Informatikprojekt

Software Composition Styles: Mixins for Piccola

Daria Spescha

Institut für Informatik und Angewandte Mathematik
University of Bern, Switzerland

Supervised by Prof. Dr. Oscar Nierstrasz

March 2004

Abstract

Piccola is a language to compose software components. It is designed to support different composition styles. Furthermore, mixins and mixin layers are a powerfull composition style. So it was an interesting question if mixins and mixin layers could be implemented with Piccola. To prove this I first simulated the structure for mixins and then implemented a small mixin layer library for graph manipulation.

Contents

1 Introduction **3**
1.1 About this document . . . 3
1.2 Resources . . . 3

2 Theory: What Are Mixins and Mixin Layers? **4**

3 A Short Introduction to Piccola **5**

4 Modelling Mixins and Mixin Layers in Piccola **6**
4.1 Modelling Classes with Piccola . . . 6
4.2 Mixins . . . 8
4.3 Components and MixinLayers . . . 9

5 Example: Mixin-Library for Graphs **11**
5.1 Base Classes and Components . . . 11
5.2 Mixins and Mixin Layers . . . 11
5.3 Visualisation with JGraph . . . 12
5.4 Using the Library . . . 12

6 Conclusion **14**

Appendices **15**

A Source Code for Mixins in Piccola **15**
A.1 Class Simulation . . . 15
A.2 Mixin . . . 16
A.3 Component . . . 17
A.4 Mixin Layer . . . 17

B Source Code of the Graph Library **18**
B.1 Base Classes . . . 18
B.2 Base Components . . . 20
B.3 Graph Traversal . . . 21
B.4 Vertex Numbering . . . 23
B.5 Cycle Checking . . . 23
B.6 JGraph Adaptation . . . 25
B.7 Visualisation . . . 27
B.8 Examples . . . 30

1 Introduction

Piccola is a composition language based on Java (and also implemented in Java). It was developed at the "Institut für Informatik und Angewandte Mathematik" at the University of Berne. Piccola is intended to be a scripting language for easy composition of software components. (For further details see section 3).

There are many composition styles, and some of them have already been modelled with Piccola (cf. [3]). An interesting style to model with Piccola is composing components with mixins and mixin layers (for definitions see 2 on page 4). A small mixin example for Piccola already exists (cf. [3, pp. 6] and [4, pp. 23]), but this is just a very small trial. Therefore, it was interesting to see if this works with more complex structures. To test this, I tried to develop a small library of components and mixin layers for graph manipulation in Piccola.

1.1 About this document

In section 2, I will first treat some theoretical aspects and introduce mixins and mixin layers. Then, I will give a short overview of JPiccola in section 3. Section 4 explains the implementation of the mixin structure in JPiccola. At last, I will exemplify the use of mixin layers by presenting the library of the graph components and mixins in section 5.

1.2 Resources

The mixin and mixin layers described in this paper are developed for JPiccola 3.7b and Java VM 1.4.2. For the visualisation I used JGraph 3.0 for Java 1.4. All extracts of sourcecode work with these resources.

2 Theory: What Are Mixins and Mixin Layers?

Mixins are classes without a fixed superclass. More precisely, mixins encapsulate funcionality without being bound to a superclass. Thus, a mixin can be instantiated with a variety of classes. By applying a mixin to a class A, a new class B is produced as subclass of class A but with the additional functionality defined by the mixin. A mixin may impose restrictions on the classes it is instantiated with, for example methods the class has to implement. Mixins are suitable for functions which are common for different classes, because you have to implement the functionality only once and you can then increment the classes by applying the mixin. One of the advantages of this technique is, that you can compose classes with exactly the functionality needed at the moment using some base classes and a mixin library.

Mixin Layers are mixins containing other mixins. Different from mixins, they are not applied to classes but to components. A **component** encapsulates a collaboration, i.e. it defines roles and assigns classes to them.

For an introduction to mixins and mixin layers read [1] and [2].

3 A Short Introduction to Piccola

Piccola is a composition language with minimal syntax. It depends on a host language. For this work, I used JPiccola which is based on Java. However there is also a version of Piccola based on Smalltalk, called SPiccola.

The basic elements of Piccola are **forms**, which basically are nested records. There are two main types of forms: Forms either bind labels to values or provide a service. The latter can be invoked, possibly with arguments. In the following example, `hello` is a form with bindings for the labels `greeting` and `do`, with `greeting` being bound to a value (`"Hello!"`) and `do` providing a service. Projection extracts bindings from forms, with a dot being the operator for this (like `hello.do()` which invokes the service of `do`).

```
hello =
    greeting = "Hello!"
    do: println greeting.

hello.do()
```

Hello!

Forms can be extended with other bindings, but they are immutable. Whenever it is necessary to update a value, for example in a while-loop, you must use variables. Piccola provides a simple implementation for variables. The service `newVar` creates a new variable for storing an arbitrary value, which can be set with `<-` and be retrieved by `*`.

Actually, JPiccola offers the possibility to (re)define operators like `+` or `*` for forms. For example for the variables decribed above, the operators `<-` and `*` were defined. The operator `$` is used equivalent to the function `toString()` in Java.

Piccola also provides a service to visually explore a form: `explore <form>`. This opens a new frame showing all labels and their bindings.

To learn more about Piccola, read [6], [4] or [5].

4 Modelling Mixins and Mixin Layers in Piccola

In this section I present the source code for simulation classes, mixins, components and mixin layers in JPiccola.

4.1 Modelling Classes with Piccola

As mixins are applied to classes, it was necessary to simulate the class structure first, because this does not exist in Piccola by default. In Java, the class `Object` is the root of all classes. The Piccola simulation looks like this:

```
Object =
    new(init):
       name(): "Object"
       toString(): name()
```

(For information about the syntax of Piccola read section 3 on page 5.)

This defines `Object` to be a form with a service named `new`. Calling `new` returns a form with the two services `name` and `toString`.

Next, I needed a service for generating new classes. Therefore, I created the service `newClass`: (The entire source code can be found in Appendix A on page 15)

```
newClass(classDef):
    [...]
    name = myName
    new(args):
       [...]
```

`newClass` provides the two services `name`, returning the name of the class, and `new`, creating a new “Instance” of this class. `newClass` takes the argument `classDef` which defines the details for the class like name, method etc. `classDef` must be a form containing bindings for the following labels:

- `name`: a string with the name of the class.
- `extends`: the superclass for this class. This is optional, the default is `Object`
- `methods`: a service returning the methods and class variables. It has the following syntax: `methods self super:` where self and super are references to the object itself (`self`) and the superclass object respectively (`super`). Thus, calls to self and super are possible in the methods.
- `initialize`: a service executed when `new` is called. This can be used to inizialize class variables. It also takes as argument references to self and super and additionaly the arguments for initialisation: `initialize self super init:`

By calling `new`, a form is returned with bindings for all methods and variables defined by `methods` where `self` and `super` are bound to this form and a form for the superclass object.

