Bachelor's thesis

„Documentation of Decisions During the Implementation Phase Through Code Annotations“

Submitted by: Arthur Kühlwein

Registration number: 2900019

E-Mail: a.kuehlwein@stud.uni-heidelberg.de

Supervisors: Prof. Dr. Barbara Paech*, Tom-Michael Hesse*

*Chair of Software Systems
of the Faculty of Mathematics and Computer Science

at the Heidelberg University

29.07.2014

Abstract

Many decisions which are made during the implemetation phase of software projects are documented either not at all or through unstructured comments within the source code. This makes the rationale, i.e. the underlying issue with its context, alternative solutions and criteria behind the decision implicit. As a consequence, reproducing the decision later on becomes very difficult. It is thus necessary to make as much of this decision knowledge as possible explicit, since decisions made during the earlier stages are frequently re-evaluated, changed or discarded during the evolution of a software project.

This thesis investigates the possibilities and limitations of documenting decision knowledge through annotations in the source code within the Eclipse IDE. For this, an annotation schema is developed and implemented as an Eclipse plug-in. It is investigated, how this schema can be made flexible in the sense that project-specific adjustments to the structure of the decision knowledge should be possible with as less time and effort as possible. Also, annotated decisions are mapped to and kept consistent with an external UNICASE documentation.

Kurzbeschreibung

Viele Entscheidungen, die innerhalb der Implementierungsphase von Softwareprojekten getroffen werden, werden oft entweder gar nicht oder nur stichpunktartig in unstrukturierten Kommentaren beschrieben. Dadurch bleibt das Entscheidungswissen, d.h. der Problemkontext mit seinen Alternativen und Kriterien, implizit, was es im Nachhinein sehr schwierig macht, die Entscheidung nachzuvollziehen. Im Rahmen der Softwareevolution werden viele Entscheidungen neu bewertet, verändert oder verworfen. Deshalb ist es notwendig, so viel Entscheidungswissen wie möglich explizit zu machen.

Diese Arbeit untersucht die Möglichkeiten und Einschränkungen, Entscheidungswissen über Annotationen im Quellcode der Eclipse IDE zu dokumentieren. Dafür wird ein Annotationsschema entwickelt und als Eclipse Plug-in implementiert. Es wird untersucht, inwieweit das Annotationsschema flexibel verändert werden kann, sodass projektspezifische Anpassungen an der Wissensstruktur mit so wenig Aufwand wie möglich umgesetzt werden können. Ferner werden annotierte Entscheidungen in eine externe UNICASE-Dokumentation überführt und mit dieser konsistent gehalten werden.

Eidesstattliche Erklärung zur Bachelorarbeit

Hiermit erkläre ich, dass ich die Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe.

Die Arbeit habe ich bisher keinem anderem Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher nicht veröffentlicht.

Heidelberg, den ....

....
(Arthur Kühlwein)

Contents

1 Introduction **1**

1.1 Background and Aim of this Thesis . . . 1

1.2 Structure of this Thesis . . . 1

2 Basic Concepts **2**

2.1 The Life Cycle of a Software Project . . . 2

2.2 Decision-Making During the Implementation Phase . . . 3

2.2.1 Naturalistic and Rational Decision-Making . . . 3

2.2.2 Information Needs in Design Decisions . . . 4

2.3 The Decision Documentation Model . . . 5

2.4 Java Code Annotations . . . 8

2.5 Eclipse . . . 8

2.5.1 Resource Markers . . . 9

2.5.2 Resource Markers vs. Java Code Annotations . . . 10

2.5.3 Extensions and Extension Points . . . 11

2.6 Existing Approaches . . . 11

2.6.1 CM^2 . . . 11

2.6.2 Lougher and Rodden's Approach . . . 13

2.6.3 SEURAT . . . 14

2.6.4 Comparison and Evaluation . . . 15

3 Approach **17**

3.1 Annotation Classes . . . 19

3.1.1 Core Annotations . . . 19

3.1.2 Augmented Annotations . . . 21

3.2 Granularity . . . 22

3.3 Locality . . . 23

3.4 Mapping Actions on Annotations to Decision Elements . . . 24

3.4.1 Creating a New Annotation . . . 25

3.4.2 Modifying an Annotation . . . 25

3.4.3 Deleting an Annotation . . . 26

3.5 Mapping Actions on Decision Elements to Annotations . . . 26

3.5.1 Creating a New Decision Element . . . 26

3.5.2 Modifying a Decision Element . . . 27

3.5.3 Deleting a Decision Element . . . 27

4 Requirements Analysis **28**
4.1 Use Cases . . . 28
4.1.1 Manage Augmented Annotation Types . . . 29
4.1.2 Associated System Functions . . . 29
4.2 UI Workflow and Mock-ups . . . 33
4.2.1 Core Annotations . . . 33
4.2.2 Augmented Annotations . . . 33
4.2.3 Code Associations . . . 33
4.3 Non-Functional Requirements . . . 37
4.3.1 Performance . . . 37
4.3.2 Intuitive UI and Workflow . . . 37
4.3.3 Robustness . . . 38
4.3.4 Extensibility . . . 38
4.3.5 UNICASE Look and Feel . . . 38

5 Design and Implementation **39**
5.1 Model and Program Logic . . . 39
5.1.1 Annotation Nature and Annotation Builder . . . 40
5.1.2 Annotations and the Annotation Manager . . . 41
5.2 User Interface . . . 43
5.2.1 Text Hover . . . 43
5.2.2 Code Associations View . . . 45
5.2.3 Augmented Annotation Manager . . . 45
5.2.4 Dialogs . . . 47
5.3 Interaction . . . 47
5.3.1 Creating a Core Annotation . . . 47
5.3.2 Creating a New Decision Element . . . 48
5.4 Quality Assurance . . . 49
5.4.1 Component Tests . . . 49
5.4.2 Integration Tests . . . 50
5.4.3 System Tests . . . 50

6 Discussion **51**

7 Conclusion **54**
7.1 Future Work . . . 54

References **55**

List of Figures 57

Appendix A Package Diagram 59

Appendix B Class Diagrams 60

B.1 Class Diagram for org.unicase.decision.annotations 60
B.2 Class Diagram for org.unicase.decision.annotations.builders 61
B.3 Class Diagram for org.unicase.decision.annotations.model 62
B.4 Class Diagram for org.unicase.decision.annotations.model.commands 63
B.5 Class Diagram for org.unicase.decision.annotations.dialogs 64
B.6 Class Diagram for org.unicase.decision.annotations.hover 65
B.7 Class Diagram for org.unicase.decision.annotations.providers 66
B.8 Class Diagram for org.unicase.decision.annotations.views 67
B.9 Class Diagram for org.unicase.decision.annotations.util 68

1 Introduction

1.1 Background and Aim of this Thesis

Rationale is the reasoning underlying the creation and use of artifacts [1, p. 3]. In the context of software development, an artifact can be anything ranging from the software project itself to a class diagram or even a single line of code. The value of documenting the rationale behind design decisions, especially in software projects, is widely acknowledged and a number of tools and frameworks exist which support the management of rationale. One such tool is UNICASE [2], which is a knowledge management tool integrating project and system knowledge directly into the Eclipse IDE [3, 5].

However, the rationale of decisions made during the implementation phase of a software project, which are referred to as *implementation decisions*, is either documented with unstructured comments within the source code or not documented at all. As a result, the rationale remains implicit, which makes it very difficult for developers to reproduce the decisions later on. This becomes particularly problematic during the evolution of a software project, where the composition of the development team can change and decisions made during earlier stages are subject to frequent re-evaluation, which may lead to a modification or rejection of the original decision.

In order to effectively use the rationale of implementation decisions, it is necessary to make this rationale explicit, with as less effort for the developer as possible. Java, which is the main programming language to be used in the Eclipse IDE, permits the creation and use of custom code annotations as of release 5.0 of the Java Platform [6]. The aim of this thesis is to investigate the possibilities and limitations of using Java-style code annotations to document decision knowledge during the implementation phase, while providing a high degree of flexibility and simplicity regarding project-specific adjustments to the structure of the decision knowledge. Also, annotated decisions are mapped to and kept consistent with an external UNICASE documentation. This structures the decision knowledge while making it explicit as well as easily accessible to all members of the development team.

For this, a custom annotation schema is developed and implemented as an Eclipse plug-in. To prove the functionality of the developed annotation schema and plug-in, decision knowledge regarding the plug-in itself is annotated using the schema.

1.2 Structure of this Thesis

Section 1 served as an introduction to the thesis' context, motivation and aim. Section 2 explains the basic concepts and provides detailed background knowledge. Section 3 describes the approach towards the development of the annotation schema. Section 4 explains the analysis of requirements for the implementation of the previously introduced schema, while Section 5 describes the implementation of the schema as a plug-in for the Eclipse IDE. We will discuss the results of our work in Section 6. Section 7 concludes this thesis and discusses future work.

2 Basic Concepts

In the following sections, we will explain the basic concepts which are relevant to our thesis.

2.1 The Life Cycle of a Software Project

The life cycle of a software project, which might last many years, consists of a number of phases. Figure 1 illustrates the different phases of a software project's life cycle along with examples of the rationale that is associated with each development phase. The main focus of our annotation schema lies in the implementation phase.

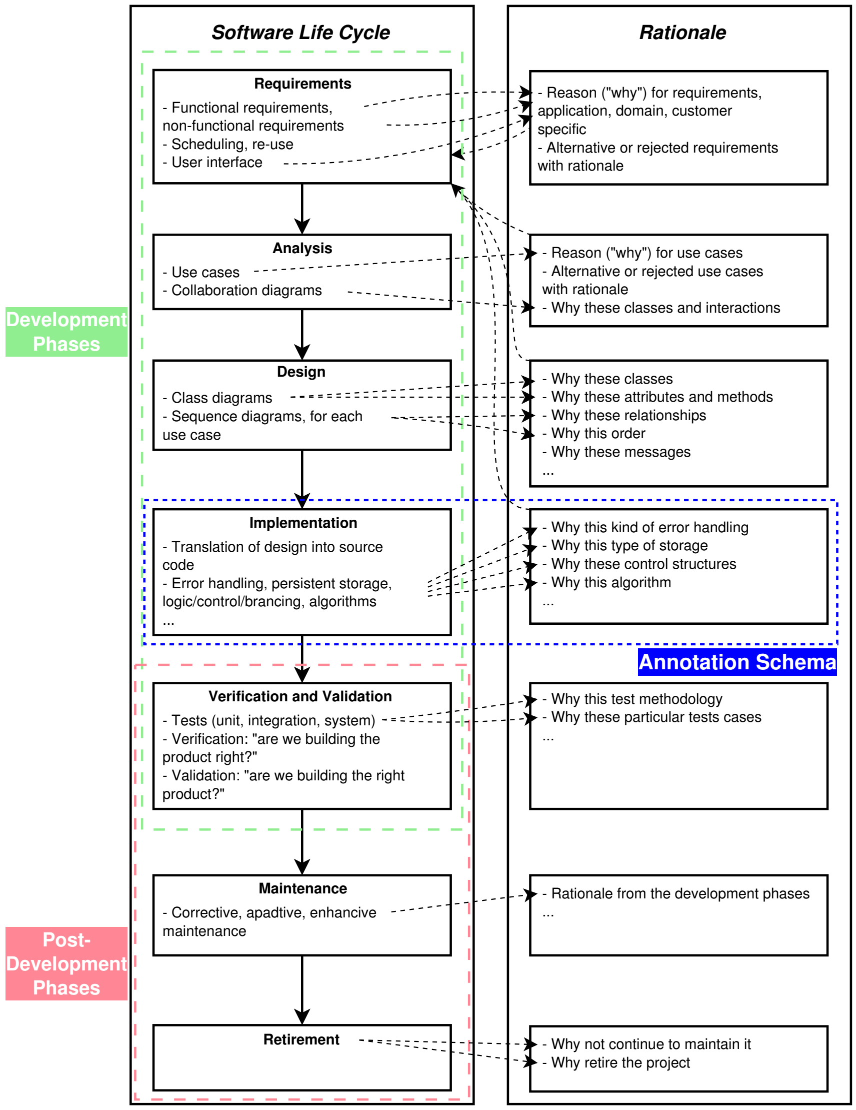

Software Life Cycle
Rationale
Requirements
- Functional requirements, non-functional requirements
- Scheduling, re-use
- User interface
- Reason ("why") for requirements, application, domain, customer specific
- Alternative or rejected requirements with rationale
Analysis
- Use cases
- Collaboration diagrams
- Reason ("why") for use cases
- Alternative or rejected use cases with rationale
- Why these classes and interactions
Development Phases
Design
- Class diagrams
- Sequence diagrams, for each use case
- Why these classes
- Why these attributes and methods
- Why these relationships
- Why this order
- Why these messages
...
Implementation
- Translation of design into source code
- Error handling, persistent storage, logic/control/branching, algorithms
...
- Why this kind of error handling
- Why this type of storage
- Why these control structures
- Why this algorithm
...
Annotation Schema
Verification and Validation
- Tests (unit, integration, system)
- Verification: "are we building the product right?"
- Validation: "are we building the right product?"
- Why this test methodology
- Why these particular tests cases
...
Maintenance
- Corrective, apadtive, enhancive maintenance
- Rationale from the development phases
...
Post-Development Phases
Retirement
- Why not continue to maintain it
- Why retire the project

Figure 1: The different phases in the life cycle of a software project along with the rationale associated with each development phase. Adapted from [14, Figure 2-1]

All phases can be categorized into development phases and post-development phases. We will only describe the implementation phase in detail, as this is the development phase our thesis is primarily concerned with.

During the implementation phase, all decisions made in the earlier phases are translated into executable source code. Naturally, there are still important decisions to be made. Capturing these decisions can be especially useful for software maintenance, where understanding the reasoning behind implementation decisions is important (see Section 2.6). The focus of application for our annotation schema lies in this phase.

2.2 Decision-Making During the Implementation Phase

To our knowledge, there exists no work explicitly dealing with the way programmers make decisions and which decision knowledge is of importance in the context of implementation decisions. However, a lot of research has been done on the subject of decision-making and relevant decision knowledge in the context of design decisions. The boundary between those neighboring development phases is blurred, i.e. the design phase contains activities related to the implementation phase and vice versa. Therefore, we can use the rich pool of knowledge regarding design decisions to - at least partially - describe the nature of implementation decisions.

In this section, we will present two papers on the nature of design decisions which are significant to our thesis, and apply this knowledge to implementation decisions.

2.2.1 Naturalistic and Rational Decision-Making

One key finding was made by Zannier et al. [4], who identified two seemingly opposing approaches by which software developers make decisions, *naturalistic decision making* (NDM) and *rational decision making* (RDM). Table 1 gives a short comparison of both approaches. As a result of their multi-case study among 25 software designers, they concluded that the approach which was chosen to make a design decision depended on the structure of the underlying problem.

	NDM	RDM
Characterized by	Appreciation for real-time scenarios	Appreciation for mathematical computation
	Judgment bias, no or few alternatives considered	Utility functions, probabilities, many alternatives considered
Problem structure	Ill-structured	Well-structured
Solution quality	Satisfactory	Optimal

Table 1: Comparison between NDM and RDM

RDM was primarily used for *well-structured problems* (WSPs), while NDM was primarily used for *ill-structured problems* (ISPs). A WSP is characterized by having apparent criteria by which an optimal solution can be achieved. A game of chess, for example, is a WSP. An ISP on the other hand is a problem which requires problem structuring, i.e. converting the ISP to a WSP. There exist no optimal solutions for an ISP, only ”good” or ”bad” ones. An example for an ISP is the design of a new house, where ”decisions made in early design sketches of the house establish structures under which following decisions are made”.

It is thus important that our annotation schema caters to both approaches, as we do not know a priori how a particular problem for an implementation decision is structured and which decision-making approach developers are going to use in order to solve this problem.

2.2.2 Information Needs in Design Decisions

Ko et al. [15], as a result of their two-month field study of 17 software developer teams at Microsoft, identified the information needs of software developers, including those involved in making design decisions. The knowledge sought in design (d_i) and writing code (c_i) was:

d_1: What is the purpose of this code?

d_2: What is the program supposed to do?

d_3: Why was the code implemented this way?

d_4: What are the implications of this change?

c_1: What data structures or functions can be used to implement this behavior?

c_2: How do I use this data structure or function?

c_3: How can I coordinate this code with this other data structure or function?

A ranking of the most frequent unsatisfied information needs was compiled out of the identified information needs. The information need d_3 was ranked on number 2 with 44% of queries being unsatisfied and a maximum observed search time of 21 minutes; note however that this ranking ”may reflect that 11 of the 17 participants’ teams were in a bug fixing phase”. Also, d_3 was rated as important by 61%, as unavailable by 37% and as inaccurate by 39% of the participating developers.

These findings prove that explicit documentation regarding the rationale of an implementation decision is generally regarded as important; however, this information need often remains unsatisfied.

Falessi et al. [16] made a number of deliberations regarding the utility of *design decision rationale documentation* (DDRD) depending on its purpose, i.e. the role of the DDRD consumer. The different roles are represented by five different DDRD use cases, of which the use case ”identification of wrong knowledge on the solution space” comprises designers and architects as beneficiaries of the DDRD.

They confirmed their deliberations in two independent studies [16, 17], showing that some DDRD information was commonly regarded as ”required” by participants belonging to different roles (see Figure 2).

As can be seen in Figure 2, issues, decisions, positions and arguments were rated as ”required” not only by more than 60% of the participating designers and architects, but also

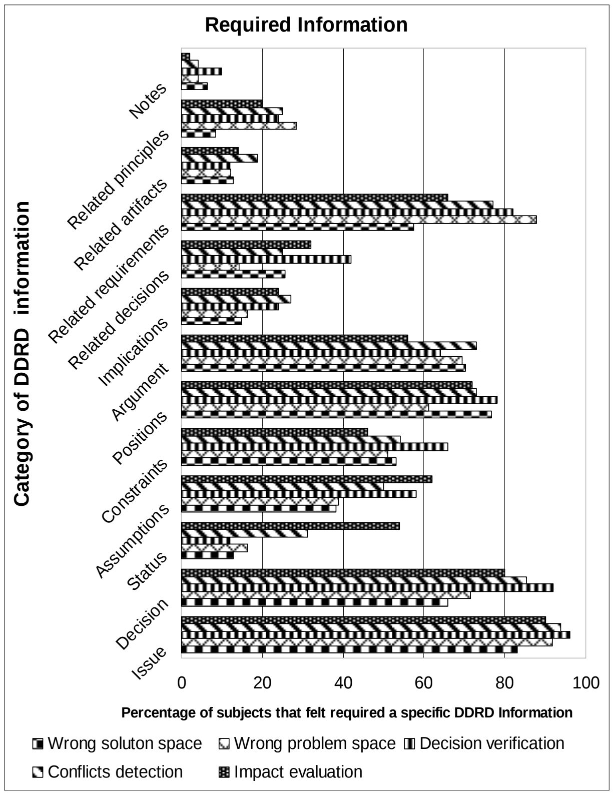


Figure 2: Rating of different DDRD information categories as "required" by subjects. From [16]

by a similar percentage of participants belonging to different roles. This implies that the explicit documentation of the rationale behind design decisions, and thus - to a certain extent - implementation decisions will be beneficial to all members of the development team.

2.3 The Decision Documentation Model

The Decision Documentation Model (*DDM*) was introduced by Hesse and Paech [5]. The model serves as a structure for documenting decision knowledge and primarily consists of two elements: *DecisionStatements* and *DecisionComponents*. A *DecisionStatement* represents a decision in its entirety along with all administrative information needed. It may depend on other *DecisionStatements* via the *dependsOn*-relation and can contain any number of *DecisionComponents*, which represent knowledge related to the decision. Each *DecisionComponent* can contain further *DecisionComponents* itself and may be linked to other knowledge elements via the *concerns*-relation. It is the parent class of all knowledge elements that describe the decision more in detail, such as arguments, questions, solutions, etc. An *AbstractKnowledgeElement* represents any knowledge element in the system. The general structure of the model is illustrated in Figure 3.

The *DecisionComponents* belonging to a *DecisionStatement* form a hierarchy among themselves. Also, they form a network of different relations.

While the DDM has been originally designed to support design decisions on a high level of abstraction, the model can be just as well used on implementation decisions; a short

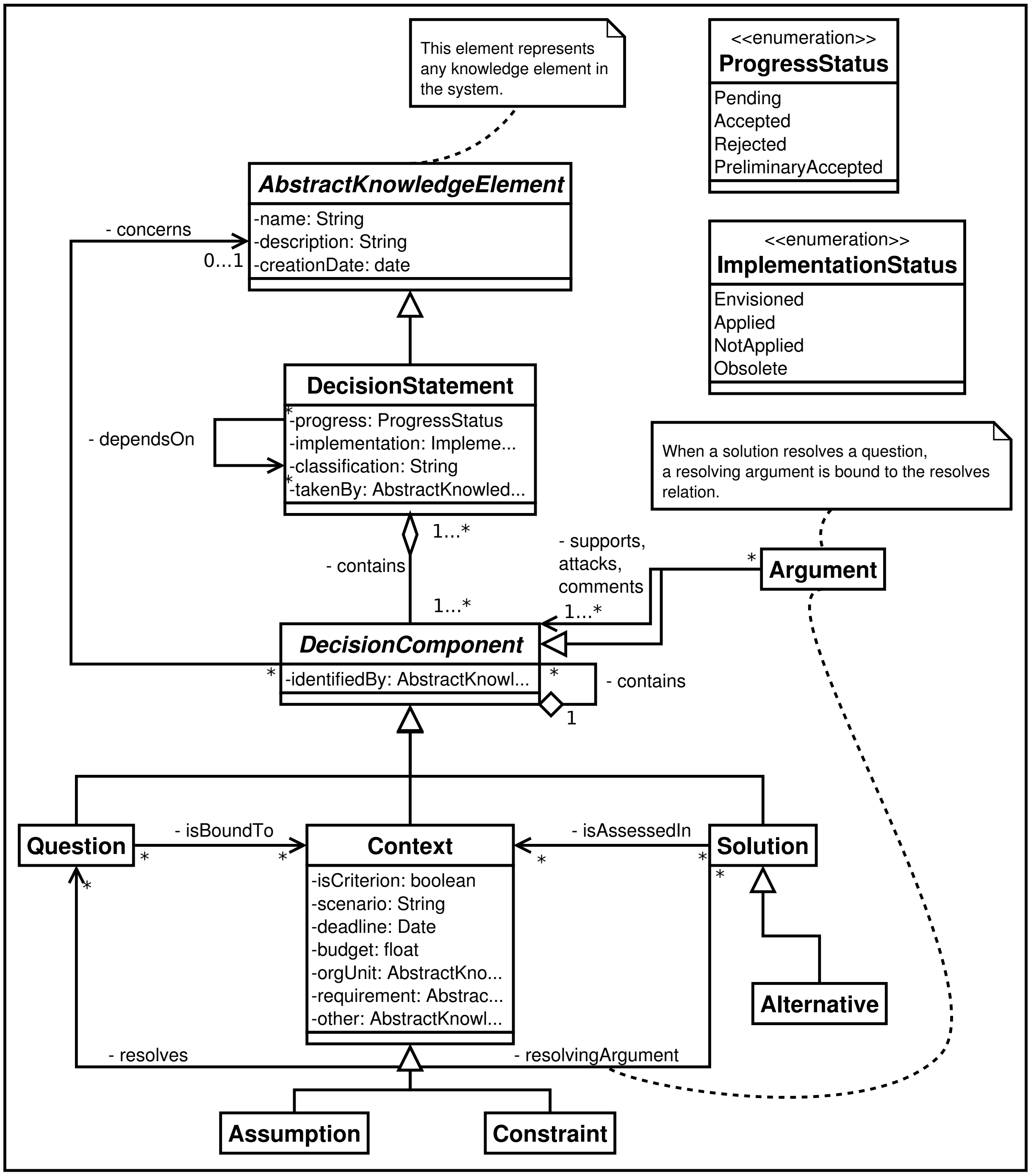


Figure 3: Structure of the DDM. Adapted from [5]

example shall illustrate this.

We assume there is a document export plug-in for a CASE tool, which exports model elements from a project into a PDF file. This functionality is provided to the user via a document export wizard, in which he or she can select the model elements that are to be exported. However, no functionality to export a project in its entirety exists at the moment, so the decision is made to implement this functionality. As the back-end functionality to export a project is already there, it is only necessary to add a wizard for users who want to export a project. The structure of applying the model to this decision is illustrated in Figure 4.

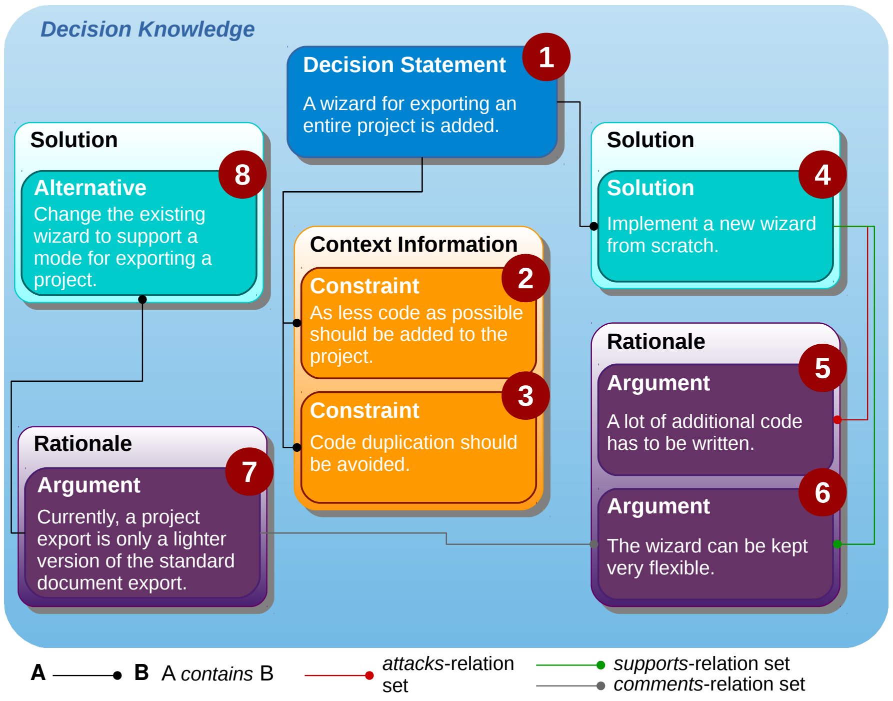

Decision Knowledge
Decision Statement
1
A wizard for exporting an entire project is added.
Solution
Alternative
8
Change the existing wizard to support a mode for exporting a project.
Solution
Solution
4
Implement a new wizard from scratch.
Context Information
Constraint
2
As less code as possible should be added to the project.
Constraint
3
Code duplication should be avoided.
Rationale
Argument
5
A lot of additional code has to be written.
Argument
6
The wizard can be kept very flexible.
Rationale
Argument
7
Currently, a project export is only a lighter version of the standard document export.
A B A contains B
attacks-relation set
supports-relation set
comments-relation set

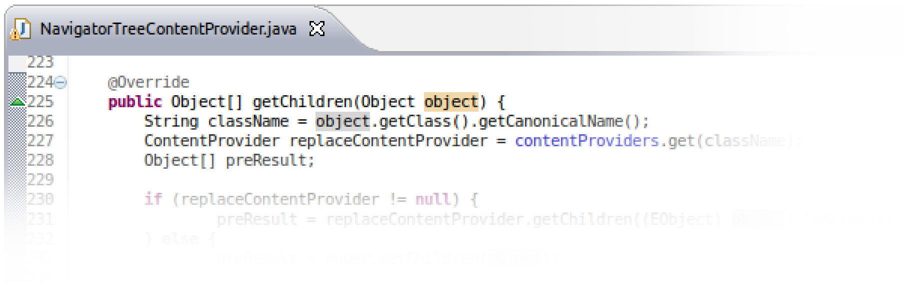
Figure 4: The DDM applied on the example. Adapted from [5]

We see that in the first iteration, the decision along with all contextual information regarding the decision is added to the decision knowledge (steps 1, 2 and 3). The contextual information consists of two constraints, stating that code addition and code duplication should be minimized. In the second iteration, a solution is proposed, suggesting that a new wizard should be implemented from scratch (step 4). An attacking argument is then added, stating that in order to do this, a lot of additional code has to be written, a lot of which might also be duplicated code from the existing wizard (step 5). A supporting argument is also added, which claims that implementing the wizard in this fashion will keep it very flexible (step 6). In the third iteration, the supporting argument in step 6 is commented by another argument, which states that in the software's current version, a project export is only a lighter version of the standard document export, eliminating the need for flexibility (step 7). To reflect this, an alternative solution is provided, suggesting that the existing document export wizard should be adjusted to support a mode for exporting a project (step 8).

The model has been partially implemented in UNICASE and will serve as the basis for both design and implementation of the annotation schema introduced in this thesis.

