An Object-Oriented Library for Tracing Requirements

Francisco A. C. Pinheiro

Universidade de Brasília
Departamento de Ciência da Computação
facp@cic.unb.br

Abstract. We present the overall design of an object-oriented library for use when developing tracing capable applications. The library is implemented as java classes and consists of definitions of the basic classes for registration of objects and relations, as well as methods for different types of tracing procedures and components for manipulating tracing results and viewing them graphically.

Keywords: Requirements traceability, object-orientation, object library, tracing environments.

1 Introduction

Requirements traceability remains an important concern for software developers, including management. It supports assessment of changes, provides guidance, and is viewed as a measure of quality. Nevertheless, its adoption and consequent benefits do not produce uniform results (Palmer 1997). There are many factors influencing the effectiveness of requirements traceability. Environmental and organizational factors are chief among them but technical factors also have an impact.

Many traceability models and techniques were proposed from tagging to indexing schemes, through traceability matrices (Davis 1990), tracing structures (Gotel 1995) and tracing languages (Pinheiro and Goguen 1996), to cite just a few. Although the basic concept of requirements tracing is quite simple: to follow links or relations, its implementation is made difficult because the proposed models and techniques are not easy to be incorporated in tools and software environments. They usually have to be hardcoded.

Some major gaps in the current traceability tools may be the result of this lack of flexibility. Jarke (1998) identifies the adaptability to project specific needs as a critical issue and Dömges and Pohl (1998) cite integration into the process, adaptation to the situation, and support for organizational knowledge creation as desirable features of tracing environments. Although these issues are not entirely technical, a tracing library may be useful to overcome some of the difficulties by making easier to develop software environments and applications exhibiting desired traceability features.

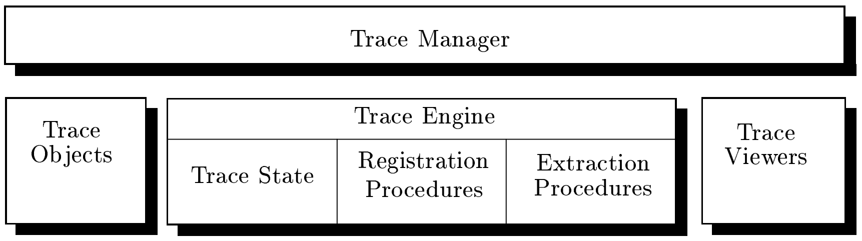

Fig. 1. Tracing Library Components

In this paper we present the overall design of an object-oriented library for tracing requirements. The library is implemented in Java and so are the examples presented here. The paper is structured in the following way: Sect. 2 describes the structure of the library and its basic classes, Sect. 3 describes the main structure of the library, the configuration automaton used to recognize tracing expressions, Sect. 4 presents the procedures used to register objects and relations, Sect. 5 discusses some of the methods available for tracing, Sect. 6 presents some of the classes used to show trace results in a window environment, and Sect. 7 contains the concluding remarks.

2 Objects and Classes

The library is composed by the four components shown in Fig. 1. The *Trace Objects* component contains the basic classes of traceable objects and their (allowed) relationships. It is an extensible part of the library, the user may and should provide his own classes. The *Tracing Engine* component comprises the classes for controlling the state of the environment, i.e., the configuration of all traceable objects and relations, as well as classes to implement procedures for registering objects and relations and for extracting tracing results. The *Trace Viewers* component contains classes providing several ways to visualize tracing results. It is mainly intended for use in window applications. The *Trace Manager* component is responsible for implementing the functionality of the library as a whole. In every application or environment using this library there will be only one instance of a class `TrcManager`. This instance controls the message passing mechanism among all other library objects.

The library comes with an abstract class `TrcObject` that is the superclass of all classes of traceable objects and relations. A traceable object is an object one can trace using the library. The same explanation is valid for traceable relations. All traceable objects should have an specific class that is a subclass of `TrcObj` and all traceable relations should have an specific class that is a subclass of `TrcRel`. This basic hierarchy is shown in Fig. 2 where `Object` is the superclass of all Java objects.

