

Improving Usability of Software Refactoring Tools

Erica Mealy, David Carrington, Paul Strooper and Peta Wyeth
School of Information Technology and Electrical Engineering
The University of Queensland, St Lucia, Qld 4072, Australia
{erica, davec, pstroop, peta}@itee.uq.edu.au

Abstract

Post-deployment maintenance and evolution can account for up to 75% of the cost of developing a software system. Software refactoring can reduce the costs associated with evolution by improving system quality. Although refactoring can yield benefits, the process includes potentially complex, error-prone, tedious and time-consuming tasks. It is these tasks that automated refactoring tools seek to address. However, although the refactoring process is well-defined, current refactoring tools do not support the full process.

To develop better automated refactoring support, we have completed a usability study of software refactoring tools. In the study, we analysed the task of software refactoring using the ISO 9241-11 usability standard and Fitts' List of task allocation. Expanding on this analysis, we reviewed 11 collections of usability guidelines and combined these into a single list of 38 guidelines. From this list, we developed 81 usability requirements for refactoring tools. Using these requirements, the software refactoring tools Eclipse 3.2, Condenser 1.05, RefactorIT 2.5.1, and Eclipse 3.2 with the Simian UI 2.2.12 plugin were studied. Based on the analysis, we have selected a subset of the requirements that will be incorporated into a prototype refactoring tool intended to address the full refactoring process.

1 Introduction

Software refactoring is a software development process designed to reduce the time and costs associated with software development and evolution. Refactoring is defined as the process of internal improvement of software without change to externally observable behaviour [9, 22]. In [17], we evaluated automated software refactoring tool support using the DESMET Feature Analysis technique [15]. The evaluation showed that usability of software refactoring

tools varied greatly, as did their level of support for the full refactoring process.

Software refactoring presents several challenges for automated tool support. During refactoring, users must synthesize and analyse large collections of data (code) to identify inappropriate or undesirable features (such as duplicated code), propose solutions to discovered issues, and perform potentially complex, error-prone and tedious transformations to rid their systems of these undesirable features. Throughout the process, the user must maintain the existing behaviour of their system in addition to maintaining or improving their own mental model and understanding of the system. This in turn means that the level of automation the tool exhibits must not negatively affect the user's understanding. Through the application of research and theory from situation awareness [4, 5, 34] and task allocation, we aim to improve the quality of support provided for software refactoring.

Usability of software refactoring tools is an area in which little research has been performed. In general, the production of and research into software development tools often overlooks the issue of usability [30]. In the area of refactoring, usability-related research has focused on understandability of error messages and assisting the user in the selection of text with a mouse cursor prior to the application of an automated refactoring transformation [20]. We believe that there are more important aspects affecting the usability of existing refactoring tools and we wish to identify and address these issues. To identify areas in which usability can be improved, we have developed usability requirements for refactoring tools and have used these requirements to analyse the state-of-the-art for refactoring tools. To achieve this, we sought usability design guidelines that could yield a set of usability requirements when combined with a definition of the tasks, users, and environment of refactoring. This definition was developed using the framework presented in the ISO 9241-11 interface design standard [13].

Guidelines have been used for both the design and evaluation of user interfaces since the early 1970s [11]. In looking at usability guidelines, we found many different sets of guidelines, rules, heuristics, maxims, etc.¹, yet no single set provided a complete set of guidelines that could be used to develop usability requirements for software refactoring tools. To manage the number of individual guidelines (we collected 120 from 11 sources), we collated and categorised the lists based on fundamental groupings that were evident across the initial 11 sources. The categories selected were based upon common terminology from within the usability community. Duplicate guidelines and those addressing a similar or closely-related concept became more prevalent, as the categorised list became larger. We distilled the categorised list of 120 initial guidelines into 29 to provide a more usable list.

The composite set of guidelines still did not provide adequate guidance for the development of usability requirements for the task of software refactoring. As a result, we defined six new guidelines for use in designing computer-based support for software refactoring. From the complete set of usability guidelines, we have developed a list of 81 usability requirements for software refactoring tools, some of which could not be readily attained through the use of the existing usability guidelines. Using these requirements, we have evaluated four existing refactoring environments: Eclipse 3.2 [7], RefactorIT 2.5.1 [29], Condenser 1.05 [18], and Eclipse 3.2 with the Simian UI 2.2.12 plugin [3]. Based on the analysis, we will identify areas that will be addressed in a prototype refactoring support tool.

The remainder of this paper is organised as follows: Section 2 presents an analysis of the task of software refactoring with respect to information required for appropriate user interface design. The specification of the refactoring usage environment (ISO 9241–11) is presented in Section 3. Section 4 presents literature on automation for situation awareness and Fitts’ List for task allocation, applying these to software refactoring. Section 5 presents the results of the collation, categorisation and distillation of existing usability guidelines. Section 6 defines additional guidelines for computer-supported tasks such as refactoring. In Section 7 we present some of the 81 identified usability requirements and analyse four of those that were derived from the additional guidelines from Section 6. Finally, Section 8 presents the analysis of four refactoring tools to identify gaps in the current state-of-the-art for software refactoring tools. The paper closes with a summary of the contributions of this paper and an identification of future work.

¹We will use “guidelines” to include all of these related concepts.

2 Software refactoring

Refactoring is the process of internal improvement of software without change to externally observable behaviour [9, 22]. Tourwé and Mens [31] identify three phases associated with the process of refactoring:

1. Identifying when an application should be refactored.
2. Proposing which refactorings should be applied.
3. Applying the selected refactorings.

