeometryDescriptorItem elements. In its minimal form it looks like the follow-

ing:

Example 5.4. Minimal GeometryDescriptor

```
<!-- a minimal geometry descritptor -->
<list name="GeometryDescriptor">
  <attribute name="Name" value="GEOMETRY_DESCRIPTOR_NAME"/>
  <attribute name="Type" value="GEOMETRY_DESCRITPOR_TYPE"/>
  <list name="CoordinateSystem">
    <attribute name="ScaleX" value="MINIMUM_VALUE, MAXIMUM_VALUE" />
    <attribute name="ScaleY" value="MINIMUM_VALUE, MAXIMUM_VALUE" />
  </list>
  <!-- optional attributes of the geometry descriptor -->
  <set name="Attributes">
    <attribute name="ATTRIBUTENAME"
      type="ATTRIBUTETYPE" value="ATTRIBUTEVALUE" />
    [...]
  </set>
  <list name="DescriptorItems">
    <list name="DescriptorItem">
      [...]
    </list>
  </list>
</list>
```

A `GeometryDescriptor` has a *Type* attribute that can be one of the following string enumerations: *Composite*, *CompositeLine* or *CompositeGroup*. Descriptor types are described in more detail in the *Elements* chapter inside the **Framework Manual**, which is distributed separately. We recommend that you browse this chapter now to acquire a better understanding of the attributes required for a more complex `GeometryDescriptor`.

5.5. Geometry Pool

A `GeometryPool` is a runtime cache of objects that represent the named `Geometry` elements defined in your `geometry.xml` file.

Loading geometries into the `GeometryPool` is performed via the `GeometryService` class, as shown below.

Example 5.5. Loading configured geometries into the GeometryPool

```
try
{
  GeometryService.loadGeometriesFromResource(
      RESOURCE_PATH + "geometry.xml", this.getClass());

  GeometryDescriptor gDesc =
      GeometryPool.get("AppLabelGeometryDescriptor");

  VisualNode visualNode =
      graphView.getVisualNodeByID(nodeA.getID());

  visualNode.getBaseComposite().applyGeometry(gDesc);
}
catch(Exception e)
```

```
{
    e.printStackTrace();
}
```

The source code in bold shows the static class method which allows you to load the xml-specified geometries into a pool object.

5.6. Geometry *Association*

In this section, you will learn how to associate or apply a `Geometry` to a node or edge. We use the same source code as in the previous example.

Example 5.6. Applying a user-defined geometry

```
try
{
  GeometryService.loadGeometriesFromResource(
      RESOURCE_PATH + "geometry.xml", this.getClass());

  GeometryDescriptor gDesc =
      GeometryPool.get("AppLabelGeometryDescriptor");

  VisualNode visualNode =
      graphView.getVisualNodeByID(nodeA.getID());

  visualNode.getBaseComposite().applyGeometry(gDesc);
}
catch(Exception e)
{
  e.printStackTrace();
}
```

The source code in bold shows you how to obtain a named `GeometryDescriptor` instance from the pool and apply that instance to a particular `VisualNode` via its embedded `BaseComposite` member. Please see the javadocs for more information about this and other composite interfaces, which are responsible for the runtime graphical representation of all visual elements.

5.7. Style *Definition*

In this section, you will learn how to create a `Style` for a visual element (`VisualNode` and `VisualEdge`). A `Style` is used in conjunction with a `Geometry` to complete a visual element's graphical representation in a view. While *geometries* are used for shapes, *styles* are used for colors, line weights, line strokes and fonts.

Once you have understood how *Items* are configured and referenced from composite *Descriptors*, you will automatically understand how `Style` configurations work. We therefore recommend that you read the previous *Geometry* sections so that you may understand the `Style` xml structures more easily.

The following xml fragment is part of the file named `styles.xml`. Here you should analyze its contents and examine the `StyleItem` named “AppLabelStyleItem” and the composite (`StyleDescriptor`) named “AppLabelStyleDescriptor”.

Example 5.7. Style definition fragment

```
...

<list name="StyleItem">
  <attribute name="Name" value="AppLabelStyleItem"/>
  <attribute name="Type" value="Label"/>
  <set name="Attributes">
    <set name="Fill">
      <attribute name="style" value="solid"/>
      <attribute name="background" type="Color" value="86,130,196"/>
    </set>
    <set name="Font">
      <attribute name="size" type="Integer" value="12"/>
      <attribute name="family" value="Verdana"/>
      <attribute name="color" type="Color" value="white"/>
      <attribute name="weight" value="bold"/>
    </set>
    <set name="Border">
      <attribute name="color" type="Color" value="102, 102, 102"/>
      <attribute name="round edges" type="Boolean" value="true"/>
    </set>
  </set>
</list>

...

<list name="StyleDescriptor">
  <attribute name="Name" value="AppLabelStyleDescriptor" />
  <attribute name="Type" value="Composite" />
  <list name="DescriptorItems">
    <set name="DescriptorItem">
      <attribute name="StyleItem" value="AppLabelStyleItem"/>
    </set>
    <set name="DescriptorItem">
      <attribute name="StyleItem" value="AppCLabelStyleItem"/>
    </set>
  </list>
</list>

...
```

Elaborate details about configuring `StyleItem` elements can be found in the *Elements* chapter inside the **Framework Manual** distributed separately. We strongly recommend that you read that chapter now to gain an in-depth understanding of all `Style` configuration possibilities available to you.

Note

We have eliminated explanatory text concerning the xml structures of styles because the concepts and naming conventions used are identical to those of geometries. If you have read this previous section, you will be able to understand and decipher any `style.xml` file.

5.8. Style Pool

In this section, you will learn how to load styles into a `StylePool`, a runtime cache of objects that rep-

resent the `Style` elements defined in your `styles.xml` file.

Example 5.8. Loading configured styles into the `StylePool`

```
StyleService.loadStylesFromResource(
    RESOURCE_PATH + "styles.xml", RepositoryExample.class);
```

Loading styles into the `StylePool` is identical to loading geometries into the `GeometryPool`. The pools are kept separate to allow you to mix and match geometries and styles among visual elements.

5.9. Style *Association*

In this section, you will learn how to associate or apply a `Style` to an element.

Example 5.9. Applying a user-defined style

```
// load style descriptor from pool
StyleDescriptor styleAppNode =
    StylePool.get("AppLabelStyleDescriptor");
// apply the appropriate style to the VisualNode
visualNodeA.getBaseComposite().applyStyle(styleAppNode);
```

Applying a style to a visual element is similar to applying a geometry. The `BaseComposite` member of any `VisualGraphObject` is used to apply it via a method call to *applyStyle()*.

5.10. Repository Element *Definition*

In this section, you will learn how to define a repository element, including its name, geometry, style and optional modeling rule.

A repository element is a **prototype** for a visual element that is duplicated inside your view documents. This means that the repository element's type, geometry, style and rule are used to create the new visual elements you add to a view (programmatically or via drag and drop). You may create as many repository elements as you require for your business application.

Similar to geometries, styles and rules, repository elements are specified in xml and loaded into a “Pool” object, which functions as a singleton.

The following xml fragment is part of the file named `elements.xml`. Here you should analyze its contents and decipher the *NodeElement* named “AppNodeElement”.

Example 5.10. Element definition

```
<list name="NodeElement">
  <attribute name="Name" value="AppNodeElement"/>
  <attribute name="Geometry" value="AppLabelGeometryDescriptor"/>
  <attribute name="Style" value="AppLabelStyleDescriptor"/>
  <attribute name="Rule" value="AppNodeRule"/>
</list>
```

Note that the above repository item has been declared a “NodeElement.” Both *NodeElements* and *EdgeElements* aggregate a named `GeometryDescriptor`, `StyleDescriptor` and an optional *Rule* inside their definitions. Please read Modeling Rules [31] to understand what these rules are and how to configure their attributes.

Taken together, the descriptors and the optional rule completely define a unique repository element that you may retrieve by name and apply to a particular visual element in a view.

5.11. Element Pool

In this section, you will learn how to load elements into an `ElementPool`, a runtime cache of objects that represent the different visual elements defined in your `elements.xml` file.

Example 5.11. `ElementPool` Loading

```
ElementService.loadElementsFromResource(
    RESOURCE_PATH + "elements.xml", RepositoryExample.class);
```

Loading elements into the `ElementPool` is performed via the `ElementService` class and resembles the manner in which one loads geometries and styles into their respective pools.