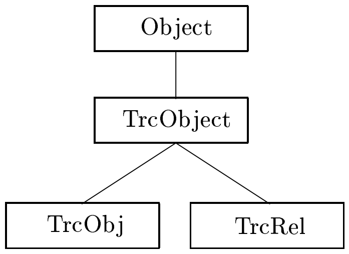

Fig. 2. Class Hierarchy

The relation objects are instances of `TrcRel`. They contain as a minimum `source` and `target` attributes for holding the related objects:

```
public class TrcRel
{
    TrcObject  source;
    TrcObject  target;
}
```

The use of the library is simple. For an application A the user should import the library classes he wants to use (in fact the only optional classes are those implementing the trace viewers). For every class `Cobj` whose objects he wants to trace, he should declare it as a subclass of `TrcObj`. In the same way, for every type of link he wants to consider for tracing an appropriate relation class should be declared as a subclass of `TrcRel`.

Once the desired classes are structured the user should provide for every created object to be registered with the trace manager.

3 Configuration Automaton

The configuration automaton is the main structure of the tracing engine component of the library. A configuration automaton is an automaton used to verify if a tracing expression is matched by any configuration of objects and relations registered in the environment. By registration of an object we mean the effective creation of the object instance and its placement under the control of the trace manager, i.e., the object becomes traceable. Similarly, the registration of a relation comprises the effective creation of a relation instance relating two objects and its placement under the control of the trace manager.

For each object *obj* and relation *rel* registered in the environment there are corresponding states S_{obj} and S_{rel} in the configuration automaton. The configuration automaton is dynamically maintained:

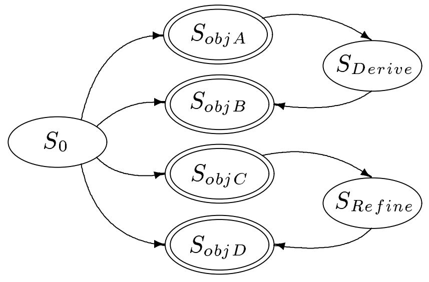

Fig. 3. Configuration Automaton

- For each object *obj* registered in the environment, a state S_{obj} is added in the automaton as well as a transition from its initial state S_0 to S_{obj}. The state S_{obj} is final and the transition is labelled with the object identifier.
- For each relation instance of class *rel* relating objects obj_1 and obj_2, a state S_{rel} and two transitions are added to the automaton. The transition from S_{obj_1} to S_{rel} is labelled with the relation class identifier *rel* and the transition from S_{rel} to S_{obj_2} is labelled with object identifier obj_2.

Figure 3 shows the states and transitions for a configuration where objects *objA* and *objB* are related by an instance of the relation class *Derive* and objects *objC* and *objD* are related by an instance of the relation class *Refine*. There are also procedures to remove states and transitions when the corresponding objects and relations are excluded.

The configuration automaton is implemented by the `CfgState` class.

```
private class CfgState
{
    int       stateType;
    Vector    stateTo;     // transitions coming out of state
    TrcObject stateId;
}
```

The `CfgState` instances contain the attribute `stateId` which holds (a pointer to) the object corresponding to this state, the attribute `stateTo` which is a list, implemented as a vector, of all arcs coming out of it, and the attribute `stateType` used to differentiate the initial, object and relation states. Each element of `stateTo` points to a state reachable from this instance of `CfgState`. Figure 4 illustrates the use of `stateTo` vectors in `CfgState` instances.

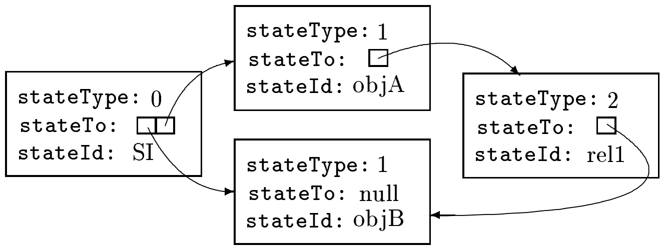

Fig. 4. `CfgState` instances

4 Registering Objects and Relations

The user of the library has to know only one method for registering objects and relations. The method `trcRegister` is polymorphic and implements all the necessary operations to create the corresponding states and transitions in the configuration automaton:

1. `trcRegister(TrcObj obj)` to register objects one wants to trace.
2. `trcRegister(TrcRel rel)` to register relations one wants to use when tracing.

