

# JTube: Lightweight Support for Structuring and Documenting Java Source Code

Patrick Nepper

December 5, 2005

## Abstract

Structure and documentation are two major factors for successful software development. Retaining control over them is difficult, because there are various forms of structural information and documentation. Java source code is still full of neglected sources of such information. Whether it is relations between methods of the same class or dependencies between source code and external documentation, we lack lightweight tools to extract and visualize the respective data. In this paper we present *JTube*, a tool that assists the developer with making use of this hidden information. *JTube* is designed to be easily usable for Java developers: it comes as an Eclipse plugin and makes use of common practices in writing Java source code, namely grouping methods, using submethods and referencing external documentation.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	The Structure of Software . . . . .	4
1.2	Internal and External Documentation . . . . .	5
1.3	Our Contribution . . . . .	5
<b>2</b>	<b>Background</b>	<b>6</b>
2.1	Problems of Structuring and Documenting Code . . . . .	6
2.2	An Agile Approach . . . . .	7
2.2.1	Eclipse and Java . . . . .	7
2.2.2	Java Comments for Structuring . . . . .	7
2.2.3	RDF for Documentation . . . . .	7
<b>3</b>	<b>JTube</b>	<b>8</b>
3.1	Overview . . . . .	8
3.2	Features . . . . .	8
3.2.1	Grouping Methods . . . . .	8
3.2.2	Recognizing Submethods . . . . .	10
3.2.3	Adding Meta Information . . . . .	11
3.3	Architecture . . . . .	12
3.3.1	Components . . . . .	12
3.3.2	Eclipse Integration . . . . .	13
3.4	Synchronization . . . . .	13
<b>4</b>	<b>Use Cases</b>	<b>14</b>
4.1	Grouping Methods . . . . .	14
4.1.1	Situation . . . . .	14
4.1.2	Problem . . . . .	14
4.1.3	Solution . . . . .	16
4.2	Submethods . . . . .	16
4.2.1	Situation . . . . .	16
4.2.2	Problem . . . . .	16
4.2.3	Solution . . . . .	17
4.3	External Documentation . . . . .	17
4.3.1	Situation . . . . .	17
4.3.2	Problem . . . . .	18
4.3.3	Solution . . . . .	18
<b>5</b>	<b>Related Work</b>	<b>19</b>
5.1	Javadoc . . . . .	19
5.1.1	Current Standard 1.5 . . . . .	19
5.1.2	Proposed Tags . . . . .	19
5.2	Eclipse . . . . .	20
5.2.1	Eclipse Task Tags . . . . .	20
5.2.2	Eclipse Markers . . . . .	20

<b>6</b>	<b>Current and Future Research</b>	<b>21</b>
6.1	Active Knowledge . . . . .	21
6.1.1	Idea . . . . .	21
6.1.2	Implementation . . . . .	21
6.1.3	User Interface . . . . .	22
6.2	Concept Mining . . . . .	22
6.2.1	Idea . . . . .	22
6.2.2	Implementation . . . . .	23
6.2.3	User Interface . . . . .	23
6.3	Web Integration . . . . .	24
<b>7</b>	<b>Conclusion</b>	<b>24</b>

# 1 Introduction

In every software development project, complexity is the major threat to on-time, on-budget and on-purpose delivery. Software development processes and methodologies try to tackle the problem of complexity by *structuring* the software into units that are easy to handle and by *documenting* all relevant information in the course of the development process.

## 1.1 The Structure of Software

The human mind is only capable of grasping a limited amount of information at the same time. Without *structure*, we are overwhelmed by the enormous amount of code that is needed to build software and therefore it becomes hard to understand how the software works. Furthermore, without structure it is inherently difficult to manage software, because managing software (e.g. its development and quality) just like managing anything else depends on the existence of artifacts that have a clearly defined scope and properties that can be evaluated. Nowadays, structuring software is a more or less standardized process in professional software development: In Java, structuring software into packages and structuring packages into classes helps developers understand and manage software on a macro level. But this is not sufficient. The need for describing the collaborations between classes (e.g. in the context of collaboration-based designs) on a more granular level than the class-level is hampered by a mere hierarchical and class-centric approach: In modern programming languages, the “*tyranny of the dominant decomposition*”—as Tarr et al. have coined it [15]—allows only for a one-dimensional (hierarchical) decomposition of a system. In Java, the dominant dimension is the *class*. Especially when it comes to the responsibilities of classes and their collaborations with other classes, a limitation to *classes* as the sole artifact can obscure the relationship between methods of different classes or even within the same class. Developments such as Hyper/J [10] or the Mixin Layers [14] try to tackle this problem by introducing new language constructs. Until such developments become integrated into standard development processes, developers are faced with some obstacles if it comes to substructuring classes into smaller units, because there are no established tools for helping the developer understand the internal structure of a class. The smallest structural units of software design are methods, but their relations are usually not displayed: In Eclipse, for example, the outline view displays all the methods contained in a class, but it is not possible to get a fast overview of the relations between these methods; if the user wants to analyze the relation between different methods, he can use Eclipse’s call hierarchy feature to see all methods and/or constructors that call the current method. As this relation is rather complex to calculate (methods often can be called from any class within the workspace), Eclipse can only offer this information on-demand and for one method at a time.

Structuring classes takes places on different levels. Part of the structure of a class is defined by the interrelation of methods contained in the class, for example the relation between methods that are only helper methods for other methods or the dependencies of methods that override or implement methods from super classes. This kind of structural information must be extracted by analyzing the dependencies between the class’s methods. On the other hand, the

structure of a class is also defined by the structural information the developer applies to the class by introducing lines of comments into the source code in order to separate groups of methods or by adding comments that explicitly relate a method to another method, a term, etc. Classes often fulfill different responsibilities, therefore developers often try to categorize methods according to these responsibilities by adjusting their order and using Java comments.

**Example.** In a recent project for a large German bank we used Java comments to categorize methods based on whether anticipated changes in the semantics of the input data would make a future code review necessary or not. It would then be easier for us to get back to all the code segments that need refurbishment, for the case the anticipated request for change (based on a change in the input data) comes in.

## 1.2 Internal and External Documentation

Besides structuring code, *documentation* is another powerful tool to handle complexity. It makes sure that the application logic can be explained in a human readable format. This is essential for developers that have to understand unknown source code and helps to map functionalities described as project deliverables to specific parts of the implementation. Although documentation does not reduce the complexity of software it allows us to get an insight into parts of the implementation without having to read or understand the complete source code. Therefore, it makes understanding, maintaining and collaboratively creating source code more efficient.