2.4 Java Code Annotations

In the Java platform, annotations are a form of meta data used on Java source code. Annotation-dependent behavior is triggered on annotated source code during compilation. Annotations start with an at-sign (@), followed by the annotation type. The Java platform provides a set of built-in annotations and, as of release 5.0 of the Java platform, permits the definition of custom annotations [6]. Figure 5 shows an example of a typical Java annotation. Here, the *@Override* annotation is used to mark methods which override a method of their super class.



NavigatorTreeContentProvider.java

```
    @Override
    public Object[] getChildren(Object object) {
        String className = object.getClass().getCanonicalName();
        ContentProvider replaceContentProvider = contentProviders.get(className);
        Object[] preResult;

        if (replaceContentProvider != null) {
                preResult = replaceContentProvider.getChildren((EObject) [illegible]
        } else {
                preResult = super.getChildren(object);
```

Figure 5: An example of a Java annotation

In order to be able to use a custom annotation, the annotation needs to be declared and an annotation processor associated with this annotation needs to be implemented [8]. This annotation processor is then responsible for handling source code decorated with this annotation.

The annotation processors are called by the *Annotation Processing Tool* (apt), which first determines what annotations are present on the source code at hand and then looks for any annotation processor factories that process the found annotations. If apt has found an annotation processor factory for an annotation, the respective annotation processor is run.

In order to develop a custom annotation processor, the *AnnotationProcessor* interface in the package *com.sun.mirror.apt* needs to be implemented. This interface has the method *process()* which is used by the apt tool to invoke the processor, which may be responsible for processing more than one annotation type.

Also, *AnnotationProcessorFactory* needs to be implemented, which returns an instance of the processor. The method *getProcessorFor()* is called by apt to get the processor, which gets an *AnnotationProcessorEnvironment* from apt. This environment provides all necessary contextual information for the processor, including references and means to communicate and cooperate with apt.

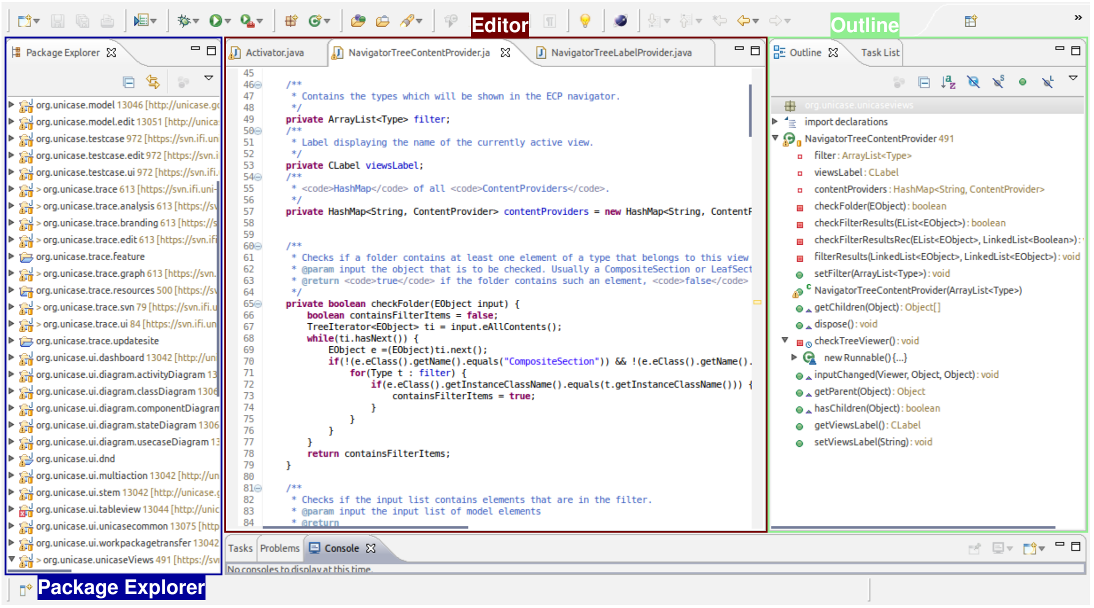
Processor factories can be found either by specifying the ”-factory” command line option or using the apt discovery procedure.

2.5 Eclipse

Eclipse is a highly extensible integrated development environment (IDE), which can, by means of various plug-ins, support a number of different programming languages. It also serves as a tool platform which can be used to develop and run applications, such as UNICASE. This is made possible by Eclipse’s Rich Client Platform (RCP), which comprises

- the core platform,
- the Equinox OSGi[1],
- the Standard Widget Toolkit (SWT)[2],
- JFace[3] and
- the Eclipse Workbench.

The Eclipse Workbench contains all views, perspectives, wizards and editors within Eclipse. A typical configuration for the Eclipse Workbench can be seen in Figure 6.


Editor
Outline
Package Explorer
Activator.java
NavigatorTreeContentProvider.ja
NavigatorTreeLabelProvider.java
Outline
Task List
Tasks
Problems
Console
org.unicase.model 13046 [http://unicase.g
org.unicase.model.edit 13051 [http://unica
org.unicase.testcase 972 [https://svn.ifi.un
org.unicase.testcase.edit 972 [https://svn.i
org.unicase.testcase.ui 972 [https://svn.ifi.
> org.unicase.trace 613 [https://svn.ifi.uni-
> org.unicase.trace.analysis 613 [https://sv
> org.unicase.trace.branding 613 [https://s
> org.unicase.trace.edit 613 [https://svn.ifi
org.unicase.trace.feature
> org.unicase.trace.graph 613 [https://svn.
org.unicase.trace.resources 500 [https://sv
> org.unicase.trace.svn 79 [https://svn.ifi.u
> org.unicase.trace.ui 84 [https://svn.ifi.un
org.unicase.trace.updatesite
org.unicase.ui.dashboard 13042 [http://un
org.unicase.ui.diagram.activityDiagram 13
org.unicase.ui.diagram.classDiagram 1306
org.unicase.ui.diagram.componentDiagram
org.unicase.ui.diagram.stateDiagram 1306
org.unicase.ui.diagram.usecaseDiagram 13
org.unicase.ui.dnd
org.unicase.ui.multiaction 13042 [http://un
org.unicase.ui.stem 13042 [http://unicase.g
org.unicase.ui.tableview 13044 [http://unic
org.unicase.ui.unicasecommon 13075 [http
org.unicase.ui.workpackagetransfer 13042
> org.unicase.unicaseViews 491 [https://sv
org.unicase.unicaseviews
import declarations
NavigatorTreeContentProvider 491
filter : ArrayList<Type>
viewsLabel : CLabel
contentProviders : HashMap<String, ContentProvider>
checkFolder(EObject) : boolean
checkFilterResults(EList<EObject>) : boolean
checkFilterResultsRec(EList<EObject>, LinkedList<Boolean>) :
filterResults(LinkedList<EObject>, LinkedList<EObject>) : void
setFilter(ArrayList<Type>) : void
NavigatorTreeContentProvider(ArrayList<Type>)
getChildren(Object) : Object[]
dispose() : void
checkTreeViewer() : void
new Runnable() {...}
inputChanged(Viewer, Object, Object) : void
getParent(Object) : Object
hasChildren(Object) : boolean
getViewsLabel() : CLabel
setViewsLabel(String) : void

```java

    /**
     * Contains the types which will be shown in the ECP navigator.
     */
    private ArrayList<Type> filter;
    /**
     * Label displaying the name of the currently active view.
     */
    private CLabel viewsLabel;
    /**
     * <code>HashMap</code> of all <code>ContentProviders</code>.
     */
    private HashMap<String, ContentProvider> contentProviders = new HashMap<String, ContentF


    /**
     * Checks if a folder contains at least one element of a type that belongs to this view
     * @param input the object that is to be checked. Usually a CompositeSection or LeafSect
     * @return <code>true</code> if the folder contains such an element, <code>false</code>
     */
    private boolean checkFolder(EObject input) {
        boolean containsFilterItems = false;
        TreeIterator<EObject> ti = input.eAllContents();
        while(ti.hasNext()) {
            EObject e =(EObject)ti.next();
            if(!(e.eClass().getName().equals("CompositeSection")) && !(e.eClass().getName().
                for(Type t : filter) {
                    if(e.eClass().getInstanceClassName().equals(t.getInstanceClassName())) {
                        containsFilterItems = true;
                    }
                }
            }
        }
        return containsFilterItems;
    }

    /**
     * Checks if the input list contains elements that are in the filter.
     * @param input the input list of model elements
     * @return
```

```
No consoles to display at this time.
```

Figure 6: The Eclipse Workbench. From left to right: Package Explorer, Editor, Outline

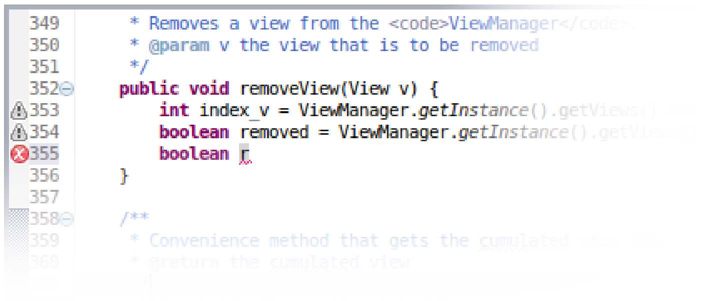
One interesting feature of the Eclipse Workbench, and the editor in particular, is the possibility to decorate resources within the Workbench using resource markers.

2.5.1 Resource Markers

Markers are light-weight objects, which are used to decorate any resource contained in the Eclipse Workbench [9]. These resources, for example, can be source code files, or the source code contained in these files as they are being displayed in the editor.

By default, the editor uses a number of different markers, for example to warn users about problematic sections or errors in their source code. An example of warning and error markers in the editor can be seen in Figure 7. Markers can show an icon on the small vertical bar on the left hand side of the editor and underline relevant code sections in that particular line with color.

[1] http://www.eclipse.org/equinox/
[2] http://www.eclipse.org/swt/
[3] http://wiki.eclipse.org/index.php/JFace



```
     * Removes a view from the <code>ViewManager</code>.
     * @param v the view that is to be removed
     */
    public void removeView(View v) {
        int index_v = ViewManager.getInstance().getViews().
        boolean removed = ViewManager.getInstance().getViews(
        boolean r
    }

    /**
     * Convenience method that gets the cumulated view
     * @return the cumulated view
     */
```

Figure 7: Active error and inactive warning markers in the Eclipse editor

Plug-ins may contribute custom markers to the Eclipse editor via the Resources plug-in [10]. These markers have to inherit their type from at least one existing marker type, which can be either from one of the five standard types defined in the Resources plug-in, or any other marker type defined in external plug-ins. The five standard types defined in the Resources plug-in are:

- *org.eclipse.core.resources.marker*
- *org.eclipse.core.resources.taskmarker*
- *org.eclipse.core.resources.problemmarker*
- *org.eclipse.core.resources.bookmark*
- *org.eclipse.core.resources.textmarker*

A marker can contain any number of attributes, which are maintained as name/value pairs, where the names are strings and the values can be any one of the supported value types. Supported value types are boolean, integer and string.

Plug-ins that contribute markers are solely responsible for creating markers, changing their attributes and removing them when they are not needed anymore.

2.5.2 Resource Markers vs. Java Code Annotations

There are similarities between Java's code annotations and Eclipse's resource markers, but also a number of differences.

Annotations mainly serve as instructions for the Java compiler and each annotation type has to be defined in a separate file. This makes the creation of annotation types during runtime difficult. Also, placing multiple annotations of the same type on one element is only supported in the Java Platform 8.0, which is the newest release as of June 2014.

Markers are more light-weight and fully utilize Eclipse's plug-in system. Marker types are defined by the plug-in and can be created and removed during runtime through Eclipse's *addContribution()* and *removeExtension()* methods. This vastly enhances flexibility compared to Java annotations.

2.5.3 Extensions and Extension Points

Each Eclipse plug-in may provide *extension points*. An extension point allows other plug-ins to contribute *extensions* to the extension point.

Plug-ins that provide extension points are also responsible for evaluating the extensions. A contribution to an extension point can comprise source code or data.

Eclipse, for example, provides a number of extension points that, among others, allow plug-ins to add additional views or extend the IDE's functionality.

Our plug-in will provide a number of extension points to enable extensibility.

2.6 Existing Approaches

In this section, we will present a selection of three existing approaches which support the documentation of decisions through annotating source code. None of these approaches deal with implementation decisions explicitly. Instead, the first two approaches mainly focus on decisions made during the maintenance phase of a software project's life cycle, while the third approach focuses on the decisions made during the design phase.

Finally, we will compare and evaluate the presented approaches with our approach.

2.6.1 CM^2

Canfora et al. [7] introduced the Cooperative Maintenance Conceptual Model (CM^2), which structures information about design rationale "as a network of linked comments and defines a maintenance rationale that extends the idea of design rationale to cover both rationale concerned with analysis and design activities (*rationale in the large*) and rationale concerned with implementation activities (*rationale in the small*)".

Of primary concern here is the representation of rationale in the small. The proposed model allows the annotation and the linkage of both source code and its design documentation using the same flexible mechanism. Every maintenance programmer can "develop a network of comments to record and transmit the decisional dynamics concerned with the implementation decisions he or she made". When the programmer starts with his or her job on a maintenance request, CM^2 associates an implementation folder to the maintenance request, to which all comments made by the maintenance programmer team will be automatically linked. Figure 8 summarizes the annotation mechanism for the rationale in the small.

The gray boxes represent comments which have been created during a single maintenance process. These comments, which are used to annotate the source code, are associated to the corresponding implementation folder. In addition, "each comment can be linked to one or more other comments also belonging to different implementation folders".

An example of source code that has been annotated using CM^2 can be seen in Figure 9.

In this example, the maintenance team is to update a supermarket chain's information system in accordance to the change of the country's currency from Lira to Euro. The supermarket's application software as well as its underlying database should be modified to handle the new currency. Out of the available proposals, the best one was selected and implemented.

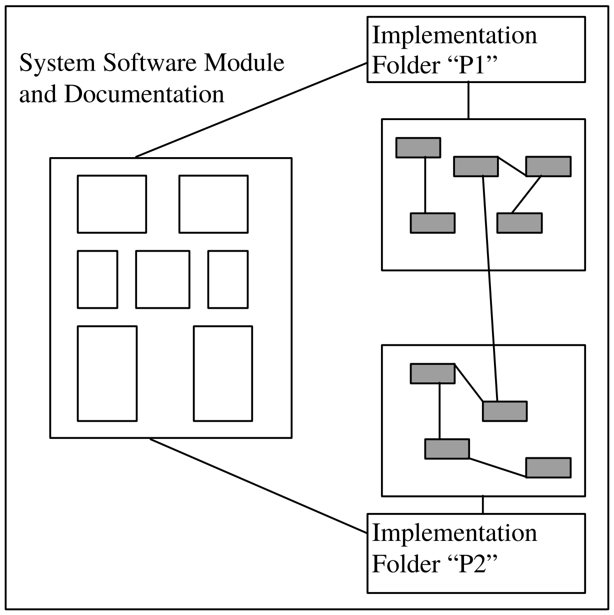
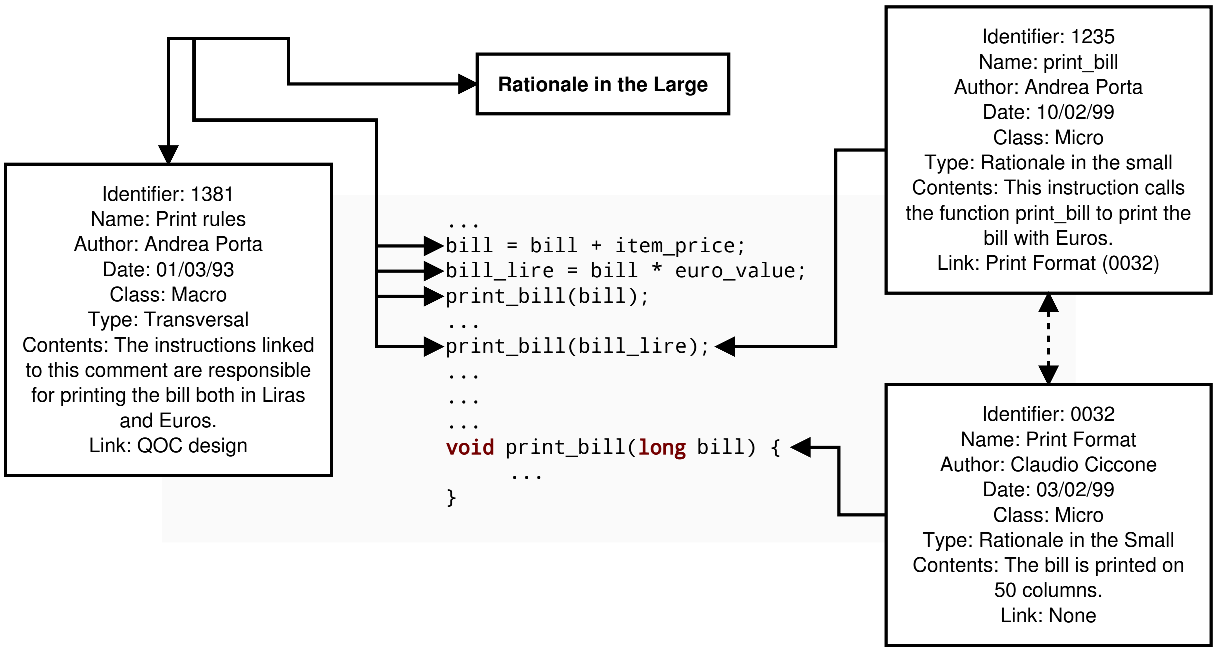

System Software Module and Documentation
Implementation Folder “P1”
Implementation Folder “P2”

Figure 8: Overview over CM^2’s annotation mechanism. From [7]



Identifier: 1381
Name: Print rules
Author: Andrea Porta
Date: 01/03/93
Class: Macro
Type: Transversal
Contents: The instructions linked to this comment are responsible for printing the bill both in Liras and Euros.
Link: QOC design

```
...
bill = bill + item_price;
bill_lire = bill * euro_value;
print_bill(bill);
...
print_bill(bill_lire);
...
...
...
void print_bill(long bill) {
      ...
}
```

Identifier: 1235
Name: print_bill
Author: Andrea Porta
Date: 10/02/99
Class: Micro
Type: Rationale in the small
Contents: This instruction calls the function print_bill to print the bill with Euros.
Link: Print Format (0032)

Identifier: 0032
Name: Print Format
Author: Claudio Ciccone
Date: 03/02/99
Class: Micro
Type: Rationale in the Small
Contents: The bill is printed on 50 columns.
Link: None

Figure 9: Example of a source code segment that has been annotated using CM^2. Adapted from [7]

The traversal link in Figure 9 directly links the comment to the rationale in the large, so maintenance programmers can quickly understand the solution to the problem in the example by analyzing the rationale in the large, links that connect this rationale to the corresponding rationale in the small and the rationale in the small itself.

2.6.2 Lougher and Rodden's Approach

Lougher and Rodden [13] presented a system which allows "members of a maintenance team to easily construct and structure a shared pool of maintenance information". The system's configuration can be seen in Figure 10.

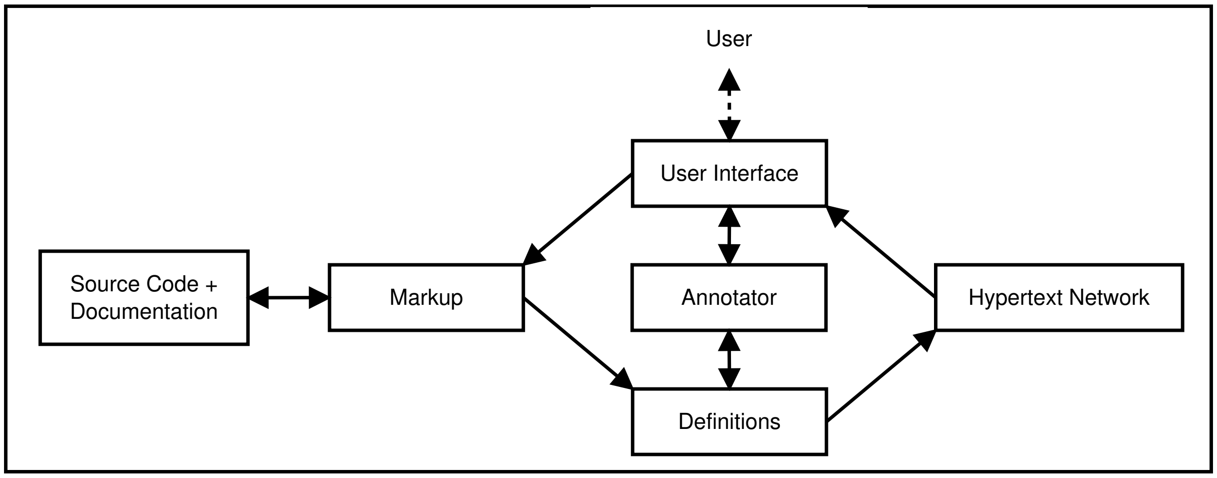


Figure 10: The configuration of the system behind Lougher and Rodden's approach. Adapted from [13]

The source code is annotated by attaching comments to the source code using a markup language "which specifies, among other things, how to recreate the hypertext network". The hypertext network allows the linkage between the maintenance rationale with the parts of the system that this rationale deals with. When the source code is loaded, it is passed up to the markup component, which consists of a parser and a generator. The parser extracts the documentation from the source code and passes it to the annotator, which incrementally builds the hypertext network from this information.

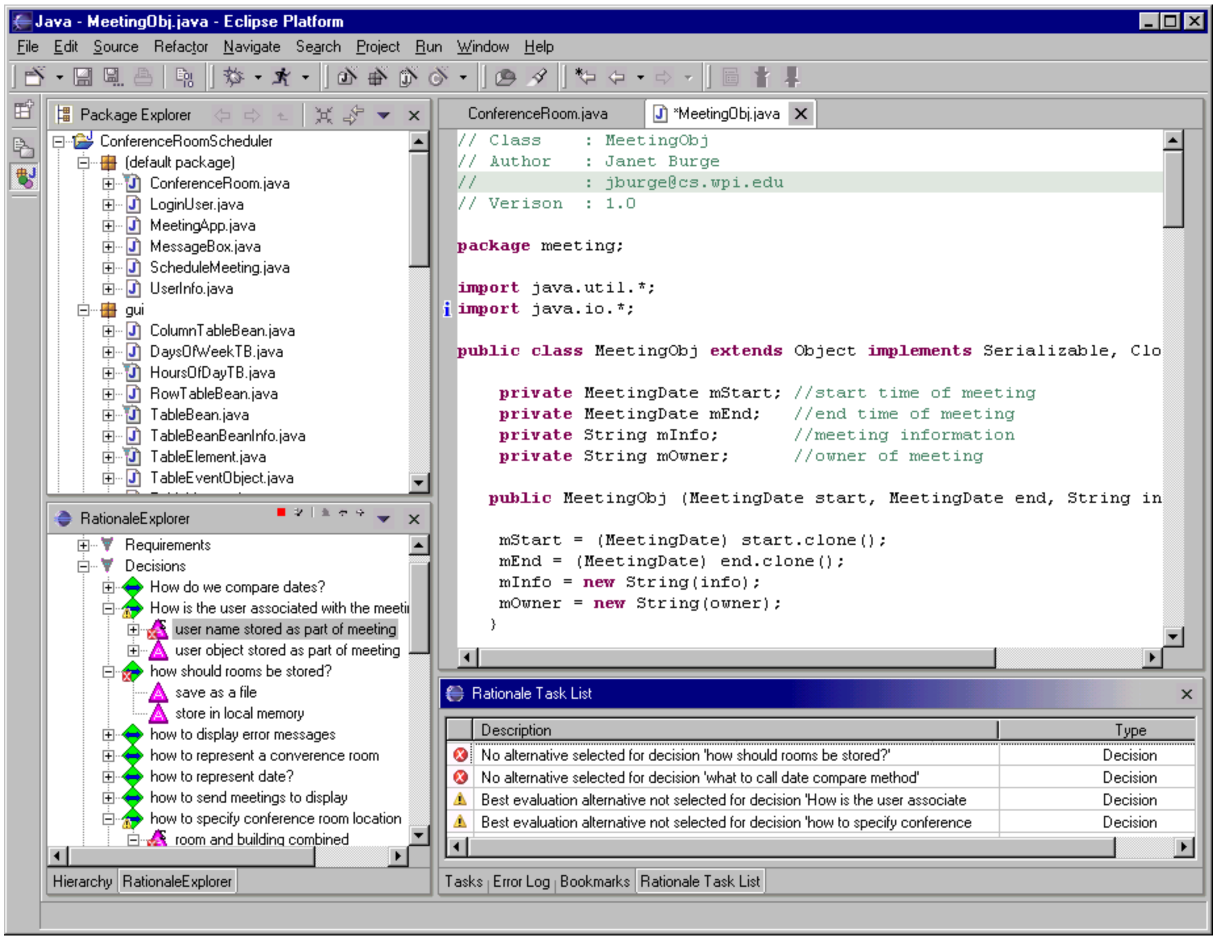
The user interface comprises an interactive editor, through which "the maintenance engineer can browse and edit the hypertext network". Source code can be freely edited, while references to annotations are kept consistent. Once the project has been saved, the "modified source code and hypertext network is passed back to the markup component, which generates the source code and embedded markup information for writing to file".

The comments are attached "by hypertext style links to various parts of the source code". These links are bi-directional and can be either linked to arbitrary sections within the source code, or arbitrary code fragments. Comments can also be attached to other comments, which enables "notes to be added quickly during modification".

2.6.3 SEURAT

Software Engineering Using RATionale (SEURAT) is a system ”developed to explore uses of design rationale” and was introduced by Burge and Brown [11]. ”It supports both the display of and inferencing over the rationale to point out any unresolved issues of inconsistencies”. SEURAT is integrated into the Eclipse IDE as a plug-in, allowing to ”connect the rationale with the code and design artifacts that it explains”.

Two additional views are contributed to the Eclipse workspace by SEURAT, which can be seen in Figure 11.



```
// Class     : MeetingObj
// Author    : Janet Burge
//           : jburge@cs.wpi.edu
// Verison   : 1.0

package meeting;

import java.util.*;
import java.io.*;

public class MeetingObj extends Object implements Serializable, Clo

    private MeetingDate mStart; //start time of meeting
    private MeetingDate mEnd;   //end time of meeting
    private String mInfo;       //meeting information
    private String mOwner;      //owner of meeting

  public MeetingObj (MeetingDate start, MeetingDate end, String in

    mStart = (MeetingDate) start.clone();
    mEnd = (MeetingDate) end.clone();
    mInfo = new String(info);
    mOwner = new String(owner);
  }
```

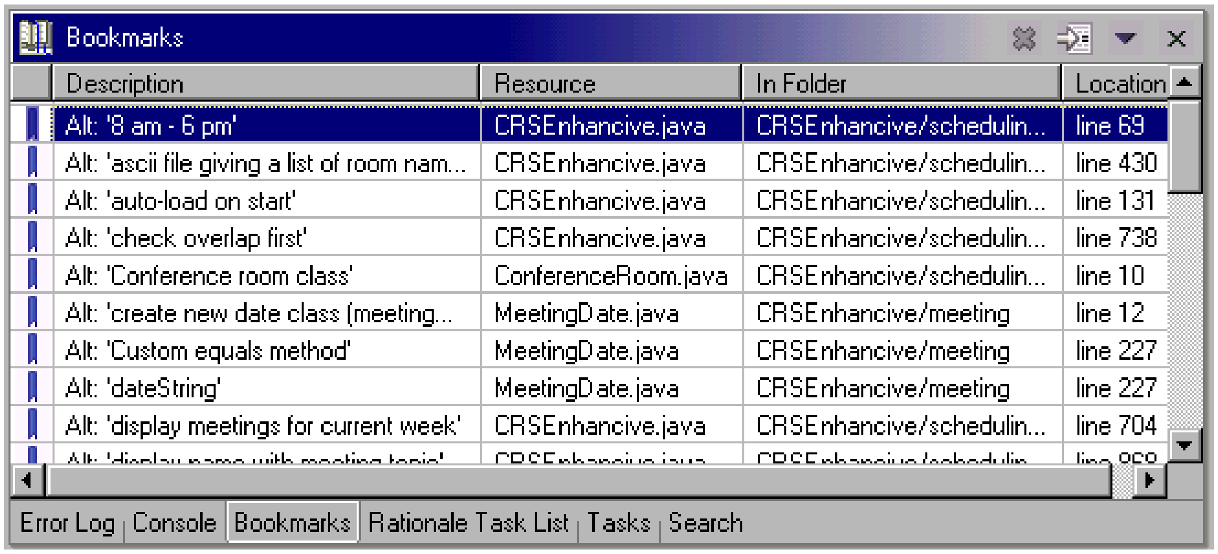
	Description	Type
⊗	No alternative selected for decision 'how should rooms be stored?'	Decision
⊗	No alternative selected for decision 'what to call date compare method'	Decision
⚠	Best evaluation alternative not selected for decision 'How is the user associate	Decision
⚠	Best evaluation alternative not selected for decision 'how to specify conference	Decision

Figure 11: SEURAT and Eclipse. From [11]

The *Rationale Explorer* serves as the main navigational tool for the rationale. The hierarchical tree view allows the user to display and edit the rationale. The *Rationale Task List* shows ”a list of errors and warnings about the rationale”. Indicators are used for the files in the Java Package Explorer to show where rationale is available.