5.12. Summary

In this chapter you learned about repository elements and how to configure their various parts. Examples of `Geometry` and `Style` items as well as composite *Descriptor* definitions were shown and explained in various sections.

Although we tried to describe these concepts at a level of detail that should not overwhelm the new user, we strongly suggest that you read those parts of the **Framework Manual** that have been referenced inside this chapter.

In the next chapter (Modeling Rules [31]), we discuss the creation and modification of rules that support and restrict users when creating and manipulating visual graphs. *Rules* were first introduced in this chapter but without the detail necessary to understand them.

Chapter 6. Modeling Rules

This tutorial provides *Java Application Programmers* exposure to critical configuration and runtime features of the Tensegrity Graph Framework. Here we give you detailed instructions on how to redefine a graph's runtime behavioral rules. These are the rules which support and restrict users while creating and manipulating visual graphs.

A rule in the Tensegrity Graph Framework makes sure that a model or a view remain well-defined according to that rule. Cardinality, association and insertion rules are just a few of the many rules programmers can take advantage of. Although we cover these topics in much greater detail in the *Elements* chapter inside the **Framework Manual**, we shall provide you with some of the key rules concepts in this chapter as well.

6.1. Tutorial At A Glance

The table below gives you some important information about this tutorial.

Table 6.1. Tutorial Aspects

Tutorial Aspect	Tutorial Description
Approximate Duration	45 Minutes
Expected Outcome	An application showing a graph document view containing selectable and dragable nodes and edges. This example gives the *Java Application Programmer* an initial exposure to modeling rules that are specified in XML and loaded at application start-up.
Source Files	The tutorial examples come with two source packages containing the java and configuration files needed to compile and run them. These are "com.tensegrity.firststeps.rules" and "com.tensegrity.firststeps.swt.rules" respectively for the AWT/Swing-based and SWT-based examples.
Creating Files	Not applicable in this tutorial
Modifying Files	rules.xml

6.2. Running The Tutorial Example

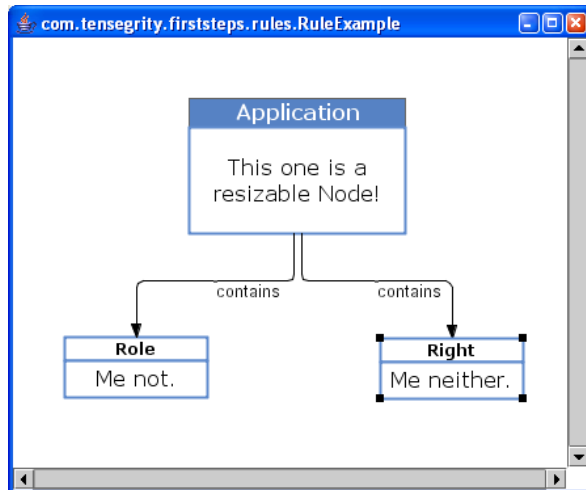
To run the tutorial example, please do the following in Eclipse:

1. Select package "`com.tensegrity.firststeps.rules`" or "`com.tensegrity.firststeps.swt.rules`" with the left mouse button.
2. Select the menu "Run".
3. Select the child menu "run as".
4. Select "Java Application"
5. The main method from the "Main.java" compiled class will be executed.
6. In the running application, try to test the rules defined in the rules.xml file.

Note

Running the SWT-based example as described above you will probably encounter an error of this type: “java.lang.UnsatisfiedLinkError: no swt-win32-3062 in java.library.path”. Please refer to the note in section “Running The Tutorial Example” of chapter “Hello Node” in order to see how to overcome this issue.

Figure 6.1. Screenshot RulesExample



In the visual graph above, a rule has been defined and attached to the root node (at the top), permitting it to be resized.

6.3. Graph Rule *Definition*

In this section, you will learn about defining graph rules inside an xml configuration file. Such a file is included with this tutorial and is aptly called “`rules.xml`”.

Below is a sample `GraphRule` defined in an attribute-based xml format. This type of rule is applied to a visual graph as a whole and is much simpler than the rules created for its embedded visual elements.

Example 6.1. Graph Rule XML Fragment

```
...

<list name="Rule">
 <attribute name="Name" type="String" value="graphrule"/>
 <attribute name="Type" type="String" value="GraphRule"/>
 <list name="Properties">
  <attribute name="EdgeInsertSinglePosition" type="Boolean" value="false"/>
  <attribute name="AutoSpaceMode" type="String" value="no autospace"/>
  <attribute name="AcceptLooseObjects" type="Boolean" value="true"/>
 </list>
</list>
```

```
...
```

There are various visual graph attributes which together form the basis of a rule. For visual graph objects, the relevant attributes are listed below.

- `EdgeInsertSinglePosition`

  Also known as an “Edge Split,” this flag deals with edges that have been configured to accept nodes dropped on them. In other words, when a user drags and drops a node on an edge which has this attribute value set to true, the nodes previously connected by that edge will lose that connection and subsequently connect to the dropped node.

  Multiply-selected nodes that are simultaneously dropped over distinct edges will also split the affected edges.

- `AutoSpaceMode`

  The given value for the attribute `AutoSpaceMode` must be one of the following enumerations. The `AutoSpaceMode` attribute specifies how nodes are moved when they are dropped onto each other.
  - horizontal autospace
  - vertical autospace
  - no autospace

- `AcceptLooseObjects`

  This flag allows or disallows “loose” elements (isolated nodes or edges) at the top-level of a visual graph. Isolated elements are not allowed in a real graph but may be permitted here for designers who perform intermediate steps towards completion of their models. If you wish to enforce a "Do it now or never" policy you would set this flag to “false”.

6.4. NodeRule *Definition*

In this section, you will learn how a `NodeRule` is defined and structured inside a given `rules.xml` or similarly named file.

There are various attributes that can be changed to customize the runtime behavior of a `VisualNode`. Below we list a few of them. You will eventually have to consult the *Rules* section of the *Elements* chapter inside the **Framework Manual** for more detailed explanations of all configuration attributes available to you. Such a discussion is unfortunately beyond the scope of this tutorial chapter and would undoubtedly duplicate information that is likely to change in future software releases.

Example 6.2. `NodeRule` XML Fragment

```
...
<list name="Rule">
 <attribute name="Name" type="String" value="nrule1"/>
 <attribute name="Type" type="String" value="NodeRule"/>
 <list name="Properties">
  <attribute name="Deletable"  type="Boolean" value="false"/>
  <attribute name="Moveable"   type="Boolean" value="true"/>
  <attribute name="Selectable" type="Boolean" value="true"/>
  <attribute name="Resizable"  type="Boolean" value="false"/>
```

```
  ...
 </list>
</list>
...
```

Notice that a `Rule` has a Name and a Type attribute. “NodeRule” is the Type string required when defining rules that are to be attached to `VisualNode` instances in your repository or visual graphs.

The tag **<Properties>** contains child attribute elements that specify different behaviors of the rule. More times than not, the attribute name is intuitively comprehensible. In those rare cases where the name alone is not enough to convey meaning, you should reference the *Elements* chapter inside the **Framework Manual** that accompanies this tutorial.

6.5. EdgeRule *Definition*

In this section, you will learn how an `EdgeRule` is defined and structured inside a given `rules.xml` or similarly named file. An `EdgeRule` definition has an identical structure when compared to a `NodeRule`. The only differences are the names and numbers of rule attributes.

Likewise, there are various attributes that can be changed to customize the runtime behavior of a `VisualEdge`. Below we list a few of them. You will eventually have to consult the *Rules* section of the *Elements* chapter inside the **Framework Manual** for more detailed explanations of all configuration attributes available to you.

Example 6.3. `EdgeRule` XML Fragment

```
<list name="Rule">
 <attribute name="Name" type="String" value="erule1"/>
 <attribute name="Type" type="String" value="EdgeRule"/>
 <list name="Properties">
  <attribute name="Deletable" type="Boolean" value="false"/>
  <attribute name="Moveable" type="Boolean" value="true"/>
  <attribute name="Selectable" type="Boolean" value="true"/>
  <attribute name="Resizable" type="Boolean" value="false"/>
  <attribute name="DetachableSource" type="Boolean" value="false"/>
  <attribute name="DetachableTarget" type="Boolean" value="false"/>
  <attribute name="EdgeInsertMode" value="edge insert disabled"/>
  <attribute name="SnapToPortDistance" value="5cm"/>
 </list>
</list>
```

There are two `EdgeRule` attributes that take an enumerated value. They are the following:

- `EdgeInsertMode`

  The edge insert mode can be one of the following constant strings:
  - “edge insert disabled”
  - “edge insert exact position”
  - “edge insert automatic center”
- `SnapToPortDistance`

  The “snap to port distance” denotes the maximum distance for an edge to automatically snap to the closest port. Setting the distance to 0 or less will effectively disable the automatic snapping of edges

to the closest port.

