

# Evaluating software refactoring tool support

Erica Glynn and Paul Strooper  
School of Information Technology and Electrical Engineering  
The University of Queensland  
St Lucia, Qld 4072, Australia  
erica@itee.uq.edu.au

## Abstract

*Up to 75% of the costs associated with the development of software systems occur post-deployment during maintenance and evolution. Software refactoring is a process which can significantly reduce the costs associated with software evolution. Refactoring is defined as internal modification of source code to improve system quality, without change to observable behaviour. Tool support for software refactoring attempts to further reduce evolution costs by automating manual, error-prone and tedious tasks. Although the process of refactoring is well-defined, tools supporting refactoring do not support the full process. Existing tools suffer from issues associated with the level of automation, the stages of the refactoring process supported or automated, the subset of refactorings that can be applied, and complexity of the supported refactorings. This paper presents a framework for evaluating software refactoring tool support based on the DESMET method. For the DESMET application, a functional analysis of the requirements for supporting software refactoring is used in conjunction with a case study. This evaluation was completed to assess the support provided by six Java refactoring tools and to evaluate the efficacy of using DESMET method for evaluating refactoring tools.*

## 1 Introduction

Software evolution can be time-consuming and tedious, and often accounts for up to 75% of costs associated with the development of software-intensive systems [23]. Lehman's laws of software evolution [16, 17] state that while the functionality of a system increases over time, there is a corresponding decrease in quality and increase in internal complexity. Refactoring is a process that helps mitigate the problems of evolution by addressing the concerns of internal complexity and quality.

Refactoring is *the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure* [11]. Refactoring tasks range in complexity from the introduction of a symbolic constant to replace a magic number, to the major re-design of a system to introduce a design pattern [12, 14, 5]. As described by Fowler [11], refactoring is a manual process consisting of multiple stages: identification of code smells, proposing a refactoring, and applying the selected refactoring. A *code smell* is a structure present in source code that suggests the possibility of refactoring [2]. Manual refactoring, like evolution, is a time-consuming, tedious and error-prone process.

Automated software refactoring assists developers by supporting the time-consuming and error-prone tasks of manual refactoring, including propagating changes to method names, etc. Although automated software refactoring provides benefits to users, problems can be introduced since these tools currently have either an inappropriate level of automation (no user interaction or customisation available), are incomplete (covering only part of the refactoring process) or are too simple (the transformations and detected issues are not sufficiently complex or sophisticated). Furthermore, tools categorised as *automatic* (implying the whole process of refactoring is automated) exhibit varying levels of automatic support for the refactoring process. The current levels of support are characterised by only a subset of the refactorings described in [11, 10] being implemented or only the automating the application of refactorings without automating the detection of the opportunities for application.

In this paper, the DESMET method [15], for evaluating software engineering methods and tools, is used to assess software refactoring tools. DESMET provides a formalised evaluation framework for assessing the appropriateness of tools for use in supporting a selected software engineering task. We have used the DESMET Fea-

ture Analysis method (incorporating a small case study) to evaluate Java refactoring support tools. The tools studied are Eclipse [9], Borland JBuilder [6], IntelliJ IDEA [13], RefactorIT [25], Condenser [20] and jCOSMO [28]. The application of the Feature Analysis has highlighted the limitations of current refactoring tools. The evaluation has also demonstrated the framework's ability to differentiate software refactoring tools.

Section 2 introduces software refactoring and issues with refactoring tool support. Section 3 presents existing comparison frameworks and requirements definitions for software development environments (including software refactoring support), highlighting shortcomings with approaches using these frameworks for evaluating refactoring tools. An introduction to DESMET tool evaluations and feature analyses is provided in Section 4, including the definition of the feature set used in this study. Information on the case study used in this project and the results of the Feature Analysis of the selected Java refactoring tools is provided in Section 5.

## 2 Software Refactoring

Refactoring is the process of internal improvement of software without change to externally observable behaviour [11, 22]. Tourwé and Mens [27] identify three phases associated with the process of refactoring:

1. Identification of when an application should be refactored.
2. Proposal of which refactorings could be applied where.
3. Application of the selected refactorings.

Fully-automatic tools [21, 7] complete the multi-stage refactoring process without user interaction, from initial identification of where it is needed through the selection and application of a specific refactoring. Static, fully-automated software refactoring tools focus on detection and removal of duplicated code [21, 1]. In the Guru fully-automatic refactoring tool [21], restructuring of inheritance hierarchies is also performed automatically for programs written in the Self programming language (the Java equivalent is Condenser [20]). Although fully-automated tools assist software developers, the lack of user input into the process causes the introduction of meaningless identifier names, lack of customisability, and negative impacts on a user's current understanding of a system [4, 8]. Semi-automated refactoring tools attempt to address the problems raised by fully-automated or fully-manual processes by retaining user input to guide the refactoring process

whilst automating the tedious, error-prone and complex sub-tasks.

Current support for semi-automated software refactoring focuses on a single stage of the refactoring process, e.g., the implementation of refactorings such as 'Extract method'. This single-stage focus unnecessarily burdens developers with tasks that could be completed automatically, such as identifying where a refactoring may be applied. Additionally, tools automating different stages of refactoring are not themselves integrated, making the linking together of the stages (i.e., passing the output of one stage into the next) the task of the user. Examples of semi-automated refactoring are the refactoring transformation support in environments such as Eclipse [9], and the code smell detection of jCOSMO [28].