The following source code creates two classes, `A` and `B` where B is a subclass of `A`. `A` and `B` have the two labels `name` and `new` as described above (see figure 1).

```
A = newClass
```

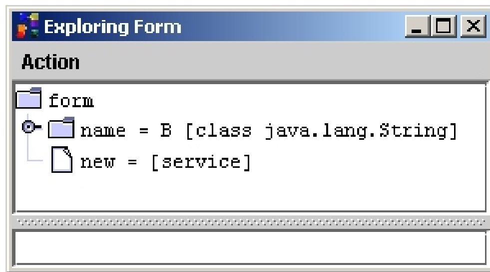

Figure 1: Exploring Class B

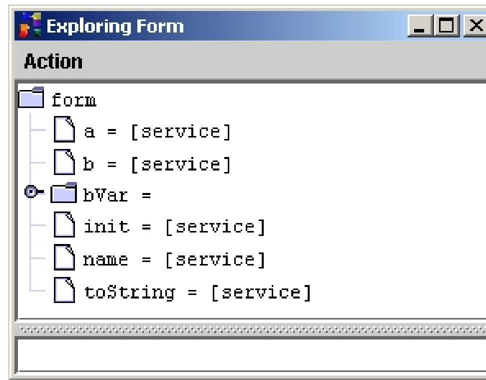

Figure 2: Exploring Instance b

```
    name = "A"
    methods self super:
       a: 1

B = newClass
    name = "B"
    extends = A
    methods self super:
       bVar = newVar()
       a: super.a() + (*bVar)
       b: self().a()
    initialize self super init:
       self().bVar <- init.var

a = A.new()
println "a.a() = " + a.a()
b = B.new(var = 3)
explore b
println "b.a() = " + b.a()
println "b.b() = " + b.b()
println "b.bVar: " + (*(b.bVar))
```

```
a.a() = 1
b.a() = 4
b.b() = 4
b.bVar: 3
```

`a` and `b` are "instances" of the created classes. They have labels for all methods as described above. (See figure 2)

To be able to use Java classes the same way as the classed defined by `newClass`, the service `adaptJavaClass` adapts the Java class to the above structure. Therefore, it is also possible to write mixins for Java classes in Piccola. Nevertheless, classes created like this are not Java classes but just Piccola "classes"!

There is one big problem in simulating classes this way: In Java, calls to `this` are resolved at the moment of the invocation. This permits us to redefine a certain behaviour in each subclass. For example, a superclass `super` contains the following code: `dothis(): this.dothat()`. Each subclass `sub` of `super` is able to overwrite `dothat`. By calling `sub.dothis()` the `dothat()` method defined by this subclass is invoked. But because Piccola uses early binding, this is not possible. `self` and `super` are bound at the moment of the class creation and then are fixed. To simulate a design equivalent to `dothis` in the example, a workaround is needed: `dothat` must be a service returning a service which can be set by a subclass. The part `methods` of the class definition must contain the following code:

```
[...]
dothatService = newVar(\():())
setDothat serv: dothatService <- serv
dothat: (*(self().dothatService))
[...]
```

This is not a nice solution but at least a way to make it possible.

4.2 Mixins

The next step is a service creating a new mixin, called `newMixin`:

```
newMixin(mixinDef):
    'mixinDef = (defaults, mixinDef)
    name = mixinDef.name
    applyMixin(class):
       [...]
    _*_ class: applyMixin(class)
```

`newMixin` creates a form containing bindings for the labels `name`, `applyMixin` and the operator `*`. `applyMixin` is the most important service of the created form: By calling it with a parameter `class`, it is applied to this class, i.e. it creates a new class as subclass of the parameter extending it with the behaviour defined by the mixin.

The syntax for creating a new mixin is similar to the one for classes: `newMixin` expects the same parameter, except for `extends`, which is missing as mixins are defined as classes with free super (cf. 2). As with classes, the methods of a mixin may have calls to `self` and `super` with `super` being a reference to the class the mixin will be applied to. Calls to `super` can at the same time be seen as constraints on the classes the mixin is instantiated with, as for example calling `super.a()` requires the method `a()` in the superclass. `*` is just an abreviation for `applyMixin`.

Here is an example (`A` is the class defined in the above example):

```
C = newMixin
    name = "C"
    methods self super:
       bVar = newVar()
       a: super.a() + (*bVar)
       b: self().a()
    initialize self super init:
```

```
        self().bVar <- init.var

D = C * A
explore C
d = D.new(var = 3)
explore d
println "d.a() = " + d.a()
println "d.b() = " + d.b()
println "d.bVar: " + (*(d.bVar))
```

a.a() = 1
d.a() = 4
d.b() = 4
d.bVar: 3

The mixin `C` defines exactly the same behaviour as the class `B` in the first example. Therefore, applying `C` to `A` creates a class identical to `B`, so `b` and `d` are equivalent instances. This demonstrates that everything we can do with subclassing can also be done with mixins. However, our mixin `C` could also be instantiated with other classes defining a method `a()`, creating absolutely different classes but implementing the methods `a()` and `b()` of `C` only once.

4.3 Components and MixinLayers

Now it is possible to work with classes and mixins in Piccola. The next step is to specify components and mixin layers. Components, as defined by [1] and [2], encapsulate collaborations, i.e. they define roles and the classes playing these roles. In fact, this is just the way I implemented components:

```
newComponent compDef:
    compDef.roles
    name = compDef.name
```

The service `newComponent` creates a form with bindings for `name` and for the roles defined. More complicated is the implementation of mixin layers: Mixin layers are applied to components, thus creating a new component. Mixin layers must be able to specify extensions for existing roles and to add new roles. Extensions for existing roles are mixins which have to be instantiated with the class belonging to this role in the supercomponent the mixin layer is applied to, whereas new roles must be classes. Therefore the service `newMixinLayer` produces a form with bindings for `apply` and the operator `**` as a shortcut for `apply`:

```
newMixinLayer layerDef:
    name = layerDef.name
    apply comp:
       [...]
    _**_ comp: apply(comp)
```

`newMixinLayer` expects an argument with the labels `name`, `extendedRoles` and `newRoles` as shown in the following example. (The classes and Mixins used are those of the above examples.)

```
myComp = newComponent
    name = "My Component"
    roles =
       role_A = A

myMixinLayer = newMixinLayer
    name = "My Mixin Layer"
    extendedRoles =
       role_A = C
    newRoles =
       role_B = B

mixComp = myMixinLayer ** myComp
```

Of course, it is also possible to define the class or mixin directly in the component or mixin layer respectively (like `myRole = newClass ...`) instead of using an already defined one. By using components and mixin layers, we can ensure that the classes using each other really provide the behaviour needed by the other: for example when working with graphs, it is important that the nodes and edges provide certain functionality the graph uses. If the graph has the ability to number the nodes, the nodes need the facility to store the given number (cf. 5).