The Rationale Task List can be used to viewed in an editor, which allows the specification of the class, method or instance variable that is associated with the rationale.

Bookmarks, which are a feature of the Eclipse IDE, are used to connect the rationale with the source code [12]. The *Bookmarks List* can be seen in Figure 12.



Description	Resource	In Folder	Location
Alt: '8 am - 6 pm'	CRSEnhancive.java	CRSEnhancive/schedulin...	line 69
Alt: 'ascii file giving a list of room nam...	CRSEnhancive.java	CRSEnhancive/schedulin...	line 430
Alt: 'auto-load on start'	CRSEnhancive.java	CRSEnhancive/schedulin...	line 131
Alt: 'check overlap first'	CRSEnhancive.java	CRSEnhancive/schedulin...	line 738
Alt: 'Conference room class'	ConferenceRoom.java	CRSEnhancive/schedulin...	line 10
Alt: 'create new date class (meeting...	MeetingDate.java	CRSEnhancive/meeting	line 12
Alt: 'Custom equals method'	MeetingDate.java	CRSEnhancive/meeting	line 227
Alt: 'dateString'	MeetingDate.java	CRSEnhancive/meeting	line 227
Alt: 'display meetings for current week'	CRSEnhancive.java	CRSEnhancive/schedulin...	line 704
Alt: 'display name with meeting topic'	CRSEnhancive.java	CRSEnhancive/schedulin	line 968

Figure 12: The Bookmarks List. From [12]

It lists the file, folder and line number of each bookmark. Double-clicking on a bookmark opens the source code in the editor.

2.6.4 Comparison and Evaluation

In this section, we will evaluate and compare each approach with our approach.

2.6.4.1 CM^2

While the rationale in the large is using a well-structured model to represent the decision knowledge, the annotations inside the source code which are representing the rationale in the small are not; instead, a network of comments is used which only link to elements in the rationale in the large, but do not use these elements themselves. This allows the linkage of source code to the rationale in the large, but not a well-structured representation of rationale in the small, i.e. decisions made in the implementation phase.

Instead, our approach will use a well-structured model, namely the DDM for both the documentation of implementation decisions as well as the documentation of decisions made on a higher level of abstraction. In this sense will not discern between the rationale in the large and the rationale in the small.

2.6.4.2 Lougher and Rodden's Approach

A strong point of the system is the ability for users to define custom comment objects to suit their documentation style and to "reflect what they wish to document". While the comment objects can be defined for any type of information, it remains unclear whether a well-defined structure is imposed on these comments and the generated documentation in general.

In principle, UNICASE is very similar to the hypertext network. However, the system presented here was introduced in late 1993 and unlike UNICASE, the system is not integrated in an IDE. We will apply the bi-directional nature of links in the system to the annotations in our approach, as we agree with the authors that ”the simple concept of linking” is ”suprisingly powerful” [13].

2.6.4.3 SEURAT

This approach utilizes Eclipse’s marker system to create simple links between the rationale and the source code, which are both fully integrated in the IDE. In our annotation schema, we will extend this idea by enhancing the functionality of these links.

3 Approach

In this section, we will describe our approach as well as the concept of the resulting annotation schema along with all key decisions.

An annotation schema which is to help programmers and developers (*users*) document their implementation decisions has to be both light-weight and flexible. It has to be light-weight, because the overhead for documenting implementation decisions has to be kept as low as possible and flexible, because the schema has to support project- and problem-specific adjustments to the structure of the decision knowledge.

The DDM (see Section 2.3), which is the basis for our annotation schema, already supports a wide number of knowledge elements and can be easily extended to support additional elements. We use the term *decision element* as a general term for decision statements and decision components which make up the decision knowledge in the DDM. So far, this model has been applied for decisions on a higher level of abstraction, so some types of decision elements are more likely to be used in the context of implementation decisions than others (cf. Section 2.2). However, it is still necessary to include all types of decision elements to guarantee flexibility.

With this information in mind, we developed the annotation schema. Figure 13 gives an overview over the relations between annotations in our schema and the decision elements in the DDM.

We take annotations as they are defined in the Java platform as the basis for our definition of an annotation in our schema. An at-sign (@) indicates the beginning of an annotation, followed the annotation type, which reflects the types of decision elements the annotation refers to. Each annotation type contains a certain set of attributes. In order to avoid obscuring source code, some attributes may be hidden and may only made visible to users through a detail view, which also allows the hidden attributes to be edited.

We chose to use this particular notation for our annotations, as it conforms with the notation of both the Java programming language and the *Javadoc* tool[4].

Initially, we intended to use annotations to provide a basic one-to-one mapping to decision elements in the DDM. However, we soon realized that the potential of these annotations is limited. Thus, there exist two classes of annotations in our schema: *core annotations* and *augmented annotations*.

Core annotations provide a basic one-to-one mapping to the decision elements in the DDM, i.e. they represent the highest level of granularity. Each type of decision element has a respective core annotation type associated with it. The type-specific attributes of these core annotations correspond with the type-specific attributes of the respective decision elements, i.e. each core annotation is linked to exactly one decision element.

Augmented annotations provide extended functionality compared to core annotations; they may contain any number of attributes and execute any number of augmented annotation commands, which can create, refer to or modify decision elements. Our annotation schema provides a core set of predefined augmented annotation types, but the user may freely define augmented annotations as well. Augmented annotations represent the medium to low levels

[4]Widely used tool for generating API documentation in HTML format - http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html

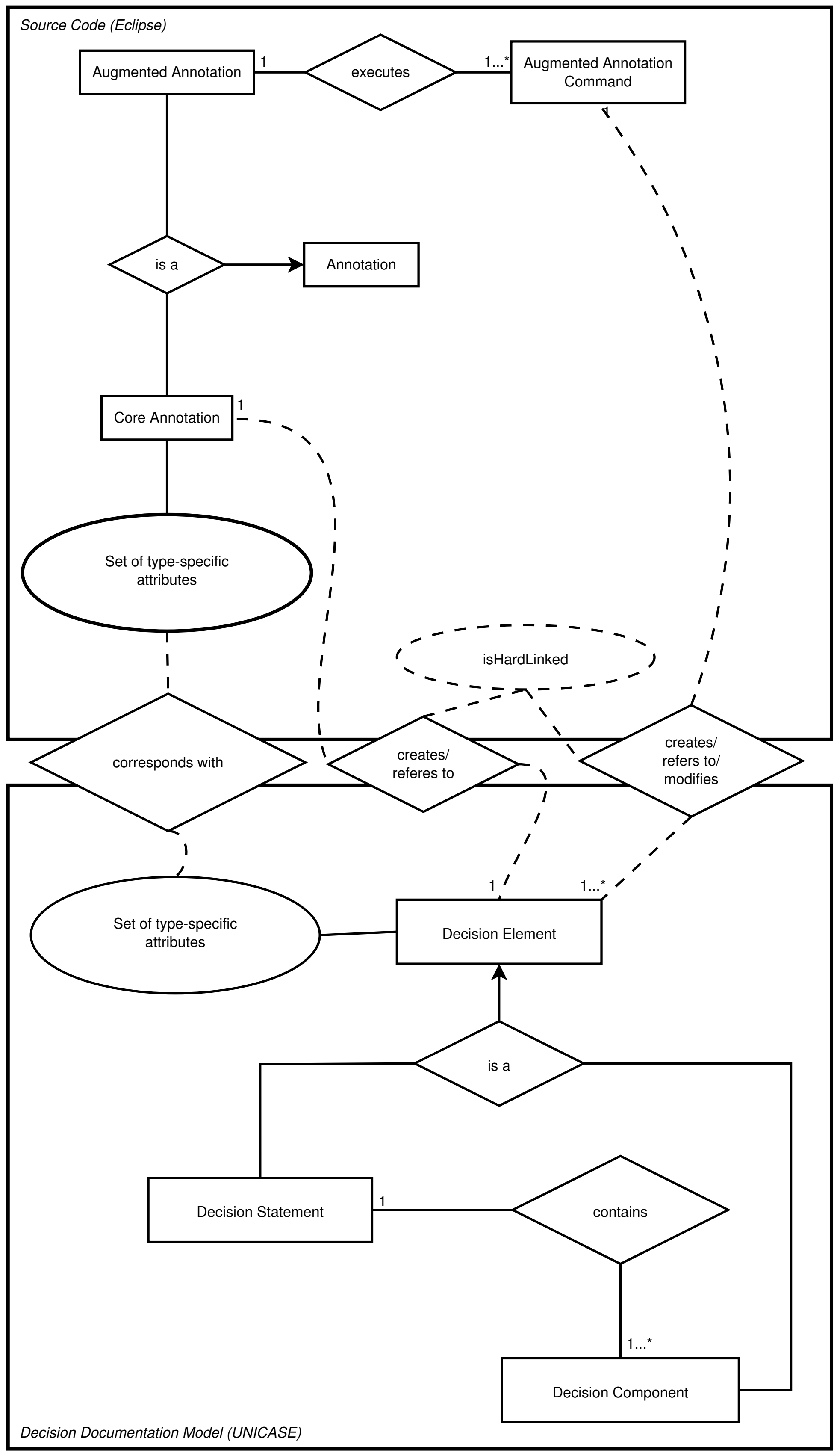


Figure 13: Relations between annotations and decision elements (Chen's notation)

of granularity. We introduced augmented annotations, because we wanted to exploit the potential of annotations in our schema beyond the basic one-to-one mapping that is provided by core annotations.

Both core annotations and augmented annotation commands may be *hard-linked* to their decision elements. In the context of core annotations, a hard link means that changes in the annotation will be applied to the referred decision element and vice versa. Also, if the core annotation is deleted, the referred decision element is deleted as well. In the context of augmented annotation commands, a hard link means that the command creates the referred decision element. If the augmented annotation that contains the command is deleted, the referred decision element is deleted as well.

Annotations in our schema can be used to

1. document implementation decisions, i.e. decisions concerned with the implementation phase. This is the primary use case for our schema. Here, annotations primarily create decision elements.

   Examples for this, which are based on our example implementation decision from Section 2.3, can be seen in Section 3.2.

2. comment on or refer to existing decisions elements. These decision elements may belong to all phases of the project's life cycle, including the implementation phase. In this case, annotations primarily refer to decision elements.

   Using our aforementioned example implementation decision, implementers could link the implementation decision to the underlying design decision by referring to the respective decision element.

There are a number of important aspects to consider for our annotation schema. Due to the nature of decision-making for implementation decisions, there are different levels of granularity at which an implementation decision may be documented. At the same time, the source code that is of significance for a given decision may be scattered across any number of source code files. Thus, the granularity and locality of annotations plays an important role. Finally, we need to consider how to map user actions to annotations and decision elements respectively.

We will discuss granularity and locality in Sections 3.2 and 3.3, and the mapping of user actions to annotations and decision elements in Sections 3.4 and 3.5 respectively.

3.1 Annotation Classes

In this section, we will present the two classes of annotations in detail.

3.1.1 Core Annotations

Each decision element available in the DDM is mapped to exactly one core annotation type in order to guarantee a high degree of flexibility. All core annotation types share a common set of attributes, of which all but one of these common attributes are hidden, as most of them are only used for internal purposes, such as referencing and persistent storage. The only common attribute which is not hidden is the core annotation's description, which refers to the description of the referred decision element.

In the following sections, the available types of core annotations will be described in detail.

3.1.1.1 @Decision

This annotation type represents a decision statement, which is the top-level decision element. In order to document an implementation decision, it is necessary to create this annotation first. It contains the following attributes:

Progress The decision's progress status. Possible values are *rejected, pending, preliminary accepted* and *accepted*.

Implementation The decision's implementation status. Possible values are *not applied, envisioned, applied* and *obsolete*.

3.1.1.2 @Alternative

This annotation type represents a solution alternative for a decision statement. To indicate that the alternative solution has been accepted as a solution for the decision at hand, it is resolved by a resolving argument. It contains no specific attributes.

3.1.1.3 @Argument

This annotation type represents an argument either supporting or attacking a decision element. It contains the following attributes:

Supports A list of decision elements the argument is supporting.

Attacks A list of decision elements the argument is attacking.

3.1.1.4 @Constraint

This annotation type represents a constraint on the given decision. It contains the following attributes:

Is Criterion Flag to indicate indicate whether the constraint is to be treated as a criterion.

Measure The measure of the criterion. Possible values are *qualitative* and *quantitative*.

Type The type of the constraint. Possible values are *as is, to be* and *not to be*.

Scenario The scenario of the constraint.

Deadline The deadline of the constraint.

3.1.1.5 @Assumption

This annotation type represents contextual information that states expectations towards the decision's environment. It contains the same attributes as the *@Constraint* annotation.

3.1.1.6 @Goal

This annotation type represents a positive aim that shall be reached when deciding how to solve the decision's problem. It contains no specific attributes.

3.1.1.7 @Issue

This annotation type represents a given decision problem. It contains the following attribute:

Error Details The error details of the issue.

3.1.1.8 @Context

This annotation type represents a description of the decision's environment. It contains the same attributes as the *@Constraint* annotation.

3.1.1.9 @Implication

This annotation type represents contextual information describing expected outcomes or effects of the decision. It contains the same attributes as the *@Constraint* annotation as well as:

Effect The effect of the implication. Possible values are *positive, indifferent, negative* and *unknown*.

3.1.1.10 @Assessment

This annotation type represents a solution with respect to a given criterion. It contains the following attributes:

Quantity The quantity of the assessment.

Quality The quality rating of the assessment. Five step rating ranging from *very good* to *very poor*. Additional value *inapplicable*.

Supports A list of decision elements the assessment is supporting.

Attacks A list of decision elements the assessment is attacking.

3.1.1.11 @Claim

This annotation type represents a solution aspect that already worked out formerly. It contains the following attribute:

Type The type of claim. Possible values are *as is, to be* and *not to be*.

3.1.2 Augmented Annotations

Augmented annotations extend the functionality of core annotations. The behavior of the augmented annotation is defined by the commands it contains. The commands are executed automatically as soon as the augmented annotation is created. The behavior of a type of augmented annotation can be defined by users, either programmatically or by means of an editor.

Our annotation schema provides a set of predefined augmented annotations, but users are able to create their own or edit the predefined augmented annotation types. Our annotation schema will provide an editor allowing users to create augmented annotations that perform a number of simple commands.

In the following sections, the available types of predefined augmented annotations will be described in detail.

3.1.2.1 @Pro

This augmented annotation creates a supporting argument. It contains only one attribute, which is the supporting argument's description. It searches upwards for the nearest instance of a core annotation inside the source code. If one such annotation has been found, it creates a supporting argument as a direct child of the decision element the core annotation refers to. If no such annotation could be found within the source code, it does nothing.

3.1.2.2 @Contra

This augmented annotation creates an attacking argument. It contains only one attribute, which is the attacking argument's description. It searches upwards for the next instance of a core annotation inside the source code. If one such annotation has been found, it creates an attacking argument as a direct child of the decision element the core annotation refers to. If no such annotation could be found within the source code, it does nothing.

3.1.2.3 @ResolvedBy

This augmented annotation indicates that an alternative has been accepted as a solution for a decision. It contains only one attribute, which is a reference to the resolving argument; this attribute is hidden and may be viewed and edited in a detail view. It searches upwards for the next instance of an *@Alternative* annotation inside the source code. If one such annotation has been found, it refers the resolving argument to the decision element the *@Alternative* annotation refers to. If no such annotation could be found within the source code, it does nothing.

3.2 Granularity

Users may want to document their implementation decisions with different levels of granularity, depending on the severity and significance of the decision. Thus, our annotation schema supports different levels of granularity.

We illustrate this using our example implementation decision from the DDM, where we will use our annotation schema to document the decision.

```
@Decision("A wizard for exporting an entire project is implemented. We
   chose to just adjust the existing document export wizard, because it
   was the simplest solution.")
```

```
      [source code]
```

Listing 1: The lowest level of granularity

Listing 1 shows the lowest level of granularity for our schema. This basically equates to an extended comment, where only the major parts of an implementation decision are captured. The overhead for programmers is minimal.

```
      @Decision("A wizard for exporting an entire project is implemented.")
      @Alternative("Implement a new wizard from scratch.")
      @Pro("The wizard can be kept very flexible.")
      @Contra("This produces a lot of duplicate code.")
      @Contra("High cost.")
      @Alternative("Adjust the existing wizard to support a project export.")
      @ResolvedBy
      [source code]
```

Listing 2: The medium level of granularity

Listing 2 shows the medium level of granularity. Here, the two alternatives are described, along with attacking and supporting arguments for the first alternative using the *@Contra* and *@Pro* augmented annotations. The second alternative has been accepted, which is indicated by the *@ResolvedBy* augmented annotation. The overhead for this level of granularity is somewhere in the middle ground between the lowest and highest level of granularity.

```
      @Decision("A wizard for exporting an entire project is implemented.")
      @Constraint("As less code as possible should be added to the project.")
      @Constraint("Code duplication should be avoided.")
      @Alternative("Implement a new wizard from scratch.")
      @Argument("A lot of additional code has to be written.")
      @Argument("The wizard can be kept very flexible.")
      @Argument("Currently, a project export is only a lighter version of the
         standard document export.")
      @Alternative("Adjust the existing wizard to support a project export.")
      [source code]
```

Listing 3: The highest level of granularity

Listing 3 shows the highest level of granularity. Here, every decision element in our example has been mapped to a core annotation. Note that only the descriptions are visible here, all other attributes are hidden. They can be accessed and edited via a detail view. This level of granularity has the highest overhead for programmers but also provides the most detailed insight into the decision.

The borders between the different levels of granularity are blurred. For example, users might document the constraints for their implementation decisions, but use augmented annotations or a mixture of *@Argument*, *@Pro* and *@Contra* annotations to formulate supporting and attacking arguments for the given alternatives.

3.3 Locality

In the listings above, we wrote our annotations in one consistent block. In some cases, it might be beneficial to clarity to write annotations right above the lines of source code that are directly concerned with them. Our annotation schema allows annotations to be distributed in this fashion. We shall illustrate this using Listing 3 as our example.

```
@Decision("A wizard for exporting an entire project is implemented.")
@Constraint("As less code as possible should be added to the project.")
@Constraint("Code duplication should be avoided.")
[source code]
        @Alternative("Implement a new wizard from scratch.")
        @Argument("The wizard can be kept very flexible.")
        /*[source code] //Source code from the first
         *[source code] //alternative, which has been
         *[source code] //rejected and commented out
         * .........
         *[source code]
         */
        @Argument("A lot of additional code has to be written.")
        @Argument("Currently, a project export is only a lighter version
           of the standard document export.")
        @Alternative("Adjust the existing wizard to support a project
           export.")
        [source code] //Source code from the accepted alternative
```

Listing 4: Distributed annotations, highest level of granularity

Listing 4 shows a distributed version of the annotations in Listing 3. The source code from the originally proposed alternative has been commented out. Directly below, the attacking arguments to the original alternative are mentioned along with the alternative.

3.4 Mapping Actions on Annotations to Decision Elements

Based on the actions users are able to perform on the annotation schema, Table 2 gives a general overview over the different cases to consider when mapping annotations in source code to decision elements. In the following sections, we will describe the individual cases in detail; if not otherwise specified, the descriptions apply to all classes of annotations.

User action	Cases to consider
Creating a new annotation	Link to existing decision element Create new decision element
Modifying an annotation	Change content Change referred decision element(s) Change annotation
Deleting an annotation	

Table 2: Overview over the general cases to consider when mapping annotations to decision elements

3.4.1 Creating a New Annotation

Creating a new annotation is done by writing an at-sign followed by the type of annotation and, where applicable, the content of the relevant visible attributes in parentheses into the source code.

The annotation is yet to be linked to a decision element and users have two options: either link the annotation to an existing decision element or create a new decision element.

Augmented annotations: The linkage to existing decision elements or creation of decision elements is done automatically for all predefined augmented annotations.

3.4.1.1 Link to an Existing Decision Element

Users select the decision element that is to be linked to the newly created annotation. The content of attributes contained in the annotation, if available, is appended locally to the decision element's attributes and the decision element is linked to the annotation. The decision element itself or any of its attributes cannot be changed.

3.4.1.2 Create a New Decision Element

Users either select the location of the decision element that is to be created or create the decision element in a default location. Once the decision element has been created, it is linked to the annotation and the decision element's attributes are filled with the content of the annotation's attributes. In this case, there exists a hard link between the annotation and the decision element, i.e. if the decision element is deleted or modified, the annotation is deleted or modified as well, and vice versa.

3.4.2 Modifying an Annotation

Once an annotation has been created, it may of course be modified. We have to differentiate between three general types of modifications.

3.4.2.1 Change Content

Changing the content of an annotation's attributes is done by making the respective changes in the annotation. If the annotation refers to existing decision elements, the changes will have no effects on the referred decision elements; only the appended content is changed. If the annotation has created a decision element, i.e. a hard link exists between the annotation and one of the referred decision elements, the content of the decision element will change with the annotation.

3.4.2.2 Change Referred Decision Element(s)

Changing one of the referred decision elements is done by selecting the new decision element. Naturally, the selected decision element has to be of the same type as the old one. Again, the content of the annotation's attributes is appended locally to the decision element's attributes, if the selected decision element already exists. If a new decision element is created instead, a hard link between the annotation and the newly created decision element is established. If a hard link existed between the annotation and one of the referred decision elements and the decision element is changed, the old decision element is deleted.

3.4.2.3 Change Annotation

Changing the annotation may or may not be done without deleting or changing its attributes and their contents. This depends on whether the old and new annotation type have a intersection with their specific set of attributes. Once the annotation has been changed, any linkage to decision elements have to be respecified. If hard links existed between the annotation and the referred decision elements and the annotation type is changed, the old decision elements are deleted and new decision elements corresponding to the annotation are created and hard linked.

Augmented annotations: It is possible to edit an augmented annotation types in an editor. When instances of a certain augmented annotation type exist in the source code and the augmented annotation type is edited, users will have two options: they can either keep the existing instances or delete them.

3.4.3 Deleting an Annotation

Deleting an annotation is done by simply removing the annotation and its contents from the source code. We have to differentiate between two cases.

If the annotation has created a decision element, i.e. a hard link exist between the annotation and one of the referred decision elements, both the annotation and the hard-linked decision element are deleted. If the decision element has any children, they are deleted as well.

If the annotation refers to an existing decision element, only the annotation is deleted. The decision element remains unchanged.

3.5 Mapping Actions on Decision Elements to Annotations

Based on the actions users are able to perform on the DDM, we need to consider only the following general cases when mapping decision elements to annotations.

1. **Creating** a new decision element
2. **Modifying** a decision element
3. **Deleting** a decision element

The individual cases will be described in detail in the following sections.

3.5.1 Creating a New Decision Element

Creating a new decision element has no effects on any annotations inside the source code. An interesting feature to have would be the possibility for users to specify locations within the source code, which correspond to the newly created decision element, creating core annotations of the corresponding type at the specified locations inside the source code. These core annotations will then be hard-linked to the newly created decision element.

While this would not really be of benefit for programmers, as they should rarely have to leave the source code domain, we believe that personnel involved in decision making at higher levels would benefit from this feature, as they could link their decisions to source code without leaving their domain.

3.5.2 Modifying a Decision Element

Modifying a decision element has no effect on any annotations that refer to it, unless the decision element has been created by an annotation.

If the decision element has been created by an annotation, i.e. a hard link exists between the annotation and the decision element, and the decision element is modified, the modification is also transferred to the annotation. This means that all attributes of the decision element that have been modified, are also modified in the same manner on the annotation.

3.5.3 Deleting a Decision Element

Deleting a decision element will render all annotations that refer to it invalid. Users may decide whether all affected annotations are deleted or their referred decision elements are manually respecified.

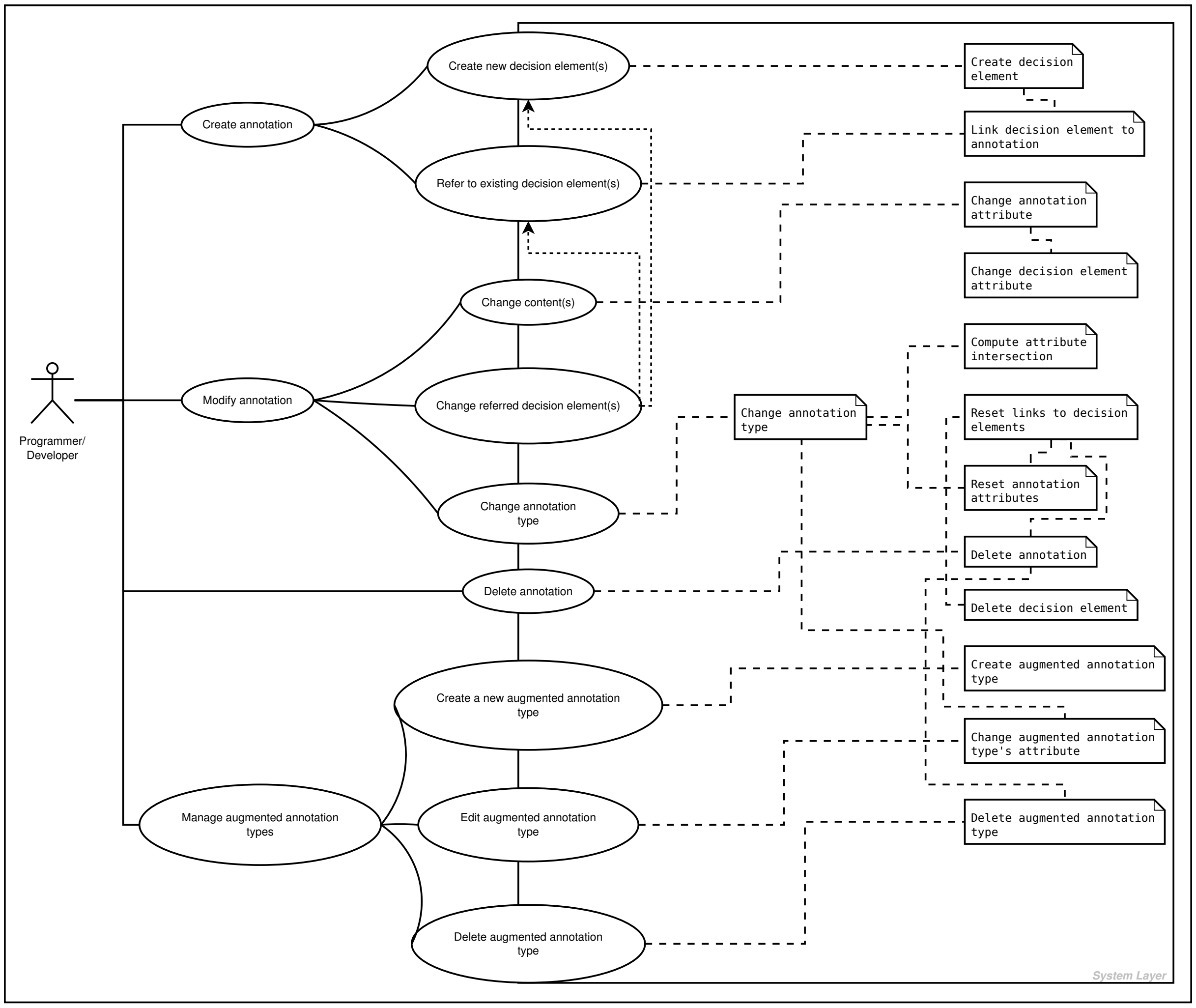
If the decision element has been created by an annotation, i.e. a hard link exists between the annotation and the decision element, and the decision element is deleted, the annotation is deleted as well. Any additional annotations that refer to the deleted decision element are rendered invalid.

4 Requirements Analysis

In the following sections, we will analyze the requirements of the implementation of our annotation schema.

4.1 Use Cases

Figure 14 illustrates the use cases and system functions we were able to identify based on our considerations from Sections 3.4 and 3.5. The use cases concerning the creation,



Programmer/ Developer
Create annotation
Modify annotation
Manage augmented annotation types
Create new decision element(s)
Refer to existing decision element(s)
Change content(s)
Change referred decision element(s)
Change annotation type
Delete annotation
Create a new augmented annotation type
Edit augmented annotation type
Delete augmented annotation type
Create decision element
Link decision element to annotation
Change annotation attribute
Change decision element attribute
Compute attribute intersection
Change annotation type
Reset links to decision elements
Reset annotation attributes
Delete annotation
Delete decision element
Create augmented annotation type
Change augmented annotation type's attribute
Delete augmented annotation type
System Layer

Figure 14: The use case diagram for our annotation schema

modification and deletion of annotations have already been described in Section 3.4. In the following sections, we will describe the remaining use cases along with all system functions associated with each use case in detail.