6.6. RuleRegistry *Population*

In this section, you will learn how to populate (load) defined rules into the `RuleRegistry`.

Loading rules from a file and applying them to a graph view is an easy task, as the following example shows.

Example 6.4. Load and apply rules from file

```
try
{
  RuleRegistry ruleRegistry=
    RuleRegistry.loadRulesFromResource(
      RESOURCE_PATH+ "rules.xml", this.getClass());

  graphView.setRuleRegistry(ruleRegistry);
}
catch(ReadException e)
{
  e.printStackTrace();
}
```

Note

It is important to remember that each view instance has its own rule registry assigned to it.

6.7. Rule *Association*

In this section you will learn how to programmatically assign a rule to an element (`VisualNode` or `VisualEdge`).

Below we add a new rule named “nrule2” to rule definition file `rule.xml`. This rule specifies a resizeable node.

Example 6.5. A rule for resizeable nodes

```
...

<list name="Rule">
  <attribute name="Name" type="String" value="nrule2"/>
  <attribute name="Type" type="String" value="NodeRule"/>
  <list name="Properties">
    <attribute name="Deletable" type="Boolean" value="false"/>
    <attribute name="Moveable" type="Boolean" value="true"/>
    <attribute name="Selectable" type="Boolean" value="true"/>
    <attribute name="Resizable" type="Boolean" value="true"/>

    ...
```

```
  </list>
</list>

...
```

In this example, we do not want all our visual nodes to be resizeable, just the root one. This requires us to retrieve that root node (variable `nodeA` is holding this root node in the graph model), get the corresponding `VisualNode` object and then apply our new rule “nrule2” to it. The following code snippet reflects these steps.

Example 6.6. Assigning a rule to a node

```
// retrieve the first node of the graph
VisualNode vNodeA=
    graphView.getVisualNodeByID(nodeA.getID());

// apply the previously defined rule
vNodeA.setRule("NormalNodeRule");
```

Adding these lines of code will result in a resizeable root node. When you get around to reading the javadocs, you will notice that rules can be set for any `VisualGraphObject` instance!

6.8. Summary

In this chapter you learned about the structure of rule elements that are persisted in an xml configuration file. Rules are defined and attached to visual graph objects at runtime and constrain their behaviors.

We strongly suggest that you read the *Elements* chapter of the **Framework Manual**, which covers rules and all possible rule attribute configurations. These attributes are described in great detail and will provide you with answers to the many questions you might have.

Chapter 7. Layout

This tutorial provides *Java Application Programmers* exposure to an important and convenient feature of the Tensegrity Graph Framework. Here we give you a design and programming introduction to layouts - algorithms which can be applied to visual graphs so that they may be automatically arranged for you.

Without a doubt, automatic layout functionality is a time-saving feature of this graph framework. Without it, you would spend countless hours manually arranging your graphs, only to find yourself repeating the mundane task whenever you insert or delete a new element.

7.1. Tutorial At A Glance

The table below gives you some important information about this tutorial.

Table 7.1. Tutorial Aspects

Tutorial Aspect	Tutorial Description
Approximate Duration	10 Minutes
Expected Outcome	An application sporting a menu that, when clicked, applies a particular layout algorithm to a `GraphView`. This example gives the *Java Application Programmer* an initial exposure to different layout components that can be easily applied to visual graphs.
Source Files	The tutorial examples come with two source packages containing the java and configuration files needed to compile and run them. These are "com.tensegrity.firststeps.layout" and "com.tensegrity.firststeps.swt.layout" respectively for the AWT/Swing-based and SWT-based examples.
Creating Files	Not applicable in this tutorial
Modifying Files	Not applicable in this tutorial

7.2. Running The Tutorial Example

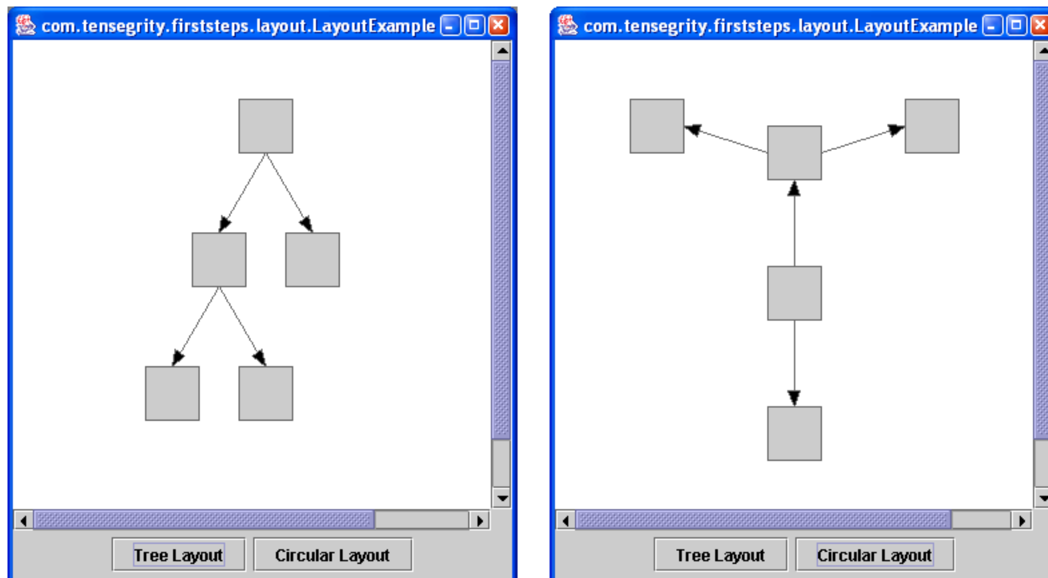
To run the tutorial example, please do the following in Eclipse:

1. Select package "`com.tensegrity.firststeps.layout`" or "`com.tensegrity.firststeps.swt.layout`" with the left mouse button.
2. Select the menu "Run".
3. Select the child menu "run as".
4. Select "Java Application"
5. The main method from the "Main.java" compiled class will be executed.
6. In the running application, press one of the "Layout" buttons. Move some nodes around and press again one of the buttons.

Note

Running the SWT-based example as described above you will probably encounter an error of this type: “java.lang.UnsatisfiedLinkError: no swt-win32-3062 in java.library.path”. Please refer to the note in section “Running The Tutorial Example” of chapter “Hello Node” in order to see how to overcome this issue.

Figure 7.1. Screenshot LayoutExample



Tree and Circular Layout Types

7.3. LayoutController

In this section, you will learn what a `LayoutController` is and how to create and apply one to a `VisualGraphView`.

The layout of a `VisualGraphView` is delegated to a `LayoutController` component. This is not the same kind of “controller” that is described in the MVC [14] tutorial in this manual. Rather it is an engine that manages the state of a layout activity, both during and after the completion of a visual graph. Moreover, the controller internally delegates certain responsibilities to a specialized `Layout` component that callers may specify using predefined string constants (also known as types).

Classes which implement the `VisualGraphView` interface provide a drawing area for visual graphs and as such utilize a `LayoutController` to automatically position visual graph elements relative to each another. Once you have retrieved this `LayoutController` from a view, you may set the layout “type” and/or directly call its *apply()* method to trigger the layout task. Remember to follow through with a call to the *updateView()* method, as shown in this example:

Example 7.1. Apply a Circular Layout to a VisualGraphView

```
// get a reference to the layout controller
```

```
GraphLayoutController layoutController =
  (GraphLayoutController) graphView.getLayoutController();

// actually apply the layout
layoutController.apply(GraphLayoutController.Circular);

// center the graph in the middle of the view
VisualOperations.translateViewportAccordingToGraph(graphView);
```

In the example above, the layout type “Circular” is passed in a separate call and specifies a specific layout from the *Tensegrity Graph API*. This type constant is used to internally instantiate a specific `Layout` component which does the actual layouting. These layout types can also be contextualized with specific parameters called a “layout context”. Please see the following sections and our javadocs for more complete information about these topics.