A single initial state `SI` with `stateType = 0`, `stateId = null`, and `stateTo = null` is created automatically as part of the initialization of the configuration automaton. When registering an object `objA` we have the following procedure:

`Sobj = new CfgState(objA)`	which creates a new instance of the class `CfgState` with attributes `stateType = 1`, `stateId = objA`, and `stateTo = null`.
`addStateObj(SI,Sobj)`	which adds the newly created state `Sobj` to the list of states reached from the initial state `SI`, that is, `Sobj` will be an element of `SI.stateTo` vector.

When registering a relation instance `rel1` of class `rel` relating objects `objA` and `objB` we have the following procedure:

`Srel = new CfgState(rel1)`	which creates a new instance of the class `CfgState` with attributes `stateType = 2`, `stateId = rel1`, and `stateTo = null`.
`addStateObj(SobjA,Srel)`	which adds the newly created state `Srel` to the list of states reached from the state `SobjA`, that is, `Srel` will be an element of `SobjA.stateTo` vector.
`addStateObj(Srel,SobjB)`	which adds the state `SobjB` to the list of states reached from the state `Srel`.

5 Tracing Methods

There are some public methods to allow the user of the library to get and manipulate tracing results.

1. `TrcGraph traceExpr(String strexpr)` returns all object and relation instances matching the tracing expression `strexpr`.

The `traceExpr` method implements tracing based on the matching of tracing expressions. A tracing expression is a pattern of objects and relations. In its most simple form a tracing expression consists of object and relation class identifiers. For example, for object identifiers obj_1, obj_2, and obj_3, and relation class identifiers $Derive$ and $Refine$,

$$\begin{gathered} obj_1\, Derive\, obj_2 \\ obj_1\, Derive\, obj_2\, Refine\, obj_3 \\ obj_1\, Derive\, obj_2\, Derive\, obj_3 \end{gathered}$$

are all tracing expressions.

Tracing expressions are used in a pattern matching procedure to verify if they can be recognized by the configuration automaton. The string `strexpr` is successfully traced if there are registered objects and relations satisfying the pattern. For example, the expression

```
objA Derive objB Derive objC
```

will trace successfully if there is an object `objA` related to an object `objB` which in turn is related to an object `objC`, all by relation instances of class `Derive`.

The tracing expression is really a regular expression where the symbols are object and relation class identifiers. Any sequence of the form

obj-identifier1 class-name obj-identifier2 will be matched whenever there are actual objects identified by *obj-identifier1* and *obj-identifier2*, related by an instance of the relation class *class-name*. The usual regular expression operators may be used. For example, the expression

```
objA Derive (objB | objC)
```

will trace successfully if there is an object `objA` related by a relation instance of class `Derive` to either an object `objB` or an object `objC`.

Tracing expressions are more flexible than usual regular expressions. For example, it is possible to express that the matching procedure should consider the attributes of objects and not their identifiers.

```
Requirement[priority = 3] Derive ObjB
```

The expression above is matched by any object of class `Requirement` that has the value 3 for its `priority` attribute and is related to object `objB` by an instance of the relation `Derive`. There are several other possibilities to write a tracing expression. All the possible forms and their formal details are presented elsewhere (Pinheiro 1997).

The result of the method `traceExpr` is an object of class `TrcGraph`, which is a graph with nodes representing objects and labelled arcs representing relationships between them. It is a convenient way of viewing trace results graphically.

There are also more specialized methods used to trace objects in both forward and backward directions.

2. `TrcGraph traceFFGraph(TrcObj obj)` returns all objects reached from `obj` in a forward direction.
3. `TrcGraph traceBWGraph(TrcObj obj)` returns all objects reached from `obj` in a backward direction.