4.1.1 Manage Augmented Annotation Types

The management of augmented annotation types is done in an editor, allowing the creation, modification and deletion of augmented annotation types.

4.1.1.1 Create a New Augmented Annotation Type

Users create a new augmented annotation type and define its behavior. A new augmented annotation type with the specified name, attributes and behavior is created.

4.1.1.2 Edit Augmented Annotation Type

Users edit a augmented annotation type. After they are satisfied with their modifications, the modifications are applied to the augmented annotation type. All existing instances of this augmented annotation types within the source code are modified accordingly. This may trigger the deletion of decision elements.

4.1.1.3 Delete Augmented Annotation Type

Users delete a augmented annotation type. The augmented annotation type along with all of its instances within the source code are deleted. This may trigger the deletion of decision elements.

4.1.2 Associated System Functions

In the following sections, we will describe the system functions associated with each use case in detail.

4.1.2.1 Create Decision Element

This function creates a decision element. Its type is based on the type that is provided as an argument. The description of the decision element is provided by the calling annotation, the name of the decision element and its location within the underlying UNICASE project is provided by the user. Annotations calling this system function may provide a set of attribute-value pairs as an additional argument to this function, based on which the decision element's attributes are set.

The decision element is then hard-linked to the calling annotation by executing the "link decision element to annotation" function.

Arguments:

1. Type of the decision element
2. Name of the decision element
3. Calling annotation
4. *Optional:* set of attribute-value pairs

Result: a decision element is created, which is hard-linked to the annotation that called this function.

4.1.2.2 Link Decision Element to Annotation

This function creates a hard-link between a decision element and an annotation, both are given as arguments to this function.

Arguments:

1. Decision element
2. Calling annotation

Result: a hard link is established between the decision element and the annotation.

4.1.2.3 Change Annotation Attribute

This function modifies an attribute of an annotation. If the attribute concerns a hard-linked decision element, the modification has to be performed on the hard-linked decision element's attribute as well, which is done by executing the ”change decision element attribute” function.

Arguments:

1. Calling annotation
2. Attribute to change
3. New attribute value

Result: the annotation's and, where applicable, decision element's attribute has been modified accordingly.

4.1.2.4 Change Decision Element Attribute

This function modifies the attribute of a decision element.

Arguments:

1. Decision Element
2. Attribute to change
3. New attribute value

Result: the decision element's attribute has been modified accordingly.

4.1.2.5 Change Annotation Type

This function changes the type of an annotation. First, it computes the intersection between the old and new annotation types' attribute sets using the ”compute attribute intersection” function. All attributes inside the intersection are transferred, all other attributes are reset using the ”reset annotation attributes” function.

Arguments:

1. Old annotation type
2. New annotation type

Result: the annotation type has been changed accordingly.

4.1.2.6 Compute Attribute Intersection

This function computes the intersection between the attribute sets of two annotation types.
Arguments:

1. First annotation type
2. Second annotation type

Result: the intersection of attribute sets of the two annotation types.

4.1.2.7 Reset Annotation Attributes

This function resets the attributes of an annotation. If a set of attributes is given as an argument to the function, only the given attributes are reset, otherwise all of the annotation's attributes are reset. Any links to decision elements are reset as well using the ”reset links to decision elements” function.
Arguments:

1. Annotation whose attributes are to be reset
2. *Optional:* set of attributes that are to be reset

Result: the annotation's attributes are reset.

4.1.2.8 Reset Links to Decision Elements

This function resets the links between an annotation and a decision element. If the link is a hard link, the decision element is deleted using the ”delete decision element” function.
Arguments:

1. Annotation to delete the links for
2. Decision element to unlink

Result: the link between the annotation and the decision element is unset.

4.1.2.9 Delete Annotation

This function deletes an annotation. Any links to decision elements are reset using the ”reset links to decision elements” function.
Arguments:

1. Annotation to delete

Result: the annotation is deleted.

4.1.2.10 Delete Decision Element

This function deletes a decision element. Any decision element which are direct or non-direct children of the decision element are deleted as well.
Arguments:

1. Decision element to delete

Result: the decision element is deleted.

4.1.2.11 Create Augmented Annotation Type

This function creates a new augmented annotation type. The type's name and attributes are provided by the user as well as its behavior.
Arguments:

1. Name of the augmented annotation type
2. Behavioral description

Result: the augmented annotation type is created.

4.1.2.12 Change Augmented Annotation Type's Attribute

This function changes an attribute of a augmented annotation type if the attribute which is to be changed has been provided; if it is left out, the attribute is added to the augmented annotation type. Calls the ”change annotation type” function to compute the attribute intersection and reset the old attributes.
Arguments:

1. Name of the augmented annotation type
2. Affected states
3. *Optional:* Attribute to change
4. New attribute

Result: the augmented annotation type's attribute is changed.

4.1.2.13 Delete Augmented Annotation Type

This function deletes a augmented annotation type. Any instances of this type within the source code are deleted using the ”delete annotation” function.
Arguments:

1. Augmented annotation type to delete

Result: the augmented annotation type is deleted along with all its instances within the source code.

4.2 UI Workflow and Mock-ups

In the following sections, we will describe the UI workflow with mock-ups for each annotation class in detail.

4.2.1 Core Annotations

Once a core annotation has been created in the source code, users will have to decide whether they want to create a new decision element associated with this core annotation or link the core annotation to an existing decision element.

We found that using a combination of hovers and dialogs would provide the best possible workflow. Initially, we considered a separate view which would allow users to edit annotations. This option was discarded, because we found that hovers would feel more natural to users.

4.2.1.1 Creating a New Decision Element

Figure 15 shows the UI mock-up for the workflow for creating a new decision element. The decision element's properties can be edited in the detail view.

4.2.1.2 Referring to an Existing Decision Element

Figure 16 shows the UI mock-up for the workflow for referring to an existing decision element. The decision element's properties cannot be edited in the detail view; instead, users will have to open the decision element in UNICASE and make their modifications there.

4.2.2 Augmented Annotations

Figure 17 shows the UI mock-up for the workflow for managing augmented annotation types. Users may create, edit and delete augmented annotation types. When creating or editing an augmented annotation type, users are able to edit the augmented annotation type's behavior.

Initially, we decided not to show a hover for augmented annotations. We soon realized that this would leave users in the dark on whether the execution of the augmented annotation's commands was successful or failed. We thus decided to show a small hover for augmented annotations providing generic information, which can be seen in Figure 18.

4.2.3 Code Associations

In order to visualize the linkage between a given decision elements and annotations, we will introduce a *code association view*. A mock-up for this view can be seen in Figure 19.

This view creates a bidirectional linkage between decision elements and annotations.

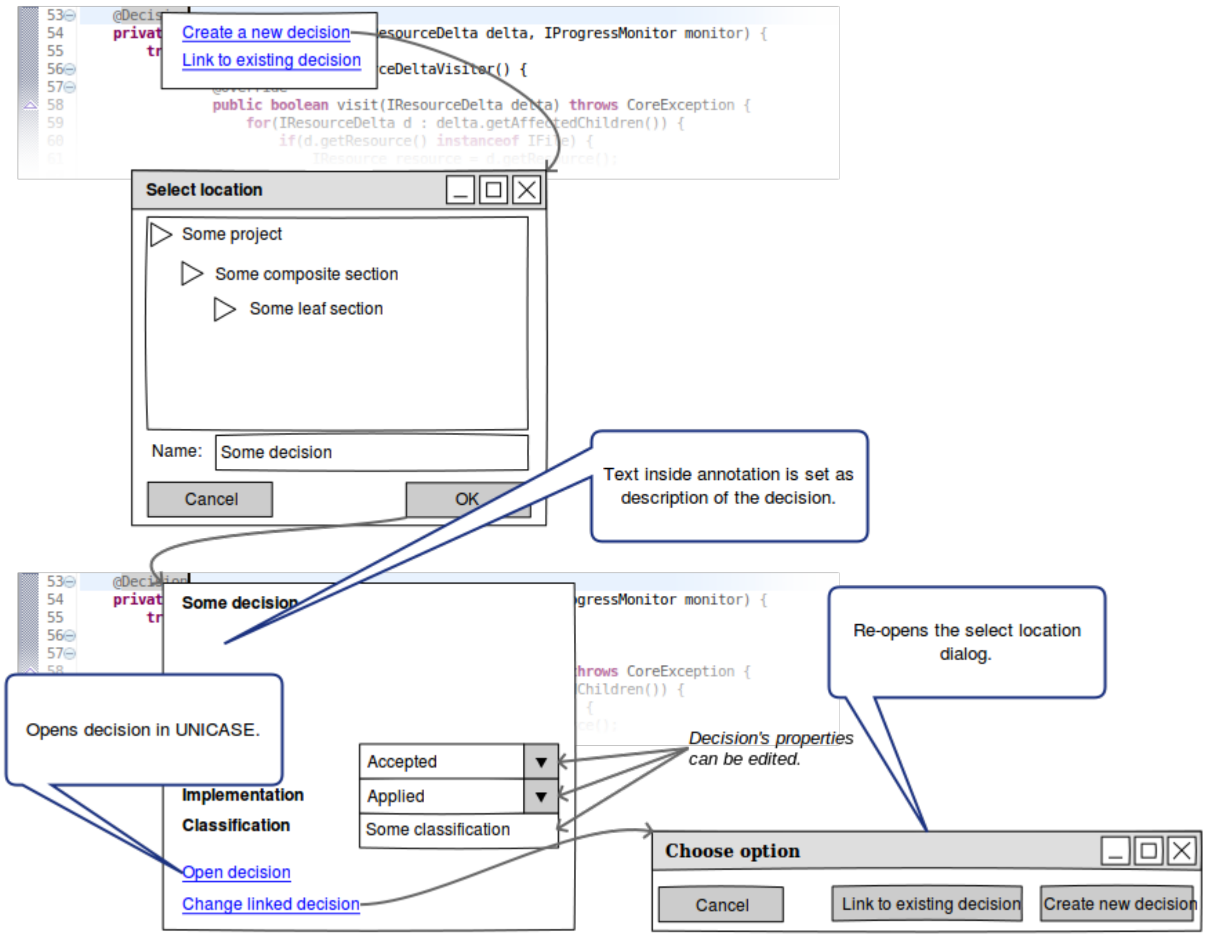
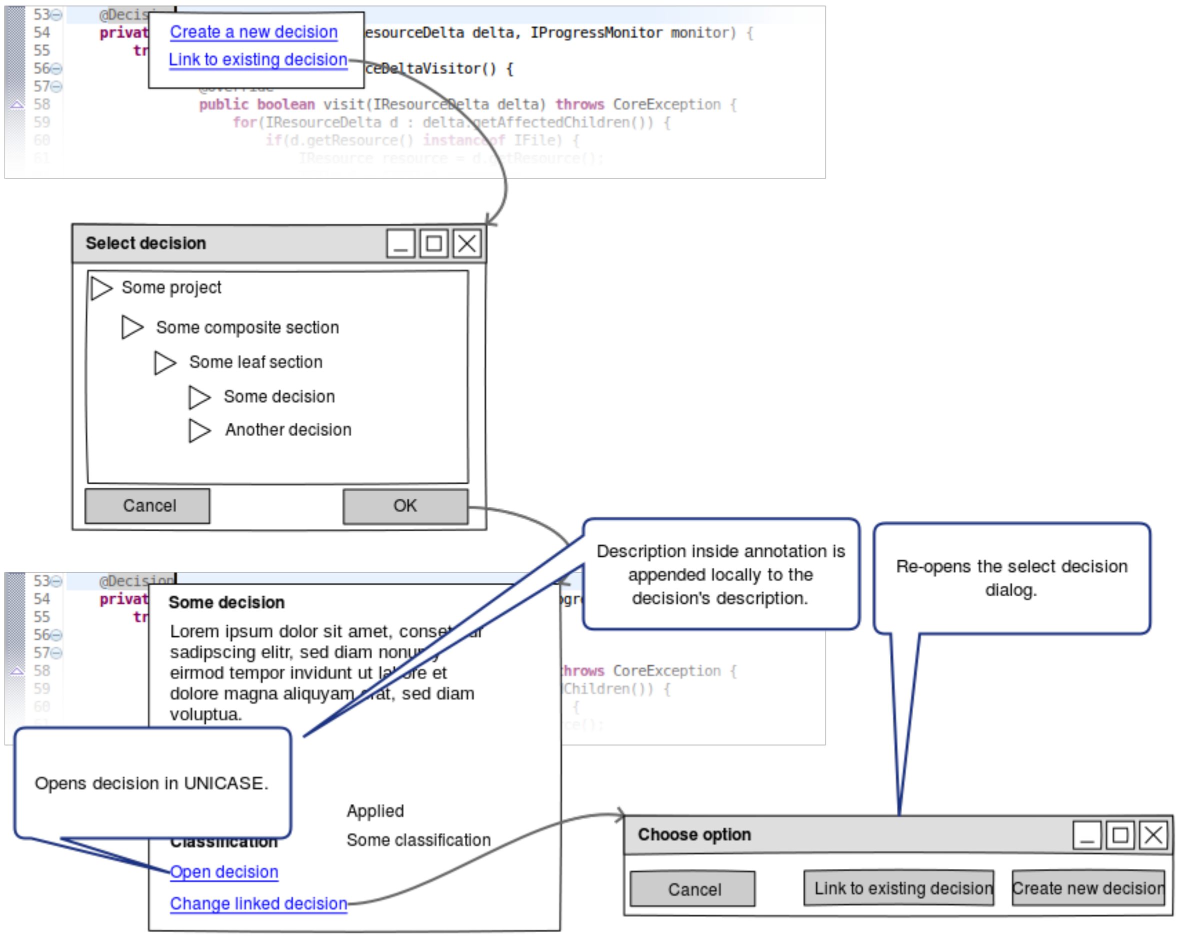
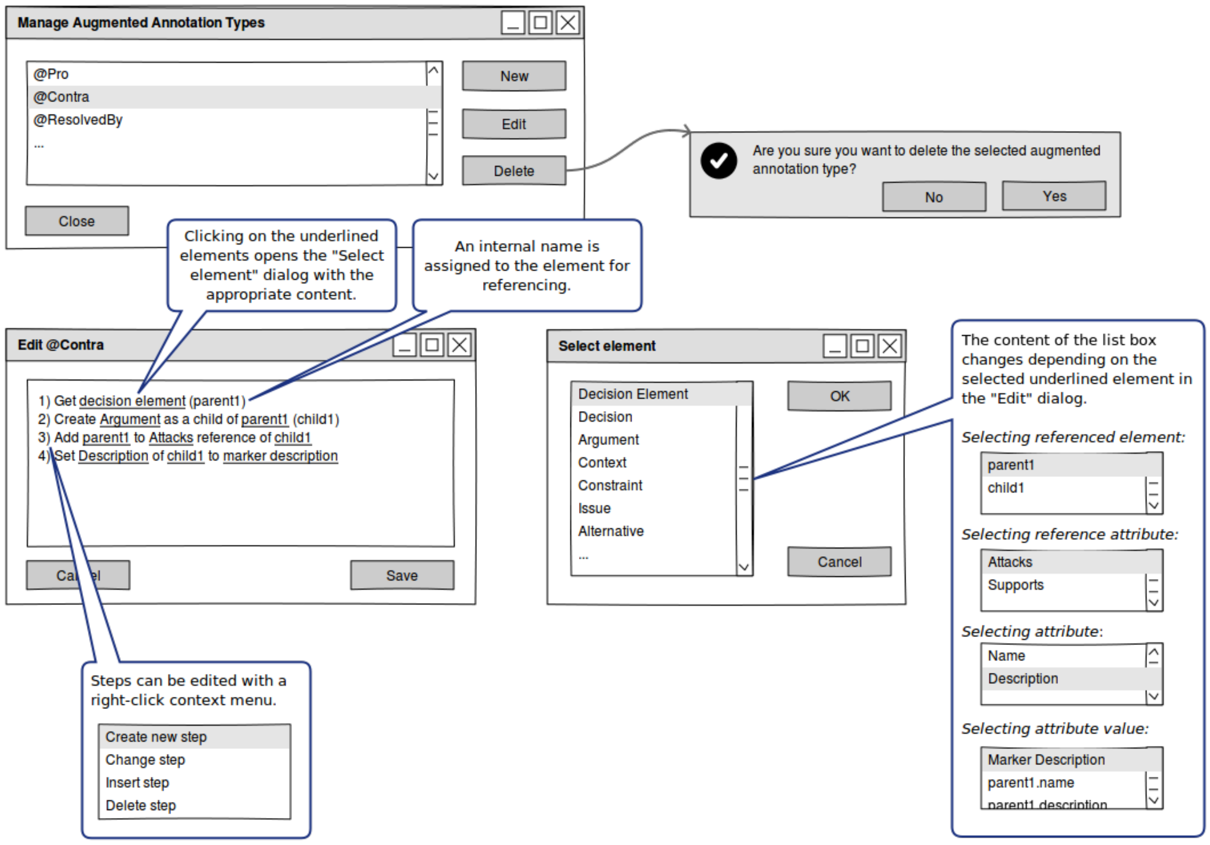


Figure 15: UI workflow for creating a new decision element



Create a new decision
Link to existing decision
Select decision
Some project
Some composite section
Some leaf section
Some decision
Another decision
Cancel
OK
Description inside annotation is appended locally to the decision's description.
Re-opens the select decision dialog.
Some decision
Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.
Opens decision in UNICASE.
Applied
Classification
Some classification
Open decision
Change linked decision
Choose option
Cancel
Link to existing decision
Create new decision

Figure 16: UI workflow for referring to an existing decision element



1) Get decision element (parent1)
2) Create Argument as a child of parent1 (child1)
3) Add parent1 to Attacks reference of child1
4) Set Description of child1 to marker description

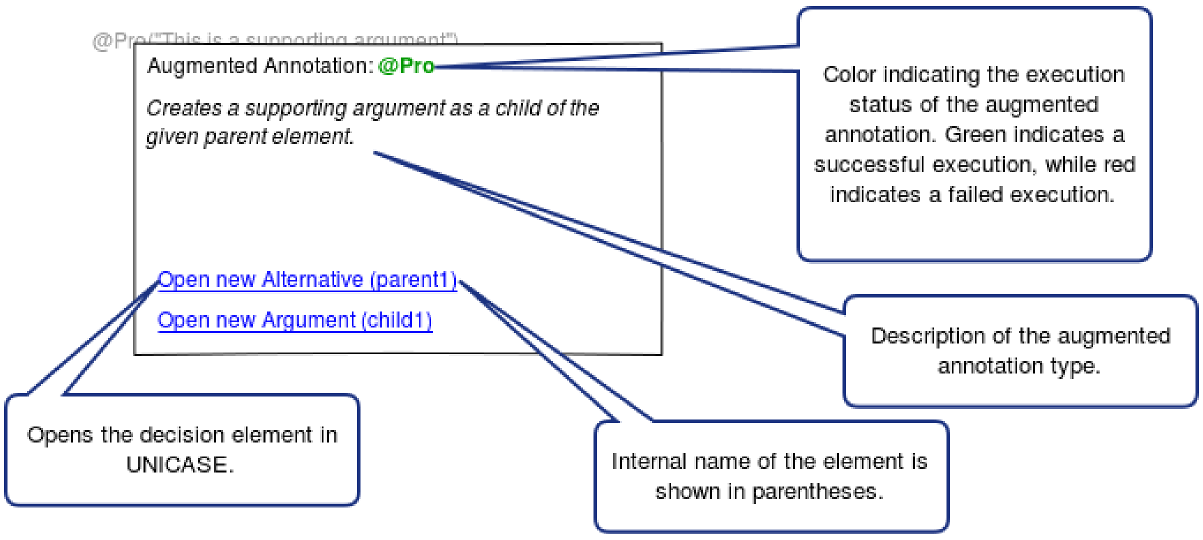
Clicking on the underlined elements opens the "Select element" dialog with the appropriate content.

An internal name is assigned to the element for referencing.

The content of the list box changes depending on the selected underlined element in the "Edit" dialog.

Steps can be edited with a right-click context menu.

Figure 17: UI workflow for managing augmented annotation types


@Pro("This is a supporting argument")
Augmented Annotation: @Pro
Open new Alternative (parent1)
Open new Argument (child1)

Creates a supporting argument as a child of the given parent element.

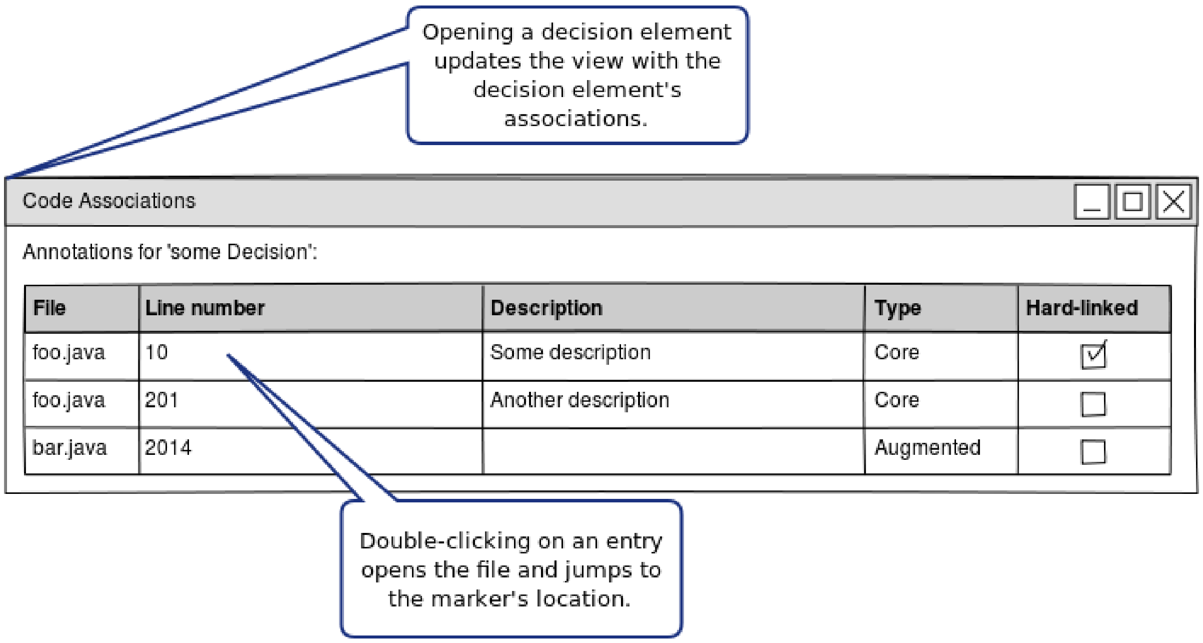
Color indicating the execution status of the augmented annotation. Green indicates a successful execution, while red indicates a failed execution.

Description of the augmented annotation type.

Opens the decision element in UNICASE.

Internal name of the element is shown in parentheses.

Figure 18: UI mock-up for the augmented annotations hover


Opening a decision element updates the view with the decision element's associations.
Code Associations
Double-clicking on an entry opens the file and jumps to the marker's location.

Annotations for 'some Decision':

File	Line number	Description	Type	Hard-linked
foo.java	10	Some description	Core	✓
foo.java	201	Another description	Core	☐
bar.java	2014		Augmented	☐

Figure 19: UI mock-up for the code associations view

4.3 Non-Functional Requirements

We identified a number of nun-functional requirements, which we believe our implementation has to fulfill in order to maximize the implementation's usability as well as the benefits to users of the implementation. In the following sections, we will describe these non-functional requirements in detail.

4.3.1 Performance

In order to minimize the interruption of the general implementation workflow, users should be able to create annotations and link them with decision elements in real time, independent of the project's size, i.e. the number and size of source code files and the number of annotations. UI elements should be responsive under all circumstances. Saving and restoring the state of annotations between user sessions should not take more than a few seconds on an average computer, even for large projects with a high number of annotations.

4.3.2 Intuitive UI and Workflow

UI elements should be easy to read and understand. It should be clear right away what a UI element does and how users are supposed to use it. The same rules apply to the workflow of our implementation. The workflow should be tailored to support the quick creation and modification of annotations in source code with as less overhead for users as possible. The interruption of the general implementation workflow should be as minimal as possible.

4.3.3 Robustness

The implementation of our annotation schema should be as bug-free as possible. Erroneous user input should be either handled or ignored, but should never break the implementation or even cause data loss.

4.3.4 Extensibility

In order to make the implementation flexible, users should be able to add new core and augmented annotations either at runtime or programmatically using extension points provided by our implementation. For this, our implementation needs to provide extension points.

4.3.5 UNICASE Look and Feel

Since the DDM is implemented in UNICASE and our annotation schema is designed to work in conjunction with the DDM, our implementation should conform to the visual appearance and layout of UNICASE.

5 Design and Implementation

The implementation of our annotation schema was developed as a plug-in for Eclipse 3.7 ”Indigo” with UNICASE 0.52. We will refer to this plug-in as the *Annotations* plug-in.

We will first describe the results and issues for the design and implementation of the plug-in’s model and program logic and then proceed with the design and implementation of the user interface and worfklow. We will then describe the interactional dynamics of the components in our plug-in and finally describe the quality assurance process.

Note that due to size constraints and in order to improve comprehensibility, we will primarily show generalized versions of diagrams illustrating only the core classes, attributes and methods and their interactions. See Appendix A for a package diagram and Appendix B for complete versions of each class diagram.

Figure 20 illustrates the different components of the Annotations plug-in.

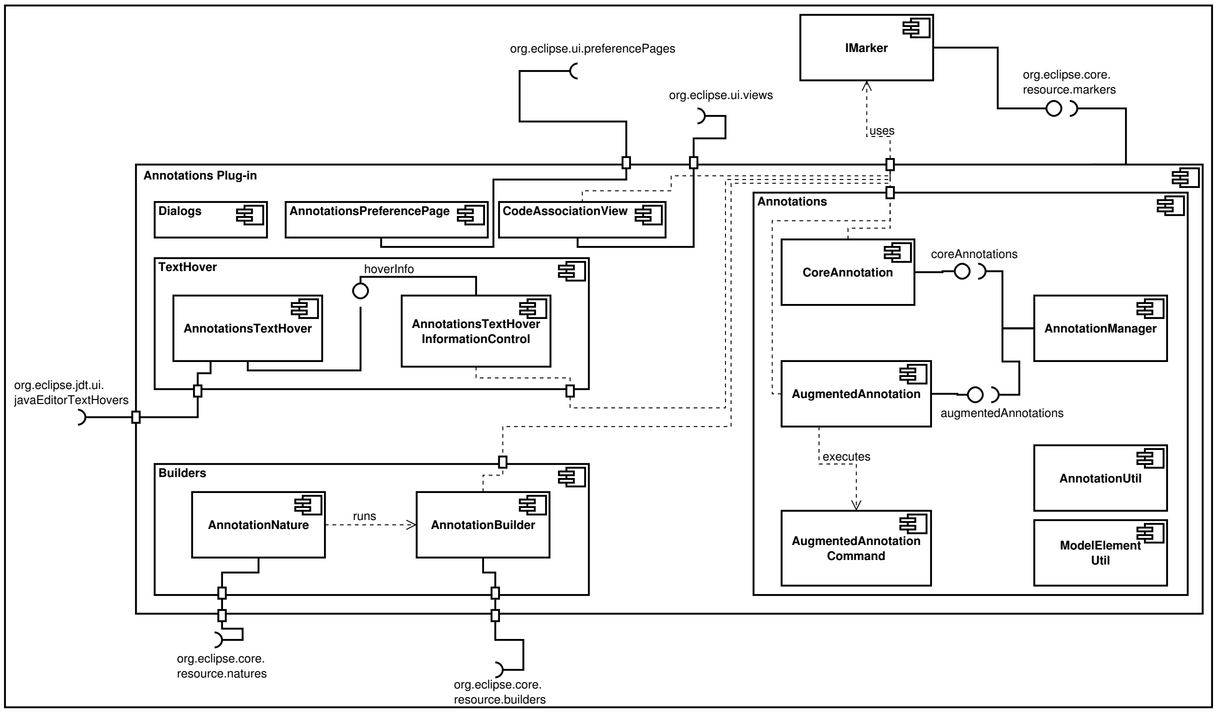


Figure 20: Component diagram for the Annotations plug-in

5.1 Model and Program Logic

In the following sections, we will describe the implementation and design for the key components of the plug-in’s model and program logic.