We distinguish between *external* documentation (e.g. PDF documents) and *inline* documentation (e.g. Javadoc). Inline documentation often references external documentation, but the dependencies (essentially the links) between inline and external documentation are often not easily manageable.

**Example (continued).** In the case of our bank project, we did not only categorize methods, but we also referenced external documentation. The Javadoc comments used for the references were mainly targeted at future developers who would have to do some maintenance on the code, whereas most of the application logic was explained in detail in external documents. As the bank already had very clearly defined requirements for their documentation process (they had their own adaptation of RUP in place), we had to stick to certain templates and the Word document format.

## 1.3 Our Contribution

In terms of structure and documentation in the above project we had to face two major drawbacks: Although every author had a clear understanding of how his code segments were structured, the information hidden in the source code and the Java comments was not easily accessible from the outside and acquiring an overview of a class's internal structure would have required us to check all the methods and Java comments in the class. Furthermore, the separation of inline (Javadoc) documentation and external documentation has further increased the complexity rather than reducing it. Keeping track of which external document has something to do with which lines of code soon went beyond an efficient level.

We believe that complex tools that impose additional processes and activities on the developer are counter-productive for reducing the complexity of

software systems. Software development tools need to be designed to make the developer's work more efficient by providing support and information at his fingertip. Our goal is to contribute to the course of agile software development by providing the developer with lightweight tools to support structuring and documenting Java source code. In this respect, agility for us means to adapt to the developer's need for easy-to-use tools that are extensions to the existing development tools. In this paper we will introduce JTube, a small Eclipse plug-in that shows how the internal structure of Java classes can be visualized and how the associations between source code segments and external documentation can be managed.

## 2 Background

### 2.1 Problems of Structuring and Documenting Code

Today, a number of tools try to organize the correlation between code, documentation and organizational information. The Rational Unified Process (RUP) for example offers the developers with activities and artifacts that can be used to plan, track and review all steps of the software development process. Although the creators of RUP argue that it can be adapted to even the needs of very small projects, it still seems to be more comfortable to use Java comments to organize source code fragments, to document relevant information that is necessary to maintain the respective code segment or to point to external sources of information. The reason for this is probably the high flexibility of Java comments and the lack of willingness to adopt highly detailed processes by the developer. As pointed out in the paper *"Making RUP Agile"* [7], it is necessary to create *"a scaled down version of RUP with as little process overhead as possible"* in order to use it for small projects.

Developers that want to manage the structure and documentation of their code from within the code unfortunately have to cope with the following issues:

1. **Unexploited Source of Information.** Using Java comments to structure code segments makes it hard to make use of this information. Although the user explicitly adds structural information to the source code, this source of information is neglected by most IDEs. For example, it is not possible to extract information on shared responsibilities from a class's methods, although the developer might have used Java comments to categorize the methods. There is no way to use the grouping information contained in such comments.
2. **Lack of Visualization.** Closely connected to the previous problem is the fact that information contained in source code is not sufficiently visualized in most IDEs. In Eclipse, for example, there is no visualization of the relationships between methods (e.g. method A is a submethod of method B) and nice features such as the call hierarchy or type hierarchy are only available for one method at a time.
3. **Documentation Scattering.** Documentation is often distributed across multiple documents, ranging from different Javadoc comments in different classes to external documentation that is referenced from within Javadoc

comments. It becomes increasingly difficult to keep this information consistent and easily accessible. There is no mechanism for getting a fast overview of how different documentation sites are interrelated and to navigate this information efficiently.

## 2.2 An Agile Approach

The “*Manifesto for Agile Software Development*” [4] declares the following values:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

We want to contribute to these values by adapting to the needs of the software developer when it comes to structuring and documenting code: If the developer is used to representing meta-information using Java comments—why not support this behavior with a mechanism to make it manageable? We believe that supporting the developer in his desire to categorize, document and interrelate code using Java comments is the basis for a powerful and lightweight tool to support agile software development. Making use of the incremental nature of Java comments and flexibly combining it with external information will furthermore help closing the gap between inline and external documentation.

### 2.2.1 Eclipse and Java

Eclipse is one of the most widely accepted IDEs for Java software development [1]. We will make use of its Java development tools and its Java search engine to analyze and visualize the internal structure of Java classes.

### 2.2.2 Java Comments for Structuring

As developers often use Java comments to insert delimiters between method groups or to explicitly tag methods with certain labels, we will extract the information contained in these comments to analyze and visualize the user-defined structure of Java classes.

### 2.2.3 RDF for Documentation

If it comes to linking inline documentation to external sources, we will be using an RDF [2] database to tag these links. RDF allows us to persist documentation in small, homogeneous units, that are the basis for efficient querying. The underlying RDF data management will be provided by Hyena, an RDF editor for software engineering. Hyena can be used to manage annotations such as documentation or categorization and for querying its database [12]. We will integrate the Hyena-based RDF support seamlessly so that the developer can add links effortless.

## 3 JTube

### 3.1 Overview

JTube is an Eclipse plug-in that supports the developer in using Java comments to structure and document code and visualizes the inner structure of Java classes by combining different sources of structural information (submethod dependencies, Java comments). JTube is designed to be a lightweight component that integrates seamlessly with the Eclipse SDK.

Our goal was to solve the problems described in Sect. 2.1 in a “hands-on” fashion, making use of current development tools while protecting the developer from having to adopt a new process. The result is JTube, an Eclipse plug-in that was implemented as a prototype to present the desired features in a Java environment:

1. **Grouping Methods.** JTube supports the developer’s desire to group methods using Java comments and makes this grouping browsable. This way, JTube exploits the structural information contained in Java comments and makes it visible.
2. **Recognizing Submethods.** JTube recognizes that developers create private methods that only function as submethods for other methods and makes this relationship visible.
3. **Adding Meta Information.** JTube helps the developer to reference arbitrary information (e.g. other code segments or external documentation) that is related to a certain line of code making use of the extensive capabilities of Hyena, a tool for RDF-based data linkage [11]. This helps to combine fast-accessible and easy-to-maintain inline documentation with in-depth external documentation.

### 3.2 Features

#### 3.2.1 Grouping Methods

In recent discussions (cf. Sect. 5.1.2) about possible extensions to the current set of Javadoc tags, a tag named “@category” has been suggested to be used for “logically grouping classes, methods, fields” [9]. Although our contribution was not inspired by this suggestion, we adopt the naming for uniformity reasons. JTube supports method grouping in a straightforward manner. By using the Javadoc @category-tag in a method-internal Java comment, a user can mark a method as belonging to one or more groups (separated by commas). Thus, the user can assign methods to groups with respect to their responsibilities (e.g. comparing objects or sorting data).

