A Comparison of Software Refactoring Tools

Jocelyn Simmonds
Ecole des Mines de Nantes
jocelyn.simmonds@eleve.emn.fr

Tom Mens
Vrije Universiteit Brussel
tom.mens@vub.ac.be

November 25, 2002

Abstract

The principal aim of this paper is to apply the Taxonomy of Software Evolution, developed by Mens et. al [1], to position various software tools that support the activity of software refactoring as part of the evolutionary process. This taxonomy is based on the mechanisms of change and the factors that impact upon these mechanisms. The goal of this taxonomy is to position concrete tools and techniques within the domain of software evolution, so that it becomes easier to compare and combine them. In this paper, we apply the taxonomy to four tools that provide explicit support for refactoring. The tools that were considered for this detailed study are the following: Smalltalk VisualWorks 7.0 [2], Eclipse 2.0 [3], Guru (for SELF 4.0) [4] and the Together ControlCenter 6.0 [5]. After a detailed discussion and comparison of these tools, we analyse the strengths, weaknesses and limitations of the evolution taxonomy that was used.

1 Introduction

In recent years, software evolution has gained importance. This is mainly because of the high maintenance costs that have been produced by non-evolutive software designs. When object-oriented languages were introduced, it was expected that maintenance costs would decrease in important amounts. This was because, unlike procedural languages, object-oriented languages permit code reuse and encapsulation. But the maintenance effort is only smaller when systems are well designed.

Unfortunately, maintenance costs stayed high. When new features were inserted into existing systems, the system designs were not changed so as to be able to accept these changes without disturbing the system as a whole. Instead, changes were inserted simply where they were possible.

To combat this problem, various tools were developed. These tools all use the idea of applying "refactorings" to existing system code, so as to better the existing design implicit in the source code. Refactorings were defined by Opdyke [6] as source-to-source transformations, that, when applied to source code, transformed it, without changing the behavior of the system.

Nowadays, various refactoring tools, techniques and mechanisms exist, each with specific application domains and characteristics. In this paper, we will try to classify some refactoring tools, using an evolution taxonomy that is based on the characterizing mechanisms of the tools [1].

This taxonomy is based on the mechanisms of change and the factors that impact upon these mechanisms. The goal of this taxonomy is to position concrete tools and techniques within the domain of software evolution, so that it becomes easier to compare and combine them.

Four tools have been chosen for this study. These tools have been picked because of the apparent differences that exist between them. It would be interesting to see if the tools are really as different as they seem.

2 Application of the Taxonomy

In this section, the "Taxonomy of Software Evolution" developed in [1], will be applied to four software development tools that provide explicit support for software refactoring: Cincom's *VisualWorks 7.0* [2], *Eclipse 2.0* [3], Together's *ControlCenter 6.0* [5] and *Guru* [**?**].

The tools where selected because of the differences that they show. *VisualWorks* is an integrated development environment (IDE) for Smalltalk that includes the well-known *Refactoring Browser* [7]. Smalltalk is a dynamic, class based object-oriented language. *Eclipse* (specifically, JDT) is a Java IDE that permits refactoring. Java, unlike Smalltalk, is a static, class based object-oriented language. *Guru* restructures and refactors SELF code in a fully automated fashion. SELF is a prototypical object-oriented language. Finally, Together's ControlCenter manages a great part of the production process, and has a Java IDE that permits refactoring.

We can notice, before applying the taxonomy, that three of the tools are IDEs that permit refactoring, while the last one (Guru) seems to be a stand-alone refactoring tool. Will this be visible during the detailed comparison of the tools?

The taxonomy will now be explained briefly, before applying it. A categorisation is proposed along six axes: temporal facets, facets of change, system facets, change process, object of change, and drivers of change. Each axis is subdivided into several facets.

2.1 Temporal Changes

Time of Change. Four times of change are defined: (T_1) the time (interval) when a *change is requested* (T_2) the time (interval) when a *change is prepared* (T_3) the time (interval) when a *change becomes available for execution* (T_4) the time (interval) when a *change is executed for the first time*

Change History. Tools can either support versioning, or not provide means of distinguishing the new versions from the old ones. In unversioned systems, new and old versions can not be used simultaneously in the same context. Versioned system may allow different versions to co-exist.

Change Frequency. Changes to a system may be performed continuously, periodically, or at arbitrary intervals.

2.2 Drivers of Change

Role of the user. Several distinct personnel roles can be identified in the lifetime of a software change. Here, one concretely identifies the users that interact with the tool.

Distribution of users. This facet identifies the physical location of the users involved in the changes. Are they making local or distributed changes?

Degree of Automation. The taxonomy distinguishes between three degrees of automation: automated, partially automated and manual software changes. Automated changes require no user intervention, while manual changes are made by the user. Partially automated is halfway between.

2.3 Facets of Change

Type of Change. Changes can be structural o semantic. Semantic changes can have an impact on the software behavior, while structural changes aim to preserve the semantics.

Effect of Change. A distinction is made between element addition, element subtraction and element modification (e.g., element renaming).

Invasiveness. This facet establishes whether evolutions and extensions are destructive or not. If they are destructive, they are call invasive, otherwise, they are non-invasive.

Safety. Two types of safety are distinguished: static and dynamic safety. A mechanism provide static safety is if can ensure, at compile-time, that the system will not crash or behave erroneously at runtime. If the mechanism has consistency checks during runtime, it provide dynamic safety.

Effort. Changes can require a high or low effort, when being included in a system.

2.4 Object of Change

Artifact. This facet specifies which software artifacts are subject to changes, when using an evolutionary tool.

Granularity. Granularity refers to the scale of the artifacts to be changed. It can range from very coarse, through medium, to a very fine degree of granularity. In this application of the taxonomy, coarse refers to the system or package level changes, medium refers to class level changes, and fine refers to method and variable level changes.

Locality. Changes can be local or global changes. Sometimes, local changes might be in fact global changes, because the local change is propagated throughout the system.

Scope. This refers to the impact that a change in one artifact might have on other artifacts, e.g., a change in the source code might require a change in the documentation.

2.5 System Facets

Availability. Once the changes have been integrated into the system, must the system be restarted so that these changes take effect? If so, the tool presents static software evolution, otherwise, it presents dynamic software evolution.

Activeness. The system is reactive if changes are driven externally, and proactive if the system autonomously drives the changes itself.

Openness. Software systems are open if they are specifically built to allow for extensions. Closed systems do not provide a framework for possible extensions.

2.6 Change Process

Plan. A change process is planned if the changes are managed in a more or less coherent manner. If not, it is a unplanned change process. Which one does the tool support?

Control. A change is controlled, if the constraints in the change process are explicit and enforced. Otherwise, the change is uncontrolled.

Measure. Which type of technique is used to measure the change? Impact can be measured by its impact on the system, or by the effort needed to produce the change.

3 VisualWorks 7.0

VisualWorks [2] is a popular Smalltalk IDE, produced by Cincom. The famous *Refactoring Browser* [7] has become the standard Smalltalk browser as of VisualWorks 7.0. To achieve this, Cincom has worked closely with Refactory, Inc. to integrate the complete refactoring browser toolset into VisualWorks. This secures VisualWorks' place as the premier extreme Programming toolset on the market. In addition to the program transformations that automate refactoring, the Refactoring Browser has buffers, drag and drop, undo, viewing resources with the correct editor, a graphical view of the class hierarchy, protection against editing old versions of a method by mistake, and Smalllint, a style checker and bug detector. The most important enhancements (and the reason why this tool is included in this report) is the tool's ability to perform behavior-preserving, refactoring transformations. In the application of the taxonomy to this tool, we will focus on its refactoring capabilities.