The return types of these methods are also `TrcGraph`, which means that a graph is returned as a result of using them. But the user may get list of objects as a result of tracing using one of the following methods:

4. `TrcList traceFFList(TrcObj obj)` returns all objects reached from `obj` in a forward direction.
5. `TrcList traceBWList(TrcObj obj)` returns all objects reached from `obj` in a backward direction.

The `traceFFList` and `traceBWList` methods result in an object of class `TrcList` which is a list containing the objects reached from `obj` together with the relations used to relate them. The result is indeed a list of triples

$$< \mathtt{source}, \mathtt{rel}, \mathtt{target} >$$

each one indicating that object `source` is related to `target` by a relation instance of class `rel`.

Given a trace result of type `TrcGraph` or `TrcList` the user may manipulate its elements. For example, there methods for traversing the graph, probing the result for specific objects, and getting the next element of the list. There are also methods to directly inquiry the existence of relationships between objects:

6. `Boolean isrelated(TrcObj source, TrcObj target)` returns true if the object `source` is related to `target` by a relation of any kind.
7. `Boolean isrelated(TrcObj source, String rel, trcObj target)` returns true if there is a relation of class `rel` relating `source` to `target`.
8. `Boolean isrelated*(TrcObj source, TrcObj target)` returns true if objects `source` and `target` are related by a chain of relations, with possible intermediate objects.

6 Viewing Results

The library has classes for manipulating tracing results in a graphical way. These widgets are provided in terms of frames to be used in a window environment.

- `TrcGraphFrame`. This class allows the viewing of trace results graphically. The nodes are mouse sensitive and activate pushdown menus with options to show object's properties. There are also buttons to select some classes from the result such that only objects of these classes are visible.
- `TrcListFrame`. This class allows the viewing of trace results in form of a list. It is also possible to selectively choose the classes of the visible objects.
- `TrcMatrixFrame`. This class allows the viewing of traceability matrices.

These classes are subclasses of the Java class `Frame`. They serve as containers to hold complex objects. Each one of these classes has an attribute to hold the data to be shown. The data for the `TrcGraphFrame` is a graph, i.e., an object of class `TrcGraph`. The data for the `TrcListFrame` is a list, i.e., an object of the class `TrcList`. The `TrcMatrix` is the class of the objects hold by the `TrcMatrixFrame`.

There are specific methods to generate traceability matrices:

- `traceMatrix(Vector Lclass, Vector Cclass)` returns a `TrcMatrix` object which is a matrix with a line for each object of class `Lclass` and a column for each object of class `Cclass` and with elements (i, j) marked if the object in line i is related to the object in column j by any relation.
- `traceMatrix(TrcRel rel, Vector Lclass, Vector Cclass)` returns an object of class `TrcMatrix` with a line for each object of class `Lclass` and a column for each object class `Cclass` and with elements (i, j) marked if the object in line i is related to the object in column j by a relation of class `rel`.

If a more detailed matrix is needed one can use the `traceMatrix` method specifying a list (class `MtrList`) of individual objects to be considered as elements of lines and columns.

– `traceMatrix(MtrList Lobj, MtrList Cobj)` returns a `TrcMatrix` object with a line for each object specified in the list `Lobj` and a column for each object specified in the list `Cobj` and with elements (i, j) marked if the object in line i is related to the object in column j by any relation.
– `traceMatrix(TrcRel rel, MtrList Lobj, MtrList Cobj)` returns a matrix object with a line for each object specified in the list `Lobj` and a column for each object specified in the list `Cobj` and with elements (i, j) marked if the object in line i is related to the object in column j by a relation of class `rel`.

It is even possible to specify specific relation classes for each pair (i, j) such that the matrix element (i, j) will be marked if the object in line i is related to the object in column j by the relation specified for (i, j). The signature for this method usage is not given here.

The data for these viewer classes may be maintained statically or dynamically. Statically, once the data is shown, say from a `TrcGraph` object, it is not modified anymore. There are specific methods to update the contents of a viewer class object, say generating a new `TrcGraph` object to be used as data source. Dynamically, every time an object or relation is registered or unregistered the data used as source may be regenerated:

1. In the event of registering a new object or relation, it is verified if the tracing result object been shown (graph, list, or matrix) should be modified. If so, then a new (graph, list, or matrix) tracing result object is generated and used as data source.
2. In the event of an object or relation being unregistered, it is verified if the object is been shown as part of any instance of a viewer class (`TrcGraphFrame`, `TrcListFrame`, or `TrcMatrixFrame`). If so, then a new (graph, list, or matrix) tracing result object is generated and used as data source.