JTube makes these group assignments visible by providing a specialized outline view (“Enhanced Outline View”): This view is in synch with the current Java editor and presents a group-centric view on the methods contained in the current Java class. The usability of the enhanced outline view is based on the basic principle for visual user interfaces—the *Visual Information Seeking Mantra* as it was described by Shneiderman in [13]:

1. Overview first

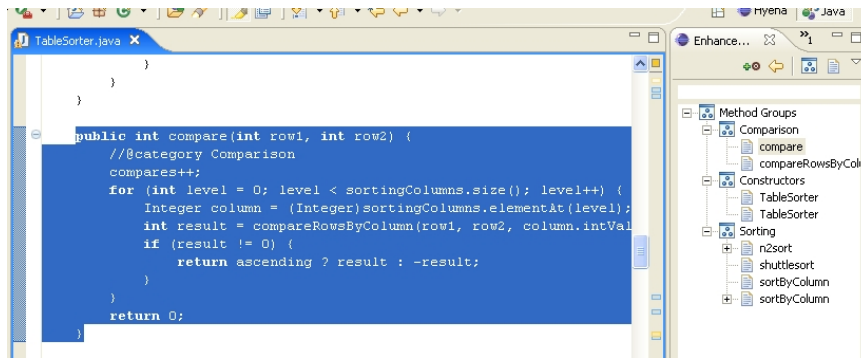


Figure 1: JTube’s Enhanced Outline View visualizes individual method group assignment

2. Zoom and filter
3. Then details-on-demand

JTube’s enhanced outline view presents the user with an *overview* of all groups that are available in the current file. For each method group JTube will present a subtree that contains all the methods that belong to the same group (cf. Fig. 1), which allows the user to *zoom in* on a group of interest. JTube furthermore offers a text field that allows the user to *filter* the displayed list of methods based on (parts of) their name. If the user wants to go into *detail*, he can then easily navigate to methods of the same group by clicking on them.

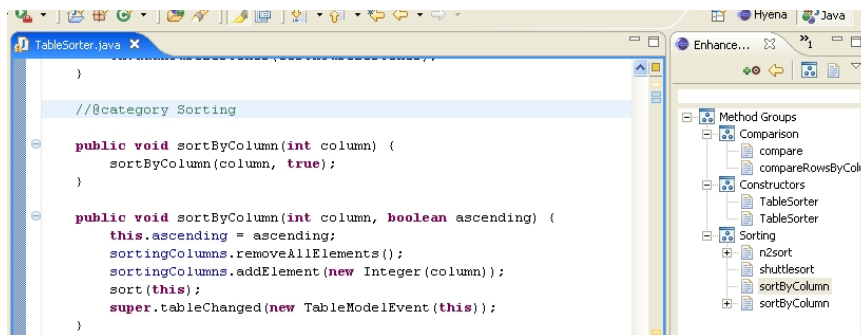


Figure 2: JTube’s Enhanced Outline View supports grouping multiple methods

In order to get even closer to how developers use Java comments to group methods, it is also possible to use the `@category`-tag in a comment that is placed between methods (i.e. in the class scope). This tells JTube to interpret such a group assignment as relevant to all the methods that follow this Java comment. Such a group assignment might span multiple methods until another group assignment, that is also placed between methods, might override the current assignment. JTube’s enhanced outline view offers a unified visualization of method groups—irrespective of whether the groups were assigned by using Java comments that were placed inside of methods or between methods (cf. Fig. 2). Furthermore, the developer can combine these two ways of grouping methods

to mix general assignments spanning multiple methods with fine-grained group assignments of individual methods. For example, the developer can assign several methods to the group “Sorting” and one of these methods additionally to the group “Filtering”. This provides the necessary flexibility in making group assignments.

To increase the usability of JTube’s method grouping feature, the user is not forced to use the Javadoc `@category`-tag. Instead, JTube also analyzes Java comments that start with “//–”, in order to allow for the use of Java comments such as

```
//----- Sorting
```

to structure Java code. This is helpful to visualize the internal structure even of old Java source files (see also the use case in Sect. 4.1). If JTube encounters a group assignment that defines no group name (e.g. “//@category”), JTube lists the methods belonging to this anonymous group as belonging to the special group “<anonymous group #X>”.

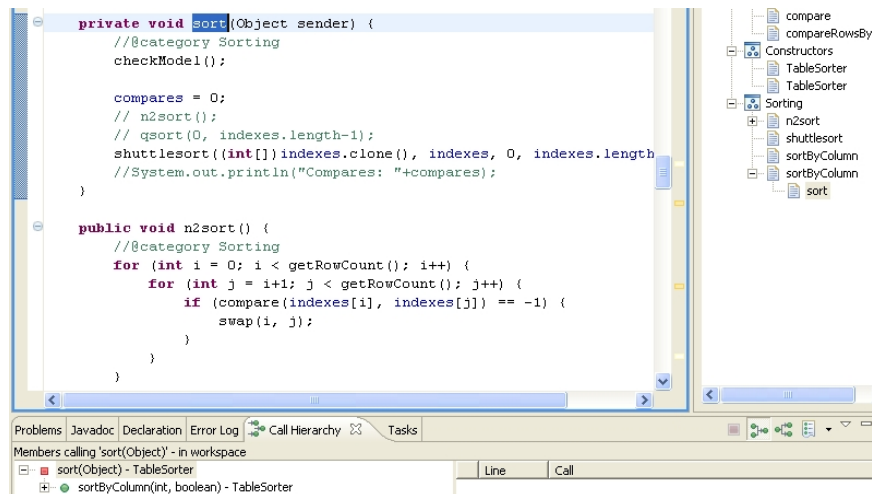


Figure 3: JTube visualizes submethods as leaves in the method-tree

### 3.2.2 Recognizing Submethods

Sometimes the source code of a single method extends over dozens of lines. In order to keep it simple, developers often refactor the source code and divide such a method into several “submethods”. As Kent Beck puts it, “*Lots of little methods named by what they are intended to do, not how they do it, make understanding the high-level structure of a computation easy.*” [3]. His “*Composite Methods Pattern*” explains the principle of how to extract such submethods from large methods. In Java, submethods are usually declared as private and are only referenced by their parent method. The standard outline view, however, presents us with a flat outline of all methods contained in the class—the dependencies between methods cannot be displayed. JTube provides an enhanced outline view that allows the user to browse methods and their depending submethods. By default, submethods are not displayed in the outline.

This keeps the outline view simple and helps understanding the basic structure of the class (cf. Fig. 1). If the user wants to access the submethods for a specific method, he simply expands the respective subtree in the enhanced outline view (cf. Fig. 3). By integrating this feature with the preceding one (method grouping), the developer has a very intuitive outline view at hand that combines two very basic ways of structuring source code. To preserve simplicity, JTube makes sure that submethods, that are members of the same method group as their parent method, are listed only once in the group listing.