3.1 Temporal Changes

Time of Change. The VisualWorks is only concerned with the coding process in a software development effort. For this reason, it does not place restrictions on the T_1. The user is usually the developer or maintainer of the system, and after he receives the request for a change, the browser helps him prepare and execute this change (T_2). Since VisualWorks has been developed as an interactive tool, dynamic pre and post condition checking is used to make the modification of source code faster (T_3). Once the refactoring has been accepted by the browser, the compilation is finished (marking the beginning of T_4) resulting in an executable program. The tool's range of action only embarks T_2 and T_3.

Change History. The browser does not provide facilities for managing different versions. VisualWorks depends on the host IDE's versioning capabilities. The browser place no restriction on the type of IDE that must be used, either a versioned or unversioned environment can be used. Although it does not version the code, it does provide undo capabilities. This is important, because the decision of where and when to apply refactorings is not always clear. The maximum length of undo history can be modified by the user.

Change Frequency. VisualWorks is made to support *continuous* modifications. Maintainers and developers can incorporate changes at any given time. The base refactorings are usually applied in refactoring chains, to accomplish certain objectives, like code cleanup or design modification.

3.2 Drivers of Change

Role of the user. The browser is "code oriented", that is, its main concern is the development and maintenance of source code. For this reason, the main user of VisualWorks is the system developer or maintainer. User requested functions can lead to code cleanup or system redesign, so as to be able to include modifications without making the system corrupt and brittle. When changes require system redesign, the system architect can also become a user of the tool, deciding what refactorings to apply and where.

Distribution of users. At the present time, all Smalltalk IDEs are local, stand-alone environments. Since the browser "inherits" its behavior from the standard browser, it is limited by what this browser can do. This means that VisualWorks may only become a distributed tool, once its family of hosts becomes distributed. But the theory behind the Refactoring Browser [1] permits the future merging of multiuser programming environments, once the underlying Smalltalk environment is extended as a multiple user distributed application.

Degree of Automation. VisualWorks is a partially automated tool. The user has to pick which refactorings to apply to the source code, with what parameters, and when to apply them. The preconditions, postconditions and effects are calculated and applied in an automatic fashion. The user input is used to make the dynamic analysis of conditions quicker, by supplying things that are hard to calculate. It also helps maintain the source code readable, since no class, method or variable names are automatically generated. This tool does not automate the decision of which refactorings to apply, that is a user decision.

3.3 Facets of Change

Type of Change. The majority of the refactorings included in VisualWorks are *structural changes*. These refactorings were presented in the PhD of William Opdyke [6]. The idea behind these refactorings is that they are behavior preserving. Preconditions are used to determine if a refactoring is legal. Don Roberts extended Opdyke's definition of refactoring by including postconditions [8].

Effect of Change. The refactorings included in VisualWorks refer directly to the addition, removal and modification of program elements: classes, methods and variables (see Table 1). But these refactorings are always applied in refactoring chains, so as to preserve behavior. This means

[1] The implicit dependency relationships that exist between different refactorings

that the result of the application of these transformation rules is altered source code. The behavior that the program offers before the alteration is still present, and possibly new behaviors have been added.

Type of Refactoring	**Refactoring**
Addition	AddClass(*className; superclass; subclasses*) AddMethod(*class; selector; body*) AddInstanceVariable(*class; varName; initClass*)
Substraction	RemoveClass(*class*) RemoveMethod(*class; selector*) RemoveInstanceVariable(*class; varName*)
Modification	RenameClass(*class; newName*) RenameMethod(*classes; selector; newName*) MoveMethod(*class; selector; destV arName; newSelector*) RenameInstanceVariable(*class; varName; newName*) PullUpInstanceVariable(*class; varName*) PushDownInstanceVariable(*class; varName*) AbstractInstanceVariable(*class; varName; getter; setter*) MoveInstanceVariable(*class; varName; destV arName*)

Table 1: Partial list of refactorings included in VisualWorks

Invasiveness. VisualWorks modifies the existing source code directly. This means that the modifications made by the browser are invasive, unless the underlying IDE provides versioning capabilities (see Section 3.1 - Change History). The tool does provide the ability to *undo* refactorings. Even though the code suffers great alterations in form and content, it is still possible to rollback to an initial state.

Safety. Safety has to be ensured to make a tool trustworthy, so as to be used without making the software testing cycle larger. In the particular case of VisualWorks, the safety is provided by Opdyke's preconditions and Robert's postconditions. The addition of the postconditions permits condition checking to be done in a dynamic fashion. The condition analysis is done by observing the state of the system at runtime, and by examining the conditions that are true prior to the execution of the program. These conditions are the ones that have resulted to be true after accepting previous refactorings. By using dynamic checking and user input, a relatively high degree of safety can be guaranteed, since the information used to accept refactorings is more complete than that offered by a static, conservative algorithm using only preconditions.

Another way to achieve satefy is by making use of *unit tests*, one of the essential ingredients of eXtreme Programming [9]. This is supported in VisualWorks by means of *SUnit*. The combination of refactoring and unit testing makes the VisualWorks IDE ideally suited to do eXtreme Programming.

Effort. The main objective of the tool is to modify code, automating a tedious and error prone task. Now the user has to determine what refactoring to apply, when and where. Instead of doing search and replace of code, the programmer has to input certain information that is needed by the tool. This requires a relatively low effort from the user, since some IDEs offer versioning capabilities [2], and the browser itself offers undo capabilities, so the user can apply an arbitrary refactoring. If he is not satisfied with the result, he can go back to an initial state, and apply another refactoring. This means that it is a *low effort* tool.

[2]If the IDE does not offer versioning, it can be done manually by the user, or using a tool like CVS

3.4 Object of Change

Artifact. The only artifact that VisualWorks modifies, is the system source code. This is because it is code oriented, automating the search and replace that programmers have done historically by hand, when cleaning up the system source code, or applying design changes. If all the artifacts involved in a software development effort are to be kept compatible, it must be done in a manual fashion, by the development team.

Granularity. As mentioned in "Effects of Change" (see Section 3.3), the basic refactorings affect classes, methods and variables. These basic refactorings can be combined into refactoring chains, generating more complex refactorings. Using the classification mentioned in the taxonomy, VisualWorks has *medium* (class refactorings) to *fine* (method and variable refactorings) granularity.

Locality. VisualWorks is a tool used to make *global* changes to the software. Since the refactorings preserve behavior and all the refactorings offered affect from the class level, downwards, the user might think that he is only making local changes. But the alterations made by a refactoring may ramify to a global level, for example, when the main object of a system is modified. The beauty of VisualWorks is that all changes seem to be local to the user, since the code transformations are done automatically - so the user really only sees the local changes.

Scope. The browser is not integrated with the software development process, it does not coordinate its activities and products with those produced by the other phases in the development process. Only the source code of the project is modified by the browser. All changes to the other artifacts produced during the software development, must be done manually. If applying only behavior preserving refactorings, the system design and documentation should change, but the requirement specification should not need to change.

3.5 System Facets

Availability. VisualWorks is a Smalltalk environment, and these environments use dynamic compilation. This means that systems can be modified "on the fly", since refactorings are behavior preserving, and evolutions should never reduce the functionalities offered by a system (so as to maintain compatibility with previous versions). This tool is highly recommended for projects that need to evolve dynamically.

Activeness. This tool is completely *reactive*, since it expects the user to initiate the refactorings and to indicate where these should be applied, and with what parameters. All user interaction with the browser is through menus and dialog boxes. The browser does not attempt to better the design by inspecting the static code or system at runtime. All the information that VisualWorks collects is just used to determine if a refactoring is legal or not.