5 Example: Mixin-Library for Graphs

After defining the structures for classes and mixins, I am now able to implement a library for graphs in Piccola. It consists mainly of two parts: first the elementary classes and components and second the mixins and mixin layers based upon the former. The complete source code can be found in appendix B on page 18

5.1 Base Classes and Components

Graphs consist mainly of three elements: vertices, edges and the graph itself. The elementary vertex consists only of a variable for the content and its set/get methods:

The edges are a little more complicated. They need at least a source and a destination vertex. Furthermore, there are different kinds of edges: directed, undirected and "undecided". All of them are defined as a subclass of a class `abstractVertex` (which is of course not abstract as used in Java).

A graph has collections of vertices and edges and methods to manipulate them: `addEdge` and `addVertex` to add an edge or a vertex to the graph and `getChildren` searching all children (or neighbours) of a vertex.

After defining the elementary classes, the base components can now be definded, each of them containing the roles `Edge`, `Vertex` and `Graph`. Depending on the characteristics of the edges, there are directed, undirected and undecided graph components. The three components differ only in the class for the role `Edge`.

To ensure that we can also work with trees, I additionally defined a classes for tree vertices and trees. The class `treeVertex` is a subclass of `simpleVertex` with an additional collection of all children of this vertex and methods to add and get children. Therefore, trees do not need edges to define connections between vertices and, unlike graphs, the class `simpleTree` does not have the collections of all vertices and edges, but only one for the roots. The tree component contains the roles `Tree` and `Vertex`. (It is a very simple implementation which does not verify that a vertex has just one parent and so on.)

5.2 Mixins and Mixin Layers

There are mixin layers for the following functionality:

- Traversing Graphs
- Numbering Graphs
- Cycle Checking
- Visualisation

Two mixins are needed for graph traversal: one for the vertex to store if it has already been traversed and one adding the ability to traverse all vertices, which is applied to the graph. While traversing the vertices, the graph calls a `visit` method on each vertex. The method may be overwritten in subclasses (cf. the mixin layer for numbering). As this is not possible with Piccola the mixin uses the work around mentioned in section 4.1 on page 6. In the graph library there is just a depth first traversal for graphs implemented but it would be easy to provide also breadth first traversal and traversal for trees. As mentioned in section

2 on page 4, mixins may have constraints on the classes they are applied to. To provide its service, the traversal mixin expects the superclass to have a collection with all vertices and a method searching all connected vertices of a given vertex. The corresponding mixin layer redefines the roles `Graph` and `Vertex` whereas `Edge` remains unchanged.

Sometimes, graphs need to number their vertices. This is a perfect example for the use of mixins: as different applications may require different orders of numbering, it makes sense to delegate the sequencing of the vertices and just number them in the given order. Hence, the mixin providing the numbering functionality depends on the graph traversal and just sets the `visit` method as described above to set the number of the vertex. Similar to the graph traversal, the numbering mixin layer affects only the roles `Graph` and `Vertex`. The latter is extended by the capability to store the obtained number.

Another nice feature is the check for cycles in a graph. The graph can perform this check on its own, i.e. it does not have constraints on the subclass and the mixin layer changes just this role.

5.3 Visualisation with JGraph

We often need visualisation for graphs as this is either helpful or even the main intention of the graph. To represent the Piccola graphs I used JGraph, a framework for visualizing graphs in Java (view http://www.jgraph.com). As JGraph is quite complex, I wrote a Piccola class for creating a visual graph. This way, it would be easy to code some mixins to broaden the graphical facilities. Unfortunately, it is not possible to use the full capacity of JGraph with Piccola because Piccola cannot subclass Java classes which would be necessary, for example to have different shapes for the vertices. However, this is just the purely graphical part which can be used as a stand-alone.

To visualize the Piccola graphs, there are also mixins visualizing the graphs and trees with the help of the class mentioned above. They implement just one method, `visualize`, visualizing all vertices and edges. The mixin layer first extends all elements with their graphical counterparts, i.e. each vertex, each edge and the graph itself know their visualisation. In a second mixin layer, the graph is extended by the feature of visualizing itself and a new role is added, the role `GraphicalGraph`.

How this visualisation looks like can be seen in figure 3. (The source code for this example can be found in the appendix.)

5.4 Using the Library

The components and mixin layers described in the last two sections can now arbitrarily be combined to get the desired behaviour. For example, we can apply the mixin layer visualizing the graph to the directed graph component to get a representation of a homepage system. Or we can combine the tree visualisation with the basic tree component as a diagrammed hierarchy for Java classes. Of course, we are also able to combine several mixin layers: first we apply the graph traversal to the undirected component, then the numbering mixin layer and finally the visualizing mixin layer. By first calling `number` and then `visualize`, a diagram with numbered vertices is created.