### 3.2.3 Adding Meta Information

A typical problem of source code documentation is that documentation that concerns cross-cutting topics is scattered over a multitude of different Java source files. Although it is possible to interrelate this information, e.g. by using Javadoc's `@see`-tag, it is hard to get an overview of all related documentation on a specific topic. In [5], Bjune and Hagemeister show that linking documentation to source code and vice versa *"can substantially aid the software development effort by maintaining traceability between work products"*. Their paper describes a showcase tool that can be used to create links between Visual C++ source code and Office documents. Our approach shares the idea of linking-up different sources of information while sticking to the tools the developer is used to, and adds a number of new ideas to facilitate synchronization for changes in the workspace and to improve collaboration with other developers. Through its integration with Hyena, JTube allows to link arbitrary information to Java resources and this information can then be manipulated, browsed and queried by the user in Hyena.

JTube allows the user to add references for lines of code to the Hyena database. The goal of this functionality is to make code positions available for reference within Hyena. This requires basically two things:

1. References for different lines of Java code must be unique.
2. References must be synchronizable with respect to code changes/deletions.

To fulfill these requirements JTube uses Eclipse markers (cf. Sect. 5.2.2) and the RDF persistency capabilities of Hyena. Using these technologies, JTube can make sure that the link between the Java source code and Hyena's symbolic reference always stays synchronized. In other words, if the user moves or deletes a file, JTube will automatically change/remove all of the file's references in the Hyena database. If another developer adds or removes a reference from/to Hyena, JTube will remove/add its link to the Hyena reference as soon as the user opens the respective file the next time. To ensure the usability of the above described functionality, JTube offers a very simple interface: JTube's enhanced outline view contributes two buttons to the view's toolbar. The first button allows the user to add a reference to the currently selected line of code to Hyena. JTube adds a decorator to the vertical toolbar to indicate the link to Hyena and to allow the user to navigate to the Hyena reference (cf. Fig. 4). JTube also contributes a command to Hyena ("Show in Java editor") that can be used to let Eclipse open a Java editor with the currently selected Java source code reference. These two features combined provide bidirectional navigation between the Java editor and Hyena.

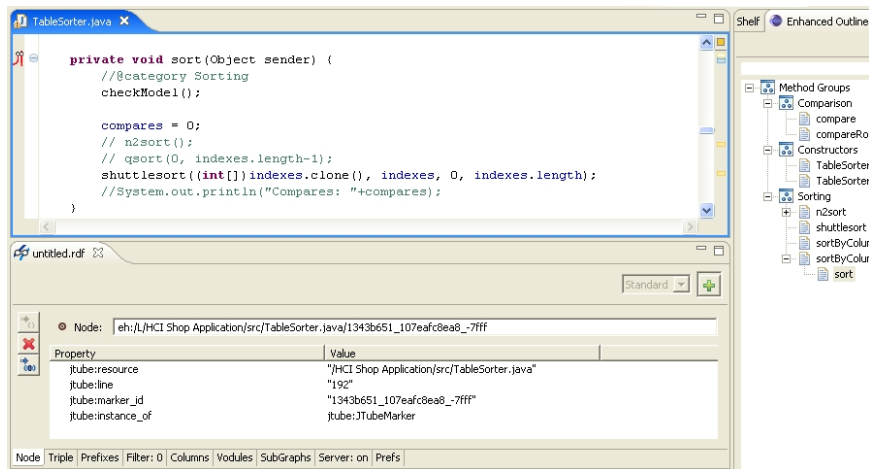


Figure 4: JTube markers link to data in Hyena

After the developer has added a reference to Hyena he can use Hyena to better manage his documentation: By linking up related pieces of documentation in Hyena the developer finally can see all related sources of information in one place. Using Hyena’s querying features, it is even possible to create different dynamic views on these networks of documentation. The second button allows the user to force a complete update from the Hyena database. This is very useful, when the developer checks out a project from a repository that already uses a Hyena RDF database, because Eclipse currently does not support sharing marker information with other developers. In order to synchronize the information contained in Hyena with the workspace, JTube will add markers to their respective Java resources for all references that were discovered in Hyena and that are currently not managed by JTube.

Furthermore, Hyena can be used to link arbitrary information to referenced code. Since Hyena is extensible and adaptable to a multitude of different fields of application, it will be possible to create tools for managing external sources of information almost without limitations. Everything that is representable as RDF can be imported to Hyena. For example, there is already an extension for an RDF-based Wiki. Linking JTube’s method references to Wiki resources might lead to an integration of Javadoc and Wiki-based documentation.

### 3.3 Architecture

JTube is a simple Eclipse plug-in that contributes a view (“Enhanced Outline”) to the Eclipse IDE. It uses some of Eclipse’s core features to search files in the workspace and to evaluate Java source code. It provides an interface to Hyena, a tool for managing information with an RDF database.

#### 3.3.1 Components

JTube consists of three main components:

1. **Data.** This component provides the basic functionality to access the

relevant data from within the Java workspace and Hyena. It encapsulates all the details of the Eclipse and Hyena integration and offers the necessary interface for querying Java elements, Hyena source code references, etc.

2. **Model.** The data model is an abstraction of the actual Java source code and contains only the data relevant for the current JTube view. This component is responsible for storing information on method groups and submethods.
3. **View.** This component contains the JTube Enhanced Outline View and all classes necessary to create this view and contribute it to the Eclipse workbench.
4. **Actions.** This component contains all actions that JTube contributes to Eclipse and that can be triggered by the user (e.g. adding references to Hyena).

### 3.3.2 Eclipse Integration

JTube has been designed to be a lightweight extension to the Eclipse SDK. Its Enhanced Outline View is contributed to the workbench by extending the Eclipse plugin extension point `org.eclipse.ui.views` and can be found in the Eclipse views collection under the category `JTube`. Since Hyena is also an Eclipse plugin, JTube also extends a Hyena extension point to integrate with Hyena (`de.hypergraphs.hyena.vodules`).

The JTube plugin makes use of several Eclipse core features:

- **File Search Engine.** To search the locations of methods that belong to the same group (cf. Sect. 3.2.1).
- **Java Search Engine.** To analyze if a method has the role of being a submethod (cf. Sect. 3.2.2).
- **Java Data Model.** To access Java methods as data objects.
- **Resource Markers.** To keep track of changes in source code positions and to enable synchronization with Hyena (cf. Sect. 5.2.2).
- **Resource Change Listener.** To be notified whenever a resource in the workspace is changed or deleted (cf. Sect. 3.4).