Openness. VisualWorks is implemented in Smalltalk. This means that it can be modified by using the reflective capabilities of Smalltalk. The authors of this tool developed a framework that permits the definition of new refactorings (based on existing ones). This framework is available, but the authors recognize that it is difficult, if not impossible, to use this framework without extensive knowledge about the theory on which refactorings are based. They propose as future work the further development of this part of the browser, for example, providing a GUI that permits refactoring definition.

3.6 Change Process

Plan. The browser is not integrated with the software development process, so it can be used in planned or unplanned change processes. However, the type of development that would most benefit from such a tool are Evolutionary Systems (*an unplanned change process*), for example, Extreme Programming [9]. This is because of all the dynamic capabilities that the tool has to offer, like dynamic checking and dynamic compilation.

Control. This tool knows exactly what conditions the system complies with, after a refactoring is applied, through the dynamic analysis of the program at runtime, the calculation of pre and

post conditions and the knowledge stored in a database of known facts. This means that the systems that are modified using this tool, go through a *controlled evolutionary process.*

Measure. VisualWorks does not include metrics of any kind. To obtain any information about the quality or modularity of the code produced by refactoring a system, it is necessary to use some tool that has this capability. This would be extremely useful, since it is not always clear what refactorings should be applied. Metrics about quality the resulting program would help the user form a better idea about the results of the refactorings that where applied.

4 Eclipse 2.0

The Eclipse project has been designed for building Integrated Development Environments (IDEs), that can be used to create applications as diverse as Web Sites, Java programs, C++ programs, etc. It has an open, extensible architecture based on plug-ins. The Eclipse SDK has a great amount of built-in functionalities, which are mostly generic. Additional tools are needed to extend the platform to work with new types. The Eclipse SDK is distributed with two important tools:

- Plug-in Development Environment (PDE)
- Java Development Tool (JDT)

The PDE is used to create new plug-ins that extend the Eclipse SDK. The JDT is a state of the art Java IDE that provides, as one of its abilities, Java code refactoring. The taxonomy is applied to the JDT, since this is the tool that permits the creation and evolution of Java projects.

4.1 Temporal Changes

Time of Change. The JDT is only concerned with the coding process in a software development effort, since it is a Java IDE. For this reason, it does not place restrictions on T_1. The user is usually the developer or maintainer of the system, and after he receives the request for a change, the advanced code editor available in the JDT helps him prepare and execute this change (T_2). The JDT uses the incremental project build capabilities offered by the Eclipse Platform, so as to have an incremental Java compiler. A project is only fully compiled once (the first time it is compiled) and the rest of the times, only partial recompilations are done. This makes T_3 much shorter than what is observed in traditional Java compilers. Once the compilation is finished (marking the beginning of T_4) the result is an executable program. The JDT Java compiler produces executable code, even when problems are found in the code. The tool's range of action only embarks T_2 and T_3. The refactorings offered by the JDT can applied to a system once the changes that have been requested, have *already* been incorporated into the system. The restructuring and refactoring process is applied at an arbitrary moment of time, but not during the change process. This means that the tool has no control over T_1 (the time when a change is requested), T_2 (the time when a change is prepared and T_3 (the time when a change becomes available for execution). Once T_3 has finalized, the tool can be applied to the system, that now includes the modifications needed so as to include the requested changes. The refactorings offered by the tool could also have been applied before T_1 or T_2, in the expectation of future changes being requested. Once the refactoring process has concluded, the changes can be executed for the first time.

Change History. Each user has a workspace, that has one or more top-level projects. Each project has a project nature, that gives it a particular personality (eg. a Java project). Projects contain regular files. The projects, files and folders that are found in a workspace, are called the workspace *resources.* Eclipse adds version and configuration management capabilities to all the projects that are in a workspace. Repositories store version-managed resources on shared servers. One workspace can access different types of repositories simultaneously. A stream maintains the shared configuration of one or more related projects for a development team. A same project can be one more than one stream, and these streams can evolve in a parallel form. This means that

software evolution in Eclipse may be carried out in a *parallel* fashion. Since Eclipse is a local application, each user works in his own workspace, so Eclipse has *parallel asynchronous changes*. All work must then be synchronized, so changes in Eclipse are *convergent*.

Change Frequency. The developers of the JDT have tried to develop an IDE for Java where it is possible to build large systems without expensive system recompilations. The ability to maintain relatively dynamical information is needed to be able to accomplish this purpose. This led to the development of the Java Model, a model used to represent Java program elements in a tree structure. A Java compiler was developed, that uses the tree structure generated by the Java Model to make *incremental project builds*.[3] This means that it is cheap to frequently build the system, since only the parts of the project that have suffered modification get recompiled. This means that the tool supports *continuous* changes.

4.2 Drivers of Change

Role of the user. Eclipse is "code oriented", that is, its main concern is the development and maintenance of source code. For this reason, the main user of the JDT is the system developer or maintainer. User requested functions can lead to code cleanup or system redesign, so as to be able to include modifications without making the system corrupt and brittle. Eclipse includes testing suites, for example, in the JDT, JUnit testing can be used. This means that the testers can also be users of this tool. Another type of user are plug-in writers, that use the PDE to extend the functionalities offered by Eclipse.

Distribution of users. The tool is a local application. This means that each person in a development team has his own workspace, accessing a local copy of the resources. This workspace is not directly accessible by the rest of the team members. This is solved by the use of repositories (see Section 4.1 - Change History). A team member can only see the changes submitted by other team members when he synchronizes his workspace with that stored in the project stream, in the repository. The repository type supports an optimistic concurrency model, so conflicting incoming and outgoing changes are detected automatically. The developer must resolve this conflict. This means that the tool supports a restricted form of *distribution*. Some new plug-ins have be created to better this aspect.

Degree of Automation. The JDT is a partially automated tool. The user has to use a Refactoring Wizard to pick which refactorings to apply to the source code, with what parameters. The preconditions and effects are calculated and applied in an automatic fashion. The user input is used to make the dynamic analysis of conditions quicker, by supplying things that are hard to calculate. It also helps maintain the source code readable, since no class, method or variable names are automatically generated. This tool does not automate the decision of which refactorings to apply, that is a user decision. It also adds features like code completion to the code editor. The JDT determines legal code completions by using the Java Compiler.

4.3 Facets of Change

Type of Change. The JDT refactorings are all classified as actions that preserve program semantics. This means that all changes that the Refactoring Wizard applies to the source code are *structural* changes. The refactorings do not alter what the program does, they just affect the form in which the program does it. The user has the freedom to insert *semantic* changes at any moment, since this tool is not just a refactoring tool, but a complete IDE.

Effect of Change. The majority of the refactorings included in the JDT refer directly to the renaming or movement of program elements: classes, methods and fields. There are more refactorings available (refer to Table 2 to see the current list[4] of refactorings available for Eclipse 2.0). All these refactorings are behavior preserving. The effect of change associated to the IDE involves directly all element manipulations: element addition, removal or modification. These activities correspond to the basic actions used by the programmer to write applications.

[3] Using dependency graphs that are calculated from the information saved using the Java Model

[4] Since Eclipse is always growing, through plug-ins, this list is subject to modification.

Rename(*field, method, class, package*)
Move(*static field, static method, class*)
Extract(*local variable, method*)
Organize imports
Inline local variables
Reorder method parameters

Table 2: The current list of refactorings available for Eclipse 2.0

Invasiveness. The Eclipse JDT is a Java IDE, so it allows the user to modify the existing source code directly. However, the changes made to the code through it, are *not destructive*, because of the version and configuration management that JDT offers. This means that the modifications made by the user are invasive, but that the tool assimilates them as non-invasive changes, by preserving old unmodified versions in the repositories. The tool does provide the ability to *undo* refactorings.