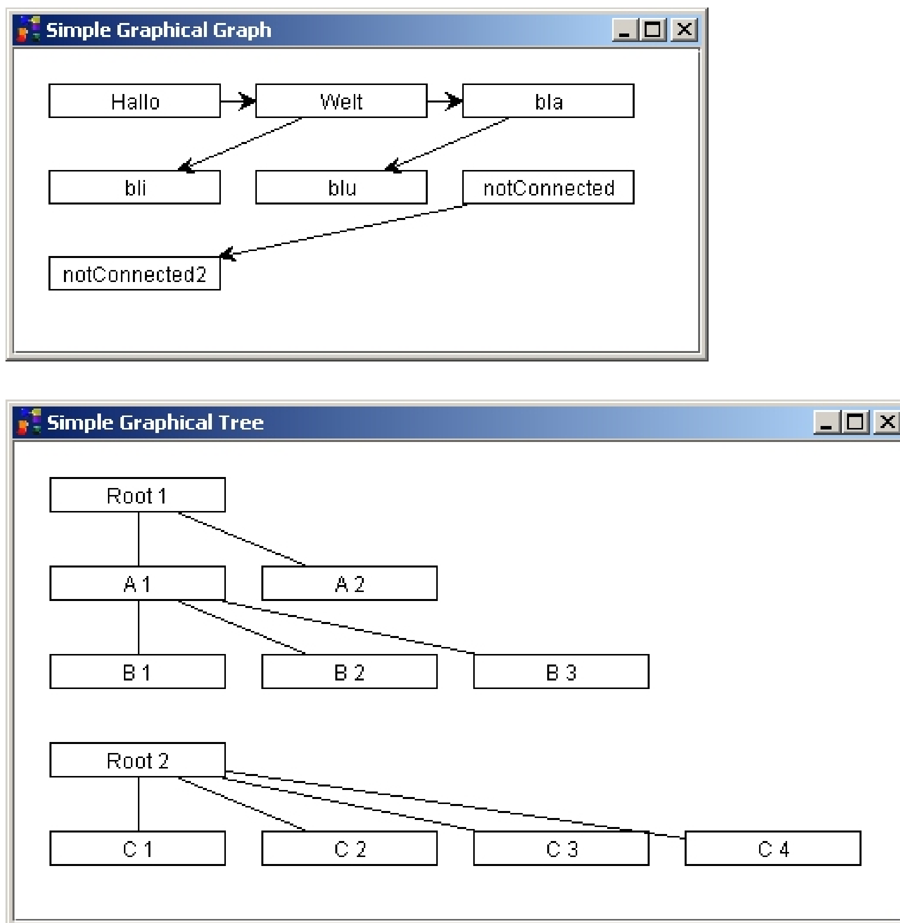

Figure 3: Two examples for the visualisation of graphs and trees

6 Conclusion

Mixins and mixin layer are a very powerful programming style. As Piccola by itself offers the ability to extend forms by new bindings, it was made easy to simulate the extension of classes by mixins. With an adequate library of base components and mixin layers for a specific domain, each software engineer can combine them to match exactly the requirements or he can easily write supplementary mixin layers.

It would be interesting to build some higher level applications based on the graph library, for example a framework for visualizing homepage systems. Like this, it would be possible to test if the concept of mixins is really usefull or if this is just valuable in theory.

As described in section 4.1 on page 6, it is a problem to overwrite methods in subclasses called by a method in the superclass. Therefore a workaraound is needed to do this all the same. Unfortunately, all methods which are used this way need to be identified when implementing the superclass. This may restrict the reuse of classes as superclasses if they do not provide this possibility for a certain method.

Appendices

A Source Code for Mixins in Piccola

A.1 Class Simulation

Code for Object, the root of the simulated class hierarchy:

```
1  # Object: used as superclass for all other classes.
2  # service "new" is the empty form.
3  # service "toString" returns the name
4  Object =
5      new(init):
6          name(): "Object"
7           toString(): name()
```

Code for newClass, the actual class simulation:

```
1  #default values for needed arguments for class creation
2  'defaults =
3      name = ""
4      extends = Object
5      initialize self super init: () #println "def init"
6
7  # newClass creates a new class
8  # with a service "new" which creates a new "instance"
9  # and bindings for self and super
10 # needs as argument a form with the following bindings:
11 # name: the name of the class. default: ""
12 # extends: the superclass. default: Object
13 # initialize: the service executed by new() call
14 # methods self super: the methods of the class.
15 # class variables can also be defined as label in methods
16 # and be initialized in the service "initialize"
17 # the methods can have references to self (call: self())
18 # and super (call: super)
19 newClass(classDef) :
20     'classDef = (defaults, classDef)
21     'superclass = classDef.extends
22     'myName = classDef.name
23     name = myName
24     new(args):
25         'selfConst = newConst()
26         'self =
27             'super = superclass.new(args)
28             super
29             name(): myName
30             toString(): super.toString() + " " + name()
```

```
31              classDef.methods selfConst super
32              init = classDef.initialize selfConst super
33          ''selfConst.init(self)
34          ''self.init(args)
35          self
```

Code for adapting Java classes to above defined class structure:

```
1  'extractClassName className:
2      'i = className.lastIndexOf(".")
3      'charArr = className.toCharArray()
4      'i = i+1
5      'sub = (Host.class "java.lang.String").new(val1= charArr, val2= i, val3=(className.len
6      sub
7
8  # adapts a java class to fit the above defined class structure
9  adaptJavaClass className:
10     'class = Host.class(className)
11     name = extractClassName(className)
12     new(args):
13         'object = class.new(args)
14         name(): name
15         object
```

A.2 Mixin

Code for newMixin, the Service creating a new mixin:

```
1  # newMixin creates a new mixin
2  # with a service "applyMixin" (expecting a class as argument)
3  # which creates a new class as application of the mixin to a class.
4  # and with an operator *: applyMixin.
5  # Needs as argument a form with the following bindings:
6  # name: the name of the Mixin. default: ""
7  # initialize: the service executed by new() call of the applied class
8  # methods self super: the methods of the mixin.
9  # variables for the class can also be defined as label in methods
10 # and be initialized in the service "initialize"
11 # the methods can have references to self (call: self())
12 # and super (call: super)
13 newMixin(mixinDef):
14     'mixinDef = (defaults, mixinDef)
15     name = mixinDef.name
16     applyMixin(class):
17         'mixinClass = newClass
18             name = name + class.name
19             initialize = mixinDef.initialize
20             extends = class
```

```
21                methods = mixinDef.methods
22            mixinClass
23        _*_ class: applyMixin(class)
```

A.3 Component

Code for newComponent, the Service creating a new component:

```
1  # creates a new component with a name and several roles.
2  # roles are specified as follows:
3  # role-name = class
4  newComponent compDef:
5      compDef.roles
6      name = compDef.name
```

A.4 Mixin Layer

Code for newMixinLayer, the Service creating a new mixin layer.

```
1  # creates a MixinLayer with a name and a service "apply"
2  # apply applies the mixin layer to a component and returns a new component
3  # with the roles of the given component and new roles (see below)
4  # expects as argument a form with the following bindings:
5  # name: name of the mixin layer
6  # newRoles: roles which are new in the mixin
7  # extendedRoles: roles which already exist in the super component
8  #         and are extended by this mixin layer. form: role-name = mixin
9  #         The mixin is applied to the class for this role in the super-component
10 newMixinLayer layerDef:
11     name = layerDef.name
12     apply comp:
13         'newComp = newComponent
14             name = name + comp.name
15             roles =
16                 comp
17                 'labelNew = newLabel("newRoles")
18                 'labelExtend = newLabel("extendedRoles")
19                 if (labelNew.exists(layerDef))
20                     then: layerDef.newRoles
21                 if (labelExtend.exists(layerDef))
22                     then:
23                         'rolesVar = newVar()
24                         ''forEachLabel
25                             form = layerDef.extendedRoles
26                             do X:
27                                 'mixin = X.project(form)
28                                 'newRole = X.bind(mixin * X.project(comp))
29                                 'rolesForm = (*rolesVar)
```

```
30                                  'rolesForm = (meta rolesForm).extend(newRole)
31                                  'rolesVar <- rolesForm
32                          (*rolesVar)
33          newComp
34      _**_ comp: apply(comp)
```