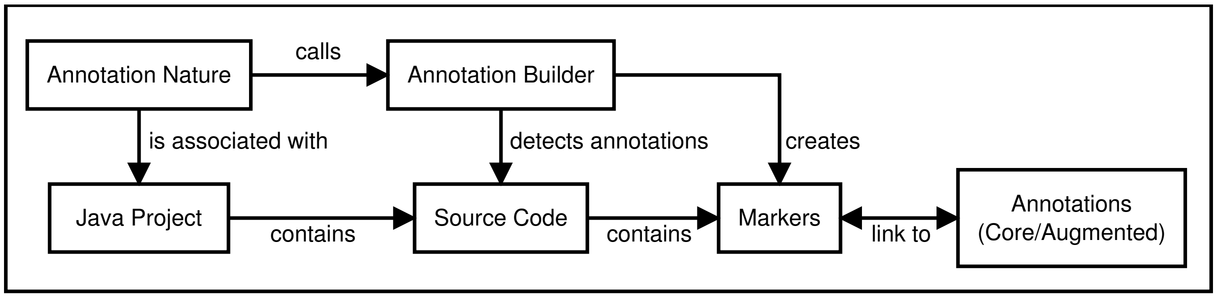


Figure 21: Configuration of the system behind the Annotations plug-in

5.1.1 Annotation Nature and Annotation Builder

For detecting and creating annotations we use Eclipse's *project natures* and *project builders* extension points. A Java project can be linked to any number of project natures. Each nature has a project builder associated with it. Each time a source code file within a Java project is saved, the project natures that are linked to it are notified. The project natures then call their respective project builders which are responsible for handling the saved source code.

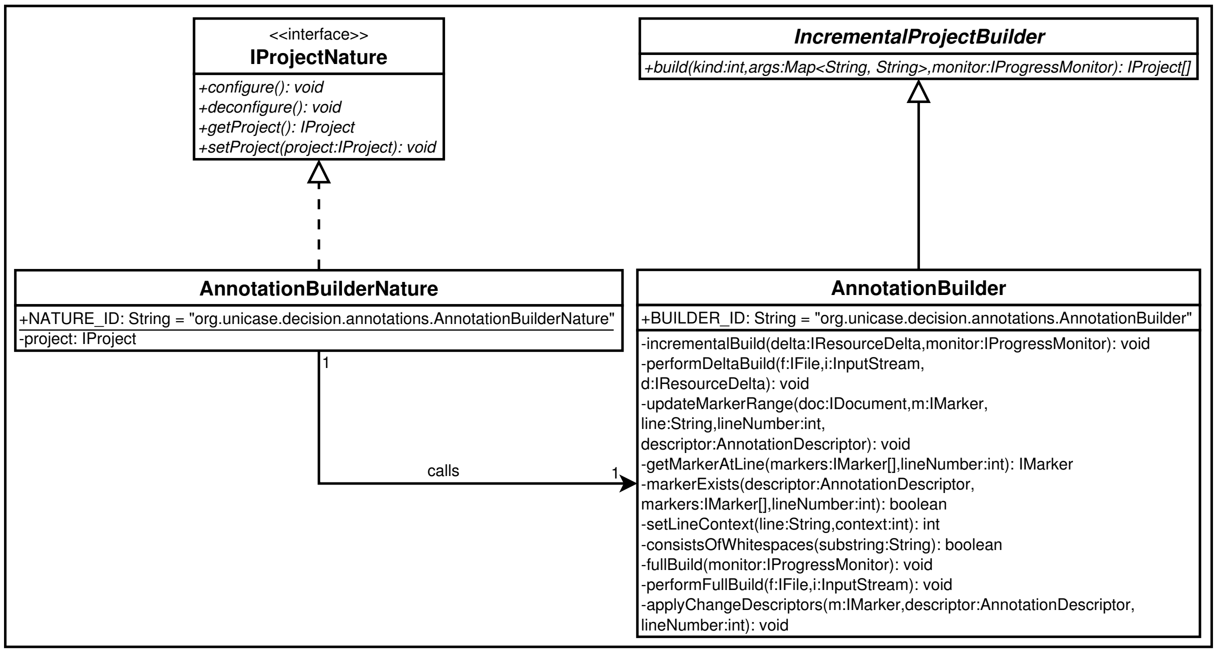
Initially, all existing Java projects were automatically associated with our annotation nature. We soon decided it would be better to allow users to associate and unassociate individual Java projects with our annotation nature, as this also allows us to specify a UNICASE project with which a given Java project is associated.

Our annotation nature can be associated and unassociated with a Java project via a context menu. Each time a Java source code file is saved and the annotation nature is associated with the file's project, the annotation builder is called. The annotation builder scans the saved source code for any instances of annotations and creates an annotation marker at the location of the annotation each time it finds such an instance. It also manages the linkage between the annotation markers and the annotation instances. The configuration of our system is illustrated in Figure 21.

However, this approach means that the source code has to be handled without using the underlying object model, which is used by Eclipse's default Java editor. Any modifications to the source code in the Java editor also affect the object model and are handled by the editor's *reconciler*. The reconciler is responsible for updating the object model on-the-fly and an incremental or even full build of the source code is only executed when the source code file is saved. We initially wanted to use the object model to handle the source code, but to our knowledge there exists no way to add a custom reconciler to Eclipse's default Java editor, so using a project builder was the next best alternative. Directly working with the source code added a number of issues, especially when updating source code and keeping the descriptions of model elements and hard-linked annotations and their markers synchronized.

In Sections 3.2 and 3.3, we wrote our annotations as ”pure” Java code annotations. We discovered that this approach raised a number of issues during the implementation. Because every Java code annotation type needs to be declared in its own class file, the creation of annotation types at runtime becomes very difficult, as the class files would need to be created or modified at runtime as well. Also, the descriptions of these Java code annotations have to be written in parentheses as well as in quotation marks, which adds unnecessary overhead. Finally, we use the Java Platform 7.0 as the basis for our implementation, which does not support multiple Java code annotations of the same type in the same scope. To bypass these issues, we embed our annotations in Java comments. This also removes the need to put annotation descriptions in parentheses and quotation marks, which eases the workflow.

The class diagram for the annotation nature and the annotation builder can be seen in Figure 22.



<<interface>>
IProjectNature
+configure(): void
+deconfigure(): void
+getProject(): IProject
+setProject(project:IProject): void
IncrementalProjectBuilder
+build(kind:int,args:Map<String, String>,monitor:IProgressMonitor): IProject[]
AnnotationBuilderNature
+NATURE_ID: String = "org.unicase.decision.annotations.AnnotationBuilderNature"
-project: IProject
1
calls
1
AnnotationBuilder
+BUILDER_ID: String = "org.unicase.decision.annotations.AnnotationBuilder"
-incrementalBuild(delta:IResourceDelta,monitor:IProgressMonitor): void
-performDeltaBuild(f:IFile,i:InputStream, d:IResourceDelta): void
-updateMarkerRange(doc:IDocument,m:IMarker, line:String,lineNumber:int, descriptor:AnnotationDescriptor): void
-getMarkerAtLine(markers:IMarker[],lineNumber:int): IMarker
-markerExists(descriptor:AnnotationDescriptor, markers:IMarker[],lineNumber:int): boolean
-setLineContext(line:String,context:int): int
-consistsOfWhitespaces(substring:String): boolean
-fullBuild(monitor:IProgressMonitor): void
-performFullBuild(f:IFile,i:InputStream): void
-applyChangeDescriptors(m:IMarker,descriptor:AnnotationDescriptor, lineNumber:int): void

Figure 22: The class diagram for the annotation nature and the annotation builder.

A project is associated with the *AnnotationBuilderNature* via the *setProject()* and *configure()* methods and unassociated with the *deconfigure()* method. Our *AnnotationBuilderNature* then calls the *AnnotationBuilder* each time a source code file is saved. The *Annotation-Builder's* main method is the *performDeltaBuild()* method, which scans the source code for annotations and creates, modifies or deletes annotation markers accordingly. As is apparent from the diagram, a number of helper methods had to be written for the *AnnotationBuilder* in order to properly handle the detection of annotations in the source code.

5.1.2 Annotations and the Annotation Manager

Figure 23 shows the generalized classes for the annotations and the annotations manager. The *AbstractAnnotation* is the parent class for both the *AugmentedAnnotation* and *CoreAn-notation* classes. Each annotation refers to a marker and is identifiable by a UUID[5]. Each marker refers to its associated annotation using the string representation of its UUID, out of which the UUID can be reconstructed.

Each *CoreAnnotation* refers to exactly one model element. The *hardLinked* flag indicates whether the *CoreAnnotation* is hard-linked to the referred model element.

[5]Universally unique identifier - http://docs.oracle.com/javase/1.5.0/docs/api/java/util/UUID.html

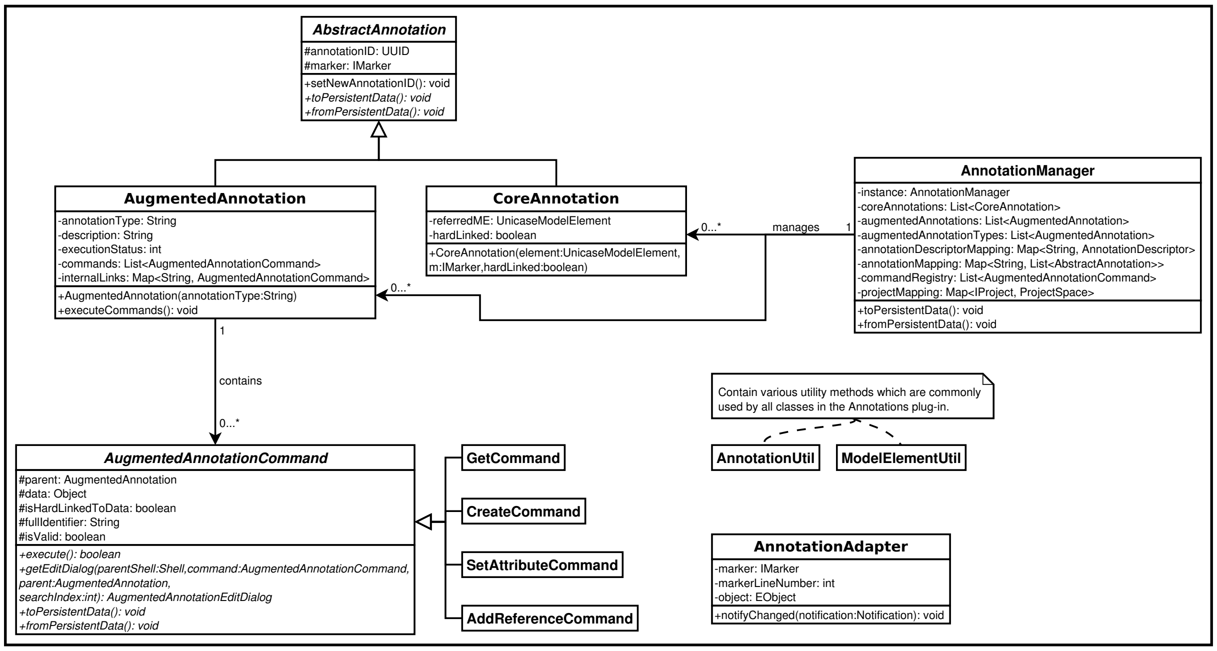

AbstractAnnotation
#annotationID: UUID
#marker: IMarker
+setNewAnnotationID(): void
+toPersistentData(): void
+fromPersistentData(): void
AugmentedAnnotation
-annotationType: String
-description: String
-executionStatus: int
-commands: List<AugmentedAnnotationCommand>
-internalLinks: Map<String, AugmentedAnnotationCommand>
+AugmentedAnnotation(annotationType:String)
+executeCommands(): void
CoreAnnotation
-referredME: UnicaseModelElement
-hardLinked: boolean
+CoreAnnotation(element:UnicaseModelElement, m:IMarker,hardLinked:boolean)
AnnotationManager
-instance: AnnotationManager
-coreAnnotations: List<CoreAnnotation>
-augmentedAnnotations: List<AugmentedAnnotation>
-augmentedAnnotationTypes: List<AugmentedAnnotation>
-annotationDescriptorMapping: Map<String, AnnotationDescriptor>
-annotationMapping: Map<String, List<AbstractAnnotation>>
-commandRegistry: List<AugmentedAnnotationCommand>
-projectMapping: Map<IProject, ProjectSpace>
+toPersistentData(): void
+fromPersistentData(): void
0...*
manages
1
0...*
1
contains
0...*
AugmentedAnnotationCommand
#parent: AugmentedAnnotation
#data: Object
#isHardLinkedToData: boolean
#fullIdentifier: String
#isValid: boolean
+execute(): boolean
+getEditDialog(parentShell:Shell,command:AugmentedAnnotationCommand, parent:AugmentedAnnotation, searchIndex:int): AugmentedAnnotationEditDialog
+toPersistentData(): void
+fromPersistentData(): void
GetCommand
CreateCommand
SetAttributeCommand
AddReferenceCommand
AnnotationUtil
ModelElementUtil
AnnotationAdapter
-marker: IMarker
-markerLineNumber: int
-object: EObject
+notifyChanged(notification:Notification): void
Contain various utility methods which are commonly used by all classes in the Annotations plug-in.

Figure 23: Generalized class diagram for the annotations and the annotation manager

Each *AugmentedAnnotation* contains a number of *AugmentedAnnotationCommands*, which are executed by the *executeCommands()* method. After its completion, the *executionStatus* attribute is set depending on whether the execution of the commands was successful or failed. The shared references between the commands are managed by the *internalLinks* attribute.

An *AugmentedAnnotationCommand* is the parent class for all commands. Each command refers to the containing *AugmentedAnnotation* via the *parent* attribute. The command may store its data with its *data* attribute. Analogously to the *hardLinked* flag in the *CoreAnnotation* class, the *isHardLinkedToData* flag indicates whether the command is hard-linked to its data. Each command provides a unique *fullIdentifer* which is used to refer to the command via the *internalLinks* attribute of the containing *AugmentedAnnotation*. The *isValid* flag indicates whether the command's references and data are valid. This flag is used when the parent *AugmentedAnnotation* is executing its command. When one of the commands is not valid, the *AugmentedAnnotation's* command status will be set to failed. The command is executed with the *execute()* method, which returns *true* or *false* depending on whether the execution of the command was successful or failed.

We implemented four different commands. The *GetCommand* searches for an annotation which refers to a decision element of a given type within the source code. The *CreateCommand* creates a decision element of a given type as a child of a decision element. The *SetAttributeCommand* sets the attribute of a given decision element to a given value. The *AddReferenceCommand* sets the reference of a given decision element to the given target element if it is single-valued or adds the target element to the reference if the reference is multi-valued. Additional commands can be added via the *org.unicase.decision.annotations.addCommand* extension point.

All *Core-* and *AugmentedAnnotations* are managed by the singleton *AnnotationManager* class, which is also responsible for data persistence with the *toPersistentData()* and *fromPer-*

sistentData() methods. The *toPersistentData()* method makes the data persistent while the *fromPersistentData()* reconstructs the data from the persistent data generated by the *toPersistentData()* method. The *AbstractAnnotation* and *AugmentedAnnotationCommand* classes have to implement their own versions of the *toPersistentData()* and *fromPersistentData()* methods, as the *AnnotationManager* delegates the persistence handling for *Core-* and *AugmentedAnnotations* to the annotations themself.

Persistence was a big issue as most of the data used by the annotation manager and the annotations could not be made persistent using Java's *Serializable* interface[6]. Instead, we make the data persistent by using string identifiers and reconstruct the data from these identifiers when the Plug-in is started again.

The *AnnotationUtil* and *ModelElementUtil* classes provide commonly used functionality that is used by all classes in the Annotations plug-in.

The *AnnotationAdapter* is a listener class that is attached each hard-linked decision element and the corresponding core annotation via its *object* and *marker* attributes respectively. Every time the decision element's description is changed, the listener is notified via the *notifyChanged()* method and the core annotation's description in the source code is refreshed accordingly.

5.2 User Interface

In this section, we will describe the implementation and design for the key components of the plug-in's user interface (*UI*).

5.2.1 Text Hover

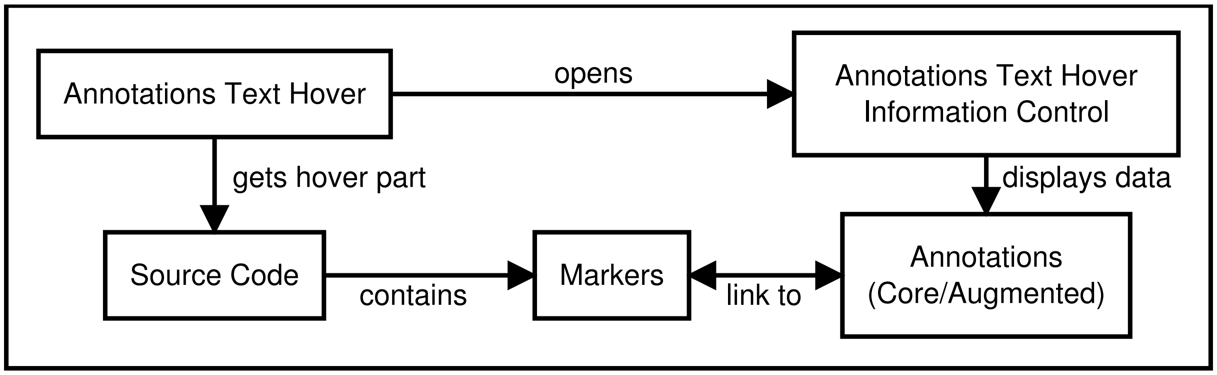
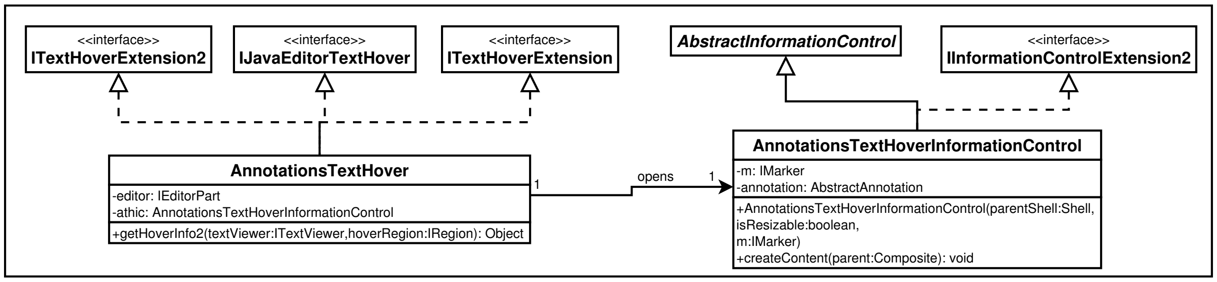


Figure 24: Configuration of the rich text hover system

For displaying the hidden attributes of our annotations in the source code, we use Eclipse's *rich text hovers*. Rich text hovers allow the definition of user-defined hover controls, which are displayed when users hover over a certain text part in the source code with the mouse.

The classes associated with the text hover can be seen in Figure 25.

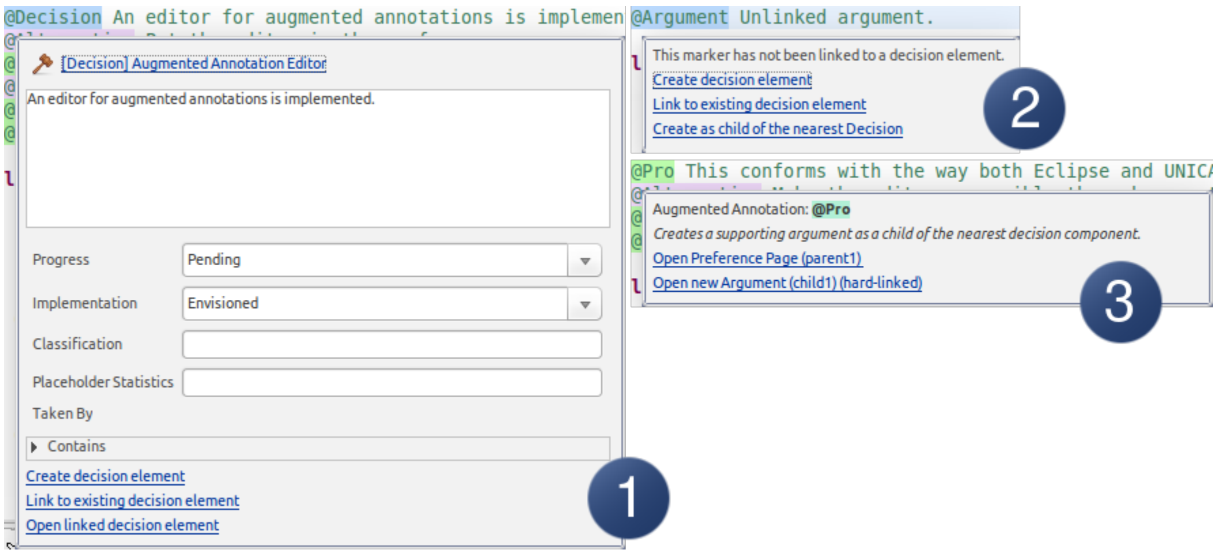
[6]Interface for serialization - http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html



<<interface>>
ITextHoverExtension2
<<interface>>
IJavaEditorTextHover
<<interface>>
ITextHoverExtension
AbstractInformationControl
<<interface>>
IInformationControlExtension2
AnnotationsTextHover
-editor: IEditorPart
-athic: AnnotationsTextHoverInformationControl
+getHoverInfo2(textViewer:ITextViewer,hoverRegion:IRegion): Object
1
opens
1
AnnotationsTextHoverInformationControl
-m: IMarker
-annotation: AbstractAnnotation
+AnnotationsTextHoverInformationControl(parentShell:Shell, isResizable:boolean, m:IMarker)
+createContent(parent:Composite): void

Figure 25: Generalized class diagram for the text hover

The *AnnotationsTextHover* class is responsible for opening the *AnnotationsTextHoverInformationControl* with the *getHoverInfo2()* method. This method gets the mouse hover region of the currently active text editor. If the hover region represents an annotation type, the *AnnotationsTextHoverInformationControl* is opened with the marker that is associated with the hover region. The content of the *AnnotationsTextHoverInformationControl* is created with its *createContent()* method. The content of the *AnnotationsTextHoverInformationControl* varies depending on the annotation that is linked to the marker. Figure 26 illustrates the different contents of the text hover.


@Decision An editor for augmented annotations is implemen

[Decision] Augmented Annotation Editor

An editor for augmented annotations is implemented.

Progress	Pending
Implementation	Envisioned
Classification	
Placeholder Statistics	
Taken By	

Contains

Create decision element
Link to existing decision element
Open linked decision element

@Argument Unlinked argument.

This marker has not been linked to a decision element.
Create decision element
Link to existing decision element
Create as child of the nearest Decision

@Pro This conforms with the way both Eclipse and UNICA

Augmented Annotation: @Pro
Creates a supporting argument as a child of the nearest decision component.
Open Preference Page (parent1)
Open new Argument (child1) (hard-linked)

Figure 26: The different contents of the text hover. Number 1: core annotation is hard-linked to a decision. Number 2: annotation is not linked. Number 3: augmented annotation

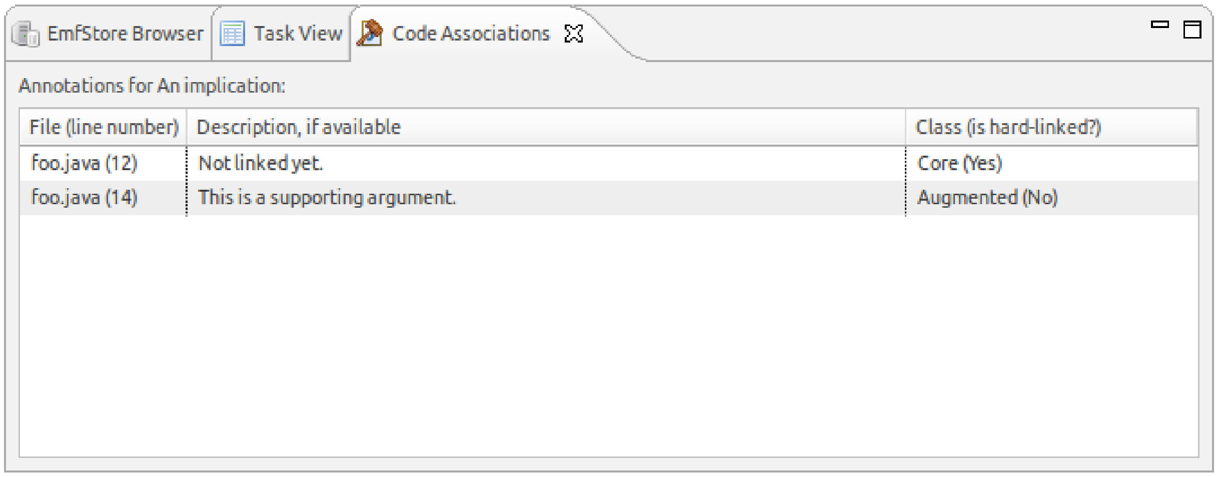
Number 1 in Figure 26 shows a core annotation that is hard-linked to a decision. Widgets to modify the decision's attributes and references are created dynamically. If the core annotation is not hard-linked, the widgets are created, but set to inactive, so users have to open the decision in UNICASE's editor in order to modify it.

Number 2 in Figure 26 shows a core annotation that is yet to be linked to a decision element. This can be the case if the annotation has either been just created or the referred decision element has been deleted, rendering the annotation invalid.

Number 3 in Figure 26 shows an augmented annotation. The augmented annotation type's name and its description are shown. Links to decision elements which are referred to or have been created by the augmented are shown as well.

5.2.2 Code Associations View

The code associations view shows all annotations that link to a given decision element (see Figure 27).



Annotations for An implication:

File (line number)	Description, if available	Class (is hard-linked?)
foo.java (12)	Not linked yet.	Core (Yes)
foo.java (14)	This is a supporting argument.	Augmented (No)

Figure 27: The code associations view

The linkage is established with the *AnnotationManager's annotationMapping*, which keeps track of the annotations that link to a given decision element.

5.2.3 Augmented Annotation Manager

The augmented annotation manager allows the modification of augmented annotation types. The classes associated with the augmented annotation manager can be seen in Figure 28.

We decided it would be best to place the augmented annotation manager in the workbench preference page of the Annotations plug-in (see Figure 29), because this conforms to the default methodology of setting plug-in preferences in both Eclipse and UNICASE. The contents of the preference page are created with the *createContents()* method of the *AnnotationsPreferencePage* class. The *EditAugmentedAnnotationsCommandsDialog* can be opened from the preference page. This dialog allows users to edit an augmented annotation type's name, description and commands.

Commands are created and edited using the *AugmentedAnnotationCommandEditDialog*, which is the super class for all command edit dialogs. All *AugmentedAnnotationCommands* have to provide their own implementation of the *AugmentedAnnotationCommandEditDialog*. The implementation should allow users to edit all of the command's parameters.

The augmented annotation manager and the various dialogs associated with it can be seen in Figure 29.

The left hand side of Figure 29 shows the preference page for the Annotations plug-in with the augmented annotation manager implemented as a table. Augmented annotation types can be added, edited or removed with a right-click context menu.