## 3.4 Synchronization

A major issue for the implementation of JTube was the synchronization between the Eclipse workspace and Hyena. Eclipse already offers everything needed to synchronize Java objects that are attached to specific lines of code in Java resources (so called “markers”). That means, whenever a user moves or changes a file or its contents, Eclipse makes sure that markers stay attached to their respective lines of code. If the user deletes a file all of its markers are deleted as well. As mentioned in [6] we only have to take care of “stale markers” that may still exist in Hyena, when they are changed or deleted by Eclipse. To achieve this we follow a two-fold synchronization process:

1. **Changes in the workspace.** In order to track changes in Java source files, we add a `ResourceChangeListener` to the Eclipse workspace that handles file move/rename operations. The listener updates Hyena by either removing marker information (file was removed) or by changing marker information (resource name/location has changed).
2. **Changes in Hyena.** If Hyena has been assigned new information from outside the current Eclipse workspace (e.g. by another software developer) it might reference non-existing markers or markers still existing in the workspace might have been removed from Hyena. To accommodate this issue, we regularly update the markers in the workspace with the information from the Hyena knowledge-base, either when a Java source file is opened or when the user explicitly orders a full update by clicking on the respective button.

This way, we provide the user with full flexibility in managing the links between JTube and Hyena: Both directions ( $JTube \rightarrow Hyena$  and  $Hyena \rightarrow JTube$ ) support adding, changing and removing links. To make the synchronization method reliable, Hyena will always have the higher precedence. The basis for this clear decision is the fact that JTube’s information relies on Eclipse markers that are only stored locally in the user’s workspace. Hyena’s information, however, can be shared with other developers and therefore deserves higher precedence in order to allow developers to collaborate (e.g. using a CVS repository).

## 4 Use Cases

JTube can be used in a variety of everyday situations. In the following section we will present use cases for each of the above described features.

### 4.1 Grouping Methods

#### 4.1.1 Situation

Knowing the dependencies between different methods is important to understanding the different responsibilities of classes and their methods. In the example of the Hyena RDF data management tool, the core class (`HyenaEngine`) fulfills different responsibilities: It provides factories for data containers and meta-containers, it offers methods that are relevant to Hyena “services” and provides different kinds of information (cf. listing in Fig. 5). The developer decided to mark the respective segments with comments such as

```
//----- Meta-Containers
or
//----- info: for broadcasting global information
```

#### 4.1.2 Problem

The problem is that the semantic connection between methods of this class will neither become clear from inside the code (if the comment gets out of sight) nor from the classes’ outline view (cf. Fig. 6)—there is no way to browse all methods belonging to the group “Meta-Containers”.

```

public class HyenaEngine
    implements Disposable, HyenaService, MetaComponent {

    ..

    private HyenaEngine(Display display) {...}

    public void dispose() {...}

    private void makeStandardContainers() {...}

    private void makeMetaComponents() {...}

    //-----

    /**
     * If Eclipse is currently running, execute {@param runnable}
     * in the SWT thread. [...]
     */
    public void syncExec(Runnable runnable) {...}

    //----- Meta-Containers

    private HyenaContainer _metaContainer;

    public <T extends MetaComponent> T getMetaComponent(
        Class<T> metaContainerImpl) {...}

    public HyenaContainer getMetaContainer() {...}

    //----- Containers

    public HyenaContainer makeContainer(String pathStr) {...}

    /**
     * Make a container and file it under a certain path
     */
    public HyenaContainer makeContainer(LabelPath path) {...}

    /**
     * Make a container. [...]
     */
    public HyenaContainer makeContainer() {...}

    ..
}

```

Figure 5: Excerpt from the source code of the class `HyenaEngine` showing the use of Java comments to group methods.

### 4.1.3 Solution

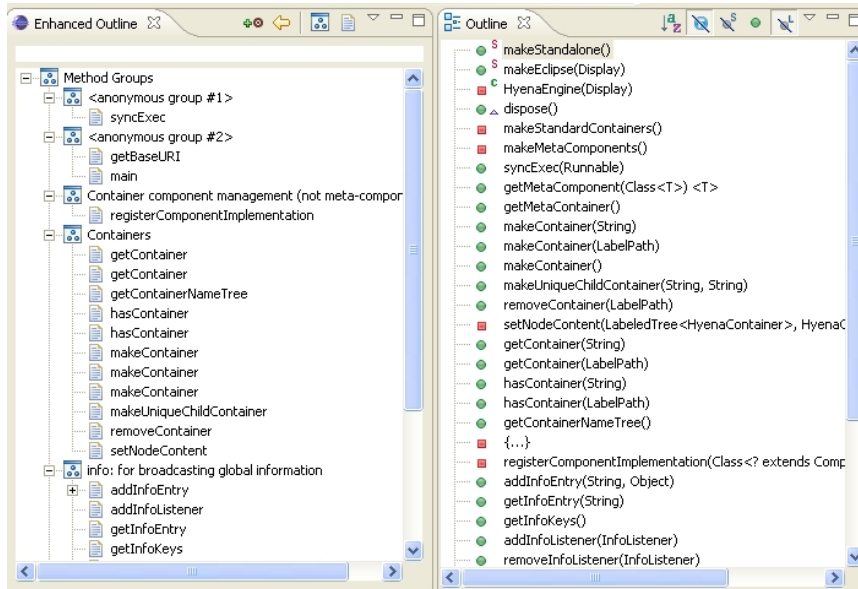


Figure 6: Comparison between JTube’s outline and the standard outline

With JTube the user can stick to his usual way of using comments to group methods. The advantage is that methods belonging to the same group(s) suddenly can be browsed using the enhanced outline (cf. Fig. 6). The structure that has been defined by the user is now explicitly visible in JTube’s enhanced outline and can be used to

1. get an overview of which groups there are
2. see which methods belong to the same group
3. find a method by looking at the group that describes its responsibility

## 4.2 Submethods

### 4.2.1 Situation

It has become a rule-of-thumb to source out segments of large methods to “sub-methods” in order to prevent monolithic structures that are difficult to understand and maintain. The Hyena class `NodeViewPage`, for example, contains the method `createControl`, that has got three submethods: `createLeftToolBar`, `createTopRightToolBar` and `updateSubjCount` (cf. Fig. 7).

### 4.2.2 Problem

Unfortunately, submethods look like “normal” methods and thus contribute to a class’s outline. If other developers want to understand the content and structure of the class, they will have a hard time fighting their way through the forest of

```

public void createControl(Composite parent) {
    //@category Control Creation

    _composite = new Composite(parent, SWT.NULL);
    ..
    createLeftToolBar(_composite);
    ..
    createTopRightToolBar(_composite);
    ..
    updateSubjCount();

    // table has been initialized, we do have a node count now
}

```

Figure 7: Source code of the method `createControl`

wild methods. What is missing in the standard outline is the big picture of the parent/submethod structure of a class (cf. Fig. 8).