B Source Code of the Graph Library

B.1 Base Classes

```
1  ###########################################################################
2  # Basic Classes: Simple Classes for Vertex, Edge, Graph, Tree             #
3  ###########################################################################
4
5  # Simple class for vertex representation
6  simpleVertex = newClass
7      name = "Vertex"
8      methods self super:
9          content = newVar("")
10         setContent cont: self().content <- cont
11         getContent: *(self().content)
12         toString: self().getContent()
13     initialize self super init:
14         self().content <- init.content
15
16 # Vertex for Trees
17 treeVertex = newClass
18     name = "Tree"
19     extends = simpleVertex
20     methods self super:
21         children = ArrayList.new()
22         getChildren: self().children
23         addChild child: self().children + child
24         toString: super.toString()
25
26 # "abstract" class representing an edge
27 abstractEdge = newClass
28     name = "Edge"
29     methods self super:
30         source = newVar()
31         dest = newVar()
32         getSource: *(self().source)
33         getDestination: *(self().dest)
34         setSource s: self().source <- s
35         setDestination d: self().dest <- d
36     initialize self super init:
37         self().dest <- init.destination
```

```
38          self().source <- init.source
39
40  # undirected edge
41  undirectedEdge = newClass
42      name = "UndirectedEdge"
43      extends = abstractEdge
44      methods self super:
45          isUnidirectional: false
46          toString:
47              'str = (*(self().source)).toString()
48              'str = str + " connected to " + (*(self().dest)).toString()
49              str
50
51  # directed edge
52  directedEdge = newClass
53      name = "DirectedEdge"
54      extends = abstractEdge
55      methods self super:
56          isUnidirectional: true
57          toString:
58              'str = (*(self().source)).toString()
59              'str = str + " connected to " + (*(self().dest)).toString()
60              str
61
62  # simple class representing an edge which can be unidirectional or bidirectional
63  simpleEdge = newClass
64      name = "Edge"
65      extends = abstractEdge
66      methods self super:
67          unidir = newVar(true)
68          isUnidirectional: *(self().unidir)
69          setUnidirectional bool: self().unidir <- bool
70          toString:
71              'str = (*(self().source)).toString()
72              'strVar = newVar(str)
73              'if (self().isUnidirectional())
74                  then: 'strVar <- (*strVar) + " unidirectional"
75                  else: 'strVar <- (*strVar) + " bidirectional"
76              'strVar <- (*strVar) + " connected to " + (*(self().dest)).toString()
77              (*strVar)
78          initialize self super init:
79              self().unidir <- init.unidirectional
80
81  # simple graph class
82  # vertices need method toString()
83  # edge needs methods getSource(), getDestination(), isUnidirectional()
84  simpleGraph = newClass
```

```
85      name = "Graph"
86      methods self super:
87          vertices = ArrayList.new()
88          edges = ArrayList.new()
89          addVertex vertex: self().vertices + vertex
90          addEdge edge: self().edges + edge
91          getChildren vertex:
92              'children = ArrayList.new()
93              'isConnection X:
94                  'if (X.getSource() == vertex)
95                      then: children + X.getDestination()
96                      else:
97                          if (!(X.isUnidirectional()))
98                              then:
99                                  if (X.getDestination() == vertex)
100                                     then: children + X.getSource()
101             'edges.foreach(isConnection)
102             children
103         toString:
104             'str = "Graph: \nVertices: \n"
105             'strVar = newVar(str)
106             'vertices.foreach(\X: strVar <- ((*strVar) + X.toString()+ "\n"))
107             'strVar <- (*strVar) + "Connections: \n"
108             'edges.foreach(\X: strVar <- ((*strVar) + X.toString() + "\n"))
109             (*strVar)
110
111 # Tree Class containing all roots
112 simpleTree = newClass
113     name = "Tree"
114     methods self super:
115         roots = ArrayList.new()
116         addRoot vertex: self().roots + vertex
117         getChildren vertex:
118             vertex.getChildren()
119         toString:
120             'str = "Tree: \nRoots: \n"
121             'strVar = newVar(str)
122             'roots.foreach(\X: strVar <- ((*strVar) + X.toString()+ "\n"))
123             (*strVar)
```

B.2 Base Components

```
1 ##########################################################################
2 # Basic Components for Graphs, Trees                                     #
3 ##########################################################################
4
5 # graph component containing simple classes for graph, vertex and edge
```

```
6  simpleGraphComponent = newComponent
7      name = "GraphComponent"
8      roles =
9          Graph = simpleGraph
10         Edge = simpleEdge
11         Vertex = simpleVertex
12
13 # graph component containing simple classes for
14 # undirected graph, vertex and edge
15 undirectedGraphComponent = newComponent
16     name = "UDGraphComponent"
17     roles =
18         Graph = simpleGraph
19         Edge = undirectedEdge
20         Vertex = simpleVertex
21
22 # graph component containing simple classes for
23 # directed graph, vertex and edge
24 directedGraphComponent = newComponent
25     name = "DGraphComponent"
26     roles =
27         Graph = simpleGraph
28         Edge = directedEdge
29         Vertex = simpleVertex
30
31 # graph component containing simple classes for trees
32 simpleTreeComponent = newComponent
33     name = "TreeComponent"
34     roles =
35         Tree = simpleTree
36         Vertex = treeVertex
```