7 Conclusions

The library components presented here are useful for incorporating traceability features into software development environments. The elements of the library basically implement structures to maintain a configuration state of objects and relations and mechanisms to update and retrieve information from it. The library does not define any tracing model and does not enforce any tracing method.

As with any library, its use should be carefully thought in advance and the tracing model to be implemented with the library should be designed to meet specific user needs. Ramesh (1998) identifies two kinds of traceability users. Low end users use traceability to

1. model dependencies among requirements,
2. allocate requirements to system components, and
3. establish links to compliance verification procedures.

High-end users additionally use traceability to

4. capture process-related information, such as design rationale, and
5. capture the evolution of various artifacts.

High-end users of traceability demand to know what information should be captured, by whom, and how it should be used. They also require that the critical elements of the software process should be traced to their stakeholders.

Of the above list of possible traceability usage, the one in item 5, referring to the evolution of artifacts, is not properly captured by any element of the library. Also, the usage mentioned in item 4 is only captured to the extent that the process-related information may be represented by a tangible artifact like a design rationale. Other types of process information that could be traced, like assignment of tasks to individuals and organizational hints to specific expertise (Rose 1998) are of a more difficult nature.

Another critical feature not addressed by the library is the generation of traceability documentation. But in this case one may easily devise ways of extending the library to do so.

The library presented here is not enough to solve many of the important problems associated with the implementation of traceability procedures. These problems are to a large extent social, related to environmental and organizational issues. Nevertheless, the building of software development environments and applications incorporating traceability features is made easier by the use of a flexible library of tracing classes. The design we presented here is of simple use. For each software environment or application been developed one should:

1. Import the tracing classes from the library.
2. Define the class hierarchy of the traceable objects and relations by declaring the classes he wants to consider as subclasses of either `TrcObj` or `TrcRel`.
3. Write down specific commands to register and unregister objects and relations. This may be facilitated by a good structuring of the application.
4. Define the interface, incorporating the trace viewer components he wants to use.

At the moment the library is in a design stage. Thus, we lack a concrete example showing its actual use. We expect the features discussed here to be fully developed in the forthcoming months as a result of a master's thesis. As part of the work yet to be done we have to address issues like the storage of tracing data in persistent media, configuration management facilities to deal with versioning of objects, and the use of the library in a multi-user, distributed environment, dealing with multiple viewpoints and web-based development.

Acknowledgements

The anonymous referees provided useful comments that led to improvements in this paper.

References

Davis, A. M.: Software Requirements: Analysis & Specification. Prentice-Hall International (1990)

Dömges, R., Pohl, K.: Adapting Traceability Environments to Project Specific Needs. Communications of the ACM **41**:12 (1998) 54–62

Gotel, O.: Contribution Structures for Requirements Traceability. Doctoral dissertation, Imperial College, Department of Computing, London, August (1995)

Jarke, M.: Requirements Tracing. Communications of the ACM **41**:12 (1998) 32–36

Palmer, J.D.: Traceability. in: R. H. Thayer and M. Dorfman (editors): Software Requirements Engineering. IEEE Computer Society Press (1997)

Pinheiro, F.A.C.: Design of a Hyper-Environment for Tracing Object-Oriented Requirements. Doctoral dissertation, University of Oxford, Oxford University Computing Laboratory, Oxford, UK (1997)

Pinheiro, F.A.C., Goguen, J.A.: An Object-Oriented Tool for Tracing Requirements. IEEE Software **13**:2 (1996) 52–64

Ramesh, B.: Factors Influencing Requirements Traceability Practice. Communications of the ACM **41**:12 (1998) 37–44

Rose, T.: Visual Assessment of Engineering Processes in Virtual Enterprises. Communications of the ACM **41**:12 (1998) 45–52