## 3 Existing evaluation frameworks

Evaluation of software engineering tools is a key process in selecting the correct environment to support software development. As such, formalised frameworks for comparing these tools are also important. For processes such as software refactoring, no specific comparison framework has been defined, instead frameworks for assessing software evolution or general development environments can be used.

The Taxonomy for Software Evolution by Mens et al. [19] is a framework for evaluating software evolution support tools. The Taxonomy defines four dimensions along which the tools are assessed: Temporal properties (when), Object of change (where), System properties (what), and Change support (how). Criteria within these categories include 'Artifact', 'Granularity', 'Impact' and 'Change propagation' for 'Object of change' as an example. Within [19], Mens et al. apply the Taxonomy to three tools, a versioning system, a refactoring tool and a dynamically evolving server, and successfully show that the Taxonomy can distinguish tools supporting different evolution sub-tasks.

In [24], Simmonds and Mens applied the Taxonomy to four software refactoring tools. The tools studied were Eclipse 2.0, Together ControlCenter 6.0, Guru (for Self 4.0) and SmallTalk VisualWorks 7.0, and were selected based upon their perceived differences. The application of the Taxonomy identified only minor differences between the tools. The differences that were identified were based upon the environments in which the refactoring support was situated (e.g., the presence or absence of version management and undo/redo facilities), and the development language supported (e.g., Java, SmallTalk or Self). The Taxonomy did not differentiate the tools based upon their support of the stages of refactoring (Guru fully auto-

mated the entire process, whereas the other tools supported only the implementation stage), nor the different levels of support provided within a stage (based on the complexity and number of different refactorings available). The interaction styles of the selected tools was also not addressed by the Taxonomy, with the difference between Guru's command-line, batch processing interface not compared to the interactive, Graphical User Interfaces (GUIs) of the other tools. The inability to evaluate refactoring tools based upon these differences provided a motivation for the development of the framework presented in this paper. To address the inadequacies highlighted by the applications in [24], the tools selected for the framework application in this paper have included those used by Simmonds and Mens, although limited to using equivalent tools supporting Java refactoring. The restriction to Java refactoring tools was made to eliminate language based differences within refactoring support, and to better mirror processes that are undertaken in tool selection in an industrial setting. The tools used in this study corresponding to the Simmonds and Mens study are Eclipse 3.1 [9] for Eclipse 2.0, Borland JBuilder [6] for Together ControlCenter 6.0<sup>1</sup> and Condenser [20] which is the Java version of Guru. IntelliJ IDEA, RefactorIT and jCOSMO were added to the set of tools studied to provide a more diverse set of tool styles (IDE, Stand-alone and plugin). Additionally, jCOSMO was added to provide an additional code smell detection tool, and IDEA and RefactorIT were selected based upon their reputation as premiere Java refactoring tools [10].

Another source of criteria when comparing software engineering tools are requirements specifications such as the STONEMAN Ada Programming Support Environment Specification (APSE) [3]. STONEMAN details requirements for development environments supporting the Ada programming language, but these requirements are equally applicable to software development environments in general. Whilst useful for evaluating full development environments, STONEMAN does not provide an appropriate way to evaluate refactoring tools because it focused on evaluating the completeness of an environment. Development environment requirements specifications, such as STONEMAN, may mandate the need for the inclusion of refactoring support, but specific requirements for support provided by the environment or any component tool are not included.

In addition to their lack of specific criteria for software refactoring, the Taxonomy for software evolution, and STONEMAN do not provide a rigorous or formal process for application. Techniques such as the DESMET

method [15], for evaluating software engineering tools and methods, provides a formalised and proven method for evaluating software engineering tools, whereas the Taxonomy and STONEMAN do not provide a formal and rigorous process for application.

Additionally, both the Taxonomy for software evolution and STONEMAN are incomplete, not accounting for user interaction and usability requirements. In [26], Toleman and Welsh emphasise that research into usability of software development tools and the application of usability heuristics and guidelines during the development of such tools is not common practice. It follows from this lowered importance placed on usability concerns, that frameworks for assessing software development tools currently do not account for these criteria.

In the specific case of software refactoring tools, existing evaluation frameworks are both too broad, evaluating the environments containing the refactoring support, and too general, since the stages of refactoring, and in particular the levels of automation for the different stages, are not considered. Due to this inability to consider the stages of refactoring, a command-line, batch processing, fully-automated, detection and transformation tool, such as Guru [21], is indistinguishable from the Graphical User Interface (GUI) based, user-driven, automated refactoring transformation support integrated into Eclipse.

To overcome the issues with existing evaluation frameworks, a DESMET Feature Analysis is used in this study. The criteria selected for the Feature Analysis include specific criteria for refactoring, usability concerns, and relevant criteria from the Taxonomy for software evolution and the STONEMAN APSE requirements specification.