### 4.2.3 Solution

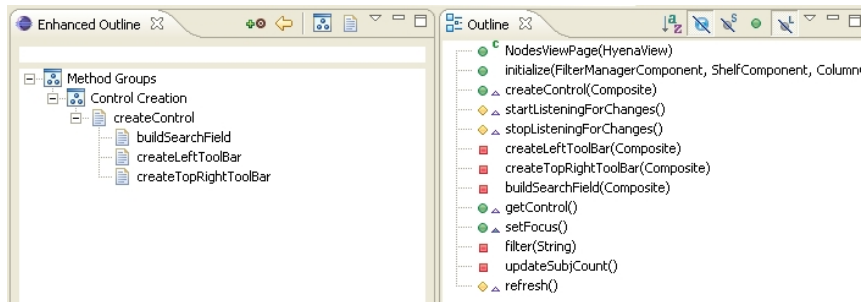


Figure 8: JTube's outline showing submethods compared with the standard outline. JTube only shows methods that are assigned to a method group.

JTube bundles methods and their respective submethods when generating an outline of a class (cf. Fig. 8). This has two major advantages:

1. The developer can easily grasp the structure of a class, as he only sees the main methods on the top level of our enhanced outline.
2. The dependencies of submethods and their respective main methods are modeled in a straightforward manner: Our enhanced outline shows main methods together with their submethods as children.

## 4.3 External Documentation

### 4.3.1 Situation

In-depth documentation and additional information is often placed outside the source code in external documents, as they have to be maintained by different

stakeholders—in most cases not the developers alone. In order to relate to this external documentation, developers often use Javadoc comments to point to external documents. In the case of documents that can be located on a network, Javadoc already provides the functionality to cope with these references. If it comes to persons, organizations, processes, etc. it gets harder to express this with Javadoc and external management of this information is hardly possible.

In our example, the software development process used by the company involves a quality assurance process consisting of an obligatory code review and follow-up reviews if necessary. The development team keeps track of the reviews using a spreadsheet that contains all class names and a status field that indicates the result of a review (“OK”, “Minor Changes”, “Follow-Up Review Necessary”). The comments made by the reviewer are either put into the spreadsheet as well or written down on a printed version of the code segment being reviewed.

### 4.3.2 Problem

The problem in this case is the lack of stability of the QA process. If developers decide to rename or move a file the reference from the external documentation to the source code breaks, because the reference simply consisted of the resource’s path and filename. Developers then have to search for the file and update the spreadsheet accordingly. With a growing number of files that are put into the QA process it gets increasingly difficult to maintain all the references.

### 4.3.3 Solution

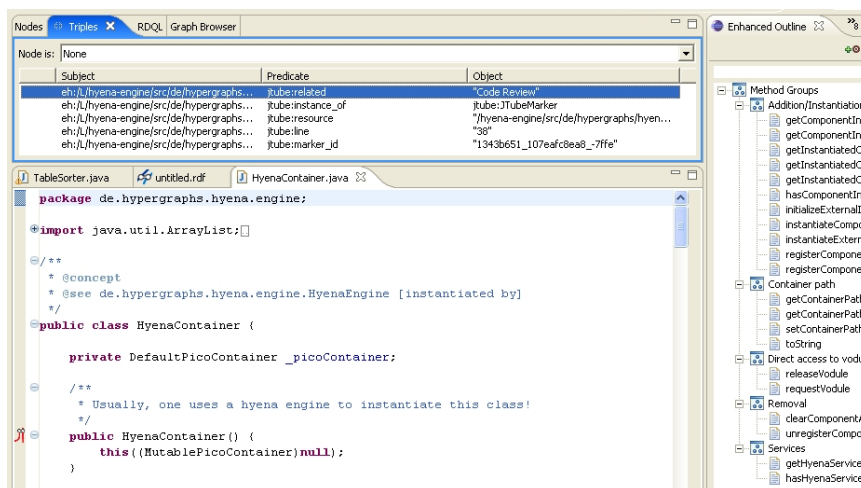


Figure 9: The class HyenaContainer and its reference in Hyena

In this case, JTube can help linking code fragments with their respective review documentation using RDF-based data linkage (cf. Fig. 9). This allows the developer to relate code segments to the review document and vice-versa. When the developer moves a file to a new location, the respective references are immediately updated. There is no need for additional maintenance.

## 5 Related Work

### 5.1 Javadoc

Most of our contribution aims at using more efficiently the structural information contained in Javadoc comments. We will therefore focus on the tag-set offered by Javadoc. Doclets and Taglets, which are themselves extremely interesting technologies but do not belong to our topic, will not be discussed here.

#### 5.1.1 Current Standard 1.5

In its current version Javadoc by default recognizes 19 different tags [8]. In our context the most relevant ones are:

1. **@author**. The **@author** tag can be used to reference one or more authors. It can be used to reference a name, an email address, etc. and thus provides one-way links between Javadoc documentation and external entities that qualify as authors.
2. **@link**. This tag can be used to create a hypertext link from a documentation position to the documentation of a specified Java element. It helps the developer to create cross-references of Javadoc documentation at arbitrary positions in the documentation.
3. **@see**. The **@see** tag is the most powerful tag of the ones discussed here. It can be used to reference a String literal, a URL or the documentation of a specified Java element. Therefore this tag provides the user with the ability to link-up different sources of information, whether it is Javadoc documentation, documents and entities identifiable using an URL or other entities that can still be described using plain text.

With these tags the developer already has the tools at hand to reference arbitrary information. If it comes to navigation, Javadoc allows the user to navigate back and forth between Javadoc documents and also from Javadoc documents to other URL-addressable information. With JTube and Hyena it should become easier to reference also non-URL information and allow navigation back to the source code. In this respect, JTube extends the capabilities of Javadoc in a natural way: By partnering with Hyena, JTube is able to provide bi-directional, synchronized links between Java source code and external information.

#### 5.1.2 Proposed Tags

With respect to future developments, Sun has already published a list of potential extensions to the list of Javadoc tags [9]. One of these, the **@category**-tag, is relevant to our topic. The idea is to use this tag to group methods using category names. This would allow for structuring Javadoc documentation beyond package and class hierarchies. The Enhanced Outline View offered by JTube already makes use of this tag to show methods according to their group assignment (cf. Fig. 1). JTube also demonstrates how different positioning of this tag (inside or outside a method) can be used to create different scopes for its validity (cf. Sect. 3.2.1).