Example 7.2. Method that creates a `GraphLayoutController`

```
// create a layout controller
GraphLayoutController layoutController =
    new GraphLayoutController();

// disable animation
layoutController.enableAnimation(false);
```

As the above example shows, a `LayoutController` instance is created by instantiating a class derived from the abstract base class `LayoutController`. The `GraphLayoutController` instance is one of four out-of-the-box concrete `LayoutController` classes which provide a specific kind of runtime layout control. Please read our javadocs for more detailed information about the `LayoutController` base and derived classes.

When working with a `VisualGraphView`, it is necessary to assign a specific `LayoutController` to it. Below you can see just how to do that.

Example 7.3. Assign a GraphLayoutController to a VisualGraphView

```
// set the newly created layout controller
graphView.setLayoutController( layoutController);
```

7.4. Layout Context (Type Configuration)

After reading this section, you will understand what we mean by a layout context.

The *Tensegrity Graph API* uses various layout engines which we described in the previous sections. We call these engines “controllers” and they are realized by classes that implement the `LayoutController` interface. The type of layout, however, is implemented by any class that implements the `Layout` interface. Layouts are classes that position visual objects according to some conceptual model. Such models include hierarchies, trees and others with names that sound rather abstract.

Each `Layout` object, therefore, represents a specific layout model or type. Moreover, it can be configured with parameters that affect the exact manner in which the specific `Layout` is conducted.

We call this layout type configuration a “Layout Context” for short. The context is nothing more than a number of attributes that parameterize the type. A list of all configuration possibilities can be found in the *Layout* chapter inside the **Framework Manual**. We urge you to read that chapter when you have a need to customize a particular layout from our framework.

7.5. Summary

In this chapter you learned about the classes which take care of the automatic layout features of a visual graph. There are a lot of details to consider when customizing a particular layout for an application. We therefore recommend that you examine the source code for the sample application we have provided and then read the *Layout* chapter inside the **Framework Manual**. There you will discover thorough descriptions of all `Layout` types as well as the many attributes that make up each layout's context.

Chapter 8. Event Handling

This tutorial provides *Java Application Programmers* exposure to some of the event-related classes in the Tensegrity Graph Framework. Specifically, we provide you with a tutorial example which allows you to view events as you interact with a sample application.

8.1. Tutorial At A Glance

The table below gives you some important information about this tutorial.

Table 8.1. Tutorial Aspects

Tutorial Aspect	Tutorial Description
Approximate Duration	10 Minutes
Expected Outcome	An application showing a number of views that the user should interact with. This example gives the *Java Application Programmer* an additional event console which shows what kinds of events are raised when the user performs some action with the mouse or keyboard.
Source Files	The tutorial examples come with two source packages containing the java and configuration files needed to compile and run them. These are "com.tensegrity.firststeps.event" and "com.tensegrity.firststeps.swt.event" respectively for the AWT/Swing-based and SWT-based examples.
Creating Files	Not applicable in this tutorial
Modifying Files	Not applicable in this tutorial

8.2. Running The Tutorial Example

To run the tutorial example, please do the following in Eclipse:

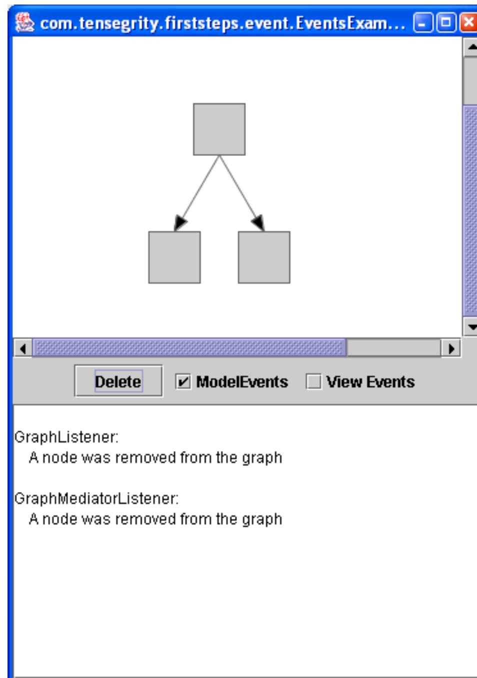
1. Select package "`com.tensegrity.firststeps.hellonode`" or "`com.tensegrity.firststeps.swt.hellonode`" with the left mouse button.
2. Select the menu "Run".
3. Select the child menu "run as".
4. Select "Java Application"
5. The main method from the "Main.java" compiled class will be executed.
6. In the running application, use the mouse and keyboard to interact with the graph view. While doing so, read the event names as they appear in the console.

Note

Running the SWT-based example as described above you will probably encounter an error

of this type: “java.lang.UnsatisfiedLinkError: no swt-win32-3062 in java.library.path”. Please refer to the note in section “Running The Tutorial Example” of chapter “Hello Node” in order to see how to overcome this issue.

Figure 8.1. Events Example Screenshot



The screenshot above shows the “Events” application after having pressed the “Delete” button two times. Because the “View Event” checkbox is unselected, no view events are logged.

8.3. Event *Registration*

In this section, you will learn about graph events and how to register for individual element and graph event notifications.

8.3.1. `GraphEvent` Concepts

An event is an object which encapsulates a change in the state of some object. Events are used to propagate these changes to yet other objects interested in responding to them. This design allows for a weak coupling of object classes that would otherwise know too much about each other, making software change difficult or even impossible.

In good object-oriented design, objects interested in receiving event notifications should play one or more “listener” roles. These roles are usually played by classes implementing listener interfaces, which are implemented on top of the business or plumbing interfaces in the application domain space.

The *Tensegrity Graph API* provides many different event and listener interfaces and classes that you may take advantage of. Listener interfaces are fixed plumbing in the *Tensegrity Graph API* and therefore you cannot substitute them with your own. You may, however, implement your own classes that

uniquely respond to any and all events.

Please have a look at the javadocs and study the numerous event class names and responsibilities. You will see events are categorized into element types and the kinds of state changes that are possible within these types. The package “com.tensegrity.graph.event” contains all event and listener interfaces and classes.

8.3.2. `GraphListener` Concepts

A `GraphListener` can be any business or non-business object interested in receiving event notifications. They are called “listeners” because that is the coupling technique. An object which throws an event knows about listeners awaiting notification only and not about some specific domain-related class or interface, such as JButton, XYZBusinessObject or anything else that is extraneous to this design pattern.

Once a listener has been registered or added to an event-throwing object, that listener will receive notifications when state changes occur. A “notification” is what happens when an event is passed to a registered listener. An actual event object does not have to be passed, however. Often it is sufficient when a method in a listener interface is invoked.

8.3.3. Listener Registration

We now give you some sample code which shows how to create and attach a `GraphListener` to an existing `Graph` instance.

Example 8.1. GraphListener Registration

```
// register for graph model events
graphModel.addGraphListener(new GraphAdapter()
{
  public void nodeRemoved(GraphEvent graphevent)
  {
    logger.logGraphListener(graphevent);
  }

  public void edgeRemoved(GraphEvent graphevent)
  {
    logger.logGraphListener(graphevent);
  }
});
```

The code above shows the instantiation of a new `GraphAdapter` instance, which implements the `GraphListener` interface.

Registration is the process of calling the *addGraphListener* method of a `Graph` object, passing it the new listener implementation.

8.4. Event *Veto*

In this section, you will learn how programmatically veto events.

To veto an event means a listener is able to stop or cancel an event or rather the action which created it. A special listener interface `VetoableGraphListener` defines an object that is able to receive graph events and subsequently call a veto.

In order to veto an event, a veto listener must throw a new instance of class `GraphEventVetoException`. A `VetoableGraphListener` implementation should never process an event like a normal `GraphListener` does.

Example 8.2. Event Veto Source Listing

```
void registerVetoListeners()
{
  // register for graph model events
  graphModel.addVetoableGraphListener(new VetoableGraphListener()
  {
   public void vetoableNodePreRemove(GraphEvent graphevent)
     throws GraphEventVetoException
   {
     logger.logVetoableGraphListener(graphevent);

     // now give it a veto
     throw new GraphEventVetoException(
       "Event work off stopped", graphevent);
   }

   public void vetoableEdgePreRemove(GraphEvent graphevent)
     throws GraphEventVetoException
   {
     logger.logVetoableGraphListener(graphevent);
   }

   public void vetoableEdgePreSplit(GraphEvent graphevent)
     throws GraphEventVetoException
   {
     logger.logVetoableGraphListener(graphevent);
   }

   public void vetoableNodePreAdd(GraphEvent graphevent)
     throws GraphEventVetoException
   {
     logger.logVetoableGraphListener(graphevent);
   }

   public void vetoableEdgePreAdd(GraphEvent graphevent)
     throws GraphEventVetoException
   {
     logger.logVetoableGraphListener(graphevent);
   }

   public void vetoableNodePreDeleteEdgeCascade(GraphEvent graphevent)
     throws GraphEventVetoException
   {
     logger.logVetoableGraphListener(graphevent);
   }
  });

  // it's the same for the view ...
}
```

The above implementation of a new `VetoableGraphListener` instance shows the various methods that can throw a veto exception, thereby canceling the atomic operation taking place in the model.