## 4 DESMET Evaluation

The DESMET method is a generic, formalised framework for evaluating software engineering methods and tools. Developed by Kitchenham et al. [15], DESMET provides a set of nine evaluation methods based upon qualitative and quantitative measurement techniques. Applications of the framework can use formal experiments, case studies, user surveys, or a combination of these to formally evaluate both methods and tools. In our evaluation, a Feature Analysis incorporating a case study was carried out.

The DESMET method is designed to be an industry-driven assessment of software tools and methods. The assessments are conducted relative to a particular environment of software development processes, tools and procedures in place at a given company(s). Note that in this paper, a full evaluation was not conducted, as a generic assessment of refactoring tools was the goal. As such,

---

<sup>1</sup>TogetherSoft was purchased by Borland in October, 2002.

some of the subjective suitability criteria, such as ‘Cost of changes to software process’ have not been assessed. Although a full evaluation has not been conducted, the feature set discussed in Section 4.2 has been defined in full so that it can be applied by others for a specific industrial environment.

#### 4.1 Feature Analysis

Using a set of required and desired features, Feature Analysis assesses the comparative applicability of a software tool (or method) in a given context. Definition of the process of Feature Analysis can be found in [15]. In a traditional Feature Analysis, each feature is scored by a boolean value for its presence or absence, or on an ordinal scale for the degree of support provided. In the case of this study, several sub-categories that have been defined contain value judgements that are context-dependent, and as such have not been assigned a numerical scale. For example, a company with its own software development department that desires customisable tools may prefer an open source tool, whereas another company without a development team may prefer a commercially supported product. This criterion is part of the ‘Supplier assessment’ category, and is an example of a criterion for which an enumerated list has been defined, e.g., Style of supplier = (Commercial | Open Source | Research), but ranking of this criteria has been left for a company choosing to apply this framework.

Section 4.2 outlines the categories of evaluation criteria for the feature set. Due to the importance and novelty of the usability and refactoring categories, a detailed breakdown of these criteria is provided. The remaining categories have been elided for the sake of brevity, however the full criteria set is available online as an appendix to this paper<sup>2</sup>. Section 5 introduces the refactoring tools evaluated and outlines the refactoring case study that was performed with each of the tools. The Feature Analysis is also presented in Section 5.

#### 4.2 Evaluation Criteria

This feature set was built from the list of top-level categories suggested by Kitchenham et al [15]. This set was further augmented with criteria from the Taxonomy for Software evolution [19], the STONEMAN Ada Programming Support Environments Specification (APSE) [3], and criteria developed by the authors through experience with techniques and tools for refactoring and general software development.

The top-level categories for this feature set are:

<sup>2</sup><http://www.itee.uq.edu.au/~erica/papers/GlynnStrooper05.pdf>

1. **Supplier assessment** The style of the supplier (e.g., Commercial, Open Source or Research) and any license restrictions (e.g., for Commercial: Annual or Perpetual, and for Open Source: the Gnu Public License (GPL)).
2. **Maturity of tool** The type and version number of the current release of the software, the perceived extent of use and the number of previous releases, used to judge the stability and maturity of this release of the tool.
3. **Economic issues** The Total Cost of Ownership (TCO) including the cost of the tool, any required software and hardware, staff training and required changed to software processes currently used.
4. **Ease of introduction** The ease with which the refactoring tool can be introduced to the company. This includes time for installation, any required software or hardware packages that are needed and the ease with which these are sourced, the complexity of the installation process, and style of software processes the tool supports, and the impact on existing processes. Note that criteria within this category do overlap with ‘3. Economic issues’, however in this category the focus is on evaluating ease with which the tool can be introduced, whereas in Economic issues, tools with a simple yet prolonged install process are compared with tools with a short yet complex install process.
5. **Reliability** Whether the tool performs without faults, or the degree to which faults occurred or can be expected. Judged based on use of the tool in a case study or example.
6. **Efficiency** The evaluation of both the tool response and processing times, and the user time required to operate the tool.
7. **Usability** Discussed in Section 4.2.1.
8. **Compatibility** How the tool fits within development environments and software processes. This includes the style of tool integration, and the openness and completeness of the overall development environment(s) the tool fits within. Note that the environments and processes considered as part of Compatibility are not restricted to those already in place at a company, whereas in ‘3. Economic issues’ and ‘4. Ease of introduction’ difference to the existing environment is considered.
9. **Refactoring Specific** Discussed in Section 4.2.2.

##### 4.2.1 Usability criteria

Usability concerns for software development environments and tools are not often evaluated [26], yet consid-

eration of these concerns is important to any style of software tool [18]. As such, usability criteria form an important part of the feature set for this analysis. In developing the Usability criteria, Lund's [18] key usability maxims were used in addition to criteria and categories suggested by Kitchenham et al. [15]. The category of Usability is further divided into Learnability and User Interface. Learnability refers to the ease with which the tool can be integrated into the environment, including how long it takes to learn to use. The User Interaction category assesses the efficacy of the tool's user interface. As a User Interface (UI) provides the means to communicate with the underlying tool, and is particularly important to the selection of what tool is used in a given environment. The style of UI that is most appropriate for a task is dependent upon the task. For instance, for tasks that are completed automatically on a server, command-line interaction with batch processing facilities may be preferred, whereas for a development workstation where all processing is done locally and the user must maintain an understanding and a high awareness of system state, a Graphical User Interface (GUI) may be preferable.