## 5.2 Eclipse

JTube is an Eclipse plugin and integrates with other important features of Eclipse. Some of the features JTube provides can already be achieved using other Eclipse functionalities. Their combination, though, is unique to JTube.

### 5.2.1 Eclipse Task Tags

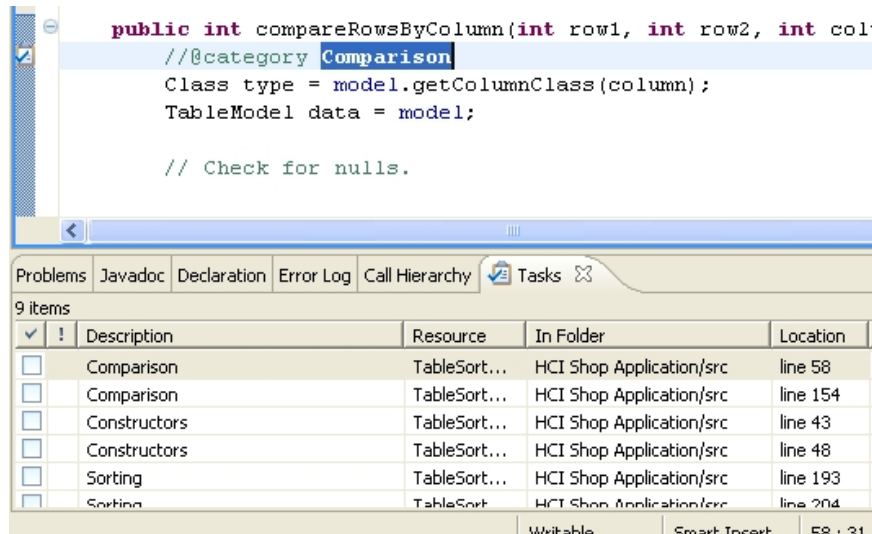


Figure 10: Eclipse Task View

Eclipse allows the developer to specify special tags to be used in comments, so called “Task Tags”, that are interpreted separately by Eclipse. Task tags can be used to categorize lines of source comments, e.g. into “todo”, “debug”, “fixme”, etc. Whenever the user adds or deletes such a task tag in a line of documentation, Eclipse automatically creates/deletes a marker (cf. 5.2.2) that will be displayed in the Tasks view. The developer can therefore categorize arbitrary lines of Java comments and get an overview of the amount of occurrences per category from the Tasks view. This functionality, though, aims at assigning tasks on a line level not on a method level. It is not possible to group the task occurrences—neither per category nor per class.

### 5.2.2 Eclipse Markers

Whenever a developer adds a task tag to a line of documentation, Eclipse generates a marker object that contains the task tag’s information and the location of the respective line of documentation. Eclipse keeps track of this marker object and makes sure that it stays synchronized with any kind of move or delete action performed on the respective file. Besides task tags, Eclipse also uses marker objects to keep track of compiler warnings, compiler errors, general information (e.g. coding style guidance) and user-defined bookmarks.

Since Eclipse already manages all the synchronization issues one is confronted with when assigning external meta-information to Java resources, we

decided to make use of this feature for synchronizing our external Hyena-based knowledge-base with the Java workspace. Eclipse markers are a great tool to keep track of changes that are performed on files or single lines of code, but unfortunately this information (i.e. the markers themselves) are not intended to be shared with other users. To accommodate this, JTube provides its own synchronization technology (cf. Sect. 3.4) for synchronizing links between code and external information in a multi-user environment.

## 6 Current and Future Research

In addition to the above described features, there are a number of features that are being discussed, but either exist only as mock-ups or are not fully implemented yet.

### 6.1 Active Knowledge

#### 6.1.1 Idea

One way to save implementation knowledge for later use is the creation of implementation templates. For example, by the time a developer decides to use a design pattern, the concrete implementation remains nothing more than a time consuming task that could easily be automated. For this reason we have implemented a spike for one-click template-based implementation, which we call Active Knowledge.

#### 6.1.2 Implementation

The process of using Active Knowledge begins with its declaration. Our JTube spike uses RDF to save the relation between code fragments and Active Knowledge. Javadoc comments are used to synchronize the RDF database with the code. Depending on the type of Active Knowledge that has been declared, JTube analyzes the code and generates the necessary code fragments of the chosen implementation template. Following its predefined heuristics JTube then modifies the code to integrate the implementation into the existing code. Our spike uses the well-known Observer Pattern. In order to let JTube know that a class uses the Active Knowledge of what an Observable is, JTube uses the following Javadoc comment to map the code fragment to its respective graph representation in the RDF graph:

```
/**
 * @see ObeserverPattern.Observable is_a
 */
public class myObservableClass {
    ...
}
```

JTube uses this information to identify classes that make use of Active Knowledge (in this case the Observable part of the Observer Pattern) and adds the statement

```
myObservableClass is_a Observable.
```

to the Hyena RDF database. In a next step JTube checks all methods in the current class to see whether their Javadoc comments contain JTube tags of the form

```
@see ObserverPattern.Trigger is_a.
```

If this is the case JTube renames the method to `<method_name>Action` and generates a proxy method that will replace the original method's implementation. This way, calling the original method will trigger the following code fragment generated by JTube:

```
public boolean <method_name> (Object myArg) {
    try {
        Object[] arguments = new Object[] {myArg};
        fire("method_name", arguments);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return <method_name>Action(myArg);
}
```

The `fire` method calls all “listeners” that have previously been registered to observe the observable method<sup>1</sup>. The return value of this proxy method is then the result of the original method. This implementation would even allow for vetoing, as the listeners are called before the actual method code is invoked.

### 6.1.3 User Interface

For the sake of simplicity, we have also implemented a one-click solution to add Active Knowledge to Java code. The developer simply selects the method he or she wants to make observable, opens the editor's context menu and chooses “Apply Active Knowledge>Observer Pattern>Make Observable”. JTube then creates the whole RDF graph (including its Javadoc reference points) automatically.

## 6.2 Concept Mining

### 6.2.1 Idea

Our Concept Mining spike lets the developer take a *concept*-oriented view at a project. It is based on the observation that names of classes, methods and fields often convey multiple concepts per name.

Take, for example, the class name `BufferedOutputStream`. The name implies that three concepts are involved: “buffering”, “output” and “stream”. We can draw this conclusion, because Java developers usually encode multiple concepts into one identifier using alternating upper and lower case characters (cf. the upper case “B”, “O” and “S”) in this example. Our JTube spike makes use of this coding convention by extracting all identifiers from a package and tokenizing them into sequences of concepts. The resulting list of tokenized identifiers is then used to discover relations between concepts in order to create a visualization of the concept graph.