Since there are also some atomic operations that can be made upon the graph view you can find a corres-

ponding interface called `VetoableVisualGraphListener`. An implementation of this interface is registered at a `VisualGraphView`. It should work in the same way as the model counterpart does except the it throws a `VisualGraphEventVetoException`.

8.5. “Big Brother” *Registration*

In this section, you will learn how to register for all event notifications.

A special interface/class pair is available in the *Tensegrity Graph API* which allows you to register for all events related to a `Graph` or a `VisualGraphView`.

Whenever you add a `GraphEventMediatorListener` to a `Graph`, you gain access to all occurring `Graph`, `Node` and `Edge` events.

Example 8.3. GraphEventMediatorListener Source Listing

```
void registerMediatorListener()
{
  graphModel.addEventMediatorListener(
          new GraphEventMediatorAdapter()
    {

    // add mediator listener to the graph model
    public void nodeRemoved(GraphEvent graphevent)
    {
      logger.logGraphMediatorListener(graphevent);
    }

    public void edgeRemoved(GraphEvent graphevent)
    {
      logger.logGraphMediatorListener(graphevent);
    }

    public void edgeSplit(GraphEvent graphevent)
    {
      logger.logGraphMediatorListener(graphevent);
    }

    public void nodeDeleteEdgeCascade(GraphEvent graphevent)
    {
      logger.logGraphMediatorListener(graphevent);
    }
  });

  // add mediator listener to the graph view
  graphView.addVisualEventMediatorListener(
    new VisualGraphEventMediatorAdapter()
    {
      public void visualNodeRemoved(VisualGraphEvent visualgraphevent)
      {
        logger.logVisualGraphMediatorListener(visualgraphevent);
      }

      public void visualEdgeRemoved(VisualGraphEvent visualgraphevent)
      {
        logger.logVisualGraphMediatorListener(visualgraphevent);
      }

      public void layout(VisualGraphEvent visualgraphevent)
```

```
        {
          logger.logVisualGraphMediatorListener(visualgraphevent);
        }

        public void visualEdgeSplit(VisualGraphEvent visualgraphevent)
        {
          logger.logVisualGraphMediatorListener(visualgraphevent);
        }
  });
}
```

The above implementation of class `GraphEventMediatorAdapter` provides an object which listens for all types of `Graph` events. Please view our javadocs for more information about this implementation class and the `GraphEventMediatorListener` interface.

8.6. Event *Logging*

If you have a look at our previous examples, you will see that many implemented methods log event descriptions to the console. All graph event classes implement interface `GraphRootEvent` respectively `VisualGraphRootEvent` which define the method *getDescription()*. This method returns a textual description of the occurred event.

Example 8.4. Event Logging Source Listing

```
// Assuming you are using a logger like log4j
logger.log(anyEvent.getDescription());
```

8.7. Summary

In this chapter you learned about some of the more important event-related classes in the *Tensegrity Graph API*. Listeners, listener registration, veto listeners and other topics were introduced in order to to give you a basic understanding of how things work.

For a more complete introduction to all event-related classes, we recommend that you view the the *Event Handling* chapter inside the **Framework Manual** as well as the javadocs distributed separately.

Chapter 9. Skeleton

Putting it all together

This tutorial provides *Java Application Programmers* with a standalone application that integrates the functionality and domain entities discussed in the previous tutorials.

This application is built on top of the Tensegrity *Skeleton* Framework, a generic application framework complete with menus, toolbars, navigators, attribute tables and other container windows.

The Tensegrity *Skeleton* Framework is designed to support different windowing toolkits. These include the Swing API and AWT (included in the Java Runtime Environment) and JFace and SWT (which is shipped with the Eclipse Development Environment).

In order to provide support for all of these different windowing toolkits, the Tensegrity *Skeleton* Framework provides a number of abstract interfaces and concrete toolkit implementations. In this tutorial, we aim to give you a fundamental understanding of our application framework by making use of the Swing API implementation.

The various sections of this tutorial will discuss the key features and some implementation details of the *Skeleton* classes and interfaces. After you have finished reading, we recommend that you reference the *Creating Applications* chapter inside the **Framework Manual**. There you will find a more complete description of the *Skeleton* Framework API.

This custom application will display a subset of the standard windows available in the framework, specifically a

- Repository View
- Layout View
- Navigator View
- Graph View
- Menu and Toolbar

9.1. Tutorial At A Glance

The table below gives you some important information about this tutorial.

Table 9.1. Tutorial Aspects

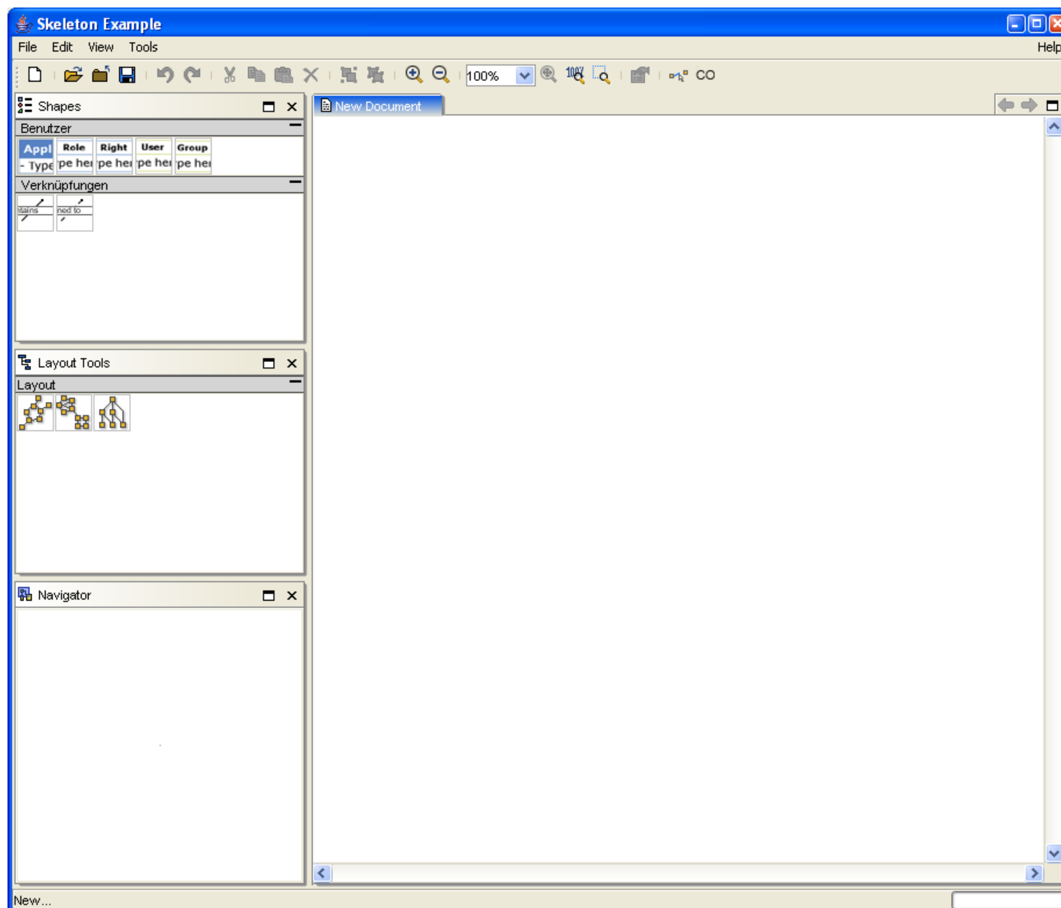
Tutorial Aspect	Tutorial Description
Approximate Duration	90 Minutes
Expected Outcome	A Swing application showing a number of views that the user should interact with. This example gives the *Java Application Programmer* exposure to and source code using the most important features of the framework.
Source Files	The tutorial example comes with with the source package “com.tensegrity.firststeps.skeleton”, which contains the Java and configuration files needed to compile and run the example.
Creating Files	Not applicable in this tutorial
Modifying Files	`elements.xml`, `geometry.xml`, `styles.xml`

9.2. Running The Tutorial Example

To run the tutorial example, please do the following in Eclipse:

1. Select the “skeleton” package with the left mouse button.
2. Select the menu “Run”.
3. Select the child menu “run as”.
4. Select “Java Application”
5. The main method from the “Main.java” compiled class will be executed.
6. In the running application, use the mouse and keyboard to interact with different views, menus and toolbar buttons. Drag and drop the domain elements from the repository container into the open document. Use the layout tools to rearrange the visual graph. Finally, discover the different commands implemented inside the menu and toolbar.