Safety. Since programs must evolve through the repository capabilities offered by this tool, it must have some sort of version compatibility checking when users synchronize changes to a shared team repository. The version and configuration management system used by the JDT assumes an optimistic concurrency model, so conflicting incoming and outgoing changes are detected automatically. The developer must resolve this conflict. The users also have at their disposal JUnit testing and a Java Debugger, as well as other testing plug-ins. This debugger supports dynamic class reloading, when the underlying Java virtual machine does.

Effort. The main objective of the tool is to create and evolve Java code. To do this, the user has at his disposal the Java Editor, an advanced editor that makes use of the Java nature that was assigned to the project. It can color keywords and syntax, make marginal annotations, format code, propose code completion and can edit code at a method level (as well as the usual source file level). Refactorings are applied using the Refactoring Wizard, which allows to pick the refactoring that will be applied and with what parameters. Before finalizing the refactoring, the proposed code modifications are showed to the user, and he can either accept these changes without modifications, or modify them and then accept them or simply not accept them at all. The refactorings are presented in such a way that the user does not need extensive knowledge about the refactoring theory to be able to apply them to his code. All this contributes to make the tool a *low* effort tool.

4.4 Object of Change

Artifact. The only artifact that the JDT modifies, is the system *source code*. This is because it is an IDE, that includes a code editor, compiler and debugger and a refactoring wizard. But projects are allowed to contain files that are not associated to its nature, for example, design documents can be kept in the project directory. Even so, if all the artifacts involved in a software development effort are to be kept compatible, it must be done in a manual fashion, by the development team. This is due to the fact that the synchronization options provided by the version and configuration management system only apply to Java related files.

Granularity. As mentioned in "Effects of Change" (see Section 4.3), the basic refactorings affect packages, classes, methods and variables. Since the JDT is also an IDE, changes of any granularity are accepted. Using the classification mentioned in the taxonomy, the JDT has *coarse* (package manipulation), *medium* (class manipulation) and *fine* (method and variable manipulations) granularity.

Locality. The JDT IDE code editor is a tool used to make *local* changes to a software. Code can be edited at a source file or method level. The IDE offers a *Refactoring Wizard*, that allows the user to make *global* changes (at package, class, method or field level) by applying refactorings. All global changes are semi-automated, the JDT modifies the source code, instead of the user.

Scope. The JDT is not integrated with the software development process, it does not coordinate its activities and products with those produced by the other phases in the development process. Only the source code of the project is modified in the editor or by the *Refactoring Wizard*. All changes to the other artifacts produced during the software development, must be done manually, unless plug-ins that support the modification of these artifacts are created. At the moment, some support plug-ins exist, but none that really coordinate and actualize all the different artifacts that exist in projects.

4.5 System Facets

Availability. Java is a static object-oriented programming language, that has a traditional write-compile-debug-execute life cycle. This is being changed by the JDT. As mentioned earlier (see Section 4.1 - Change Frequency) the Eclipse tool provides incremental project build support, and this is used by the JDT Java compiler to incrementally build Java projects. This compiler even generates executables when the compiler has found problems while compiling. The debugger permits dynamic class reloading, if the Java virtual machine used has Hot Swap capabilities. But systems must still be restarted so that changes can become effective. This means that the system evolution offered by the JDT is *static*.

Activeness. This tool is completely *reactive*, since it is an IDE. It also expects that the user initiate the refactorings and to indicate where these should be applied, and with what parameters. All user interaction with the IDE is through the Java editor. The JDT does not attempt to better the design by inspecting the static code or system at runtime. It just uses the information gathered by the Java compiler (using the Java Model) to suggest completions, or to make corrections, for example, suggest keyword completion or correct the spelling of keywords.

Openness. The Eclipse Platform has been designed so as to have an open, extensible architecture. This architecture is based on plug-in technology. The *Eclipse SDK* has a great amount of built-in functionalities, which are mostly generic. Additional tools (plug-ins) are needed to extend the platform to work with new types. The *Eclipse SDK* is distributed with two tools: the *Plug-in Development Environment (PDE)* and the *Java Development Tool (JDT)*. The PDE is used to create new plug-ins that extend the *Eclipse SDK*. The JDT is a state-of-the-art Java IDE. Since the JDT is build as a plug-in to the base Eclipse platform, it is also extensible through plug-ins. Some plug-ins for the JDT already exist, for example, testing and database plug-ins. A comprehensive list of the plug-ins available for the Eclipse platform is available on the Eclipse project website.

4.6 Change Process

Plan. The IDE is not integrated with the software development process, so it can be used in planned or unplanned change processes. However, the creators of the tool indicate that this tool encourages exploratory programming. This is because of all the incremental capabilities that the tool has to offer, like incremental compilation. There are plug-ins that permit a better coordination amongst a distributed team of development, so as to allow the communication of the existence of new releases and the necessity for team synchronization. This allows a development team to follow a *planned* change process, but, by default, the change process is *unplanned*.

Control. The tool has no knowledge about the state of the system, after the user manually modifies the project source code. As an IDE, it uses advanced information to be able to offer more complex editing options to the user, by incrementally compiling the code and generating dependency graphs with this information. However, the tool has no real "understanding" of this information. It is not possible to know all the code combinations that the user can input. The JDT does have controlled evolution, in the application refactoring context, since refactorings are defined as *behavior preserving* actions. So, this tool presents both types of evolution: *uncontrolled* in the IDE context, and *controlled* in the refactoring context.

Measure. The JDT does not include metrics of any kind. This should not be considered as a serious draw back since, by using the Eclipse plug-in capabilities, the JDT can be extended so

as to include these functionalities. The obvious choice of metric would be code quality metrics because the JDT is not integrated with the software development process. At the Eclipse Platform level, it should be possible to add plug-ins with more general metrics, that include different types of projects, for example UML projects [5] that interact with code projects, like Java projects.

5 Guru (for SELF 4.0)

Guru is a refactoring tool developed by Ivan Moore, for the SELF programming language. This language is a prototypical object-oriented language, so there is no differentiation between class and instance objects. Objects are related by inheritance relationships. This tool is used for restructuring inheritance hierarchies and refactoring methods of SELF programs, in a fully automatic manner.[6] The restructuring process is supposed to be behavior preserving.[7] This is done by maximizing the sharing features and by refactoring shared expressions into new methods. This betters the inheritance structures that exist in the system, and new abstractions can be found.

5.1 Temporal Changes

Time of Change. Guru is applied to a system once the changes that have been requested, have *already* been incorporated into the system. The restructuring and refactoring process is applied at an arbitrary moment of time, but not during the change process. This means that the tool has no control over T_1 (the time when a change is requested), T_2 (the time when a change is prepared and T_3 (the time when a change becomes available for execution). Once T_3 has finalized, the tool can be applied to the system, that now includes the modifications needed so as to include the requested changes.[8] If the tool is not applied to the system, then it is immediately available for execution. If the tool is applied, there is a time delay before the system can be executed. This delay can be quite considerable, depending on the amount of objects included in the restructuring process. Examples cited by Moore in his work indicate time intervals of up to 8 hours - this is attributed to the fact that the restructuring and refactoring processes occur simultaneously, and also to the current implementation of the tool. Once the restructuring process has concluded, the changes can be executed for the first time.

Change History. This tool is applied directly to the source code that one wants to better, design-wise. Guru gathers all the information necessary for the restructuring from the existing source code. Once it has determined the new object inheritance hierarchy and the new methods that will be added, the source code is modified. This is done automatically by obtaining the information that Guru has saved in parse trees, so existing code is replaced with generated code. Guru, as a tool, is *unversioned*.