The criteria within the **Learnability** sub-category are:

- 7.1.1. Quality of documentation** The quality of the available or supplied documentation including manuals, tutorials, examples and APIs. Also included are the amount and accuracy of available training material, e.g., tutorials that are up-to-date with the latest features and user interface, and the availability of online help, e.g., user and developer forums.
- 7.1.2. Learning curve** How much time and ongoing effort is required to become a master at using a tool. Note that the more complex and less intuitive the user interface and the more disruptive the technology to existing practices, the larger the learning curve.
- 7.1.3. Training effort** The amount of effort required to train staff to use the tool. Closely related to the inclusion or availability of tutorials and examples.

The criteria within the **User Interface** sub-category are:

- 7.2.1. Interaction style** The way in which the program physically interacts with the user, i.e., through a GUI or via the command-line, and the style of interaction, i.e., interactive or batch.
- 7.2.2. Responsiveness** How the interface responds to the user, includes both the concerns from '6. Efficiency' (processing time and necessary user time), and assessment of the measures that are taken to ensure correct user expectations of these times, e.g., progress bars, changed cursors.
- 7.2.3. Complexity of interface** Whether the user interface is intuitive, understandable, busy or cluttered. The

appropriateness of the number of steps, in the user interface, required to complete actions is also included.

- 7.2.4. Extent of user interaction required** The reactive or proactive nature of the tool, the level of user guidance required/allowed (modification and voiding of proposed actions), and the level of task automation. These measures can give an idea of whether the tool appropriately supports maintaining user understanding of the program being refactored/developed.

#### 4.2.2 Refactoring specific criteria

To allow for appropriate evaluation and comparison of refactoring tools, criteria specific to the refactoring process need to be considered. As discussed in Section 3, an issue with using existing comparison frameworks for evaluating refactoring support is that they fail to adequately distinguish between tools that support the same (or even different) refactoring tasks.

The refactoring specific sub-categories for this feature set are:

- 9.1. Refactoring stages supported** The refactoring stages (Detection, Proposal, Transformation) supported by the tool, and the level to which they are supported (not supported (0), supported manually (1) or supported automatically (2)). If multiple stages are supported the style of integration (i.e., at the file level, using a wizard or through the user manually entering data from one stage into the next) is also considered.
- 9.2. Maturity/Complexity of refactorings supported** As described by Fowler [10] in his 'Refactoring Rubicon', refactorings vary in both complexity and maturity. For example, the 'Rename field' (or method) refactoring is a simple search and replace action, whereas the 'Extract method' refactoring requires source code analysis. The maturity/complexity of the code smell or refactorings is judged according to this definition.
- 9.3. Language characteristics** Whether the tool is generic or specific to a given development language, and what (if any) language-specific features are required (e.g., multiple inheritance). A generic tool may be preferred if different projects use different languages, and a language-specific tool when all development is conducted in a single language.
- 9.4. Behaviour preservation** Since preservation of behaviour is part of the definition of refactoring, a tool which does not assure preservation of behaviour, even

at the highest level (i.e., using composition of atomic refactoring stages to ensure preservation), is not considered a refactoring tool. This criterion forms part of the required feature set for a refactoring tool.

**9.5. Customisability** Whether the tool is ‘Open’, allowing definition of additional code smells/refactorings; and ‘Customisable’, existing rules for detection, proposal or transformation can be modified or voided. This criterion assesses the ability to customise during development and through saved preferences in the tool.

**9.6. Artifacts supported** The artifacts that are supported by the tool. The minimum requirement is support of code, however support for refactoring requirements, design, build and testing artifacts may be desirable. Facilities for propagating changes across artifact types are also considered if more than one stage is supported, e.g., propagation of changes made during code refactoring to the design.

**9.7. Time of change** When the refactoring the tool support takes place. As refactoring tools can operate both statically (i.e., during development), or dynamically (i.e., during program execution), this criterion is used to categorise refactoring tools based on when changes occur.

## 5 Application

In this case study the feature set is applied to six software refactoring tools. These tools vary on style of supplier, stage(s) supported, number and complexity of code smells/refactorings and whether they are stand-alone, an integrated part of a development environment, or a plugin for multiple IDEs. The selected tools are: Eclipse [9], IntelliJ IDEA [13], Borland JBuilder [6], RefactorIT [25], Condenser [20], jCOSMO [28]. Eclipse, IDEA and JBuilder are Java software development environments with inbuilt refactoring transformation support. RefactorIT is a refactoring plugin for the Eclipse, JBuilder, Oracle JDeveloper, Sun ONE Studio and Netbeans development environments. Condenser and jCOSMO are prototype, stand-alone, code smell detectors. Of these tools, Condenser and jCOSMO are research tools from academic institutions, Eclipse is an open source project, and IDEA, JBuilder and RefactorIT are commercially supported products.