Figure 9.1. Skeleton Example Screenshot



9.3. Application Frame

This section will provide you with a general explanation of the windows composed in a *Skeleton*-based application as well as an understanding of where the application places them.

An application based upon the *Skeleton* Framework must supply different windows that represent the different views and editors of the graphical user interface. In every case, an application has a top-level window that *contains* and manages any number of child windows. A child window may, for instance, be a menubar, toolbar, repository view, layout toolbox, navigator view or an editor to model or manipulate a graph.

Within the Tensegrity *Skeleton* Framework, each window is defined by a unique public interface. This interface defines the window type as well as the methods needed to manage it.

The top-level window is an implementation of the interface `ApplicationFrame` and consists of components and containers. Components implement the `BaseComponent` interface and represent things like menubars, toolbars and statusbars. Containers, on the other hand, implement the `Container` interface and represent views and editors like the layout toolbox, the repository view and the navigator view.

The `ApplicationFrame` is divided into four distinct locations:

1. Menubar Component. The location of the menubar is directly beneath the title of the top-level window.
2. Toolbar Component. The location of the toolbar is usually just below the menubar.
3. Statusbar Component. The location of the statusbar is at the bottom of the top-level window.
4. Containers. Between the toolbar and statusbar are located all concrete *Containers* .

All child windows (Components and Containers) are requested by the `ApplicationFrame` calling different methods on itself. These methods must be overridden whenever a concrete `ApplicationFrame` class is implemented to ensure that the correct concrete child window types are returned.

Note

Concretion in the context of the Tensegrity *Skeleton* Framework is done in two steps. Firstly, you specify the windowing toolkit implementation of your choice. In this example, it is the Swing API. Secondly, you must derive from this implementation base class. Our `ApplicationFrame` implementation class therefore belongs to this class hierarchy: `ApplicationFrame ==> SwingApplicationFrame ==> ExampleApplicationFrame`

9.3.1. The Menubar Component

The menubar is represented by a concrete implementation of the `MenuBar` interface. It is requested from an `ApplicationFrame` instance through its method `getMenuBar()`.

Each item of a menu embeds a `Command` object. `Command` classes are explained in a later section in more detail.

A *Skeleton*-based application creates a set of default menus such as File, Edit, View, Tools, Window and Help. Each of these default menus references a default command object. During application startup, default menus are added to the `ApplicationFrame` instance automatically.

In order to modify the default menus or to add your own menu to the menubar, you have to override the corresponding `LauncherTask`. In the case of a custom menu, you will have to write a command class first, then create a `LauncherTask` to create and insert the menu item. `LauncherTask` classes are explained in more detail in a later section.

9.3.2. The Toolbar Component

A toolbar is represented by a concrete implementation of the `ToolBar` interface. It is requested from an `ApplicationFrame` instance through its method `getToolbars()`. As the method name implies, there can be more than one `ToolBar` instance. Each button in a toolbar embeds a `Command` object. `Command` classes are explained in a later section in more detail.

A *Skeleton*-based application creates a default toolbar. If you wish to hide it you should return *null* when overriding the method `getToolBars()` in your `ApplicationFrame` derived class. It is also possible to modify the default toolbar. To do so, you will have to write a command class first, then create a `LauncherTask` to create and insert the toolbar button. `LauncherTask` classes are explained in more detail in a later section.

9.3.3. The Statusbar Component

The statusbar is represented by a concrete implementation of the `StatusBar` interface. It is requested from an `ApplicationFrame` instance through its method `getStatusBar()`.

The statusbar displays a description the currently selected and executed command as well as other useful information. It is automatically linked to the commands embedded in the menubar and toolbar. A statusbar is provided by default. The statusbar may be hidden or modified in the same way as the menubar and toolbar by returning *null* in the `getStatusBar()` method.

9.3.4. Containers

A `Container` is an element of the graphical user interface that cannot be handled in the same way as a *Component* (menubar, toolbar, statusbar). Containers represent complex views and editors that depend on the document types handled by the application.

Note

In the *Tensegrity Graph API*, a `Container` is a word used in a graphical context and is not the same thing as the word “Container” used in a other programming contexts, such as in pure AWT or in a J2EE or another IoC [1] application. Here we are referring to the main application window's many embedded graphical panels. Each panel is capable of holding various tools or complex elements and is therefore called a `Container`, thereby assisting users with the selection of repository elements, navigation, viewing and setting attribute values and more.

Examples of containers needed for a graph application include a repository view, from which users may drag elements and drop them into a document, a navigator, which provides a miniaturized overview of the entire graph, and a layout toolbox, in which a user may select different layout types, change their context attributes and apply them to a graph document.

The Tensegrity *Skeleton* Framework defines several standard containers that are needed or helpful in a graph application. The following list gives an overview of these standard containers (some of these are

[1] Inversion of Control (IoC) is an architectural pattern in which parent containers manage the lifecycle and dependencies of child components.

used in this tutorial's application, others not):

1. A RepositoryContainer, which provides the elements a graph may consist of.
2. A LayoutToolsContainer, which provides users with several layout algorithms.
3. A NavigatorContainer, which provides users with an overview of the entire graph, allowing them to easily navigate through the graph or choose an appropriate zoom factor.
4. An OutlinerContainer, which provides users with an outline of the currently active document.
5. An AttributeTreeContainer, which provides users with the attributes of the currently selected elements, allowing them to be modified.

Finally, it is possible to create a custom Container by implementing this interface and introducing another concrete container type needed for your specialized application. This is not explained in this tutorial, however. Please refer to the **Framework Manual** for detailed information on this subject.

9.4. Commands

Within the *Tensegrity Graph Framework*, a command class implements an action which is either application-specific or document-specific. 'New Document', 'Save Document' and 'Close Document' are examples of application-specific commands while 'Cut', 'Copy', 'Paste' and 'Delete' are document-specific commands.

By separating Command objects from the UI elements, it is possible for a *Skeleton*-based application to use the same command objects from within a menubar and a toolbar. Furthermore, Command objects can be used in application code that you write elsewhere.

The Tensegrity *Skeleton* Framework provides a lengthy set of standard Command classes for a graph document. These classes may be used right away. It is also possible, however, to write your own Command classes by implementing the Command interface.

When implementing the Command interface, the code which does the action has to be placed inside the method perform(). The following example illustrates this.

Example 9.1. The ToggleAssignEdgeModeCommand

```
/**
 * Command to toggle the edge types used when in 'edge-creation-mode'.
 * It toggles between 'contain' and 'assigned to' edges.
 */
public class ToggleAssignEdgeModeCommand extends GraphDocumentCommand
{
  /** unique constant to look up the local string for this command */
  public final static String MENU_EDIT_TOGGLE_EDGE_MODE =
        "menu.edit.toggle_assign_edge";

  // constants specifying the geometries and styles
  // for the different edge types
  private final static String ASS_EDGE_GEO_DES =
        "AssignEdgeGeometryDescriptor";
  private final static String ASS_EDGE_STY_DES =
        "AssignEdgeStyleDescriptor";
  private final static String CON_EDGE_GEO_DES =
```

```
        "ContainEdgeGeometryDescriptor";
  private final static String CON_EDGE_STY_DES =
        "ContainEdgeStyleDescriptor";

  private boolean assignMode=false;

  /**
   * Constructor specifying the <code>ApplicationFrame</code> and a
   * unique id for this command.
   * @param applicationFrame the <code>ApplicationFrame</code> this
   * command is used within.
   * @param command an unique id for this command.
   */
  public ToggleAssignEdgeModeCommand(
      ApplicationFrame applicationFrame, String command)
  {
      super(applicationFrame, command);
  }

  public void perform(Object args)
  {
    super.perform(args);
    // get a reference to the active graph document
    GraphDocument graphDocument =
      (GraphDocument)
          getApplicationFrame().getMDIComponent().getActiveDocument();

    VisualGraphView vgv =graphDocument.getGraphPanel().getVisualGraph();

    // set corresponding geometries and styles
    vgv.setDefaultEdgeGeometry(
        assignMode ? CON_EDGE_GEO_DES : ASS_EDGE_GEO_DES);
    vgv.setDefaultEdgeStyle(
        assignMode ? CON_EDGE_STY_DES : ASS_EDGE_STY_DES);

    // toggle mode field
    assignMode =!assignMode;
  }
}
```