If the user has not saved his source code, by either having an independent CVS or simply by making a copy of the file, his original code is lost. No undo operation is mentioned in the Guru literature.

Change Frequency. Guru is an automatic tool, so no user interaction is needed. This allows the *frequent* application of the tool, with a minimal effort in programming hours during the restructuring process. The user only has to indicate which objects are to be considered by the process. The author recommends that this tool be used after each design change. There are, however, two important restrictions that limit the frequency of application of the tool. The first one is the duration of the process - it depends largely on the quality of the current inheritance hierarchy and the amount of objects that are to be considered in the restructuring process. The second limitation is that since the tool is automatic, automatically generated code is inserted in the source code, with automatically generated object and method names. This makes the code

[5] Rational Corp. is studying the development of a UML plug-in for Eclipse.

[6] From this point on, the restructuring - refactoring process will be referred to just as the *restructuring process* (refactoring is implied)

[7] There are no formal demonstrations that indicate that Moore's algorithms are behavior preserving, but experimental results have been favorable

[8] The tool could also have been applied before T_1 or T_2, in the expectation of future changes being requested.

less understandable, specially if many automatic restructurings are applied in a rapid succession. The tool does permit the renaming of automatically generated names, replacing the method calls automatically.

5.2 Drivers of Change

Role of the user. The tool has been developed as a support tool for the SELF programmer. The main user of the Guru tool is the system developer or maintainer. This is because of the automatically generated code that is inserted into the original source code; the person that uses the tool has to know the system structure quite well, so as to understand the changes that were introduced in the inheritance hierarchy, as well as the new methods that were generated. Since all newly introduced elements have automatically generated names, it is recommended that these names be replaced by more descriptive names, so as to contribute with the maintenance of the system.

Distribution of users. Guru is applied by the system developer (or maintainer) directly to his local copy of the system source code. This means that the tool makes only *local* changes. The tool has to examine the entire SELF environment when applying changes, so as to be certain that the automatically generated names are unique in the environment. This makes it a bad candidate for a future distributed implementation, as this would mean the necessary introduction of more checks, thus introducing more time delays in the tool execution.

Degree of Automation. This tool is 100 % *automated*. It requires user intervention to determine when it should be applied, and to indicate which objects must be considered by it. Once Guru has started the restructuring process, it is fully automated. It generates a new inheritance hierarchy and refactors the methods, from the information that is gathered from the source code. This means that it generates code automatically, as well as new object and method names. It then proceeds to replace the original code with generated code, where necessary.

5.3 Facets of Change

Type of Change. The algorithms used in the implementation of Guru are supposed to be *behavior preserving*. There is no formal proof of this fact. However, practical demonstrations, that included both code restructuring and code refactoring, have shown that the system behavior has not been altered. This is specially true in the examples presented by Moore. He used Guru to restructure the inheritance hierarchies of fundamental classes of the SELF environment. The environment presented no apparent change in behavior. This is not a formal proof of behavior conservation, but it is a good result. Therefore, the changes introduced by Guru are *structural* changes.

Effect of Change. Given a certain set of classes to restructure, Guru can *add*, *subtract* or *modify* elements, where elements are objects or methods. Instance variables are treated as a set of two methods (the corresponding accesor and mutator methods[9]) in the SELF language, so the Guru tool includes variable manipulation through method manipulation. Methods are refactored by moving common expressions to separate methods, and invoking them there. Guru restructures the system, so as to reach an "optimal" state that preserves the program behavior. Optimal is taken to mean that duplicate methods are removed, method sharing is maximized, and redefinition of methods is avoided.

Invasiveness. Guru replaces original system code with automatically generated code. This makes the tool *highly invasive*. The original method code is ignored (with its corresponding format and documentation) and is totally replaced by expressions extracted from the tool's parse tree. There are also problems with the automatically generated code - it presents an excess of brackets. The tool does not provide versioning mechanisms (see Section 5.1 - Change History). The user must provide his own versioning capabilities.

Safety. Only practical demonstrations have shown that Guru is behavior preserving. These demonstrations have involved crucial parts of the SELF environment, applying restructuring to

[9]Read-only instance variables only have an accesor method

fundamental classes (see Section 5.3 - Type of Change). This means that the system should be *backward compatible*. The literature on the tool does not indicate that any checks (static or dynamic) are applied.

Effort. The changes made by Guru are high impact changes, since potentially, the whole system structure can change. This is not usually the case. In most cases, only a few classes are added. On the other hand, quite a lot of common expressions are factored out of the methods, and are included in new methods. But since the Guru is an automated tool[10], these changes are done automatically by the tool. The user is liberated from all responsibility. The tool even provides a method renaming function. This lets the user easily rename automatically named methods and classes, by specifying only the desired name. Guru will do all the name replacements and message actualizations. This makes Guru a *low effort* tool.

5.4 Object of Change

Artifact. The only artifact that Guru modifies, is the system *source code*. This is because it is an automatic source code optimizer. It only relies on the information stored in the source code to determine an optimal[11] design. All compatibility with other software artifacts must be introduce manually, by the software development team.

Granularity. As mentioned in "Effect of Change" (see Section 5.3), the restructuring and refactoring process affects objects and methods (variables are included as methods - a SELF property). Using the classification mentioned in the taxonomy, Guru has *medium* (object restructuring) to *fine* (method and variable refactorings) granularity. Since the tool also performs refactoring of entire inheritance hierarchies, it can also be classified as coarse-grained.

Locality. Changes are *global* within the set of objects being optimized (these objects are not necessarily related through inheritance relationships). Moreover, changes in a set of objects might mean that objects outside this set have to be modified as well, since the new inheritance hierarchy is generated by flattening the existing inheritance hierarchy. Methods change name, so all the messages that refer to a certain method have to reflect this change. This means that changes could embark the whole SELF environment, and not only the system to which the studied objects belong.

Scope. Guru is not integrated with the software development process, it does not coordinate its activities and products with those produced by the other phases in the development process. Only the source code of the project is modified, so as to include the automatically generated code. All changes to the other artifacts produced during the software development, must be introduced manually by the development team.

5.5 System Facets

Availability. The tool is used once the requested changes have been incorporated into the system. When the tool is applied to restructure a system, the optimization process could take hours to complete, based on the amount of objects considered in the restructuring process. After the modified system has been obtained, it could suffer more changes, for example, the user could rename the automatically generated methods. For these reasons, it can be said that Guru supports *static software evolution* processes.

Activeness. Guru requires user intervention to determine when it should be applied, and to indicate which objects must be considered by it. Once Guru has started the restructuring process, it is fully automated. The system modifications are determined through two main algorithms, one that does the automatic inheritance hierarchy restructuring and the other, the automatic method refactoring. The tool just executes the algorithms, it does not make decisions as to which refactorings should be applied. Moore proposes, as future work, the development of more refactoring algorithms, and that the tool must decide which algorithm should be applied, given some conditions and a context. This means that, currently, the tool is *reactive*.

[10] After the initial user interaction.

[11] See Section 5.3 - Effect of Change

Openness. The Guru tool is implemented in SELF, and it is the author's opinion that good object oriented design principals were used during the development of the tool. This was demonstrated[12] by applying the Guru tool on its own structure. Guru produced similar inheritance structures as those developed. This means that the tool should be relatively easy to extend, if the person introducing the tool extensions has a good knowledge of the existing system structure, the theory behind the algorithms employed by Moore and the SELF language. Since it might be difficult to meet all three conditions, this tool is classified as *partially open*.