During the Feature Analysis, a case study was used to provide a common foundation on which to base the evaluation. Any supplied documentation and tutorials were also used prior to the application of the tool to the case study program to assess the training attributes of the tool. The case study program selected was a library application

catering for a variety of styles of items (at desk, references, books, periodicals, etc.). Before commencing the tool evaluations, the Library program was studied and manually refactored to identify the applicable refactorings and ideal design. This measure was put in place to mitigate any effects that increasing familiarity with the case study program as the evaluation progressed could have on the results. Details of the initial system and the characteristics of the system after (manual) refactoring appear in Table 1.

	Initial	After refactoring
Number of classes	12	15
Lines of Code (LOC)	319	250
Duplicate LOC	51	0

**Table 1. Library system: Before and After**

The refactorings from Fowler [11] that could be applied to the program were: ‘Create superclass’, ‘Extract method’, ‘Pull up method’ and ‘Pull up field’ refactorings, based on the presence of the ‘Duplicate code’ code smell.

The process taken during the evaluation of the refactoring tools, was to first locate, download and install the tools. Upon installation, any tutorials or introductory help manuals provided with the tool were used, followed by the application of the case study. Throughout these initial stages, quantified measurements, including installation time, and time taken to complete the case study were recorded. After the completion of the case study, the Feature Analysis evaluation was completed for each tool. A final collation of the results was completed once each tool was individually evaluated.

### 5.1 Data

This section presents the results of the evaluation of the six Java refactoring tools. For each of the categories outline in Section 4.2, the scores have been converted to a 5 point ordinal scale to allow for comparison. Due to their importance in differentiating refactoring tools (discussed in Section 3), the Usability and Refactoring categories are also given an overall higher weighting than the other categories. For Usability the weighting given is a multiplication factor of 1.5, for Refactoring the weighting is 3. Within the categories, the criteria are scored on weighted scales according to level of importance (the full definition of the feature set with scales is available online<sup>3</sup>). Note that the level of importance for a given criteria takes into account its status as a mandatory or desirable requirement as defined by the DESMET method. To compute the

<sup>3</sup><http://www.itee.uq.edu.au/~erica/papers/GlynnStrooper05.pdf>

normalised total for each category, the results for each of the weighted criteria are summed, and then converted to a score out of 5. The calculation of the normalised scores for the Refactoring Category is provided in Table 2. To calculate the scores, the totals for the weighted criteria are summed, e.g., the ‘Stages supported’ criteria is a ten point scale, with 3 points for each stage supported automatically, 1.5 for semi-automatically, and 0 for a stage not explicitly supported, and one point for if support for multiple stages is integrated. Complexity/Maturity of refactorings is a 15 point scale, due to its ability to differentiate tools supporting the same stage of refactoring. Of similar importance is Behaviour preservation, whose support is a mandatory requirement. Finally, customisability, whilst desirable is not mandatory and is an overlapping criteria with the Usability category, as such it is scored out of 1. The remaining criteria are qualitative, and whilst useful for distinguishing refactoring tools have not been judged for this evaluation. From the total possible score of , the results for each tool are then converted to a score out of 5.

Tables 3 and 4 summarise the results by providing cumulative scores for each category (determined by adding together the criteria scores within the category), with any comments or results from qualitative criteria also included. The total possible score for each category is provided in brackets.

## 5.2 Discussion

From the evaluations and the data presented in Section 5.1, conclusions can be drawn about both the quality of refactoring support provided by the studied tools and the efficacy of the developed evaluation framework. Conclusions that can be reached relating to the refactoring tools are:

- None of the tools selected fully support the refactoring process as a whole.
- Refactoring implementation tools are comparable in maturity, reliability, efficiency and usability, but distinguishable on supplier, economic issues and refactoring specific criteria.
- The tools supporting the detection phase are not yet industrial strength.
- Usability of refactoring tools requires further research/consideration.

It is clear from the results of the Feature Analysis that refactoring tools currently available, such as those evaluated in this study, do not provide adequate support for the refactoring process in its entirety, as defined by Fowler [11] and Tourwé and Mens [27]. This lack of support is

demonstrated by low scores for every tool in the ‘Refactoring specific’ category, the highest being 2.5 out of a possible 5. The slightly higher score for RefactorIT reflects the inclusion of ‘software audits’ which, whilst not including the sophistication of the code smells identified by Fowler [11], does score above the other tools due to its ability to detect and propose remedies for potential stylistic errors in source code. An example of a stylistic error found by RefactorIT is “Constant field’s name should be upper case” [25]. Note that this can be identified in Eclipse using the Code formatter, and is not considered a code smell, nor its remedy a refactoring.

The data collected also shows that the tools supporting refactoring implementation (Eclipse, JBuilder, IDEA and RefactorIT) are mature, reliable and efficient to use. One means of differentiating these tools is on the complexity of the refactoring supported. In the case of IDEA and RefactorIT, the higher scores for the refactoring specific criteria is partially based on supporting more and more complex refactorings. Supplier assessment and economic issues are also means of distinguishing these tools as commercial versus open source style of supplier is important in an industrial context.

Another conclusion that can be drawn from the data relates to the immaturity of support for detection of code smells. Both the code smell detection tools, Condenser and jCOSMO, are prototype tools that are not mature or reliable enough for use in an industrial setting. Additionally, RefactorIT, the only of the commercial tools to attempt detection, possesses the capability to detect and propose solutions for code smells less complex and sophisticated than those defined by Fowler [11]. The data also shows that integration of stages, even if not supported by a single tool or environment, is an issue that needs to be addressed by all the studied tools except for RefactorIT, whose integrated detection, proposal and transformation technique for stylistic errors was sufficient.