B.3 Graph Traversal

```
1  # Mixin for elements which have to be traversed
2  # adds a new variable indicating if the element already was traversed
3  traversedMixin = newMixin
4      name = "traversed"
5      methods self super:
6          traversed = newVar(false)
7          isTraversed: (*(self().traversed))
8          setTraversed bool:
9              self().traversed <- bool
10         toString: super.toString()
11
12 # Mixin for DepthFirstTraversal of a Graph
13 # constraint for application: class needs:
```

```
14  # - List "vertices" containing all vertices
15  # - method "getChildren(Vertex) getting all children (connected vertices)
16  DFTMixin = newMixin
17      name = "DepthFirstTraversal"
18      methods self super:
19          visitMethod = newVar(\X: ())
20          setVisitMethod doit: self().visitMethod <- doit
21          getVisitMethod:
22              (*(self().visitMethod))
23          getFirstUntraversedVertex:
24              'i = newVar(0)
25              'vert = newVar()
26              'found = newVar(false)
27              'loop
28                  while: ((*i) < self().vertices.size()) & !(*found)
29                  do:
30                      'if (!(self().vertices.get((*i)).isTraversed()))
31                          then:
32                              'found <- true
33                              'vert <-self().vertices.get((*i))
34                      'i <- ((*i) + 1)
35              (*vert)
36          def traverseConnected args:
37              '(self().getVisitMethod()) (args.vertex)
38              'args.vertex.setTraversed( true )
39              'args.list + args.vertex
40              'children = super.getChildren(args.vertex)
41              'forDo X:
42                  if (!(X.isTraversed()))
43                      then:
44                          traverseConnected(vertex = X, list = args.list)
45              'children.foreach(forDo)
46          traverse:
47              'traversing = ArrayList.new()
48              'temp = newVar(getFirstUntraversedVertex())
49              'loop
50                  while: (*temp) != ()
51                  do:
52                      traverseConnected(vertex=(*temp), list= traversing)
53                      temp <- getFirstUntraversedVertex()
54              traversing
55          toString: super.toString()
56
57  # Mixin Layer for Depth First Traversed Graphs.
58  # Component needs roles Graph and Vertex
59  DFTGraphML = newMixinLayer
60      name = "DFTMixinLayer"
```

```
61      extendedRoles =
62          Graph = DFTMixin
63          Vertex = traversedMixin
```

B.4 Vertex Numbering

```
1   # Mixin for Elements with number
2   numberedMixin = newMixin
3       name = "Numbered"
4       methods self super:
5           number = newVar(-1)
6           setNumber X: self().number <- X
7           getNumber: *(self().number)
8           toString: "" + self().getNumber() + ": " + super.toString()
9
10  # Mixin for Graphs numbering the vertices
11  # constraint for application: class needs:
12  # - setVisitMethod(method)
13  # - traverse()
14  vertexNumberingGraph = newMixin
15      name = "VertexNumbering"
16      methods self super:
17          actNumber = newVar(1)
18          numberVertex X:
19              X.setNumber (*(self().actNumber))
20              self().actNumber <- (*(self().actNumber) + 1)
21          numberVertices:
22              'super.setVisitMethod(self().numberVertex)
23              super.traverse()
24          toString: super.toString()
25
26  # Mixin Layer for numbered Graphs.
27  # Component needs roles Graph and Vertex
28  NumberingML = newMixinLayer
29      name = "VertexNumbering"
30      extendedRoles =
31          Graph = vertexNumberingGraph
32          Vertex = numberedMixin
```

B.5 Cycle Checking

```
1   # Mixin for Collection: adding method addAllDistinct
2   collectionMixin = newMixin
3       methods self super:
4           addAllDistinct col:
5               'addDistinct X:
6                   if (!(super ? X))
```

```
                    then: super + X
            col.foreach(addDistinct)

# Defining an ArrayList with method addAllDistinct
'extArrayList = collectionMixin * (adaptJavaClass "java.util.ArrayList")

# Mixin for CycleChecking in a Graph.
# constraint for application: class needs:
# - getChildren(vertex)
cycleCheckingMixin = newMixin
    name = "CycleChecking"
    methods self super:
        getReachable X:
            'reachable = extArrayList.new()
            'oldSize = newVar(reachable.size())
            'reachable ++ self().getChildren(X)
            'loop
                while: (*oldSize) < reachable.size()
                do:
                    oldSize <- reachable.size()
                    'templist = extArrayList.new()
                    'templist ++ reachable
                    'doit X: templist.addAllDistinct(self().getChildren(X))
                    reachable.foreach(doit)
                    reachable.addAllDistinct(templist)
            reachable
        hasCycles:
            'cycles = newVar(false)
            'i = newVar(0)
            'temp = self().vertices.get((*i))
            'loop
                while: (((*i) < self().vertices.size()) & !(*cycles))
                do:
                    'temp = self().vertices.get((*i))
                    'reachable = self().getReachable(temp)
                    'if (reachable ? temp)
                        then: cycles <- true
                    'i <- ((*i) + 1)
            (*cycles)
        toString():
            'str = newVar(super.toString())
            'if (self().hasCycles())
                then: 'str <- (*str) + "Graph with cycles \n"
                else: 'str <- (*str) + "Graph without cycles \n"
            (*str)

# Mixin Layer for Graphs with cycle checking ability
```

```
54 # Component needs role Graph
55 cycleML = newMixinLayer
56     name = "CycleChecking"
57     extendedRoles =
58         Graph = cycleCheckingMixin
```

B.6 JGraph Adaptation

```
1  'loadCore "swing.picl" root
2
3  JFrame = Host.class "javax.swing.JFrame"
4  JScrollPane = Host.class "javax.swing.JScrollPane"
5  BorderFactory = Host.class "javax.swing.BorderFactory"
6  Hashtable = Host.class "java.util.Hashtable"
7  Rectangle = Host.class "java.awt.Rectangle"
8  Color = Host.class "java.awt.Color"
9  Map = Host.class "java.util.Map"
10 Object = Host.class "java.lang.Object"
11 ArrayList = Host.class "java.util.ArrayList"
12
13 DefaultGraphCell = Host.class "org.jgraph.graph.DefaultGraphCell"
14 JGraph = Host.class "org.jgraph.JGraph"
15 DefaultGraphModel = Host.class "org.jgraph.graph.DefaultGraphModel"
16 GraphConstants = Host.class "org.jgraph.graph.GraphConstants"
17 DefaultPort = Host.class "org.jgraph.graph.DefaultPort"
18 DefaultEdge = Host.class "org.jgraph.graph.DefaultEdge"
19 ConnectionSet = Host.class "org.jgraph.graph.ConnectionSet"
20
21 EdgeDecoration =
22     ARROW_CIRCLE = GraphConstants.ARROW_CIRCLE
23     ARROW_CLASSIC =    GraphConstants.ARROW_CLASSIC
24     ARROW_DIAMOND = GraphConstants.ARROW_DIAMOND
25     ARROW_DOUBLELINE = GraphConstants.ARROW_DOUBLELINE
26     ARROW_LINE = GraphConstants.ARROW_LINE
27     ARROW_NONE = GraphConstants.ARROW_NONE
28     ARROW_SIMPLE = GraphConstants.ARROW_SIMPLE
29     ARROW_TECHNICAL = GraphConstants.ARROW_TECHNICAL
30
31 newPortVertex args:
32     vertex = args.vertex
33     port = args.port
34
35 SimpleJGraph = newClass
36     name = "SimpleJGraph"
37     methods self super:
38         model = DefaultGraphModel.new()
39         graph = JGraph.new(model)
```

```
40          attributes = Hashtable.new()
41          connectionSet = ConnectionSet.new()
42          cellsList = ArrayList.new()
43          name = newVar()
44          'vertexDef =
45              content = ""
46              BorderColor = Color.black
47              Bounds = (val1=20, val2=20, val3=40, val4=20)
48              Background = Color.white
49              Opaque = true
50          'edgeDef =
51              content = ""
52              LineEnd = EdgeDecoration.ARROW_SIMPLE
53              LineColor = Color.black
54              EndFill = true
55          addVertex args:
56              'args = (vertexDef, args)
57              'vert = DefaultGraphCell.new(args.content)
58              'attr = GraphConstants.createMap()
59              'attributes.put(val1 = vert, val2 = attr)
60              'bounds = Rectangle.new(args.Bounds)
61              'GraphConstants.setBounds(val1=attr, val2=bounds)
62              'GraphConstants.setBorderColor(val1=attr, val2=args.BorderColor)
63              'GraphConstants.setBackground(val1=attr, val2=args.Background)
64              'GraphConstants.setOpaque(val1=attr, val2=args.Opaque)
65              'p = DefaultPort.new()
66              'vert.add(p)
67              'pv = newPortVertex(vertex = vert, port = p)
68              'cellsList + vert
69              pv
70          connect args:
71              'args = (edgeDef, args)
72              'edge = DefaultEdge.new(args.content)
73              'attr = GraphConstants.createMap()
74              'attributes.put(val1=edge, val2=attr)
75              'GraphConstants.setLineEnd(val1=attr, val2=args.LineEnd)
76              'GraphConstants.setEndFill(val1=attr, val2=args.EndFill)
77              'GraphConstants.setLineColor(val1=attr, val2=args.LineColor)
78              'connectionSet.connect(val1=edge, val2=args.source.port,
79                                     val3=args.target.port)
80              'cellsList + edge
81              edge
82          pack:
83              'cells = cellsList.toArray()
84              'model.insert(val1=cells, val2=attributes, val3=connectionSet,
85                            val4=(), val5=())
86          show:
```

```
87              frame = JFrame.new((*(self().name)))
88              frame.getContentPane().add(JScrollPane.new(graph))
89              frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
90              frame.pack()
91              frame.setVisible(true)
92      initialize self super init:
93          'self().graph.setSelectNewCells(true)
94          'self().name <- (title="", init).title
```