5.6 Change Process

Plan. The tool is not integrated with the software development process, so it can be used in planned or unplanned change processes. This tool is used for restructuring inheritance hierarchies and refactoring methods of SELF programs, in a fully automatic manner. This is done by maximizing the sharing features and by refactoring shared expressions into new methods. This means that the form of the output of the changes is always the same: there is no code duplication. This can be considered as a *planned change*.

Control. The tool is limited to the application of some predetermined restructuring rules. It has no knowledge of the runtime state of the system, it only works from the information that is in the source code. The generated restructuring and refactoring only reflect what exists implicitly in the old system structure. This is not always what *should* exist in the system. The algorithms that are used guarantee[13] that the resulting system has no duplicated methods. This means that the systems that are modified using this tool, go through a *controlled evolutionary process*.

Measure. Guru has no design quality metrics incorporated, but Moore suggests two possible metrics. These are *Message sends* and *Overriding methods*.[14] The first one refers to the total number of potential message sends in all the methods, in all the classes that belong to the restructured inheritance hierarchy. This metric represents the source code "length", since in SELF, it is more informative to know how many potential messages are going to be sent, than how many lines of code there are. Also, since one of the conditions of the restructuring is that no methods or expressions are duplicated, the resulting structure will always have less lines of code than the original code, even if the resulting design is worse. Less potential messages means a better design, since the methods are well distributed within the inheritance hierarchy. The second metric, Overriding methods, refers to the number of methods which override other methods from inside and outside the classes that were restructured. If there is too much method overriding, the design is of poor quality. Currently, these metrics are not incorporated in Guru.

6 The Together ControlCenter 6.0

The ControlCenter [5] application development tool has been created by TogetherSoft, with the specific purpose of streamlining the software development process. Application modeling is used to ensure that a system meets the business needs. The ControlCenter provides a closer synchronization between the application design and its transformation into code. As a single development environment, it covers various phases of the software development life cycle, from design, to coding, to testing, to final deployment. Usually, when a complex IT infrastructure is developed, multiple platforms and tools are used by multiple participants. This is simplified by the ControlCenter, that has a collaborative application development approach. It is a unified environment for application design, development and deployment. Additionally, it supports software evolution, by providing the developer with source code refactorings. The need for a tool to support this is clear in this product; its previous version only offered two refactorings, while this one offers ten additional refactorings, for a total of twelve refactorings.

[12] In practice

[13] Again, no formal demonstration exists.

[14] These metrics are only applicable to OO systems, like those developed in SELF, since they are based on OO properties.

6.1 Temporal Changes

Time of Change. The ControlCenter is concerned with various phases of the software development cycle, and their corresponding artifact synchronization. When changes are requested, usually by the requirement analysts, these have to be communicated to the system designer (T_1). The normal (and recommended) form of proceeding is that the system designer include the changes in the system design diagrams. This means that the corresponding code is synchronized with the altered design, so the change is partially incorporated into the source code. Now it is the responsibility of the system developer or maintainer to finish adding the change to the system. These two steps represent T_2. It is also possible for the developer to incorporate the changes directly, without passing through the designer. This is not a problem, since the system design will be automatically synchronized with the source code modifications. Once the changes have been prepared, the system is compiled and tested (T_3). The environment provides a numerous set of audits for this purpose. Finally, the system can be executed for the first time (T_4). The refactorings offered by ControlCenter can applied to a system once the changes that have been requested, have *already* been incorporated into the system. The restructuring and refactoring process is applied at an arbitrary moment of time, but not during the change process. This means that the refactorings have no control over T_1 (the time when a change is requested), T_2 (the time when a change is prepared) and T_3 (the time when a change becomes available for execution). Once T_3 has finalized, reafctorings can be applied to the system, that now includes the modifications needed so as to include the requested changes. The refactorings offered by the tool could also have been applied before T_1 or T_2, in the expectation of future changes being requested. Once the refactoring process has concluded, the changes can be executed.

Change History. The tool has fully integrated version control support. It supports a wide range of versioning tools. The ControlCenter has a file-based architecture, so as to include non-ASCII files, like documentation and diagrams, in the historic repositories. To support versioning of these non-ASCII files, any file-based versioning system is supported. The tool is sold preconfigured for the CVS VS 1.11, which is bundled with the environment. An added bonus is that the ControlCenter has a graphical CVS interface. All this means that the tool is *versioned*. The characteristics of the software evolution depend directly on the versioning software used, but most versioning systems permit at least parallel divergent changes.

Change Frequency. The ControlCenter acts a project artifact manage, with automatic synchronization between artifacts. Since its main use is as a system development environment, it should support frequent changes. This is also true when used as a maintenance tool. Changes can be applied to the system in a *continuous* manner, as they are required, since the automatic synchronization of artifacts will provide artifact consistency. This makes it an ideal tool for performing refactorings, everytime the requirements change. Rollback onto previous versions is done through stored versions in the CVS repository.

6.2 Drivers of Change

Role of the user. This tool simplifies and integrates the analysis, design, implementation, deployment and debugging phases of complex applications. This means that the users are the team members involved in these phases, with their predefined roles, like analysts, designers, developers and testers. The analysts and designers use the modeling capabilities provided by the ControlCenter. This includes full UML support for system analysis and design. The developers (or maintainers) and testers use the development capabilities offered by the tool, such as the advanced code editor, debugger or audits (like unit testing). The refactoring capabilities will only be used by the *developers*. A function provided by the tool to this effect, is workspace configuration by specifying the user's role in the development process. This affects which views are visible by default.

Distribution of users. The tool is a local application. This means that each person in a development team has his own workspace, accessing a local copy of the resources. This workspace is not directly accessible by the rest of the team members. This is solved by the use of repositories

(see Section 6.1 - Change History). A team member can only see the changes submitted by other team members when he synchronizes his workspace with that stored in the project stream, in the repository.

Degree of Automation. The ControlCenter can be used as a partially automated refactoring tool. The user has to use a refactoring menu to pick which refactorings to apply to the source code, with what parameters. The preconditions and effects are calculated and applied in an automatic fashion. The tool checks that all the changes, introduced by the refactorings, are correctly propagated. The user input is used to make the dynamic analysis of conditions quicker, by supplying things that are hard to calculate. It also helps maintain the source code readable, since no class, method or variable names are automatically generated. This tool does not automate the decision of which refactorings to apply, that is a user decision. It also adds features like code completion to the code editor.

6.3 Facets of Change

Type of Change. The ControlCenter refactorings are all classified as actions that preserve program semantics. This means that all changes that are applied by the refactorings to the source code are *structural* changes. The refactorings do not alter what the program does, they just affect the form in which the program does it. The user has the freedom to insert *semantic* changes at any moment, since this tool is not just a refactoring tool, but a development environment that includes a complete IDE. Semantic changes are also introduced into the source code (automatically), when the system design is modified. This last type of changes have to be completed by the system developer.

Effect of Change. The majority of the refactorings included in the ControlCenter refer directly to the renaming or movement of program elements: classes, methods and variables. The reafactorings affect the source code directly, but modifyinh the UML design will also introduce changes into the code. There are more refactorings available (refer to Table 3 to see the list of refactorings available for the TogetherSoft ControlCenter 6.0). All these refactorings are behavior preserving. Additionally, the developer can ask the system to show a list of derived classes/interfaces, base classes, implementing classes, or overrides to clarify a refactoring. The effect of change associated to the IDE part of the tool involves directly all element manipulations: element addition, removal or modification. These activities correspond to the basic actions used by the programmer to write applications.

Invasiveness. The ContolCenter has as a Java IDE[15], so it allows the user to modify the existing source code directly. However, the changes made to the code through it, are *not destructive*, because of the version and configuration management that the ControlCenter offers. This means that the modifications made by the user are invasive, but that the tool assimilates them as non-invasive changes, by preserving old unmodified versions in the repositories. The tool does provide the ability to *undo* refactorings.