From studying the results for the Usability criteria, we have concluded that the refactoring tools that were part of integrated development environments with Graphical User Interfaces (GUIs) are the most usable. Additionally, the inclusion of both tutorials and manuals for documentation contributed to tools gaining higher usability scores. To achieve a maximum score for usability, which none of the studied tools did, an integrated approach to refactoring support which balances automation and user interaction was needed.

From the data presented, it is clear that the developed feature set achieved the goal of providing a mechanism to evaluate refactoring support. The use of a DESMET Feature Analysis incorporating a case study, allowed both

	Eclipse	IDEA	JBuilder	Condenser	jCOSMO	RefactorIT
9.1 Stage supported (10)						
– Detection	0	0	0	3	3	3
– Proposal	0	0	0	0	0	3
– Implementation	3	3	3	0	0	3
– Integrated	0	0	0	0	0	1
9.2 Complexity (18)						
– Detection	0	0	0	6	2	2
– Proposal	0	0	0	0	0	2
– Implementation	2	4	2	0	0	4
9.3 Language	Java	Java	Java	Java	Java	Java
9.4 Behaviour Preservation (5)	5	5	5	5	5	5
9.5 Customisability (1)	1	0	0	0	1	0
9.6 Artifact Supported	Code	Code	Code	Code	Code	Code
9.7 Time of change	Dev	Dev	Dev	Dev	Dev	Dev
<b>Total (34)</b>	11	12	10	14	11	23
<b>Normalised Total (5)</b>	1.6	1.7	1.4	2.0	21.6	3.4

**Table 2. Refactoring Data Summary**

	Eclipse	IDEA	JBuilder
<b>1. Supplier assessment</b>	Open Source	Commercial	Commercial
		(perpetual)	(perpetual)
<b>2. Maturity of tool (5)</b>	5	5	5
<b>3. Economic issues (5)</b>	5	4	2
		US\$99 - 499	US\$499 - 3500
<b>4. Ease of Introduction (5)</b>	5	4	4
		No inbuilt tutorial	No inbuilt tutorial
		Project setup issues	Project setup issues
<b>5. Reliability (5)</b>	5	5	5
<b>6. Efficiency (5)</b>	3	3	3
	Single refactoring stage	Single refactoring stage	Single refactoring stage
<b>7. Usability (5)</b>	4	4	4
Documentation (5)	5	4	4
Learning Curve (3)	2.5	2.5	2.5
UI Style (5)	GUI and Interactive (5)	GUI and Interactive(5)	GUI and Interactive(5)
Level of User Interaction (5)	Too high (3)	Too high (3)	Too high (3)
<b>8. Compatibility (5)</b>	4	4	4
	Full-featured IDE	Full-featured IDE	Full-featured IDE
<b>9. Refactoring Specific (5)</b>	1.6	1.7	1.4
<b>TOTAL (52.5)</b>	37.8	32.7	30.4

**Table 3. Evaluation Data Summary (1)**

comparative evaluation of the studied tools and an assessment of refactoring tools in the general sense, especially in regards to their completeness. The inclusion of the refactoring specific criteria provided a way to differentiate tools supporting the same stage of the refactoring process,

not previously possible using existing evaluation frameworks. The definition of the criteria for Usability provided a means to differentiate refactoring support tools, and demonstrated that more work is required on developing the usability of software refactoring tools.

	<b>Condenser</b>	<b>jCOSMO</b>	<b>RefactorIT</b>
<b>1. Supplier assessment</b>	Academic (open source)	Academic (open source)	Commercial (perpetual)
<b>2. Maturity of tool (5)</b>	1	1	4
<b>3. Economic issues (5)</b>	4	4	4
	Free	Free	US\$43 - 378
	Additional training costs	Additional training costs	
	Potential process change	Potential process change	
<b>4. Ease of Introduction (5)</b>	0	0*	4
	Complex install	Failed install(*)	No inbuilt tutorial
	Not all packages included	Complex install	
<b>5. Reliability (5)</b>	3	0*	5
	Developer's comment		
	Case study successful		
<b>6. Efficiency (5)</b>	4	0*	5
	Single refactoring stage		Integrated
<b>7. Usability (5) * 1.5</b>	1	0	4.5
Documentation (5)	1	0	4
Learning Curve (3)	1.5	0	2
UI Style (5)	Commandline and Batch(0)	Commandline and Batch(0)	GUI and Interactive(5)
User Interaction (5)	Too low (1)	Too low (1)	Appropriate (5)
<b>8. Compatibility (5)</b>	1	1	5
	Stand-alone	Stand-alone	5 IDEs + Stand-alone
<b>9. Refactoring Specific (5) * 3</b>	2	1.6	3.4
<b>TOTAL (52.5)</b>	16.5	7.6	37.15

**Table 4. Evaluation Data Summary (2)**

## 6 Conclusion

In this paper, we have presented a framework for evaluating software refactoring support based upon the DESMET Feature Analysis technique. This framework was then used to evaluate six Java refactoring tools. In defining the criteria for this framework we have outlined deficiencies in using existing general frameworks for evaluating software evolution and development environments to evaluate refactoring support, and have highlighted the importance of including refactoring specific criteria, and criteria to assess usability. From the data gained during the evaluation, we have demonstrated that existing support for refactoring is inadequate due to its incompleteness, and in the case of code smell detection tools, their relative immaturity. Usability of refactoring tools was also raised as an issue requiring future study. The data also shows the effectiveness of the framework in being able to distinguish between tools supporting the same stage of refactoring.