Whenever a `Command` object is created, it becomes available to other parts of your application code. Every `Command` object is registered with an application wide registry called the `CommandRegistry`. The following code shows how a `Command` object may be accessed:

Example 9.2. Accessing a created Command

```
// get the reference to the command registry
final CommandRegistry commandRegistry =
    getApplicationFrame().getCommandRegistry();
// get the reference to a certain command
Command command = commandRegistry.get(
    ToggleAssignEdgeModeCommand.MENU_EDIT_TOGGLE_EDGE_MODE);
```

9.5. Preferences

The first step on working with customized preferences is to set up an xml-file containing all desired ap-

plication settings. You should copy the basic `defaultprefs.xml` from the skeleton project and edit it to fit your needs. Setting up this file is all you have to do to provide persistent settings and have a comfortable way of editing them via the *Preference* Dialog. Let's have a look at the structure of this file:

Example 9.3. An example `preferences.xml` file

```
<list>
 <!-- a first group / tab: GUI related -->
 <set name="prefs.tab.gui">
  <set name="prefs.grp.common">
   <attribute name="prefs.common.language" value="Deutsch">
    <constraint>(value in ('Deutsch' | 'English')</constraint>
   </attribute>
  </set>
  <set name="prefs.grp.surface">
   <attribute name="prefs.surface.showtooltips"
              type="Boolean" value="true"/>
  </set>
 </set>

 <!-- a second group / tab: files and folders -->
 <set name="prefs.tab.paths">
  <set name="prefs.grp.paths">
   <attribute name="prefs.path.lastdir"
         type="com.tensegrity.gui.template.DirectoryLocationAttribute"
         value="" />
  </set>
 </set>
</list>
```

You can access the `Preference` instance of your application with a call to method `getPreferences()` defined in interface `Application`. For example, if you want to figure out which language was set you call

```
application.getPreferences().get("prefs.common.language");
```

Section Loading Preferences [54] shows how to load a custom *preferences* file with the help of a `LaunchTask`.

9.6. Launch Tasks

After reading this section, you should understand what a `LaunchTask` is and how it is used in the context of an `ApplicationFrame`.

Basically speaking, `LaunchTask` implementations are used to decouple application-specific initialization code from the `ApplicationFrame` class, instances of which are needed for any standalone application based on the *Skeleton* framework.

Even though the `ApplicationFrame` class acts like a controller, it should not be extended with specialized initialization code, which is only required once during application startup.

The `ApplicationFrame`-derived class **`SwingApplicationFrame`** decouples this specialized code by reading a list of “launch tasks” when starting up and executing them sequentially. You may override the `getLauncherTask()` method of the parent `SwingApplicationFrame` class and re-

turn a new list. Otherwise, you may use the default list without any modification.

The *Skeleton* provides a basic framework for all currently required `LaunchTask` types. There are LaunchTasks to load files (`LoadFileLaunchTask`), LaunchTasks to create and initialize ToolBars (AbstractToolBarLaunchTask), Menus (`AbstractMenuLaunchTask`), as well as LaunchTasks to create Commands (`CommandLaunchTask`). Either use these superclasses as the basis for your own implementations or extend the default `LaunchTask` implementations of the Skeleton framework.

The following sections show the `LaunchTask` implementations used in this chapter's sample application.

9.6.1. Loading Preferences

Application preferences may differ for every *Skeleton*-based application you create. These preferences are configured in an xml file and the preference dialog of you running application will reflect the structure of this file. You will return the file's path in the method `getDefaultPreferences()` of class `CustomPreferencesLaunchTask`. You can always count on a similar look and navigation structure when setting preferences.

In order to store the settings of a user, a file needs to be written to a directory on the host system. This file is located in the directory {user.home} and the file name is returned in the method `getFilename()`. The following code snippet shows how these customizations are made in the sample application.

Example 9.4. Overridden Methods for Files

```
/**
 * This LauchTask specifies the preferences to use.
 */
public class CustomPreferencesLaunchTask
    extends LoadPreferencesLaunchTask
{

  /**
   * Constructor specifying the ApplicationFrame.
   * @param applicationFrame the ApplicationFrame
   */
  public CustomPreferencesLaunchTask(ApplicationFrame applicationFrame)
  {
    super(applicationFrame);
  }

  /**
   * Returns the path to the initial preference file relative
   * to the load class specified in method getLoadClass().
   */
  protected String getDefaultPreferences()
  {
    return "../resource/xml/defaultprefs.xml";
  }

  /**
   * All changes to the initial application settings get written
   * out to this file. It will be located in folder {user.home}.
   */
  protected String getFilename()
  {
    return "skeleton_example.prefs";
  }
```

```
  /**
   * The class to use to load the initial preference file.
   */
  protected Class getLoadClass()
  {
    return Main.class;
  }
}
```

You may wish to refer to the **Framework Manual** for more information on this special topic.

9.6.2. Setting Up Commands

In order to create custom commands for a *Skeleton*-based application you can provide your own `CommandLaunchTask` implementation and override the `perform()` or `postCreate()` method. In our sample application we used the latter approach:

Example 9.5. Overridden `postCreate()` Method

```
// this method is called after the common commands were set up
public void postCreate(Object object)
{
    // register circular...
    new LayoutCommand(getApplicationFrame(), LAYOUT_CIRCULAR_IDS,
            GraphLayoutController.Circular);

    // ..., energy layout...
    new LayoutCommand(getApplicationFrame(), LAYOUT_ENERGY_IDS,
            GraphLayoutController.Energy);

    // ...and hierarchical top-bottom
    new LayoutCommand(getApplicationFrame(), LAYOUT_HIERARCHICAL_IDS,
        GraphLayoutController.HierarchicalToBottom);

    // toggle the edge types used in edge creation mode
    new ToggleAssignEdgeModeCommand(
            getApplicationFrame(),
            ToggleAssignEdgeModeCommand.MENU_EDIT_TOGGLE_EDGE_MODE);
}
```

9.6.3. Menu and Toolbar

The Skeleton creates a set of default menus called “File”, “Edit”, “View”, “Tools”, “Window” and “Help”. Each of these menus has a corresponding default command. To add a custom command to the menubar or toolbar of the application, you have to create it first. In the next step, you can add the command to the UI. For the sample application this is done in class `CustomToolbarLaunchTask`.

Example 9.6. Adding Commands to the Toolbar

```
// Add edge creation mode toggle button to toolbar.
public void postCreate(Object object)
{
```

```
  SwingToolBar toolbar=
      (SwingToolBar) getApplicationFrame().getToolBars()[0];

  // Edge creation mode
  final String ecIconURL ="icons/menu/edgeCreation.png";
  addToToolBarToggleButton(
    toolbar,
    GUIItem.MENU_EDIT_TOGGLE_EDGE_CREATION,
    ecIconURL, ecIconURL, ecIconURL, ecIconURL,
    GUIResource.class);

  final String conModeIconURL = "../resource/images/con_mode.png";
  final String conModePressedIconURL = "../resource/images/ass_mode.png";

  addToToolBarToggleButton(
      toolbar,
      ToggleAssignEdgeModeCommand.MENU_EDIT_TOGGLE_EDGE_MODE,
      conModeIconURL,
      conModePressedIconURL,
      conModeIconURL,
      conModePressedIconURL,
      Main.class);
}
```

9.6.4. Element Definitions

The element definitions describing geometries, styles and rules are specified in a file which is loaded in the method `perform()` of class `LoadElementDefinitionsLaunchTask`.