Safety. Version compatibility checking must be implemented in the CVS used by the development team, since it is not offered by the environment. When refactorings are applied, the tool checks that all the changes, introduced by the refactorings, are correctly propagated. This ensures safety of evolution through refactoring. Additionally, all artifacts produced by the tool are kept synchronized, so there is no worry of design and source code mismatch. The documentation is generated from the information that is found in the project. This means that the documentation always corresponds to the real state of the project. The users also have at their disposal JUnit testing and a Java Debugger, as well as other auditing tools.

Effort. The main objective of the tool is to develop complex information systems. This is done by supporting the different phases of the development process. Each user type has at his disposal various tools that permit him to do his work in a fast and easy way, and also coordinated with the rest of the participants in the process. The analysts and designers use the UML modeling capabilities offered by the tool. The developers use an advanced code editor. The testers use the

[15]C++ support is also included, but Java is the main language

Refactoring	Effect
Move class	Moves the selected class to a different package
Pull up attribute/operation	Moves the selected attribute or operation to its superclass
Push down attribute/operation	Moves the selected attribute or operation to its subclass
Encapsulate attribute	Makes the selected attribute private, provides accessors, and changes all usages in the project
Self-encapsulate attribute	Creates getting and setting operations and uses only those operations to access the attribute
Extract interface	Creates a new interface from one or more selected classes
Extract superclass	Creates an ancestor class for selected operations and moves the operations to the new ancestor class
Renaming in method bodies	Renaming for operation parameters, local variables, and properties, and optionally rename usages in the project
Extract method	Extracts a method from a class

Table 3: The list of refactorings available for the TogetherSoft ControlCenter 6.0, with the effect produced per refactoring

testing framework provided by the tool. Refactorings are applied using a refactoring wizard, which allows to pick the refactoring that will be applied and with what parameters. The refactorings are presented in such a way that the user does not need extensive knowledge about the refactoring theory to be able to apply them to his code. This makes it a *low effort* tool. Even the documentation is generated automatically, and document styles can be created using a wizard.

6.4 Object of Change

Artifact. The ControlCenter modifies the system *design*, *source code*, *tests* and *documentation*. This is because it is a complete system development environment, that includes a UML modeling tool, a code editor, a compiler, a debugger, a refactoring wizard, a testing framework, a documentation generator and a deployment wizard. The main artifacts are kept coordinated by automatic synchronization.

Granularity. As mentioned in "Effects of Change" (see Section 6.3), the basic refactorings affect packages, classes, methods and variables. Since the ControlCenter also includes an IDE, changes of any granularity are accepted. Other artifacts that can be changed are the system design and documentation. Using the classification mentioned in the taxonomy, the ControlCenter has *coarse* (design, documentation and package manipulation), *medium* (class manipulation) and *fine* (method and variable manipulations) granularity.

Locality. The IDE part of the tool offers a Refactoring Wizard, that allows the user to make *global* changes to the source code directly (at package, class, method or variable level), by applying refactorings.

Scope. The ControlCenter is the tool on the market that currently offers the highest level of integration of software artifacts, which are produced by the different phases of the software development process. This is done by integrating the different phases into one tool. Each phase is

represented by its main activities and artifacts. In this way, all the phases "speak" a same common language. The artifacts are kept consistent with the changes by automatic synchronization.

6.5 System Facets

Availability. Java is a static OOL, that has a traditional write-compile-debug-execute life cycle. The ControlCenter has a deployment automation expert, that allows the application to be deployed to a server configuration easily. These server configurations are determined by the development team, through visual deployment diagrams. This makes it easier to make the system available, but it does not change the fact that the tool offers *static* system evolution.

Activeness. This tool is completely *reactive*. The users must enter all the input that corresponds to the software design and source code. It also expects that the user initiate the refactorings and to indicate where these should be applied, and with what parameters. The tool does not attempt to better the design by inspecting the static code or system at runtime. It just uses the information gathered by the Java compiler to suggest completions, or to make corrections, for example, suggest keyword completion or correct the spelling of keywords.

Openness. The ControlCenter has been designed so as to have an open, extensible architecture. Third-party tools can be integrated with the ControlCenter using the Integration API, or by using the Integration Wizards provided by the tool. It is easier to incorporate third-party tools through the Wizards, than by using the API. TogetherSoft provides a list of compatible third-party tools.

6.6 Change Process

Plan. This tool is highly integrated with the software development process. It offers a set of metrics to measure the system quality. The ControlCenter does not, however, include any project management tools. This is the only main activity missing from the tool, since Analysis, Design, Implementation, Testing and Installation activities are all present in some form. Since the artifacts produced by a software are automatically synchronized, this tool is recommended for developments that need to follow a *unplanned change process*, so as to make use of the extensive functionalities offered by the tool, like automatic synchronization.

Control. The tool has no knowledge about the state of the system, after the user manually modifies the project source code. It only actualizes the information of the other artifacts, so that all the artifacts are kept synchronized. This is a form of partial control. The ControlCenter present controlled evolution, in the application refactoring context, since refactorings are defined as *behavior preserving* actions. So, this tool presents both types of evolution: *uncontrolled* in the IDE context, and *controlled* in the refactoring context.

Measure. The ControlCenter includes a Quality Assurance module. This module has audits and metrics. Some of the metrics that exist for Java are: cohesion, coupling, complexity, and volume/size. These can all be classified as *software change impact analysis* metrics. The user can define his own custom metrics, using the framework provided by the tool. The ControlCenter also has a Quality Assurance expert tool. It permits the generation of web pages, with the results of the metrics in a standard tabular form, or a graphical form. Distribution graphs for the metric results can also be requested, with an advice panel that displays tips about the high and low metric values. Metric results can be saved and later reloaded. The tool also permits the comparison of metric results.

7 Tool Comparison

The software evolution tools that were evaluated, were picked because of the apparent differences that existed between them (see Section 2 - Application of the Taxonomy). The idea that these tools did not share any characteristics was a subjective opinion, based on previous knowledge, and the fact that these tools are oriented towards different languages and purposes. *VisualWorks* refactors

Smalltalk code. Smalltalk is a dynamic, class based object-oriented language. *Eclipse JDT* is a Java IDE that permits refactoring. Java, unlike Smalltalk, is a static, class based object-oriented language. *Guru* restructures and refactors SELF code. SELF is a prototypical object-oriented language. Finally, *ControlCenter* manages a great part of the production process, and has a Java IDE that permits refactoring.

We will now be able to see if the apparent differences that exist between these tools are "real". In other words, was our first impression correct? The other thing we can notice, before comparing, is that three of the tools are IDEs that permit refactoring, while the last one (Guru) is a stand-alone refactoring tool. Will this relationship be visible in the comparison?

After looking at Table 4, the tools share many of the characteristics. It is now possible to see that most of the tools compared depend on external versioning systems. This could be explained by the variety of versioning tools that exist on the market, like *CVS*, *StarTeam*, *Microsoft Visual SourceSafe*, *Rational ClearCase*, *PVCS*, *Perforce*, etc. All the evaluated tools target the developer as the main user. The evaluated tools are all local applications. The tools all show some amount of automation, but all act as support tools in the development process (they are reactive tools). With the advances in user interfaces, these tools are more user friendly, which makes them easier to use. Applications are no longer built with the following lemma in mind: "if the user wants to use it, he will have to learn". The tools have approached the user, and not the other way round. All the tools manage at least medium and fine elements. They all make global changes.