As has been demonstrated by the results of the evaluations, current levels of refactoring support provided by software tools fall short of the desired level. In work to fol-

low on from this paper, a full usability study will be conducted to determine exact user requirements for software refactoring tools. From the identified requirements, and using the results of this evaluation, an integrated approach to software refactoring support will be developed. This approach will build on the strengths identified within current refactoring transformation support, and additionally incorporate detection of code smells (as defined by Fowler [11]) and refactoring proposal to provide a support mechanism for the entire refactoring process. Being careful to balance automation with user interactivity, the support will assist users by removing error-prone, time-consuming and tedious tasks, whilst assisting the user to maintain an accurate mental model of the software that is being refactored.

## Acknowledgement

The authors wish to thank Anthony MacDonald and Michael Lawley for discussions during the early stages of this project.

The work reported in this paper has been funded in part by the Co-operative Research Centre for Enterprise Dis-

tributed Systems Technology (DSTC) through the Australian Federal Government's CRC Programme (Department of Education, Science and Training).

Erica Glynn is supported by an Australian Postgraduate Award funded by the Australian Federal Government Department of Education, Science and Training.

## References

- [1] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Working Conference on Reverse Engineering*, pages 98–107, 2000.
- [2] K. Beck and M. Fowler. Bad smells in code. In M. Fowler, editor, *Refactoring: Improving the Design of Existing Code*, chapter 3, pages 75–88. Addison Wesley, 1999.
- [3] J. Buxton. DoD requirements for Ada programming support environments, STONEMAN. Technical Report AD-A100 404, US Department of Defense, February 1980. Available from <http://www.adahome.com/History/Stoneman>, Accessed August 2003.
- [4] F. W. Callis. Problems with automatic restructurers. *ACM SIGPLAN Notices*, 23:13–21, March 1987.
- [5] M. Ó Cinnéide. Automated refactoring to introduce design patterns. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 722–724. ACM Press, 2000.
- [6] Borland Software Corporation. JBuilder. <http://www.borland.com/us/products/jbuilder/index.html>, Accessed September, 2005.
- [7] P. Ebraert, T. D'Hondt, and T. Mens. Enabling dynamic software evolution through automatic refactoring. In *Proceedings of the Workshop on Software Evolution Transformations (SET2004)*, pages 3–7, November 2004.
- [8] M. Endsley. Automation and situation awareness. In R. Parasuraman and M. Mouloua, editors, *Automation and human performance: Theory and applications*, pages 163–181, 1996.
- [9] Eclipse Foundation. Eclipse 3.1. <http://www.eclipse.org/>, Accessed September, 2005.
- [10] M. Fowler. Refactoring home page. <http://refactoring.com/>, Accessed October 2005.
- [11] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.
- [12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [13] JetBrains. IntelliJ IDEA Version 5.0. <http://www.jetbrains.com/idea/>, Accessed September, 2005.
- [14] J. Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2005.
- [15] B. A. Kitchenham. Evaluating software engineering methods and tool: Parts 1 – 12. *ACM SIGSOFT: Software Engineering Notes*, (21(1)–23(5)), 1996–98.
- [16] M. Lehman. Laws of program evolution - rules and tools for programming management. In *Proceedings Infotech State of the Art Conference, Why Software Projects Fail?*, volume 11, pages 1–25, April 1978.
- [17] M. Lehman and J. Ramil. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 11(1):15–44, November 2001.
- [18] A. Lund. Expert ratings of usability maxims. *Ergonomics in Design*, (3):15–20, July 1997.
- [19] T. Mens, J. Buckley, M. Zenger, and A. Rashid. Towards a taxonomy of software evolution. In *Proc. Workshop on Unanticipated Software Evolution*, pages 1–18, Warhau, Poland, March 2003.
- [20] I. Moore. Condenser 1.05. <http://condenser.sourceforge.net/>, Accessed September, 2005.
- [21] I. Moore. Guru - a tool for automatic restructuring of Self inheritance hierachies. In *Technology of Object-Oriented Language Systems (TOOLS)*, volume 17, pages 267–275. Prentice-Hall, 1995.
- [22] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Department of Computer Science, 1992.
- [23] S. Rugaber. Program comprehension. *Encyclopedia of Computer Science and Technology*, 35(20):341–368, 1995.