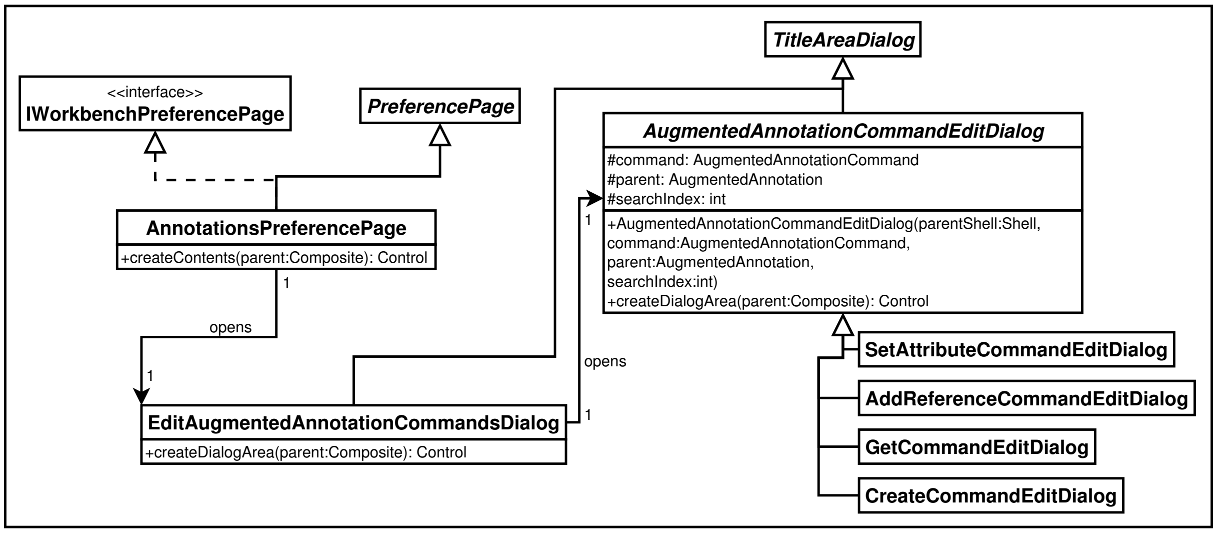
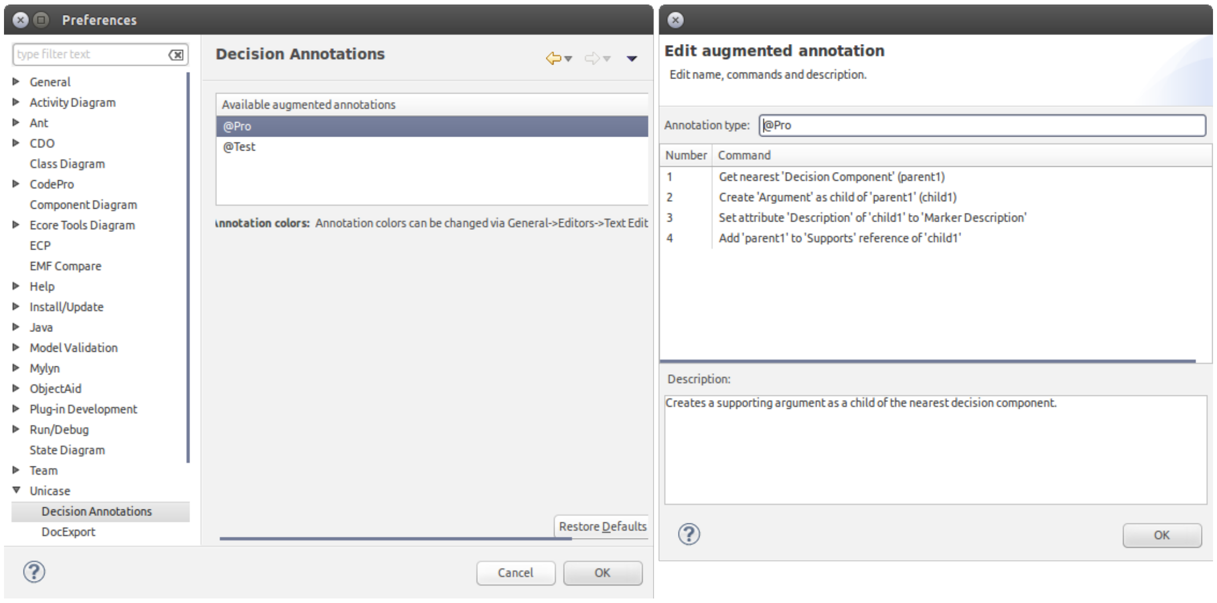

TitleAreaDialog
<<interface>>
IWorkbenchPreferencePage
PreferencePage
AugmentedAnnotationCommandEditDialog
#command: AugmentedAnnotationCommand
#parent: AugmentedAnnotation
#searchIndex: int
+AugmentedAnnotationCommandEditDialog(parentShell:Shell, command:AugmentedAnnotationCommand, parent:AugmentedAnnotation, searchIndex:int)
+createDialogArea(parent:Composite): Control
AnnotationsPreferencePage
+createContents(parent:Composite): Control
1
opens
1
1
opens
1
EditAugmentedAnnotationCommandsDialog
+createDialogArea(parent:Composite): Control
SetAttributeCommandEditDialog
AddReferenceCommandEditDialog
GetCommandEditDialog
CreateCommandEditDialog

Figure 28: Generalized class diagram for the augmented annotation manager



Decision Annotations

Available augmented annotations
@Pro
@Test

Annotation colors: Annotation colors can be changed via General->Editors->Text Edit

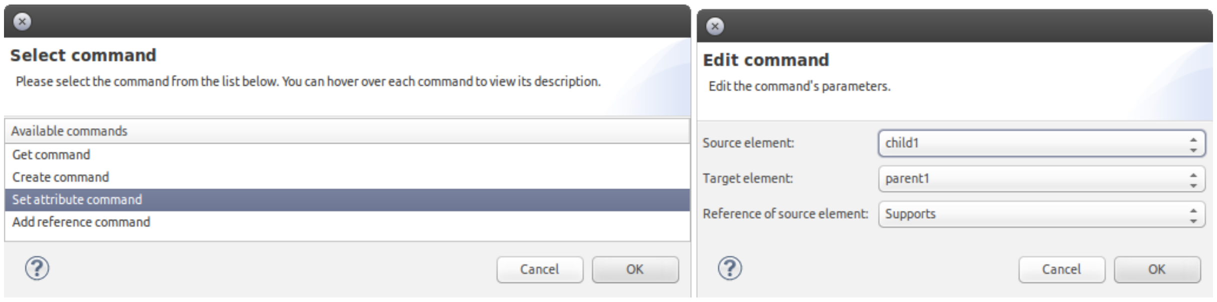
Edit augmented annotation
Edit name, commands and description.

Annotation type: @Pro

Number	Command
1	Get nearest 'Decision Component' (parent1)
2	Create 'Argument' as child of 'parent1' (child1)
3	Set attribute 'Description' of 'child1' to 'Marker Description'
4	Add 'parent1' to 'Supports' reference of 'child1'

Description:
Creates a supporting argument as a child of the nearest decision component.

Figure 29: The various elements of the augmented annotation manager. Left: the manager in the plug-in's workbench preference page. Right: the edit dialog for the *@Pro* augmented annotation



Select command
Please select the command from the list below. You can hover over each command to view its description.

Available commands
Get command
Create command
Set attribute command
Add reference command

Edit command
Edit the command's parameters.

Source element:	child1
Target element:	parent1
Reference of source element:	Supports

Figure 30: The various elements of the augmented annotation manager. Left: the command selection dialog. Right: the edit dialog for the add reference command

The right hand side of Figure 29 shows the augmented annotation editor for the *@Pro* augmented annotation. As with the augmented annotation manager, commands can be created, edited, inserted or removed with a right-click context menu.

The left hand side of Figure 30 shows the command selection dialog, which allows users to select the command they want to add to the augmented annotation type. The list of available commands is retrieved from the *commandRegistry* of the *AnnotationManager*.

The right hand side of Figure 30 shows the command editor for the add reference command. Its parameters can be adjusted using combo boxes.

5.2.4 Dialogs

We also implemented a number of different dialogs, which we will describe shortly here.

5.2.4.1 Model Element Selection Dialog

This dialog is used by the text hover for core annotations to either select the location of a decision element that is to be created or to select the decision element that the core annotation is to refer to. The selection is done with a selection tree which displays the contents of the UNICASE project the source code file's Java project is linked to. In the former case, the dialog also contains a text field to specify the name of the decision element.

5.2.4.2 Project Linkage Dialog

This dialog is shown whenever users relink a Java project, that is already linked to a UNICASE project. It gives users the option to disable annotations in the given Java project - in which case all annotations and any hard-linked decision elements are deleted - or to change the linked UNICASE project.

5.3 Interaction

To illustrate the interactional dynamics of the components in our plug-in, we will use a typical usage scenario. In our scenario, a programmer wants

1. to create a core annotation (see Figure 31) and
2. create a new decision element, which is hard-linked to the core annotation (see Figure 32),

in order to document an implementation decision.

5.3.1 Creating a Core Annotation

Figure 31 shows a generalized sequence diagram describing the key components and interactions involved in the creation of a core annotation.

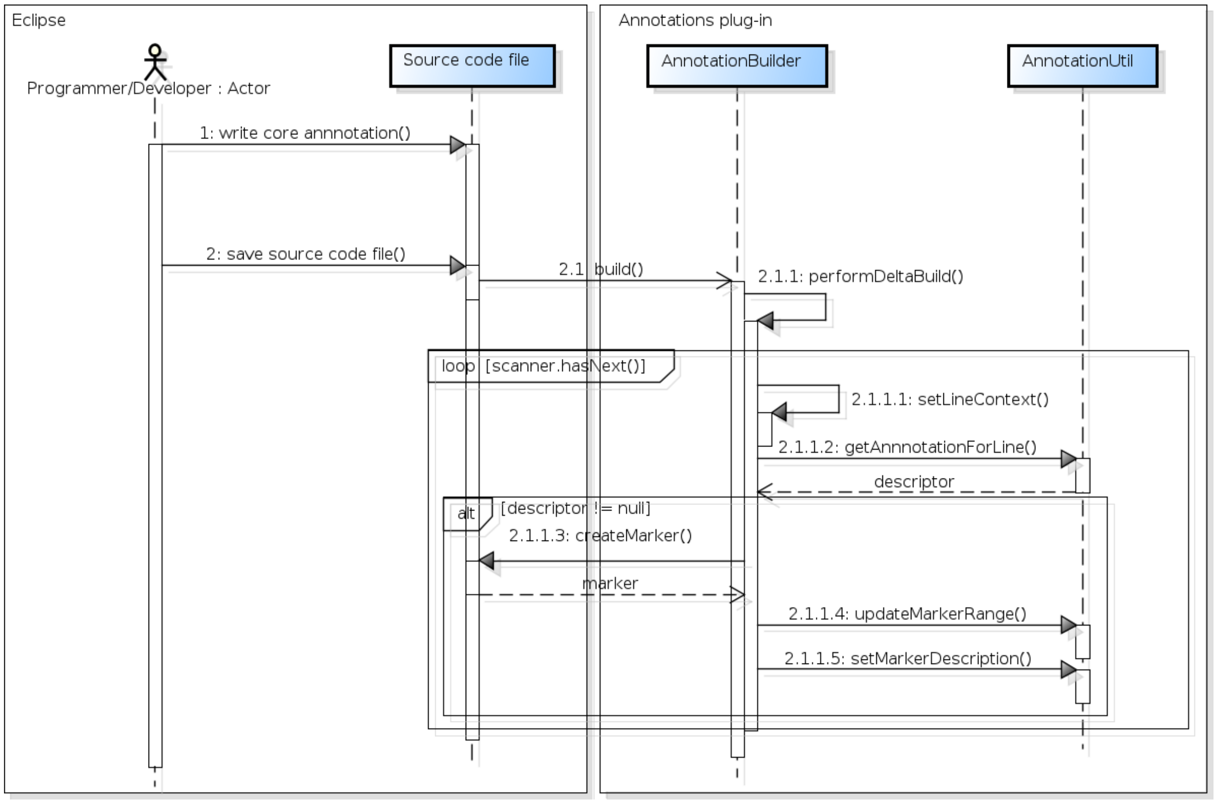

Eclipse
Programmer/Developer : Actor
Source code file
1: write core annnotation()
2: save source code file()
2.1 build()
Annotations plug-in
AnnotationBuilder
AnnotationUtil
2.1.1: performDeltaBuild()
loop [scanner.hasNext()]
2.1.1.1: setLineContext()
2.1.1.2: getAnnnotationForLine()
descriptor
alt [descriptor != null]
2.1.1.3: createMarker()
marker
2.1.1.4: updateMarkerRange()
2.1.1.5: setMarkerDescription()

Figure 31: Generalized sequence diagram for the build process

The programmer in our scenario first writes the core annotation and a small description in the source code and then saves the source code file, which triggers the build process. The *AnnotationBuilder* is called and scans each line of the source code file with its *performDeltaBuild()* method. First, the *line context* is set using the *setLineContext()* method. The line context determines the type of comment the given line is embedded in. We differentiate between no comment, single-line comment (`//`) and multi-line comment (`/**`, `/*` and `*`).

Then, the annotation at the given line is determined using the *getAnnotationForLine()* method, which returns an *AnnotationDescriptor* or *null* if no annotation could be found for the given line. An *AnnotationDescriptor* contains all the necessary information to create a marker for an annotation. Using this information, a marker for the annotation is created at the given line using the *createMarker()* method of the containing source code file. To draw a colored box around the annotation, the *updateMarkerRange()* method is called. Finally, the marker's description is set to the description given in the source code file using the *setMarkerDescription()* method.

5.3.2 Creating a New Decision Element

Figure 32 shows a generalized sequence diagram describing the key components and interactions involved in the linkage of a core annotation to a decision element. After the programmer in our scenario has created a core annotation, he hovers over the annotation with the mouse. This calls the *AnnotationsTextHover*, which creates a new instance of the *An-*

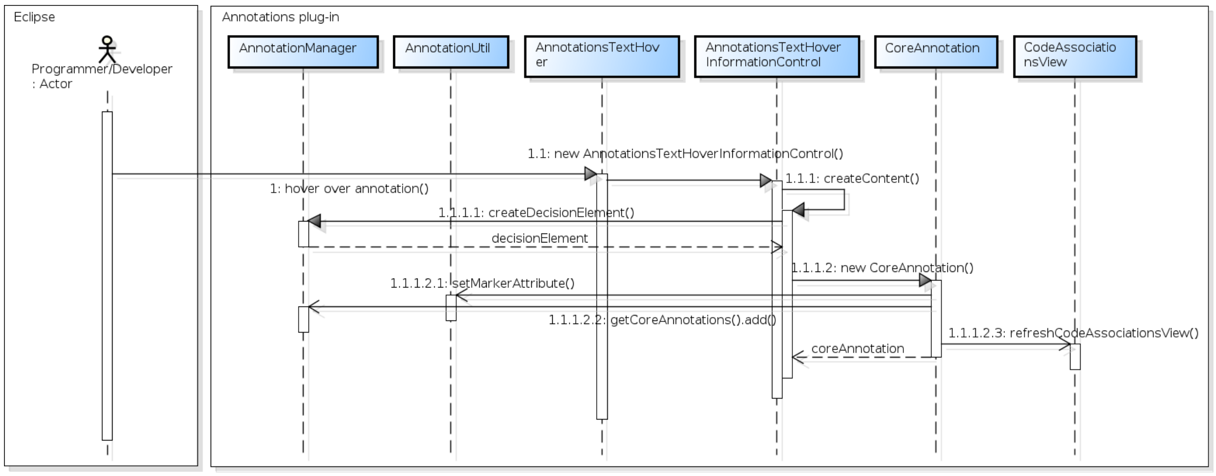

Eclipse
Programmer/Developer : Actor
Annotations plug-in
AnnotationManager
AnnotationUtil
AnnotationsTextHover
AnnotationsTextHover InformationControl
CoreAnnotation
CodeAssociationsView
1: hover over annotation()
1.1: new AnnotationsTextHoverInformationControl()
1.1.1: createContent()
1.1.1.1: createDecisionElement()
decisionElement
1.1.1.2: new CoreAnnotation()
1.1.1.2.1: setMarkerAttribute()
1.1.1.2.2: getCoreAnnotations().add()
1.1.1.2.3: refreshCodeAssociationsView()
coreAnnotation

Figure 32: Generalized sequence diagram for the linking process

notationsTextHoverInformationControl, i.e. the text hover. The content of the text hover is created with its *createContent()* method. Since our core annotation is yet to be linked, the text hover will look like Number 2 in Figure 26. The programmer selects the option to create a new decision element and selects the appropriate location and name for the new decision element with the *ModelElementSelectionDialog*.

This triggers the *createDecisionElement()* method of the *AnnotationManager*, which creates a new decision element of the given type. A new instance of *CoreAnnotation* is created, which is hard-linked to the new decision element and refers to the marker that is the input of the text hover. The link between the marker and the core annotation is established by setting the marker's UUID to the UUID of the core annotation using the marker's *setMarkerAttribute()* method. The core annotation is registered in the *AnnotationManager* by adding it to the list of core annotations, which is accessed by calling *getCoreAnnotations()*. Finally, the code associations view is refreshed using the *refreshCodeAssociationsView()* method.

5.4 Quality Assurance

Throughout the implementation, small tests were conducted manually each time a new feature was implemented or an existing feature was modified. As the implementation neared its completion, we designed and performed a number of tests. The full documentation of the conducted tests along with the corresponding test protocols can be found in the UNICASE documentation that accompanies this thesis.

5.4.1 Component Tests

The component tests were conducted manually. Clearly, automated unit tests with a testing framework like JUnit[7] would have been the better choice, but setting up the testing environment was found to be too time and resource consuming.

[7]JUnit testing framework - http://junit.org/

The tested functionality included the correct detection of all core annotations as well as predefined and user-defined augmented annotations.

5.4.2 Integration Tests

Inter-class communication within this plug-in is manageable, so integration tests were not deemed necessary in the context of this implementation.

5.4.3 System Tests

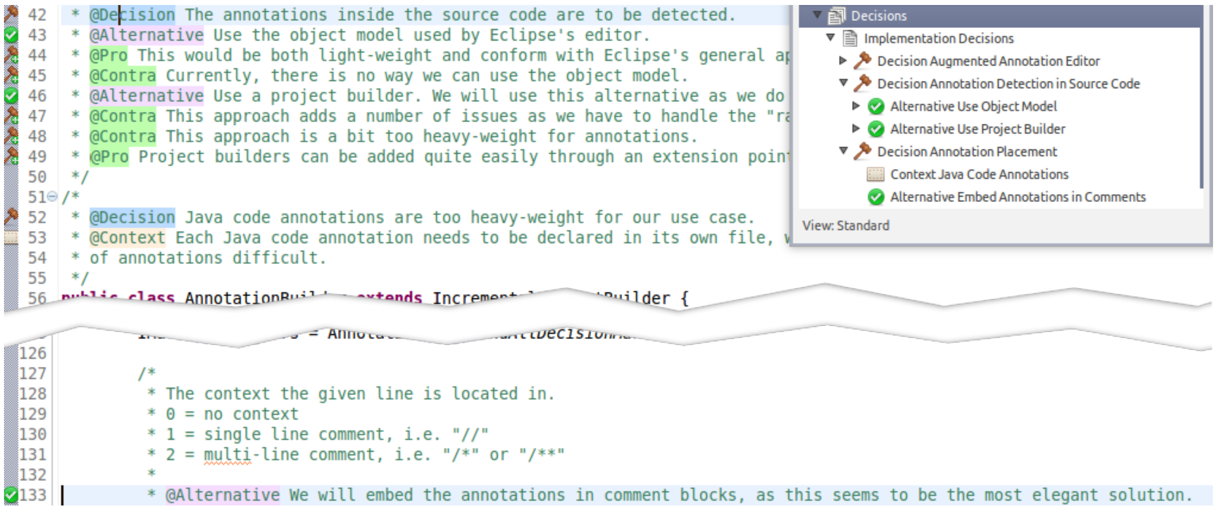
To test the interplay of critical UI and system components with the underlying classes, a number of system tests were conducted. The tests were conducted by manually performing the actions specified in the test case description. The system test cases included

- creating a core annotation,
- creating an augmented annotation,
- executing an augmented annotation with valid data,
- executing an augmented annotation with invalid data,
- creating a new decision element with an unlinked core annotation,
- creating a new decision element with a linked core annotation,
- creating a new decision element with a hard-linked core annotation,
- referring to a decision element with an unlinked core annotation,
- referring to a decision element with a linked core annotation,
- referring to a decision element with a hard-linked core annotation,
- changing the description of a hard-linked core annotation from within the source code,
- changing the description of a hard-linked core annotation from within the decision element,
- converting a single-line description into a multi-line description,
- changing the description of a linked core annotation,
- changing an attribute of a hard-linked decision element,
- deleting a core annotation and
- deleting an augmented annotation.

Because most of the major issues, especially those regarding the program logic, had been already fixed during the implementation, most of the system tests were conducted successfully.

6 Discussion

In order to test the usability and effectiveness of our annotation schema and its implementation, we used it to document the key implementation decisions made during the design and implementation of the Annotations plug-in. Two examples can be seen in Figure 33.



```
 * @Decision The annotations inside the source code are to be detected.
 * @Alternative Use the object model used by Eclipse's editor.
 * @Pro This would be both light-weight and conform with Eclipse's general ap
 * @Contra Currently, there is no way we can use the object model.
 * @Alternative Use a project builder. We will use this alternative as we do
 * @Contra This approach adds a number of issues as we have to handle the "ra
 * @Contra This approach is a bit too heavy-weight for annotations.
 * @Pro Project builders can be added quite easily through an extension poin
 */
/*
 * @Decision Java code annotations are too heavy-weight for our use case.
 * @Context Each Java code annotation needs to be declared in its own file, w
 * of annotations difficult.
 */
public class AnnotationBui      extends Increment        Builder {
[illegible]

        /*
         * The context the given line is located in.
         * 0 = no context
         * 1 = single line comment, i.e. "//"
         * 2 = multi-line comment, i.e. "/*" or "/**"
         *
         * @Alternative We will embed the annotations in comment blocks, as this seems to be the most elegant solution.
```

```
Decisions
  Implementation Decisions
    Decision Augmented Annotation Editor
    Decision Annotation Detection in Source Code
      Alternative Use Object Model
      Alternative Use Project Builder
    Decision Annotation Placement
      Context Java Code Annotations
      Alternative Embed Annotations in Comments
View: Standard
```

Figure 33: Two example decisions regarding the annotation builder from the source code of the Annotations plug-in with the corresponding decision elements in the UNICASE project

We found that the annotation schema was easy to use. The workflow is simple: first, the annotation and its description have to be written in a comment block. Then, the source code file has to be saved, so the annotation is registered. Finally, the annotation has to be linked to a decision element and, if necessary, its attributes have to be adjusted.

We found that the use of our predefined *@Pro* and *@Contra* augmented annotations rapidly sped up the documentation process when providing supporting or attacking arguments for an alternative. For augmented annotations, the linkage of the annotation to a decision element is automated, which removes the need to use UI elements altogether.

The code associations view allows a quick traversal between a decision element and the annotations that refer to it. We also found that it gave a concise overview over the relations between decision elements and annotations.

However, there are a number of issues. The decision elements in the DDM form an implicit multi-level hierarchy with a decision as the root element. While this relationship is easy to visualize with a tree view, it becomes very difficult to visualize in the source code with the given editor. Our implementation does not visualize this relationship in the source code at all.

Using Eclipse's rich text hovers for the modification of annotations, while easy to use, do interrupt the workflow, because users have to constantly switch between keyboard and mouse in order to document their implementation decisions.

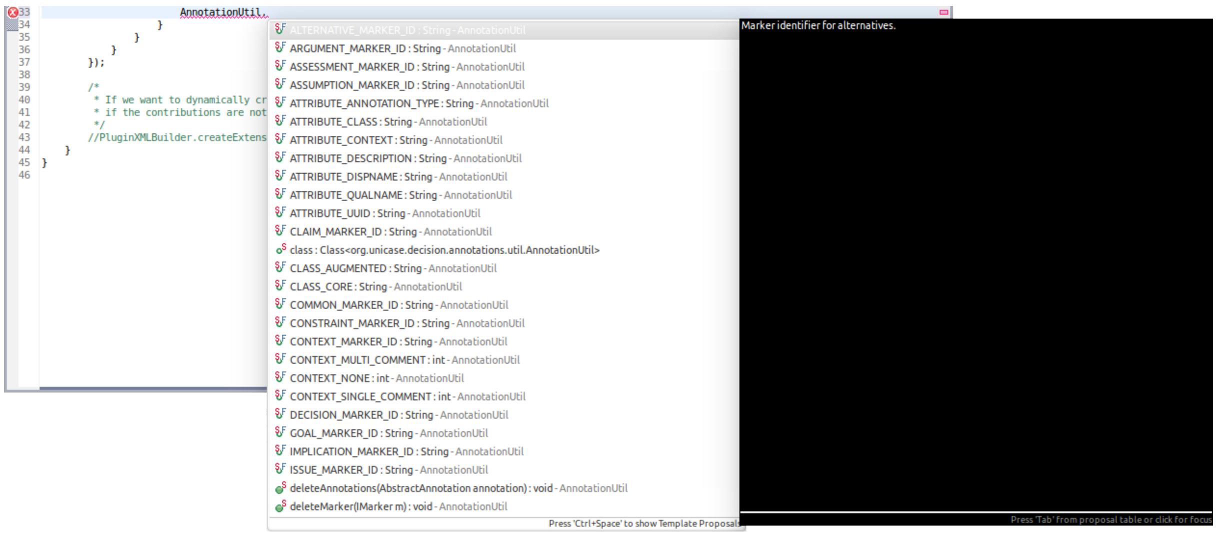
This interruption could be minimized by allowing all attributes and references of referred decision elements to be edited from within the source code by introducing separate annotation types which allow the modification of the referred decision element's attributes and references. Listing 5 illustrates this approach for the decision from our example in Section 2.3.

```
[start of comment block]
 * ...
 * @Decision A wizard for exporting an entire project is implemented.
 * @Decision#progress pending
 * @Decision#implementation envisioned
 * ...
[end of comment block]
[source code]
```

Listing 5: Setting attributes using annotations

In Listing 5, users can type in the annotation followed a number sign (”#”) and the attribute’s or reference’s internal name to indicate that they want to set the referred model element’s attribute or reference to the specified value.

However, this approach would also require users to have detailed knowledge about the attributes and references of each decision element, as this information cannot be seen in the source code perspective. Users would need to switch between the UNICASE and source code perspectives if they wanted to see which attributes and references the referred decision element has. This issue could be resolved by using Eclipse’s *content assist* feature, which can be seen in Figure 34. The content assist is shown in a small hover window when users type in methods or attributes of class instances in Eclipse. In the context of annotations, the content assist could show the available attributes and references as well as the possible values for each attribute and reference when users type annotations in the source code.



```
                                    AnnotationUtil.
                                }
                            }
                        }
                    });

                    /*
                     * If we want to dynamically cr
                     * if the contributions are not
                     */
                    //PluginXMLBuilder.createExtens
                }
            }
```

```
ALTERNATIVE_MARKER_ID : String - AnnotationUtil
ARGUMENT_MARKER_ID : String - AnnotationUtil
ASSESSMENT_MARKER_ID : String - AnnotationUtil
ASSUMPTION_MARKER_ID : String - AnnotationUtil
ATTRIBUTE_ANNOTATION_TYPE : String - AnnotationUtil
ATTRIBUTE_CLASS : String - AnnotationUtil
ATTRIBUTE_CONTEXT : String - AnnotationUtil
ATTRIBUTE_DESCRIPTION : String - AnnotationUtil
ATTRIBUTE_DISPNAME : String - AnnotationUtil
ATTRIBUTE_QUALNAME : String - AnnotationUtil
ATTRIBUTE_UUID : String - AnnotationUtil
CLAIM_MARKER_ID : String - AnnotationUtil
class : Class<org.unicase.decision.annotations.util.AnnotationUtil>
CLASS_AUGMENTED : String - AnnotationUtil
CLASS_CORE : String - AnnotationUtil
COMMON_MARKER_ID : String - AnnotationUtil
CONSTRAINT_MARKER_ID : String - AnnotationUtil
CONTEXT_MARKER_ID : String - AnnotationUtil
CONTEXT_MULTI_COMMENT : int - AnnotationUtil
CONTEXT_NONE : int - AnnotationUtil
CONTEXT_SINGLE_COMMENT : int - AnnotationUtil
DECISION_MARKER_ID : String - AnnotationUtil
GOAL_MARKER_ID : String - AnnotationUtil
IMPLICATION_MARKER_ID : String - AnnotationUtil
ISSUE_MARKER_ID : String - AnnotationUtil
deleteAnnotations(AbstractAnnotation annotation) : void - AnnotationUtil
deleteMarker(IMarker m) : void - AnnotationUtil
```

Marker identifier for alternatives.

Press 'Ctrl+Space' to show Template Proposals
Press 'Tab' from proposal table or click for focus

Figure 34: Eclipse’s content assist feature

Due to the current technical limitations of the annotation builder, annotations have to be placed in *well-formed* Java comments, or else the building process will fail. Listing 6 shows well-formed Java comments while Listing 7 shows ill-formed comments.

```java
// Well-formed
/*
 * Well-formed
 */
```

Listing 6: Well-formed Java comments

```java
/* Ill-formed */
/*
   Ill-formed
 */
int n = 0; // Ill-formed
int n = 0; /* Ill-formed */
```

Listing 7: Ill-formed Java comments

Another issue lies with the implementation of our annotation schema itself. We think that the rationale of many of the decisions made during the implementation phase of our plug-in does not require explicit documentation, as we believe that the benefit of documenting this rationale is outweighed by the overhead from creating the documentation.

For example, we think that the rationale behind the choice of data structures in our implementation is not worth documenting. While there exists a non-functional requirement that dictates a reasonable level of performance, we discovered that the choice of data structures did not have a significant impact on the performance of our implementation.

Naturally, the benefit gained from documenting these seemingly ”trivial” implementation decisions depends on the impact that these decisions have on the project's functionality within the context of its requirements. For example, we believe that projects in the field of high performance computing would benefit greatly from the explicit documentation of the rationale behind the choice of data structures and algorithms.

For each project, the apparent benefits of explicitly documenting implementation decisions and the generated overhead have to be weighed up. We tried to minimize the overhead with our annotation schema and its implementation in particular, but there will always be some overhead, as making a decision and documenting the rationale behind it are two separate processes.

7 Conclusion

In this thesis, we introduced and implemented an annotation schema which allows the explicit documentation of implementation decisions. Annotations are written in the source code and linked to decision elements. We implemented our annotation schema as a plug-in for Eclipse 3.7, which works in conjunction with UNICASE 0.52.