B.7 Visualisation

```
1   ###########################################################################
2   # Mixin and MixinLayers for Graph Visualization with JGraph               #
3   ###########################################################################
4
5   # Mixin adding a graphical component to the model component
6   graphicalMixin = newMixin
7       name = "Graphical"
8       methods self super:
9           graphicalComp = newVar()
10          setGraphicalComponent comp: self().graphicalComp <- comp
11          getGraphicalComponent: (*(self().graphicalComp))
12          toString: super.toString()
13
14  # Mixin for visualizing a graph
15  # constraint for application: class needs:
16  # - getGraphicalComponent()
17  # - list edges
18  # - list vertices
19  # Edge and Vertex need methods set/getGraphicalComponent
20  visualizeMixin = newMixin
21      name = "Visualize"
22      methods self super:
23          visualize:
24              'graph = self().getGraphicalComponent()
25              'vertWidth = 100
26              'vertHeight = 20
27              'x = newVar(20)
28              'y = newVar(20)
29              'deltaX = 120
30              'deltaY = 50
31              'maxX = 300
32              'for
33                  from: 0
34                  to: self().vertices.size()-1
35                  do i:
36                      'temp = self().vertices.get(i)
```

```
37                     'gTemp = graph.addVertex
38                         content = temp.toString()
39                         Bounds = (val1 = (*x), val2 = (*y),
40                                 val3 = vertWidth,
41                                 val4 = vertHeight)
42                     'temp.setGraphicalComponent(gTemp)
43                     'x <- ((*x) + deltaX)
44                     'if ((*x) > maxX)
45                         then:
46                             'x <- 20
47                             'y <- ((*y) + deltaY)
48             'for
49                 from: 0
50                 to: self().edges.size()-1
51                 do i:
52                     'temp = self().edges.get(i)
53                     'cont = newVar("")
54                     'contLabel = newLabel("getContent")
55                     'if (contLabel.exists(temp))
56                         then: cont <- temp.getContent()
57                     'arrow = newVar()
58                     'if (temp.isUnidirectional())
59                         then: arrow <- EdgeDecoration.ARROW_CLASSIC
60                         else: arrow <- EdgeDecoration.ARROW_NONE
61                     'gTemp = graph.connect
62                         content = (*cont)
63                         LineEnd = (*arrow)
64                         source = temp.getSource().getGraphicalComponent()
65                         target = temp.getDestination().getGraphicalComponent()
66                     'temp.setGraphicalComponent(gTemp)
67             graph.pack()
68             graph.show()
69
70 # Mixin for visualizing a tree
71 # constraint for application: class needs:
72 # - getGraphicalComponent()
73 # - list roots
74 # Vertex needs methods set/getGraphicalComponent
75 visualizeTreeMixin = newMixin
76     name = "Visualize"
77     methods self super:
78         'visualizeLayer args:
79             'graph = self().getGraphicalComponent()
80             'vertices = args.vertices
81             'x = newVar(20)
82             'y = args.y
83             'vertWidth = 100
```

```
84            'vertHeight = 20
85            'for
86                from: 0
87                to: vertices.size()-1
88                do i:
89                    'temp = vertices.get(i)
90                    'gTemp = graph.addVertex
91                        content = temp.toString()
92                        Bounds = (val1 = (*x), val2 = y, val3 = vertWidth,
93                                    val4 = vertHeight)
94                    'temp.setGraphicalComponent(gTemp)
95                    'x <- (*x) + args.deltaX
96        visualize:
97            'graph = self().getGraphicalComponent()
98            'vertWidth = 100
99            'vertHeight = 20
100           'x = newVar(20)
101           'y = newVar(20)
102           'deltaX = 120
103           'deltaY = 50
104           'for
105               from: 0
106               to: self().roots.size()-1
107               do i:
108                   'actRoot = self().roots.get(i)
109                   'parentList = ArrayList.new()
110                   'parentList + actRoot
111                   'childrenList = ArrayList.new()
112                   'childrenList ++ actRoot.getChildren()
113                   'visualizeLayer
114                               vertices = parentList
115                               y = (*y)
116                               deltaX = deltaX
117                   y <- (*y) + deltaY
118                   'loop
119                       while: !(childrenList.isEmpty())
120                       do:
121                           visualizeLayer
122                               vertices = childrenList
123                               y = (*y)
124                               deltaX = deltaX
125                           y <- (*y) + deltaY
126                           'connectChildren X:
127                               'X.getChildren().foreach(\Y:
128                                   'graph.connect
129                                       LineEnd = EdgeDecoration.ARROW_NONE
130                                       source = X.getGraphicalComponent()
```

```
131                                         target = Y.getGraphicalComponent())
132                             'parentList.foreach(connectChildren)
133                             'parentList.clear()
134                             'parentList ++ childrenList
135                             'childrenList.clear()
136                             'parentList.foreach(\X: childrenList ++ X.getChildren())
137             'graph.pack()
138             'graph.show()
139
140 # Mixin Layers for visualisation
141 graphicalML = newMixinLayer
142     name = "Graphical"
143     extendedRoles =
144         Graph = graphicalMixin
145         Edge = graphicalMixin
146         Vertex = graphicalMixin
147
148 visualizeML = newMixinLayer
149     name = "Visualized"
150     extendedRoles =
151         Graph = visualizeMixin
152     newRoles =
153         GraphicalGraph = SimpleJGraph
154
155 # Mixin Layers for visualisation of Trees
156 graphicalTreeML = newMixinLayer
157     name = "Graphical"
158     extendedRoles =
159         Tree = graphicalMixin
160         Vertex = graphicalMixin
161
162 visualizeTreeML = newMixinLayer
163     name = "Visualized"
164     extendedRoles =
165         Tree = visualizeTreeMixin
166     newRoles =
167         GraphicalTree = SimpleJGraph
```