- [24] J. Simmonds and T. Mens. A comparison of software refactoring tools. Technical Report vub-prog-tr-02-15, Programming Technology Lab, November 2002.
- [25] Aqris Software. RefactorIT 2.5. <http://www.refactorit.com/>, Accessed September, 2005.
- [26] M. Toleman and J. Welsh. Systematic evaluation of design choices for software development tools. *Software – Concepts and Tools*, 19:109–121, 1998.
- [27] T. Tourwé and T. Mens. Identifying refactoring opportunities using logic meta programming. In *Proceedings of 7th European Conference on Software Maintenance and Reengineering*, pages 91–100. IEEE Computer Society, 2003.
- [28] E. van Emden and L. Moonen. jCOSMO 0.2. <http://www.cwi.nl/projects/renovate/javaQA/intro.html>, Accessed September, 2005.

## **A Feature Set**

1. Supplier assessment
  - 1.1. Style of supplier: Commercial/Open Source (OSS) Community/Research
  - 1.2. License restrictions
2. Maturity of tool
  - 2.1. Number of versions released
  - 2.2. Current Release number
  - 2.3. Current Release type
  - 2.4. Perceived extent of use
3. Economic issues (Total cost of ownership)
  - 3.1. Cost of tool
  - 3.2. Ancillary costs
    - 3.2.1. Cost of packages/software/hardware required
    - 3.2.2. Cost of changes to software process model in place at company
    - 3.2.3. Cost of staff training
4. Ease of introduction
  - 4.1. Installation time
  - 4.2. Required packages
  - 4.3. Complexity of installation process
  - 4.4. Style of process supported
  - 4.5. Changes to process required
5. Reliability
6. Efficiency
  - 6.1. Tool response time (related to Responsiveness in 7.2)
  - 6.2. Necessarily user time (related to Complexity in 7.2)
7. Usability
  - 7.1. Learnability
    - 7.1.1. Quality of Documentation: includes APIs, manuals and examples
      - 7.1.1.1. Training quality: availability of work though examples and ease of use in case study
      - 7.1.1.2. Online help: levels of help available (manuals, forums, FAQs, technical support)
    - 7.1.2. Learning curve
    - 7.1.3. Training effort
  - 7.2. User Interface (modified from Kitchenham since not all are graphical)
    - 7.2.1. Interaction style:
      - 7.2.1.1. GUI(preferred) or command line
      - 7.2.1.2. interactive (preferred) or batch processing
    - 7.2.2. Responsiveness: includes both delay & processing time to complete actions & general time response time
    - 7.2.3. Complexity of interface: relating to busyness of the interface, number of steps required to complete an action, intuitiveness/understandability (e.g., sensible naming of menu items, etc)
    - 7.2.4. Extent of user interaction required
      - 7.2.4.1. Activeness: Reactive, Proactive or mixture of both
      - 7.2.4.2. Level of user guidance: voidable or proposed actions/transformations (e.g., if a given refactoring is proposed but the user does not want it to be performed can it be voided)
      - 7.2.4.3. Level of automation (Fully automated |Semi-automated |Fully manual)

- 8. Compatibility
  - 8.1. With existing tools (enclosing environment characteristics)
    - 8.1.1. Integration style
    - 8.1.2. Openness
    - 8.1.3. Completeness
      - 8.1.3.1. Configuration/History/Change management
      - 8.1.3.2. Artifacts supported
      - 8.1.3.3. Distribution support
      - 8.1.3.4. Runtime support
        - 8.1.3.4.1. Execution facilities (i.e., inbuilt VM etc)
        - 8.1.3.4.2. Debugging facilities: stop, step, replay etc.
        - 8.1.3.4.3. Fault recording and reporting
        - 8.1.3.4.4. Measurement
      - 8.1.3.5. Security support
      - 8.1.3.6. Process control
      - 8.1.3.7. Design support
      - 8.1.3.8. Requirements specification support
      - 8.1.3.9. Development support
        - 8.1.3.9.1. Editing
        - 8.1.3.9.1. Undo/redo facilities
        - 8.1.3.9.2. Syntax highlighting
        - 8.1.3.9.3. Compilation support
        - 8.1.3.9.4. Verification support
        - 8.1.3.9.5. Build/Release support
    - 8.1.4. Supplier assessment (as above)
    - 8.1.5. Maturity of tool (as above)
    - 8.1.6. Economic issues (as above)
  - 8.2. With software processes
    - 8.2.1. Software processes supported
    - 8.2.2. Software processes incompatible
- 9. Refactoring specific
  - 9.1. Refactoring stages supported
    - 9.1.1. Stages automated
    - 9.1.2. Integration of stages supported (mechanism (user |database |filesystem))
  - 9.2. Maturity/Complexity of refactorings supported judged against Fowlers Refactoring Rubicon
  - 9.3. Language supported, genericity & language specific features required (e.g., multiple inheritance)
  - 9.4. Behaviour preservation (safety)
  - 9.5. Customisability
    - 9.5.1. Openness additional detections/proposals/refactorings can be added
    - 9.5.2. Existing rules (i.e. for detection, proposal & refactorings) can be modified (During development |Through saved preferences |Both)
    - 9.5.3. Existing rules can be voided (Statically |Dynamically |Both)
  - 9.6. Artifacts supported
    - 9.6.1. Code |Design |Documentation
    - 9.6.2. Propagation support for propagating changes to other artifacts
  - 9.7. Time of change (refactoring style): static (compile-time) |dynamic (load-time |run-time)