The functionality and usability of our annotation schema and its implementation was tested on the source code of the implementation itself by documenting key decision made during its design and implementation phase using our annotations. We found that the annotation schema and its implementation in particular were easy to use and had considerable potential.

7.1 Future Work

An important aspect that received little attention in our thesis was the aspect of collaboration. By nature, software projects are highly collaborative, as there often is a large team behind these projects, where team members share information and work on separate pieces of source code in parallel. While UNICASE already supports collaboration through the use of the EMF-Store framework [18], the Annotations plug-in currently does not. The difficulty of implementing this collaboration aspect for the Annotations plug-in remains in question.

To truly assess the functionality and usability as well as the effectiveness of our annotation schema and its implementation, we will need to use this schema in a variety of different software projects.

Of particular interest are projects, whose implementation and the decisions behind the implementation have a great impact on the functionality of the project.

UNICASE comprises a number of elements that model different aspects of the software engineering process. In principle, our annotation schema could be used to refer to these model elements as well. System functions for example are modeled as the element *System-Function* in UNICASE. Our annotation schema could be used to link system functions in the actual source code to the corresponding system functions in UNICASE.

References

[1] J.E. Burge, J.M. Carroll, R. McCall, and I. Mistrík, ”Rationale-Based Software Engineering”. Springer-Verlag, 2006.

[2] UNICASE (Unified CASE-Tool), https://teambruegge.informatik.tu-muenchen.de/wiki/projects/unicase/. Retrieved on April 2014.

[3] Eclipse Integrated Development Environment, https://www.eclipse.org/. Retrieved on April 2014.

[4] C. Zannier, M. Chiasson and F. Maurer, ”A Model of Design Decision Making Based on Empirical Results of Interviews with Software Designers” in *Information and Software Technology*, Vol. 49, No. 6, pp. 637-653, Elsevier, 2007.

[5] T. Hesse and B. Paech, ”Supporting the Collaborative Development of Requirements and Architecture Documentation” in *Proceedings of the third International Workshop on the Twin Peaks of Requirements and Architecture (TwinPeaks’13)*, pp. 22-26, IEEE, 2013.

[6] Annotations, http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html. Retrieved on April 2014.

[7] G. Canfora, G. Casazza, and A. De Lucia, ”A Design Rationale Based Environment for Cooperative Maintenance” in *Internation Journal of Software Engineering and Knowledge Engineering*, Vol. 10, No. 5, pp. 627-645, World Scientific Publishing, 2000.

[8] Getting Started with the Annotation Processing Tool (apt), http://docs.oracle.com/javase/1.5.0/docs/guide/apt/GettingStarted.html. Retrieved on May 2014.

[9] Resource Markers - Eclipse Help, http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2FresAdv_markers.htm. Retrieved on May 2014.

[10] IMarker Interface - Eclipse Help, http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fcore%2Fresources%2FIMarker.html. Retrieved on May 2014.

[11] J.E. Burge and D.C. Brown, ”An Integrated Approach for Software Design Checking Using Design Rationale” in *1st International Conference on Design Computing and Cognition (DCC’04)*, pp. 557-576, Kluwer Academic Press, 2004.

[12] SEURAT: Software Engineering Using Design Rationale, dissertation defense by J.E. Burge, http://web.cs.wpi.edu/∼dcb/Papers/SEURAT/Burge-Diss-Defense.pdf. Retrieved on May 2014.

[13] R. Lougher and T. Rodden, ”Group Support for the Recording and Sharing of Maintenance Rationale” in *Software Engineering Journal*, Vol. 8, No. 6, pp. 295-306, IET, 1993.

[14] J.E. Burge, ”Software Engineering Using design RATionale”. Dissertation submitted to the Faculty of the Worcester Polytechnic Institute, 2005.

[15] A.J. Ko, R. DeLine and G. Venolia, ”Information Needs in Collocated Software Development Teams” in *29th International Conference on Software Engineering (ICSE 2007)*, pp. 344-353, IEEE, 2007.

[16] D. Falessi, G. Cantone and P. Kruchten, ”Value-Based Design Decision Rationale Documentation: Principles and Empirical Feasibility Study” in *Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, pp. 189-198, IEEE, 2008.

[17] D. Falessi, R. Capilla and G. Cantone, ”A Value-Based Approach for Documenting Design Decisions Rationale: A Replicated Experiment” in *Proceedings of the 3rd International Workshop on Sharing and Reusing Architectural Knowledge (SHARK 2008)*, pp. 63-70, ACM, 2008.

[18] M. Koegel and J. Helming, ”EMFStore - a Model Repository for EMF models” in *Proceedings of the 32nd International Conference for Software Engineering (ICSE 2010)*, pp. 307-308, IEEE, 2010.

List of Figures

1 The different phases in the life cycle of a software project along with the rationale associated with each development phase. Adapted from [14, Figure 2-1] . . . 2
2 Rating of different DDRD information categories as ”required” by subjects. From [16] . . . 5
3 Structure of the DDM. Adapted from [5] . . . 6
4 The DDM applied on the example. Adapted from [5] . . . 7
5 An example of a Java annotation . . . 8
6 The Eclipse Workbench. From left to right: Package Explorer, Editor, Outline 9
7 Active error and inactive warning markers in the Eclipse editor . . . 10
8 Overview over CM^2's annotation mechanism. From [7] . . . 12
9 Example of a source code segment that has been annotated using CM^2. Adapted from [7] . . . 12
10 The configuration of the system behind Lougher and Rodden's approach. Adapted from [13] . . . 13
11 SEURAT and Eclipse. From [11] . . . 14
12 The Bookmarks List. From [12] . . . 15
13 Relations between annotations and decision elements (Chen's notation) . . . 18
14 The use case diagram for our annotation schema . . . 28
15 UI workflow for creating a new decision element . . . 34
16 UI workflow for referring to an existing decision element . . . 35
17 UI workflow for managing augmented annotation types . . . 36
18 UI mock-up for the augmented annotations hover . . . 36
19 UI mock-up for the code associations view . . . 37
20 Component diagram for the Annotations plug-in . . . 39
21 Configuration of the system behind the Annotations plug-in . . . 40
22 The class diagram for the annotation nature and the annotation builder. . . . 41
23 Generalized class diagram for the annotations and the annotation manager . 42
24 Configuration of the rich text hover system . . . 43
25 Generalized class diagram for the text hover . . . 44
26 The different contents of the text hover. Number 1: core annotation is hard-linked to a decision. Number 2: annotation is not linked. Number 3: augmented annotation . . . 44
27 The code associations view . . . 45
28 Generalized class diagram for the augmented annotation manager . . . 46

29 The various elements of the augmented annotation manager. Left: the manager in the plug-in's workbench preference page. Right: the edit dialog for the *@Pro* augmented annotation . . . 46

30 The various elements of the augmented annotation manager. Left: the command selection dialog. Right: the edit dialog for the add reference command 46

31 Generalized sequence diagram for the build process . . . 48

32 Generalized sequence diagram for the linking process . . . 49

33 Two example decisions regarding the annotation builder from the source code of the Annotations plug-in with the corresponding decision elements in the UNICASE project . . . 51

34 Eclipse's content assist feature . . . 52

A.1 Package diagram for the Annotations plug-in . . . 59

B.1 Class diagram for the *org.unicase.decision.annotations* package . . . 60

B.2 Class diagram for the *org.unicase.decision.annotations.builders* package . . . 61

B.3 Class diagram for the *org.unicase.decision.annotations.model* package . . . 62

B.4 Class diagram for the *org.unicase.decision.annotations.model.commands* package . . . 63

B.5 Class diagram for the *org.unicase.decision.annotations.dialogs* package . . . 64

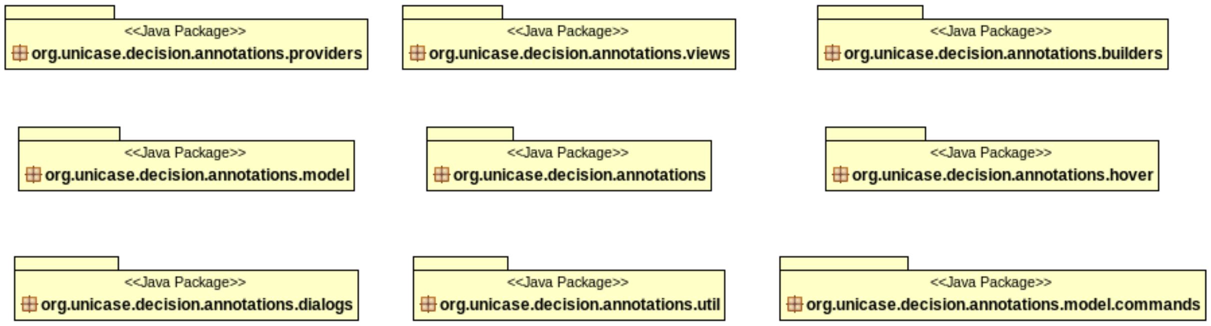
B.6 Class diagram for the *org.unicase.decision.hover* package . . . 65

B.7 Class diagram for the *org.unicase.decision.annotations.providers* package . . . 66

B.8 Class diagram for the *org.unicase.decision.annotations.views* package . . . 67

B.9 Class diagram for the *org.unicase.decision.annotations.util* package . . . 68

A Package Diagram



<<Java Package>>
org.unicase.decision.annotations.providers
<<Java Package>>
org.unicase.decision.annotations.views
<<Java Package>>
org.unicase.decision.annotations.builders
<<Java Package>>
org.unicase.decision.annotations.model
<<Java Package>>
org.unicase.decision.annotations
<<Java Package>>
org.unicase.decision.annotations.hover
<<Java Package>>
org.unicase.decision.annotations.dialogs
<<Java Package>>
org.unicase.decision.annotations.util
<<Java Package>>
org.unicase.decision.annotations.model.commands

Figure A.1: Package diagram for the Annotations plug-in

org.unicase.decision.annotations: This package contains classes responsible for controlling the Annotation plug-in's life cycle.

org.unicase.decision.annotations.builders: This package contains the builder and nature classes.

org.unicase.decision.annotations.model: This package contains the annotation manager as well as the core and augmented annotations.

org.unicase.decision.annotations.model.commands: This package contains the augmented annotation commands.

org.unicase.decision.annotations.dialogs: This package contains all dialogs used in the Annotations plug-in.

org.unicase.decision.annotations.hover: This package contains the text hover and hover controls.

org.unicase.decision.annotations.providers: This package contains content and label providers for the dialogs.

org.unicase.decision.annotations.views: This package contains the views used in the Annotations plug-in.

org.unicase.decision.annotations.util: This package contains various utility classes.

B Class Diagrams

B.1 Class Diagram for org.unicase.decision.annotations

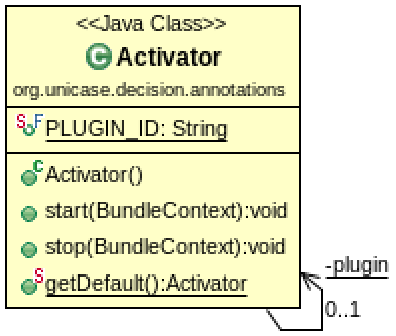
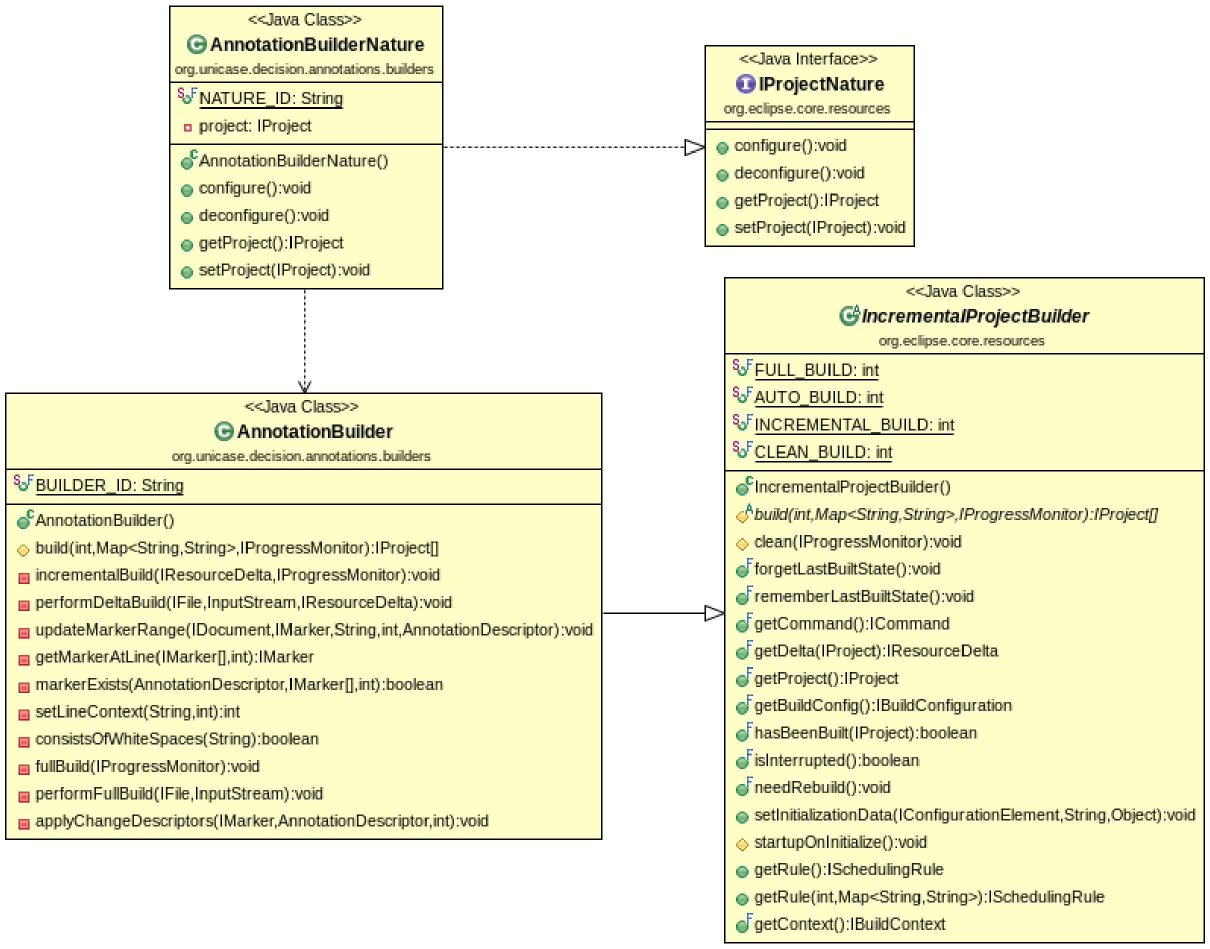


Figure B.1: Class diagram for the *org.unicase.decision.annotations* package

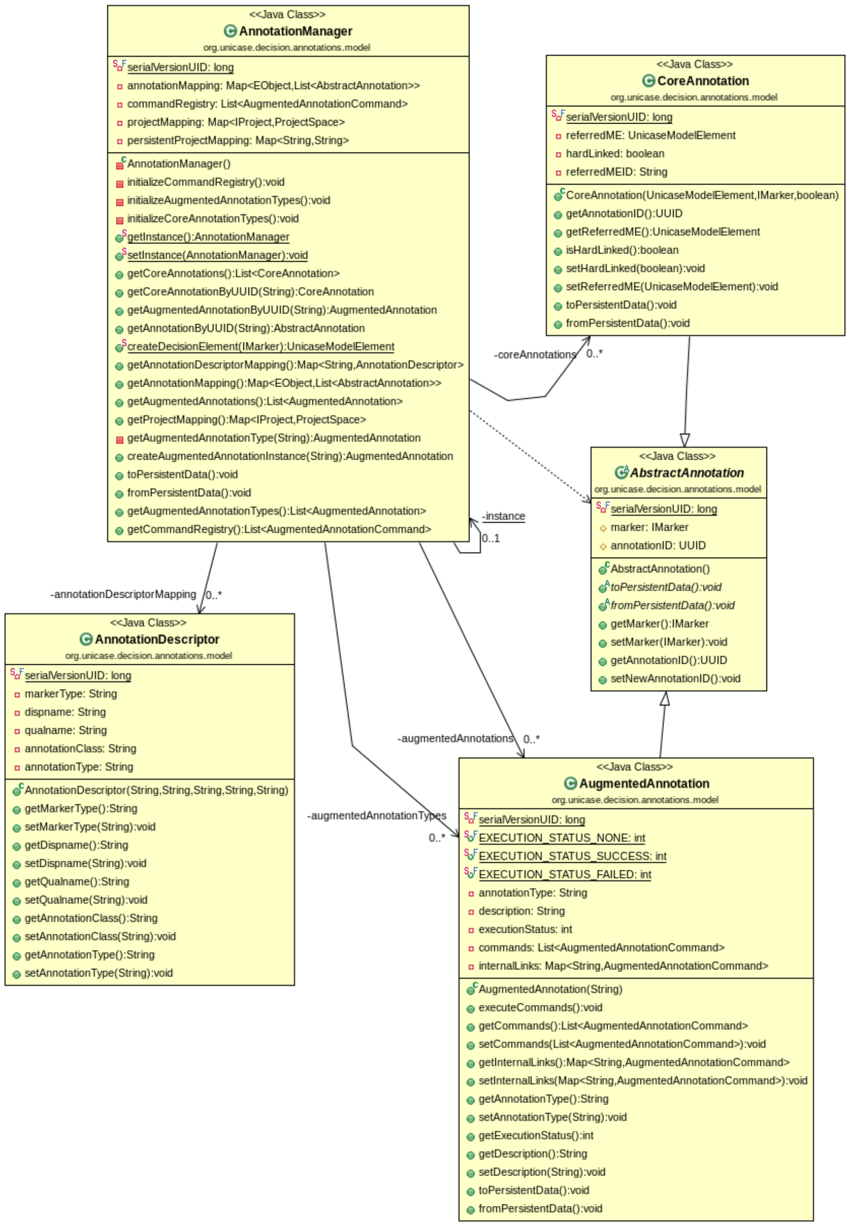
B.2 Class Diagram for org.unicase.decision.annotations.builders



<<Java Class>>
AnnotationBuilderNature
org.unicase.decision.annotations.builders
NATURE_ID: String
project: IProject
AnnotationBuilderNature()
configure():void
deconfigure():void
getProject():IProject
setProject(IProject):void
<<Java Interface>>
IProjectNature
org.eclipse.core.resources
configure():void
deconfigure():void
getProject():IProject
setProject(IProject):void
<<Java Class>>
IncrementalProjectBuilder
org.eclipse.core.resources
FULL_BUILD: int
AUTO_BUILD: int
INCREMENTAL_BUILD: int
CLEAN_BUILD: int
IncrementalProjectBuilder()
build(int,Map<String,String>,IProgressMonitor):IProject[]
clean(IProgressMonitor):void
forgetLastBuiltState():void
rememberLastBuiltState():void
getCommand():ICommand
getDelta(IProject):IResourceDelta
getProject():IProject
getBuildConfig():IBuildConfiguration
hasBeenBuilt(IProject):boolean
isInterrupted():boolean
needRebuild():void
setInitializationData(IConfigurationElement,String,Object):void
startupOnInitialize():void
getRule():ISchedulingRule
getRule(int,Map<String,String>):ISchedulingRule
getContext():IBuildContext
<<Java Class>>
AnnotationBuilder
org.unicase.decision.annotations.builders
BUILDER_ID: String
AnnotationBuilder()
build(int,Map<String,String>,IProgressMonitor):IProject[]
incrementalBuild(IResourceDelta,IProgressMonitor):void
performDeltaBuild(IFile,InputStream,IResourceDelta):void
updateMarkerRange(IDocument,IMarker,String,int,AnnotationDescriptor):void
getMarkerAtLine(IMarker[],int):IMarker
markerExists(AnnotationDescriptor,IMarker[],int):boolean
setLineContext(String,int):int
consistsOfWhiteSpaces(String):boolean
fullBuild(IProgressMonitor):void
performFullBuild(IFile,InputStream):void
applyChangeDescriptors(IMarker,AnnotationDescriptor,int):void

Figure B.2: Class diagram for the *org.unicase.decision.annotations.builders* package

B.3 Class Diagram for org.unicase.decision.annotations.model



<<Java Class>>
AnnotationManager
org.unicase.decision.annotations.model
serialVersionUID: long
annotationMapping: Map<EObject,List<AbstractAnnotation>>
commandRegistry: List<AugmentedAnnotationCommand>
projectMapping: Map<IProject,ProjectSpace>
persistentProjectMapping: Map<String,String>
AnnotationManager()
initializeCommandRegistry():void
initializeAugmentedAnnotationTypes():void
initializeCoreAnnotationTypes():void
getInstance():AnnotationManager
setInstance(AnnotationManager):void
getCoreAnnotations():List<CoreAnnotation>
getCoreAnnotationByUUID(String):CoreAnnotation
getAugmentedAnnotationByUUID(String):AugmentedAnnotation
getAnnotationByUUID(String):AbstractAnnotation
createDecisionElement(IMarker):UnicaseModelElement
getAnnotationDescriptorMapping():Map<String,AnnotationDescriptor>
getAnnotationMapping():Map<EObject,List<AbstractAnnotation>>
getAugmentedAnnotations():List<AugmentedAnnotation>
getProjectMapping():Map<IProject,ProjectSpace>
getAugmentedAnnotationType(String):AugmentedAnnotation
createAugmentedAnnotationInstance(String):AugmentedAnnotation
toPersistentData():void
fromPersistentData():void
getAugmentedAnnotationTypes():List<AugmentedAnnotation>
getCommandRegistry():List<AugmentedAnnotationCommand>
<<Java Class>>
CoreAnnotation
org.unicase.decision.annotations.model
serialVersionUID: long
referredME: UnicaseModelElement
hardLinked: boolean
referredMEID: String
CoreAnnotation(UnicaseModelElement,IMarker,boolean)
getAnnotationID():UUID
getReferredME():UnicaseModelElement
isHardLinked():boolean
setHardLinked(boolean):void
setReferredME(UnicaseModelElement):void
toPersistentData():void
fromPersistentData():void
-coreAnnotations
0..*
<<Java Class>>
AbstractAnnotation
org.unicase.decision.annotations.model
serialVersionUID: long
marker: IMarker
annotationID: UUID
AbstractAnnotation()
toPersistentData():void
fromPersistentData():void
getMarker():IMarker
setMarker(IMarker):void
getAnnotationID():UUID
setNewAnnotationID():void
-instance
0..1
-annotationDescriptorMapping
0..*
<<Java Class>>
AnnotationDescriptor
org.unicase.decision.annotations.model
serialVersionUID: long
markerType: String
dispname: String
qualname: String
annotationClass: String
annotationType: String
AnnotationDescriptor(String,String,String,String,String)
getMarkerType():String
setMarkerType(String):void
getDispname():String
setDispname(String):void
getQualname():String
setQualname(String):void
getAnnotationClass():String
setAnnotationClass(String):void
getAnnotationType():String
setAnnotationType(String):void
-augmentedAnnotations
0..*
-augmentedAnnotationTypes
0..*
<<Java Class>>
AugmentedAnnotation
org.unicase.decision.annotations.model
serialVersionUID: long
EXECUTION_STATUS_NONE: int
EXECUTION_STATUS_SUCCESS: int
EXECUTION_STATUS_FAILED: int
annotationType: String
description: String
executionStatus: int
commands: List<AugmentedAnnotationCommand>
internalLinks: Map<String,AugmentedAnnotationCommand>
AugmentedAnnotation(String)
executeCommands():void
getCommands():List<AugmentedAnnotationCommand>
setCommands(List<AugmentedAnnotationCommand>):void
getInternalLinks():Map<String,AugmentedAnnotationCommand>
setInternalLinks(Map<String,AugmentedAnnotationCommand>):void
getAnnotationType():String
setAnnotationType(String):void
getExecutionStatus():int
getDescription():String
setDescription(String):void
toPersistentData():void
fromPersistentData():void

Figure B.3: Class diagram for the *org.unicase.decision.annotations.model* package

B.4 Class Diagram for org.unicase.decision.annotations.model.commands

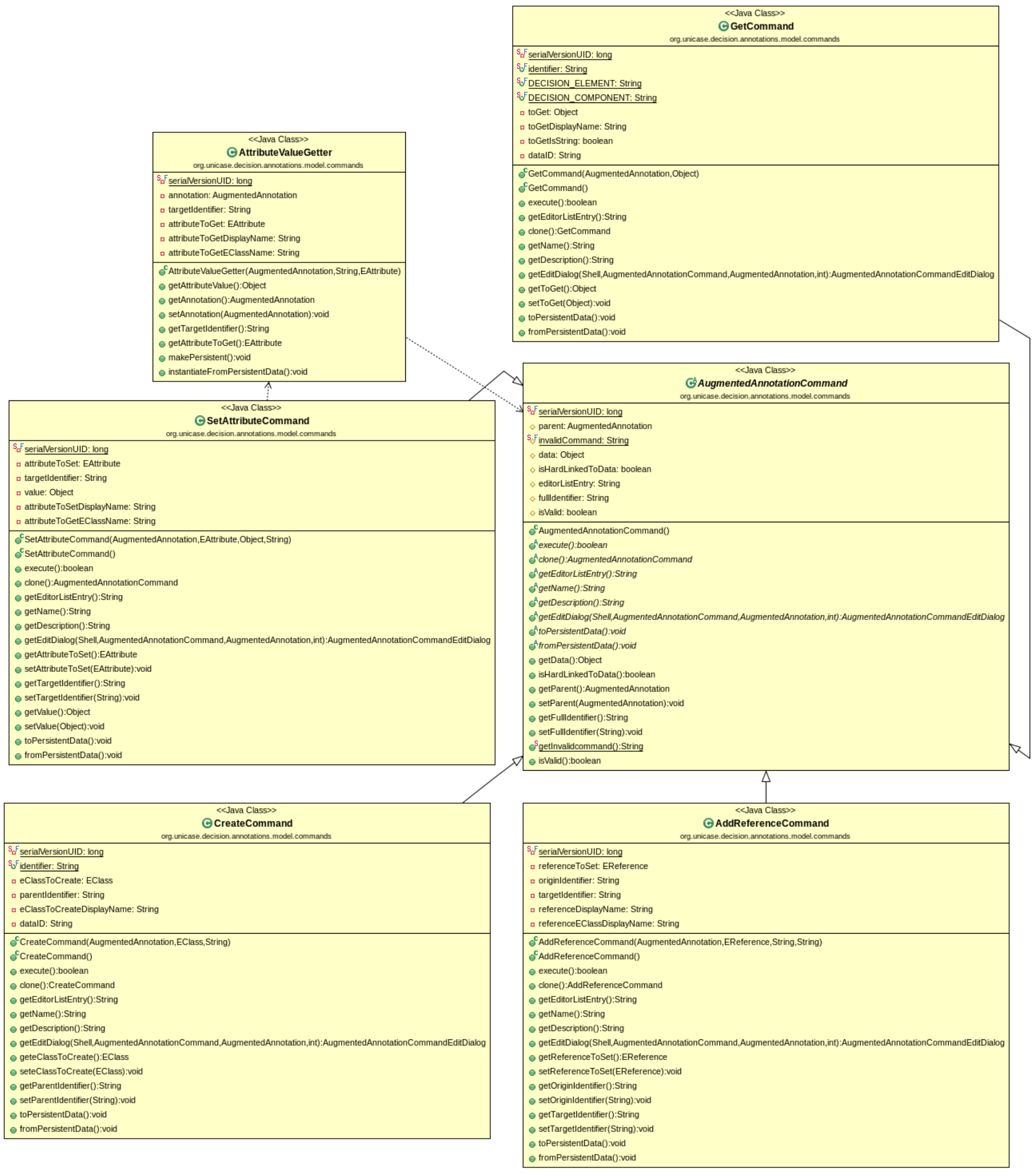
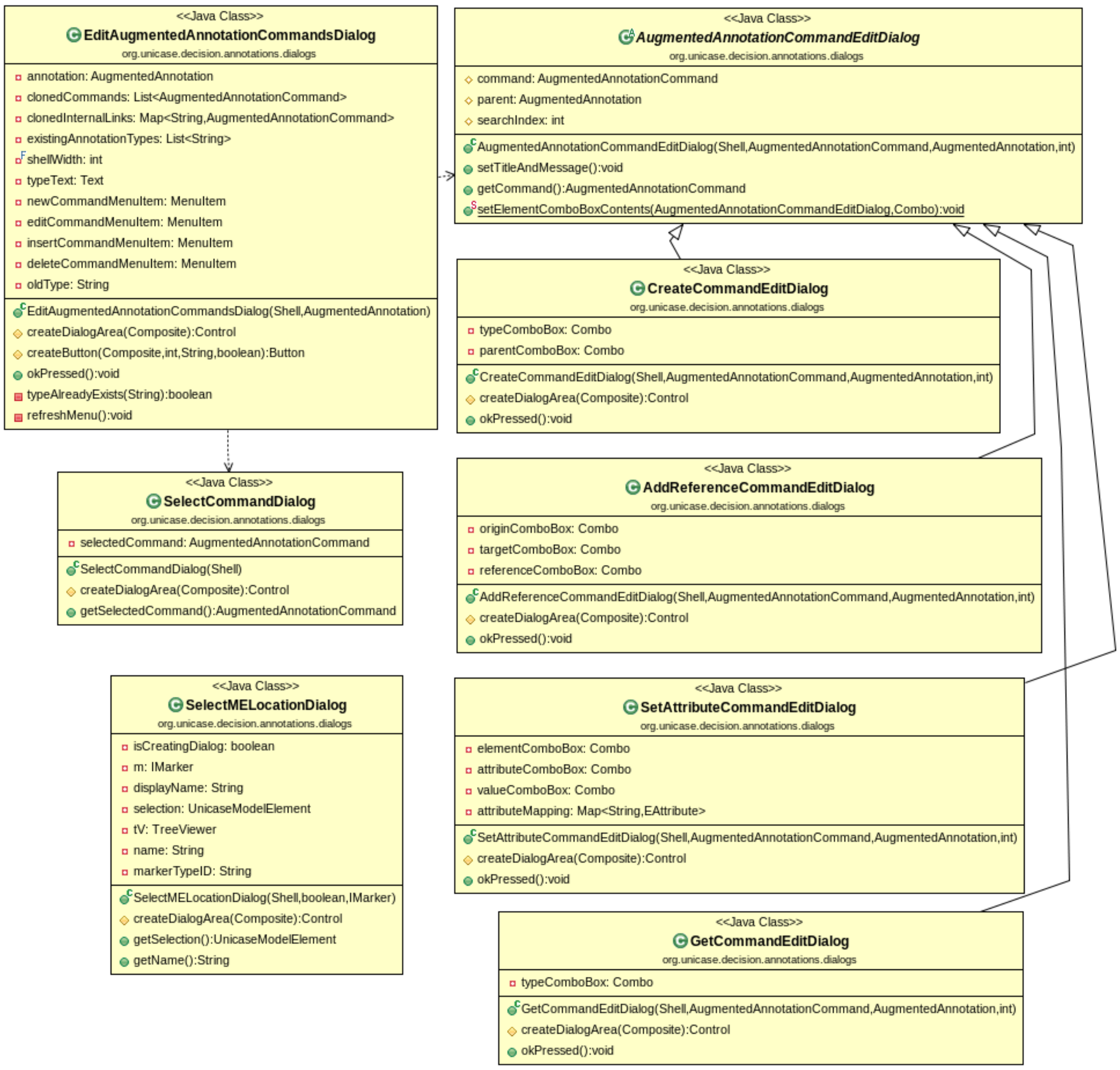