<sup>1</sup>JTube also creates the necessary method to allow listeners to register for observable methods.

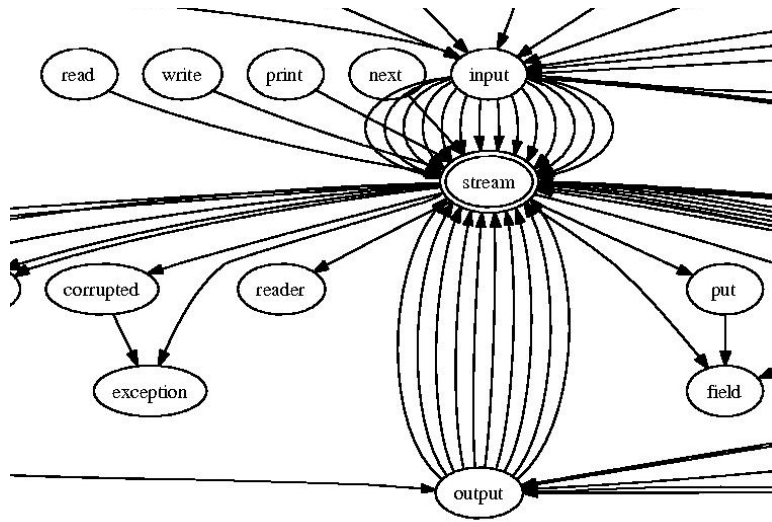


Figure 11: Excerpt of JTube’s graph output of concepts related to the concept “stream” in the java.io package.

### 6.2.2 Implementation

Given a specified package and a concept to look for (e.g. a `stream`), JTube calls a Java Doclet that traverses all classes in the package. It extracts the class, method, and field names that relate to the given concept and uses a simple tokenizer to split up names that contain multiple concepts into single concepts, e.g. the identifier `BufferedOutputStream` becomes `Buffered`, `Output`, `Stream`. After applying a normalization to lower case characters, JTube generates a dot-file containing the concept graph to be displayed by Graphviz.

### 6.2.3 User Interface

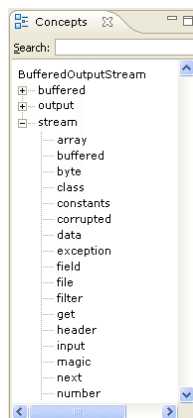


Figure 12: Concept View

Eclipse developers are used to look at a project from different perspectives using Eclipse’s Outline, Type Hierarchy or Call Hierarchy views. We have created a mock-up for a Concept view (see Fig. 12) that can be added to Eclipse. It can be used to display all concepts related to the current code selection. The developer can use this view to get a rough overview of the current conceptual context.

In a future implementation JTube should provide the possibility to choose a concept from the list or alternatively use a textbox and tab completion to look for a specific concept. The JTube spike provides the option to display a graph that visualizes the concept’s relation to other concepts in the package (see Fig. 11).

### 6.3 Web Integration

A further idea for future research would be the integration of JTube’s capabilities into a web service. Hyena already offers an RDF-based wiki (“Wikked”) that could be easily extended to display JTube RDF data. This could be used for publicizing JTube information to the web and to collaborate over a network.

Currently, the information displayed by JTube is only accessible from within the Eclipse Workbench. Using Hyena’s web interface, JTube could soon be extended with a web interface. One of the major advantages of sharing JTube’s information using a web interface might be further integration of the documentation process. Having a web interface it would become very easy to manage internal and external sources of documentation.

## 7 Conclusion

In the previous discourse we have seen how lightweight tools such as JTube can increase the usability of scattered documentation and structural information contained in submethod dependencies and source code comments. The advantage of our agile approach lies in the fact, that the developer receives JTube’s support without having to change his way of working. By integrating JTube in a next step with an RDF-based wiki, the level of collaboration can be substantially increased and foster the interaction between developers and other stakeholders in the software development process.

**Acknowledgments.** JTube has been developed at the research unit for Programming and Software Engineering of the Ludwig Maximilian University (LMU) Munich. Thanks to Axel Rauschmayer for his academic guidance and continued support during the whole course of this project and Markus Schuster for spending numerous hours with lively discussions during breaks over a cup of steaming hot java.

## References

- [1] Java IDE Market Share Survey. [http://www.qa-systems.com/products/qstudioforjava/ide\\_marketshare.html](http://www.qa-systems.com/products/qstudioforjava/ide_marketshare.html), 2003.

- [2] RDF Primer, W3C Recommendation. <http://www.w3.org/TR/rdf-primer/>, 2004.
- [3] Kent Beck. *Kent Beck's Guide to Better Smalltalk: A Sorted Collection*. Cambridge University Press, Cambridge, 1998.
- [4] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas. Manifesto for Agile Software Development. <http://agilemanifesto.org/>, 2001.
- [5] Geir A. Bjune, Jack R. Hagemeister. Hyper-Trace: Using Hypertext Linking for Traceability. In: *Proceedings of the Eighth IASTED International Conference on Software Engineering and Applications*, S. 641–646, November 2004.
- [6] Eclipse Foundation. Eclipse 3.1 Online Documentation. <http://help.eclipse.org/help31/>, November 2005.
- [7] Michael Hirsch. Making RUP agile. In: *OOPSLA '02: OOPSLA 2002 Practitioners Reports*, S. 1–ff, New York, NY, USA, 2002. ACM Press.
- [8] Sun Microsystems Inc. javadoc - The Java API Documentation Generator. <http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/javadoc.html>, November 2005.
- [9] Sun Microsystems Inc. Proposed Javadoc Tags. <http://java.sun.com/j2se/javadoc/proposed-tags.html>", September 2005.
- [10] Harold Ossher, Petri Tarr. Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In: *Proc. Symp. Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.
- [11] A. Rauschmayer. Hyena: A Semantic Web Enabled Editor for Software Engineers. Submitted for publication.
- [12] A. Rauschmayer. RDF as a Data Structure for Software Engineering. <http://www.pst.ifi.lmu.de/~rauschma/hyena/poster-hyena.pdf>.
- [13] Ben Shneiderman. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In: *Proceedings of the Conference on Visual Languages 96*, September 1996.
- [14] Yannis Smaragdakis, Don Batory. Implementing Layered Design with Mixin Layers. In: Eric Jul (Hg.), *Proc. 12<sup>th</sup> Europ. Conf. Object-Oriented Programming (ECOOP)*, *Lect. Notes Comp. Sci.* Bd. 1445, S. 550–570, 1998.
- [15] P. Tarr, H. Ossher, W. Harrison, Jr. S. M. Sutton. N degrees of separation: Multidimensional separation of concerns. In: *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, S. 107–119, May 1999.