B.8 Examples

An example for a simple tree and a simple graph.

```
1 graphicalGraphComponent = visualizeML ** (graphicalML ** simpleGraphComponent)
2 visualGraph = graphicalGraphComponent.GraphicalGraph.new(
3         title = "Simple Graphical Graph")
4 graphModel = graphicalGraphComponent.Graph.new()
5 graphModel.setGraphicalComponent(visualGraph)
6
```

```
7  vert1 = graphicalGraphComponent.Vertex.new(content = "Hallo")
8  vert2 = graphicalGraphComponent.Vertex.new(content = "Welt")
9  vert3 = graphicalGraphComponent.Vertex.new(content = "bla")
10 vert4 = graphicalGraphComponent.Vertex.new(content = "bli")
11 vert5 = graphicalGraphComponent.Vertex.new(content = "blu")
12 vert6 = graphicalGraphComponent.Vertex.new(content = "notConnected")
13 vert7 = graphicalGraphComponent.Vertex.new(content = "notConnected2")
14 graphModel.addVertex(vert1)
15 graphModel.addVertex(vert2)
16 graphModel.addVertex(vert3)
17 graphModel.addVertex(vert4)
18 graphModel.addVertex(vert5)
19 graphModel.addVertex(vert6)
20 graphModel.addVertex(vert7)
21 edge1 = graphicalGraphComponent.Edge.new(source=vert1, destination=vert2,
22                     unidirectional=false)
23 edge2 = graphicalGraphComponent.Edge.new(source=vert2, destination=vert3,
24                     unidirectional=false)
25 edge3 = graphicalGraphComponent.Edge.new(source=vert3, destination=vert5,
26                     unidirectional=false)
27 edge4 = graphicalGraphComponent.Edge.new(source=vert2, destination=vert4,
28                     unidirectional=false)
29 edge5 = graphicalGraphComponent.Edge.new(source=vert6, destination=vert7,
30                     unidirectional=false)
31 graphModel.addEdge(edge1)
32 graphModel.addEdge(edge2)
33 graphModel.addEdge(edge3)
34 graphModel.addEdge(edge4)
35 graphModel.addEdge(edge5)
36
37 graphModel.visualize()
38
39 graphicalTreeComponent =
40         visualizeTreeML ** (graphicalTreeML ** simpleTreeComponent)
41 visualTree=graphicalTreeComponent.GraphicalTree.new(
42             title = "Simple Graphical Tree")
43 treeModel = graphicalTreeComponent.Tree.new()
44 treeModel.setGraphicalComponent(visualTree)
45 root1 = graphicalTreeComponent.Vertex.new(content = "Root 1")
46 treeModel.addRoot(root1)
47 root2 = graphicalTreeComponent.Vertex.new(content = "Root 2")
48 treeModel.addRoot(root2)
49 a1 = graphicalTreeComponent.Vertex.new(content = "A 1")
50 root1.addChild(a1)
51 a2 = graphicalTreeComponent.Vertex.new(content = "A 2")
52 root1.addChild(a2)
53 b1 = graphicalTreeComponent.Vertex.new(content = "B 1")
```

```
54  a1.addChild(b1)
55  b2 = graphicalTreeComponent.Vertex.new(content = "B 2")
56  a1.addChild(b2)
57  b3 = graphicalTreeComponent.Vertex.new(content = "B 3")
58  a1.addChild(b3)
59  c1 = graphicalTreeComponent.Vertex.new(content = "C 1")
60  root2.addChild(c1)
61  c2 = graphicalTreeComponent.Vertex.new(content = "C 2")
62  root2.addChild(c2)
63  c3 = graphicalTreeComponent.Vertex.new(content = "C 3")
64  root2.addChild(c3)
65  c4 = graphicalTreeComponent.Vertex.new(content = "C 4")
66  root2.addChild(c4)
67
68  treeModel.visualize()
```

References

[1] Yannis Smaragdakis and Don Batory, **“Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs”**, ACM TOSEM, vol. 11, no. 2, April 2002, pp. 215-255

[2] Yannis Smaragdakis and Don Batory, **“Implementing Layered Design with Mixin Layers”**, Proceedings ECOOP ’98, Eric Jul (Ed.), LNCS, vol. 1445, Brussels, Belgium, July 1998, pp. 550570

[3] Oscar Nierstrasz and Franz Achermann, **“Supporting Compositional Styles for Software Evolution”**, Proceedings International Symposium on Principles of Software Evolution (ISPSE 2000), IEEE, Kanazawa, Japan, November 2000, pp. 11-19

[4] Franz Achermann and Oscar Nierstrasz, **“Applications = Components + Scripts A Tour of Piccola”**, Software Architectures and Component Technology, Mehmet Aksit (Ed.), Kluwer, 2001, pp. 261-292

[5] Franz Achermann, Markus Lumpe, Jean-Guy Schneider and Oscar Nierstrasz, **“Piccola a Small Composition Language”**, Formal Methods for Distributed Processing A Survey of Object-Oriented Approaches, Howard Bowman and John Derrick (Eds.), pp. 403-426, Cambridge University Press, 2001

[6] Oscar Nierstrasz, Franz Achermann and Stefan Kneubhl, **“A Guide to JPiccola”**, Technical Report, no. IAM-03-003, Institut für Informatik, June 2003, Technical Report, Universität Bern, Switzerland