All these similarities show that the differences that were initially seen as fundamental, were not. This is because the initial differences were based on the tool problem domain, and not the tool characteristics. The use of the taxonomy made this error of differentiation clear.

There is, however, a tool that does differ somewhat from the others, namely *Together ControlCenter 6.0*. It is the only tool evaluated that includes more phases of the development process, other than the implementation. Its use of automatic artifact synchronization should speed up the development process. This is important, since project development is no longer just a question of source code development, but of the necessity of a fast development phase, with a resulting software that is reasonably stable and that can be mass produced.

Guru also differs from the others in that it is fully automated, and not really integrated in an IDE.

VisualWorks is the only evaluated tool that is fully interactive. It requests information from the user, on a "need to know" basis. The other tools only use the information given by the user when applying a refactoring. The only one of these three that gives some freedom in this aspect, is JDT, since it permits user modification of the output produced by a refactoring, before accepting it.

The ControlCenter is the tool that is less open, since it is a commercial tool. It is extended through third-party tools. This makes the evolution of the tool more difficult, since TogetherSoft has no say in the development of third-party tools. On the other hand, the other tools are all open in some form: the source code of VisualWorks (through reflection) and Guru are available; Eclipse is extended through plug-ins.

8 Conclusion

The taxonomy developed by Mens et. al [1] was successfully applied to four software tools that support refactoring in the evolutionary software development process. The differences that were first identified between the tools, that motivated the tool selection, were found to be minor differences, after applying the taxonomy. The reason for this is that the initial differences were based on the tool problem domains. While the tool problem domains were different, the tools shared many characteristics, like change history, automation, user, distribution, etc.

By observing the comparison table (see Table 4), one can see that the IDE tools share more characteristics, than an IDE tool with a refactoring tool. The same can be said about the characteristics of the refactoring tools. This result was expected. If this had not happened, it would have probably meant that the taxonomy was badly defined.

The tool that manages to differentiate itself from the tools that were evaluated is the *Together ControlCenter 6.0*. It is the only evaluated tool that includes various phases of the development process into the evolutionary process, other than the implementation phase. Its use of automatic artifact synchronization removes the mismatch between system design and source code, after modifications. This is important, since project development is no longer just a question of source code development, but of the necessity of a fast development phase, with a resulting software that is reasonably stable and that can be mass produced.

One of the possible reasons for this differentiation in functionality is that the Together ControlCenter is a commercial tool. The other tools are open tools. It is the hope of the Eclipse community that their tool (Eclipse) becomes as rich in functionality as, for example, the Together ControlCenter. The difference is that the first system is extended through third-party tools, which are externally developed tools, and the second is extended through plug-ins developed specially for it. This means that the evolution of the Eclipse tool should be easier than ControlCenter's because TogetherSoft has no real control over the third party tools that are developed. It would be interesting to see in a few years which tool has a biggest share of the market.

The Smalltalk community seems to be happy with VisualWorks, and ports to different flavors of the language are being worked on. The Smalltalk community, unlike the Java community, does not need the development of IDEs, since it already has this type of structure in its environment (the Smalltalk browser IS the Smalltalk IDE). Additionally, Smalltalk can be extended by modifying its browsers directly.

The Guru tool is very interesting, since it is fully automated. But what makes this tool powerful, is the fact that the new inheritance structure is created independently of the existing inheritance structure. It uses algorithms to effectively find better inheritance hierarchy designs. This puts it in a category of its own.

The taxonomy itself was not easy to apply at all times. Some tools provided excessive documentation in some aspects, but ignored others almost completely. This makes the application of the taxonomy difficult in some cases. In this study, the hardest tool to evaluate was the Together ControlCenter 6.0, since no implementation details were available because it is a commercial tool. The easiest tool to evaluate was VisualWorks.

9 Future work

As immediate future work, we want to apply the taxonomy to more refactoring tools, such as jFactor [10], XRefactory [11], etc...

It would also be interesting to generate a library of the application of the taxonomy to different evolutionary mechanisms and tools. When an evolutionary tool is needed, one only has to know which features are useful for one's project, and which ones will hamper it. In this way, it would be easier for development teams to pick the adequate evolutionary tool for their product.

It is clear from the tool comparison and the conclusions, that future software evolution support tools should incorporate at least the design phase support, apart from implementation support. Specifically, future tools should consider the inclusion of source code and design refactorings. The ControlCenter artifact synchronization is an important step in this direction.

All the tools that were evaluated do not help the user decide which refactorings to apply, and where. Once the refactoring to be applied is selected, the application process is low effort, since the tools apply these changes automatically. But the refactoring selection process is still a medium to high effort activity. This effort can be lowered by the existence of tools that support the selection process. These tools would have to determine the quality of the existing design, and decide which changes could make this quality higher.

10 Acknowledgements

This research is funded by the FWO Project G.0452.03 “A formal foundation for software refactoring” and is carried out in the context of the scientific networks “Formal Foundations of Software Evolution” and “Research Links to Explore and Advance Software Evolution” financed by the Fund for Scientific Research - Flanders and the European Science Foundation, respectively.

References

[1] Mens, T., Buckley, J., Rashid, A., Zenger, M.: Towards a taxonomy of software evolution. Technical Report vub-prog-tr-02-05, Vrije Universiteit Brussel (2002) Submitted to Workshop on Unanticipated Software Evolution, Warshau, Poland, 2003.

[2] Cincom: Smalltalk VisualWorks (2002)

[3] eclipse.org: Eclipse (2002)

[4] Moore, I.: Automatic inheritance hierarchy restructuring and method refactoring. In: Proc. Int'l Conf. OOPSLA '96. ACM SIGPLAN Notices, ACM Press (1996) 235–250

[5] TogetherSoft: ControlCenter (2002)

[6] Opdyke, W.F.: Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks. PhD thesis (1992)

[7] Roberts, D., Brant, J., Johnson, R.: A refactoring tool for Smalltalk. Theory and Practice of Object Systems **3** (1997) 253–263

[8] Roberts, D.: Practical Analysis for Refactoring. PhD thesis, University of Illinois at Urbana-Champaign (1999)

[9] Beck, K.: Extreme Programming Explained: Embrace Change. Addison Wesley (2000)

[10] Instantiations: jFactor (2002)

[11] XRef-Tech: XRefactory (2002)

Facet	**VisualWorks**	**Eclipse**	**Guru**	**ControlCenter**
time of change	T_2, T_3	before T_1 or T_2 after T_3	before T_1 or T_2 after T_3	before T_1 or T_2 after T_3
change history	irrelevant	parallel/async.	unversioned	versioned
frequency	continuously	continuously	occasionally	continuously
role	developer / designer	developer / tester	developer	developer
distribution	local	local	local	local
automation	semi-automatic	semi-automatic	fully automated	semi-automatic
type	structural	structural / semantic	structural	structural / semantic
effect	alteration	any	alteration	any
invasiveness	non-invasive	non-invasive	highly invasive	non-invasive
safety	behavior -preserving	high safety	behavior -preserving	high safety
effort	low effort	low effort	virtually no effort	low effort
artifact	Smalltalk source code	Java source code	SELF source code	design (UML) Java code tests documentation
granularity	medium fine	coarse medium fine	coarse medium fine	coarse medium fine
locality	global	global	global	global
scope	source code	source code	source code	design / source code
availability	dynamic	static	static	static
activeness	reactive	reactive	reactive	reactive
openness	source available reflection	plug-in architecture	source available	integration API Wizards
plan	indifferent	unplanned	planned	unplanned
control	controlled refactoring	controlled refactoring	controlled	controlled refactoring
measure	no built-in support	no built-in support	2 proposed OO metrics	quality assurance

Table 4: Comparison of the software evolution tools that were evaluated