<<Java Class>>
GetCommand
org.unicase.decision.annotations.model.commands
serialVersionUID: long
identifier: String
DECISION_ELEMENT: String
DECISION_COMPONENT: String
toGet: Object
toGetDisplayName: String
toGetIsString: boolean
dataID: String
GetCommand(AugmentedAnnotation,Object)
GetCommand()
execute():boolean
getEditorListEntry():String
clone():GetCommand
getName():String
getDescription():String
getEditDialog(Shell,AugmentedAnnotationCommand,AugmentedAnnotation,int):AugmentedAnnotationCommandEditDialog
getToGet():Object
setToGet(Object):void
toPersistentData():void
fromPersistentData():void
<<Java Class>>
AttributeValueGetter
org.unicase.decision.annotations.model.commands
serialVersionUID: long
annotation: AugmentedAnnotation
targetIdentifier: String
attributeToGet: EAttribute
attributeToGetDisplayName: String
attributeToGetEClassName: String
AttributeValueGetter(AugmentedAnnotation,String,EAttribute)
getAttributeValue():Object
getAnnotation():AugmentedAnnotation
setAnnotation(AugmentedAnnotation):void
getTargetIdentifier():String
getAttributeToGet():EAttribute
makePersistent():void
instantiateFromPersistentData():void
<<Java Class>>
AugmentedAnnotationCommand
org.unicase.decision.annotations.model.commands
serialVersionUID: long
parent: AugmentedAnnotation
invalidCommand: String
data: Object
isHardLinkedToData: boolean
editorListEntry: String
fullIdentifier: String
isValid: boolean
AugmentedAnnotationCommand()
execute():boolean
clone():AugmentedAnnotationCommand
getEditorListEntry():String
getName():String
getDescription():String
getEditDialog(Shell,AugmentedAnnotationCommand,AugmentedAnnotation,int):AugmentedAnnotationCommandEditDialog
toPersistentData():void
fromPersistentData():void
getData():Object
isHardLinkedToData():boolean
getParent():AugmentedAnnotation
setParent(AugmentedAnnotation):void
getFullIdentifier():String
setFullIdentifier(String):void
getInvalidcommand():String
isValid():boolean
<<Java Class>>
SetAttributeCommand
org.unicase.decision.annotations.model.commands
serialVersionUID: long
attributeToSet: EAttribute
targetIdentifier: String
value: Object
attributeToSetDisplayName: String
attributeToGetEClassName: String
SetAttributeCommand(AugmentedAnnotation,EAttribute,Object,String)
SetAttributeCommand()
execute():boolean
clone():AugmentedAnnotationCommand
getEditorListEntry():String
getName():String
getDescription():String
getEditDialog(Shell,AugmentedAnnotationCommand,AugmentedAnnotation,int):AugmentedAnnotationCommandEditDialog
getAttributeToSet():EAttribute
setAttributeToSet(EAttribute):void
getTargetIdentifier():String
setTargetIdentifier(String):void
getValue():Object
setValue(Object):void
toPersistentData():void
fromPersistentData():void
<<Java Class>>
CreateCommand
org.unicase.decision.annotations.model.commands
serialVersionUID: long
identifier: String
eClassToCreate: EClass
parentIdentifier: String
eClassToCreateDisplayName: String
dataID: String
CreateCommand(AugmentedAnnotation,EClass,String)
CreateCommand()
execute():boolean
clone():CreateCommand
getEditorListEntry():String
getName():String
getDescription():String
getEditDialog(Shell,AugmentedAnnotationCommand,AugmentedAnnotation,int):AugmentedAnnotationCommandEditDialog
geteClassToCreate():EClass
seteClassToCreate(EClass):void
getParentIdentifier():String
setParentIdentifier(String):void
toPersistentData():void
fromPersistentData():void
<<Java Class>>
AddReferenceCommand
org.unicase.decision.annotations.model.commands
serialVersionUID: long
referenceToSet: EReference
originIdentifier: String
targetIdentifier: String
referenceDisplayName: String
referenceEClassDisplayName: String
AddReferenceCommand(AugmentedAnnotation,EReference,String,String)
AddReferenceCommand()
execute():boolean
clone():AddReferenceCommand
getEditorListEntry():String
getName():String
getDescription():String
getEditDialog(Shell,AugmentedAnnotationCommand,AugmentedAnnotation,int):AugmentedAnnotationCommandEditDialog
getReferenceToSet():EReference
setReferenceToSet(EReference):void
getOriginIdentifier():String
setOriginIdentifier(String):void
getTargetIdentifier():String
setTargetIdentifier(String):void
toPersistentData():void
fromPersistentData():void

Figure B.4: Class diagram for the *org.unicase.decision.annotations.model.commands* package

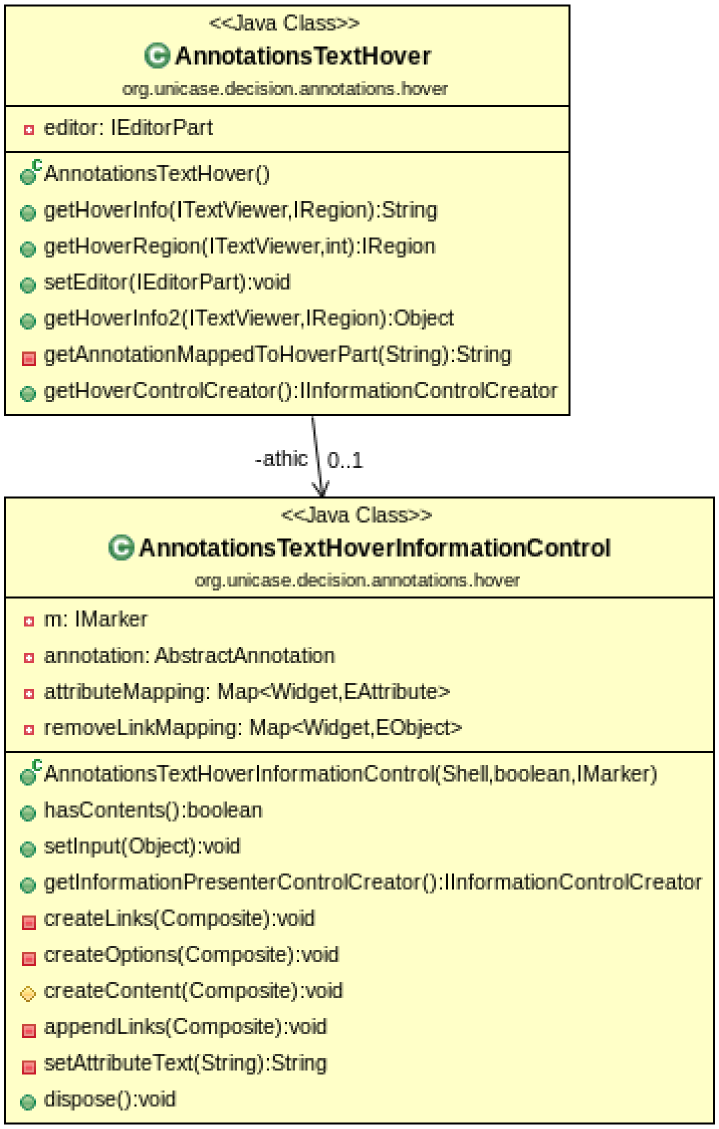
B.5 Class Diagram for org.unicase.decision.annotations.dialogs



<<Java Class>>
EditAugmentedAnnotationCommandsDialog
org.unicase.decision.annotations.dialogs
annotation: AugmentedAnnotation
clonedCommands: List<AugmentedAnnotationCommand>
clonedInternalLinks: Map<String,AugmentedAnnotationCommand>
existingAnnotationTypes: List<String>
shellWidth: int
typeText: Text
newCommandMenuItem: MenuItem
editCommandMenuItem: MenuItem
insertCommandMenuItem: MenuItem
deleteCommandMenuItem: MenuItem
oldType: String
EditAugmentedAnnotationCommandsDialog(Shell,AugmentedAnnotation)
createDialogArea(Composite):Control
createButton(Composite,int,String,boolean):Button
okPressed():void
typeAlreadyExists(String):boolean
refreshMenu():void
<<Java Class>>
AugmentedAnnotationCommandEditDialog
org.unicase.decision.annotations.dialogs
command: AugmentedAnnotationCommand
parent: AugmentedAnnotation
searchIndex: int
AugmentedAnnotationCommandEditDialog(Shell,AugmentedAnnotationCommand,AugmentedAnnotation,int)
setTitleAndMessage():void
getCommand():AugmentedAnnotationCommand
setElementComboBoxContents(AugmentedAnnotationCommandEditDialog,Combo):void
<<Java Class>>
CreateCommandEditDialog
org.unicase.decision.annotations.dialogs
typeComboBox: Combo
parentComboBox: Combo
CreateCommandEditDialog(Shell,AugmentedAnnotationCommand,AugmentedAnnotation,int)
createDialogArea(Composite):Control
okPressed():void
<<Java Class>>
SelectCommandDialog
org.unicase.decision.annotations.dialogs
selectedCommand: AugmentedAnnotationCommand
SelectCommandDialog(Shell)
createDialogArea(Composite):Control
getSelectedCommand():AugmentedAnnotationCommand
<<Java Class>>
AddReferenceCommandEditDialog
org.unicase.decision.annotations.dialogs
originComboBox: Combo
targetComboBox: Combo
referenceComboBox: Combo
AddReferenceCommandEditDialog(Shell,AugmentedAnnotationCommand,AugmentedAnnotation,int)
createDialogArea(Composite):Control
okPressed():void
<<Java Class>>
SelectMELocationDialog
org.unicase.decision.annotations.dialogs
isCreatingDialog: boolean
m: IMarker
displayName: String
selection: UnicaseModelElement
tV: TreeViewer
name: String
markerTypeID: String
SelectMELocationDialog(Shell,boolean,IMarker)
createDialogArea(Composite):Control
getSelection():UnicaseModelElement
getName():String
<<Java Class>>
SetAttributeCommandEditDialog
org.unicase.decision.annotations.dialogs
elementComboBox: Combo
attributeComboBox: Combo
valueComboBox: Combo
attributeMapping: Map<String,EAttribute>
SetAttributeCommandEditDialog(Shell,AugmentedAnnotationCommand,AugmentedAnnotation,int)
createDialogArea(Composite):Control
okPressed():void
<<Java Class>>
GetCommandEditDialog
org.unicase.decision.annotations.dialogs
typeComboBox: Combo
GetCommandEditDialog(Shell,AugmentedAnnotationCommand,AugmentedAnnotation,int)
createDialogArea(Composite):Control
okPressed():void

Figure B.5: Class diagram for the *org.unicase.decision.annotations.dialogs* package

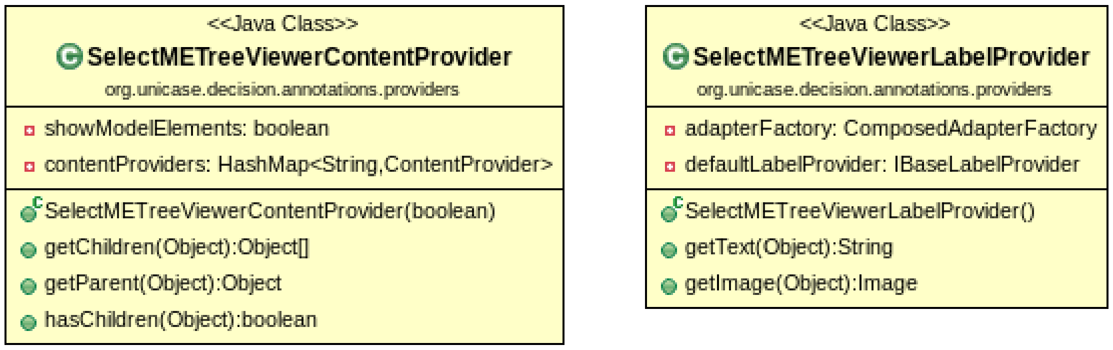
B.6 Class Diagram for org.unicase.decision.annotations.hover



<<Java Class>>
AnnotationsTextHover
org.unicase.decision.annotations.hover
editor: IEditorPart
AnnotationsTextHover()
getHoverInfo(ITextViewer,IRegion):String
getHoverRegion(ITextViewer,int):IRegion
setEditor(IEditorPart):void
getHoverInfo2(ITextViewer,IRegion):Object
getAnnotationMappedToHoverPart(String):String
getHoverControlCreator():IInformationControlCreator
-athic
0..1
<<Java Class>>
AnnotationsTextHoverInformationControl
org.unicase.decision.annotations.hover
m: IMarker
annotation: AbstractAnnotation
attributeMapping: Map<Widget,EAttribute>
removeLinkMapping: Map<Widget,EObject>
AnnotationsTextHoverInformationControl(Shell,boolean,IMarker)
hasContents():boolean
setInput(Object):void
getInformationPresenterControlCreator():IInformationControlCreator
createLinks(Composite):void
createOptions(Composite):void
createContent(Composite):void
appendLinks(Composite):void
setAttributeText(String):String
dispose():void

Figure B.6: Class diagram for the *org.unicase.decision.hover* package

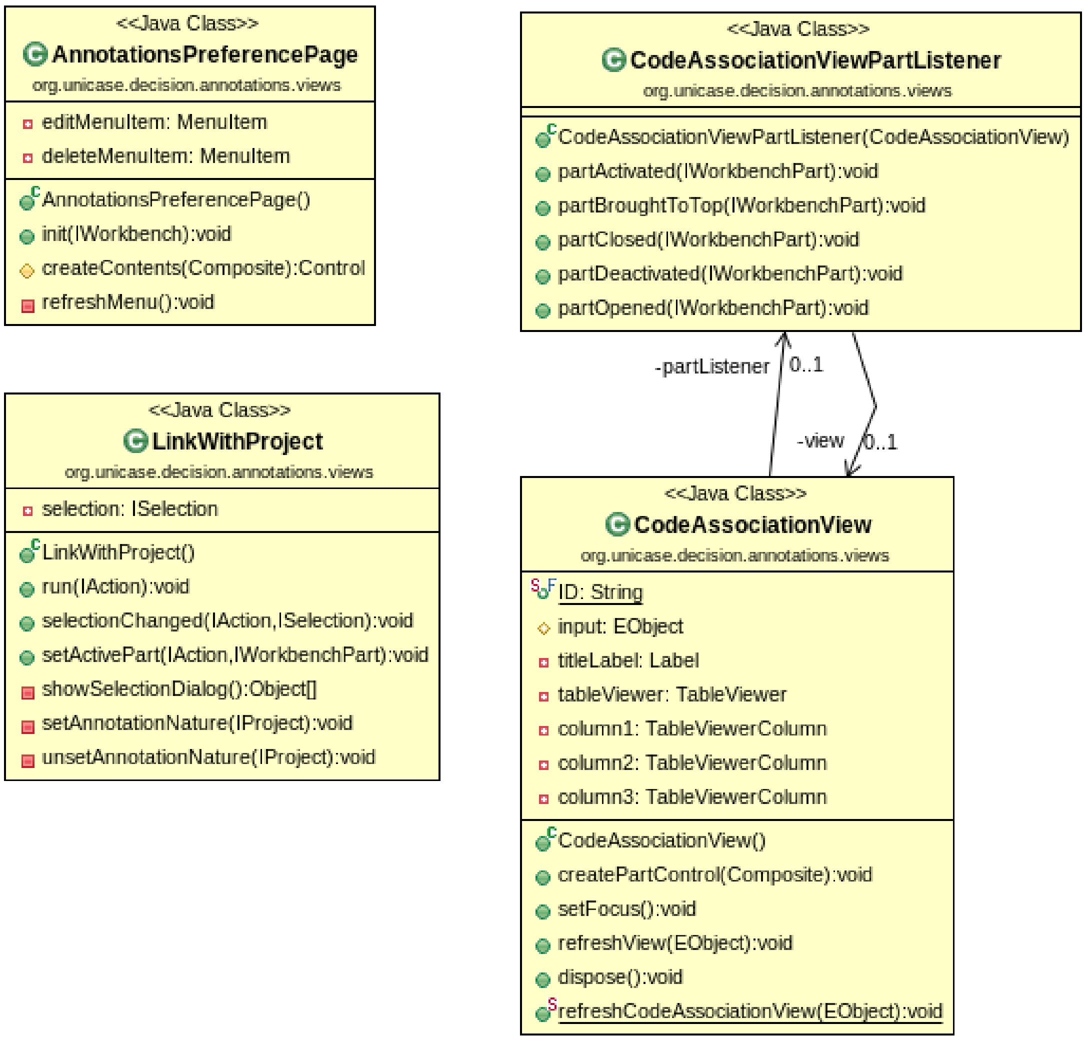
B.7 Class Diagram for org.unicase.decision.annotations.providers



<<Java Class>>
SelectMETreeViewerContentProvider
org.unicase.decision.annotations.providers
showModelElements: boolean
contentProviders: HashMap<String,ContentProvider>
SelectMETreeViewerContentProvider(boolean)
getChildren(Object):Object[]
getParent(Object):Object
hasChildren(Object):boolean
<<Java Class>>
SelectMETreeViewerLabelProvider
org.unicase.decision.annotations.providers
adapterFactory: ComposedAdapterFactory
defaultLabelProvider: IBaseLabelProvider
SelectMETreeViewerLabelProvider()
getText(Object):String
getImage(Object):Image

Figure B.7: Class diagram for the *org.unicase.decision.annotations.providers* package

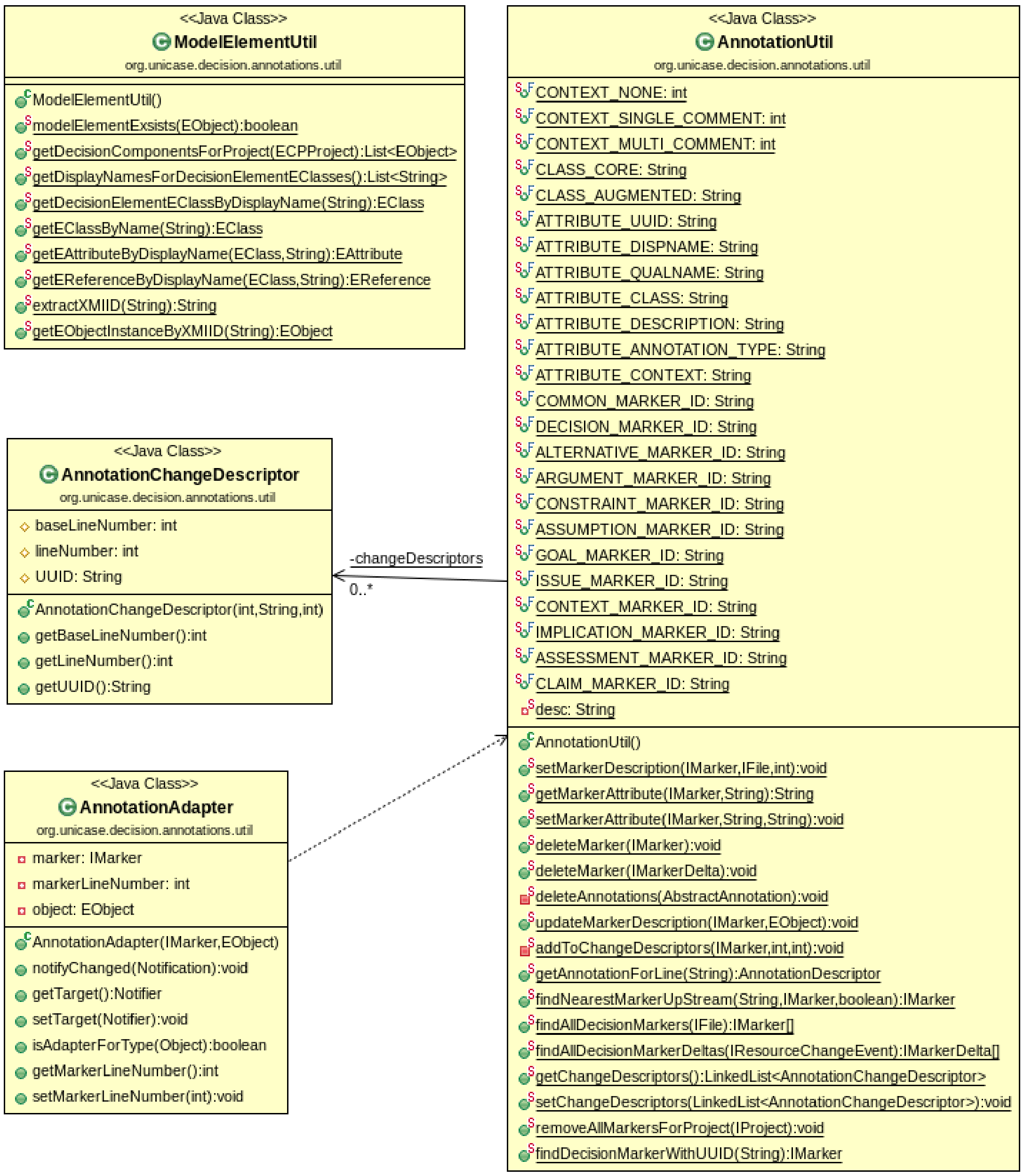
B.8 Class Diagram for org.unicase.decision.annotations.views



<<Java Class>>
AnnotationsPreferencePage
org.unicase.decision.annotations.views
editMenuItem: MenuItem
deleteMenuItem: MenuItem
AnnotationsPreferencePage()
init(IWorkbench):void
createContents(Composite):Control
refreshMenu():void
<<Java Class>>
LinkWithProject
org.unicase.decision.annotations.views
selection: ISelection
LinkWithProject()
run(IAction):void
selectionChanged(IAction,ISelection):void
setActivePart(IAction,IWorkbenchPart):void
showSelectionDialog():Object[]
setAnnotationNature(IProject):void
unsetAnnotationNature(IProject):void
<<Java Class>>
CodeAssociationViewPartListener
org.unicase.decision.annotations.views
CodeAssociationViewPartListener(CodeAssociationView)
partActivated(IWorkbenchPart):void
partBroughtToTop(IWorkbenchPart):void
partClosed(IWorkbenchPart):void
partDeactivated(IWorkbenchPart):void
partOpened(IWorkbenchPart):void
-partListener
0..1
-view
0..1
<<Java Class>>
CodeAssociationView
org.unicase.decision.annotations.views
ID: String
input: EObject
titleLabel: Label
tableViewer: TableViewer
column1: TableViewerColumn
column2: TableViewerColumn
column3: TableViewerColumn
CodeAssociationView()
createPartControl(Composite):void
setFocus():void
refreshView(EObject):void
dispose():void
refreshCodeAssociationView(EObject):void

Figure B.8: Class diagram for the *org.unicase.decision.annotations.views* package

B.9 Class Diagram for org.unicase.decision.annotations.util



<<Java Class>>
ModelElementUtil
org.unicase.decision.annotations.util
ModelElementUtil()
modelElementExsists(EObject):boolean
getDecisionComponentsForProject(ECPProject):List<EObject>
getDisplayNamesForDecisionElementEClasses():List<String>
getDecisionElementEClassByDisplayName(String):EClass
getEClassByName(String):EClass
getEAttributeByDisplayName(EClass,String):EAttribute
getEReferenceByDisplayName(EClass,String):EReference
extractXMIID(String):String
getEObjectInstanceByXMIID(String):EObject
<<Java Class>>
AnnotationUtil
org.unicase.decision.annotations.util
CONTEXT_NONE: int
CONTEXT_SINGLE_COMMENT: int
CONTEXT_MULTI_COMMENT: int
CLASS_CORE: String
CLASS_AUGMENTED: String
ATTRIBUTE_UUID: String
ATTRIBUTE_DISPNAME: String
ATTRIBUTE_QUALNAME: String
ATTRIBUTE_CLASS: String
ATTRIBUTE_DESCRIPTION: String
ATTRIBUTE_ANNOTATION_TYPE: String
ATTRIBUTE_CONTEXT: String
COMMON_MARKER_ID: String
DECISION_MARKER_ID: String
ALTERNATIVE_MARKER_ID: String
ARGUMENT_MARKER_ID: String
CONSTRAINT_MARKER_ID: String
ASSUMPTION_MARKER_ID: String
GOAL_MARKER_ID: String
ISSUE_MARKER_ID: String
CONTEXT_MARKER_ID: String
IMPLICATION_MARKER_ID: String
ASSESSMENT_MARKER_ID: String
CLAIM_MARKER_ID: String
desc: String
AnnotationUtil()
setMarkerDescription(IMarker,IFile,int):void
getMarkerAttribute(IMarker,String):String
setMarkerAttribute(IMarker,String,String):void
deleteMarker(IMarker):void
deleteMarker(IMarkerDelta):void
deleteAnnotations(AbstractAnnotation):void
updateMarkerDescription(IMarker,EObject):void
addToChangeDescriptors(IMarker,int,int):void
getAnnotationForLine(String):AnnotationDescriptor
findNearestMarkerUpStream(String,IMarker,boolean):IMarker
findAllDecisionMarkers(IFile):IMarker[]
findAllDecisionMarkerDeltas(IResourceChangeEvent):IMarkerDelta[]
getChangeDescriptors():LinkedList<AnnotationChangeDescriptor>
setChangeDescriptors(LinkedList<AnnotationChangeDescriptor>):void
removeAllMarkersForProject(IProject):void
findDecisionMarkerWithUUID(String):IMarker
<<Java Class>>
AnnotationChangeDescriptor
org.unicase.decision.annotations.util
baseLineNumber: int
lineNumber: int
UUID: String
AnnotationChangeDescriptor(int,String,int)
getBaseLineNumber():int
getLineNumber():int
getUUID():String
-changeDescriptors
0..*
<<Java Class>>
AnnotationAdapter
org.unicase.decision.annotations.util
marker: IMarker
markerLineNumber: int
object: EObject
AnnotationAdapter(IMarker,EObject)
notifyChanged(Notification):void
getTarget():Notifier
setTarget(Notifier):void
isAdapterForType(Object):boolean
getMarkerLineNumber():int
setMarkerLineNumber(int):void

Figure B.9: Class diagram for the *org.unicase.decision.annotations.util* package