Example 9.7. Loading Element Definitions

```
/**
 * Loads the graph element definitions from the resource file
 * "geometry.xml", "styles.xml", "rules.xml" and "elements.xml".
 * Afterwards the corresponding pools are filled with this definitions.
 */
public class LoadElementDefinitionsLaunchTask extends LaunchTask
{
  private final static String RES_PATH = "../resource/xml/";
  private ApplicationFrame appFrame;

  /**
   * Constructor specifying the ApplicationFrame to use.
   * @param applicationFrame Application context for this LaunchTask.
   */
  public LoadElementDefinitionsLaunchTask(
          ApplicationFrame applicationFrame)
  {
    this.appFrame= applicationFrame;
  }

  public String getName()
  {
    return "Loading Element Definitions";
  }

  public int run() throws Exception
  {
```

```
    GraphGeometryService.loadGeometriesFromResource(
      RES_PATH + "geometry.xml", ExampleApplicationFrame.class);

    StyleService.loadStylesFromResource(
        RES_PATH + "styles.xml", ExampleApplicationFrame.class);

    RuleRegistry rr =RuleRegistry.loadRulesFromResource(
        RES_PATH + "rules.xml", ExampleApplicationFrame.class);

    ((SwingApplicationFrame)appFrame).setRuleRegistry(rr);

    ElementService.loadElementsFromResource(
        RES_PATH + "elements.xml", ExampleApplicationFrame.class);

    return LAUNCHTASKSTATUS_SUCCESS;
  }
}
```

9.6.5. Populating the Repository Container

After the element definitions have been loaded, they can be placed inside a so-called `RepositoryItemPanel`. The following code snippet shows you how to do this. Please have a look at class `CustomRepositoryLaunchTask` in the sample application as well.

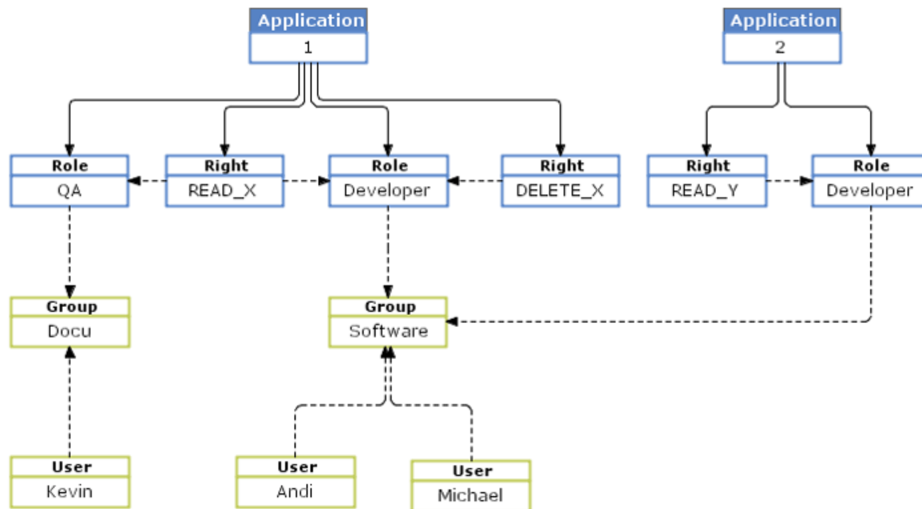
Example 9.8.

```
// Create Custom Node Repository
RepositoryItemPanel nodeRepIP= new RepositoryItemPanel(2, 2);
nodeRepIP.setBorder(BorderFactory.createEmptyBorder(2, 2, 2, 2));

final UIManager uiMan = getApplicationFrame().getUIManager();

VisualGraphObjectRepositoryItem tmpRepItem =
    new VisualGraphObjectRepositoryItem(
        getApplicationFrame().getDragContext(),
        ElementPool.get("AppNodeElement"),
        uiMan.getText("node.appl"));
tmpRepItem.setIcon(SwingUtil.getResourceIcon(
            "../resource/images/app_node.png",
            Main.class,
            tmpRepItem, false));
nodeRepIP.addJComponent(tmpRepItem);
```

9.7. Modeling Our Example

In this section we will model our “User Administration” example as introduced in the Introduction chapter. You should try to build a graph with a structure as shown in the following figure.

Figure 9.2. User Administration Graph



The repository of the sample application of this chapter holds all needed node and edge types. Here is a step-by-step guide for your first modeling steps:

1. Drag an *Application* node (the leftmost) from the repository to the graph document. Double-click in the lower part of this node an type "1". The editing process is terminated by clicking in an empty area of the document or by pressing **STRG** + **ENTER**.

2. Drag a *Right* and a *Rule* node (next to the *Application* node) into the graph document. Double-click in the lower part of these nodes and type "Read_X" for the *Right* node and "Developer" for the *Role* node.

3. Connect the *Application* node with the *Right* node. To do so, activate the *edge-creation-mode*. This is done by clicking the corresponding toggle button in the toolbar:

   Now move the mouse cursor over the *Application* node and drag to the *Right* node. You have created a *contain* edge. Do the same for the *Role* node.

4. Connect the *Right* and *Rule* node (in this direction) with an *assigned to* edge. To do so click on the toggle button labeled "CO" right from the *edge-creation-mode* toggle button. The label switches to "AS". Now drag your edge and a new *assigned to* edge is created.

You should have all you need to create the remaining part of the sample graph. Have fun!

9.8. Summary

Creating a Graph Framework from the ground up is a lot of work. We take it one step further by providing you with an out-of-the-box Application Framework that will help you create a standalone graph application within a very short time.

The *Skeleton* Framework achieves this goal by encapsulating the types of windows a typical graph application requires behind generic interfaces. Some of these interfaces already have default implementations that you may use without any customization.

We hope that this manual and all the tutorials in it have helped you answer your first and most obvious questions. If not, we would be pleased to hear from you so that we may answer those questions and improve this introductory manual.

Our big manual, the **Framework Manual**, is packed full of information that we could not possibly cover in this one. We therefore urge you to look there if you are dealing with a specific aspect of this framework that is still unclear or has not been covered here.

Many thanks for choosing Tensegrity!

Bibliography

Books

[GoF99] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Copyright © 1995 Addison-Wesley longman, Inc.. Brian H. Kernighan. 0-201-63361-2. Addison-Wesley Publishing Company. *Design Patterns*.

[Java-Specification] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. Copyright © 2000 Sun Microsystems, Inc.. 0-201-31008-2. Addison-Wesley Publishing Company. *The Java Language Specification Second Edition*.

Manuals

[TenMan] Copyright © 2004, 2005 Tensegrity Software GmbH. *The Tensegrity Graph Framework*.

Web

[Eclipse] *The Eclipse Platform [http://www.eclipse.org/]* .

[TenSW] *Tensegrity Software [http://www.tensw.com/]* .

Index

A
ApplicationFrame (see Skeleton)

C
ClientServerGraphController, 17
Commands, 51, 55
Composite, 24
Containers (see Skeleton)

E
Eclipse, 4
Edge, 11
EdgeRule (see Rules)
Element (see Repository)
ElementPool, 30
Event, 41
    Logging, 46
    Mediator, 45
    Veto, 43

G
Geometries (see Repository)
Geometry, 24
GeometryDescriptor, 24, 25
GeometryDescriptorItem, 25
GeometryItem, 24
GeometryPool, 26
Graph, 9
    Controller, 17
    Definition, 9
    Listener, 43
    Model, 8
    View, 9
GraphEvent, 42
GraphEventMediatorAdapter, 45
GraphEventMediatorListener, 45
GraphGraphModelFactory, 9
GraphLayoutController, 38
GraphListener (see Graph)

H
Hello Node, 7

I
Installation, 4

L
Launch Task (see Skeleton)
Layout, 37
    LayoutController, 38
Listener
    Registration, 43

M
Mediator (see Event)
Menubar (see Skeleton)
ModelBasedGraphController, 17
MVC, 14

N
Node, 10
NodeRule (see Rules)

P
Preferences, 52
    Loading, 54
Primitive, 24

R
Repository, 21, 29
    elements.xml, 29
    geometry.xml, 22
    rules.xml (see Rules)
    styles.xml, 27
RuleRegistry (see Rules)
Rules, 31
    for Edges, 34
    for Graphs, 32
    for Nodes, 33
    Registry, 35

S
Skeleton, 47
    ApplicationFrame, 49
    Containers, 50
    Launch Task, 53
    Menubar, 49
    Statusbar, 50
    Toolbar, 50
Statusbar (see Skeleton)
StyleDescriptor, 27
StyleItem, 27
Style Pool, 28
Styles (see Repository)

T
Toolbar, 55 (see Skeleton)

V
VetoableGraphListener, 43
VisualEdge, 12
VisualGraphView, 9
VisualNode, 11