The first stage of refactoring can also be described as the identification of *code-smells*. Beck and Fowler [1] define code-smells as “*certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring*” [1, pg 75]. Like coding style violations, code-smells are issues that exist in a system that do not prevent the system from performing the function it was developed for (unlike compiler errors), but inhibit the developer’s ability to maintain, understand or modify a piece of code. It is code-smell instances that are remedied or eliminated by the proposal and application of appropriate refactoring transformations.

3 ISO 9241–11 usability specification for software refactoring tool support

The following specification of the usage environment of software refactoring tools, according to ISO 9241–11 [13], has been broken into sections on the goals of software refactoring tools and the context in which software refactoring occurs.

3.1 Goals

The major goal of software refactoring is to improve internal quality of a software system whilst not changing its observable behaviour. Furthermore, it is the goal of software refactoring tool support to assist a software developer to perform software refactoring in the most efficient and effective way. Additionally, as software refactoring is only part of the software development process, another goal of software refactoring tool support is to not hinder the developer’s ability to understand and reason about the software system being refactored and developed.

3.2 Context of use

For software refactoring tools, the context of use, as outlined in the following sections, specifies the user set, the component sub-tasks of software refactoring, and the environment and equipment needed.

3.2.1 Users

Software refactoring tool users are mid- to high-level computer experts. Software refactoring tool users are also software developers, with knowledge of the language they are developing in and refactoring theory and practices. The level of experience in each of these areas (software development, language and refactoring theory and practice) can vary from novice to expert.

3.2.2 Tasks

As outlined in Section 2, refactoring can be broken down into three tasks: detection of code-smells, proposal of transformations to remove code-smells, and the application of the selected refactoring transformation. Within these tasks, sub-tasks include program analysis to identify issues and potential ways to remedy them within the code base; and reasoning to determine the severity and legitimacy of a potential code-smell and for selecting the *best* refactoring transformation to remove an identified code-smell without changing the behaviour of the program. The application of refactoring transformations is a generally mechanical process, which can be error-prone, repetitive, time-consuming and tedious, and are ideally completed automatically.

3.2.3 Equipment and environment

The physical equipment and environment for a refactoring tool is a computer with an operating system and potentially an integrated software development environment with refactoring support.

The social and cultural environment for a software refactoring tool includes the trust that a developer has in the tool performing the code-smell and refactoring transformation tasks correctly and completely. The safety and security requirements that are placed on the system being refactored play an important part in the level of trust required. Systems with safety-critical requirements will require a higher level of trust in the refactoring support. The size of the system being refactored and the style of software development life-cycle (e.g., agile or waterfall) also form part of the social and cultural environment of software refactoring tool use. In general, software refactoring tool users must maintain an awareness of the structure and an understanding of the code that they are refactoring, therefore an important environmental requirement for software refactoring tools is that they assist the user to develop and maintain this understanding.

4 Automation and task allocation

Automation is often introduced to aid users by reducing effort and cognitive load [4, 34]. In the context of software refactoring, automation allows the ability to introduce more thorough, complex or subtle code-smell detection mechanisms and better matching of refactoring transformations to code-smell instances. We believe the existence of a refactoring tool with this level of automation would lead to both more systems being refactored and more refactorings being applied to these systems, by reducing the effort required for refactoring and improving system quality.

Although automation can bring benefits, the addition or increase of automation in any system has inherent problems, which can include: mistrust (reliability and calibration) and over-trust (complacency) [4, 34]. For complicated applications, in which computer-based support is used to assist the user operating on and understanding a system, over-automation can have the further problem of interfering with the user's understanding [4, 5]. Due to these issues, the addition or increase of automation in a system (such as a software refactoring support tool) requires careful analysis and consideration in order to embody the appropriate level of automation.

Sheridan [26] identifies eight levels of automation, as shown in Figure 1. These levels range from no automation, through levels of human-guided or supervised automation, to completely autonomous processes. Using these eight levels, a process can be classified as to the level of automation it exhibits. Sheridan identifies four general stages in which automation can occur. These stages are 'acquire', 'analyse', 'decide', and 'implement'. For the three-stage process of software refactoring, code-smell detection maps to the acquire and analyse stages, the proposal of the appropriate refactoring transformations maps to the analyse and decide stages, and the application of refactoring transformations maps to the implementation stage.

In allocating tasks between computers and users, the requirements placed upon the user to maintain an understanding of the software system being refactored must be considered. A user's *situation awareness* is defined in terms of their perception and comprehension of a system's state and the projection to future states and required actions, relevant to the completion of a particular task [5]. For refactoring, situation awareness applies to the developer's understanding of the system, including its structure and conventions, and projections of how an actions will affect their ability to maintain and re-design the system. Endsley [4] argues that to minimise the negative effects from the introduction of automation, systems should be de-

1. The computer offers no assistance: the human must do it all.
2. The computer suggests alternative ways to do the task.
3. The computer selects one way to do the task.
4. The computer selects one way to do the task, and executes that suggestion if the human approves.
5. The computer selects one way to do the task, and executes that suggestion if the human approves, or allows the human a restricted time to veto before automatic execution.
6. The computer selects one way to do the task, and executes automatically, then informs the human.
7. The computer selects one way to do the task, and executes automatically, then informs the human only if asked.
8. The computer selects, acts, and ignores the human.

Figure 1. Sheridan's levels of automation [26]

signed to maximise user involvement whilst reducing the load that would have been associated with doing the task manually. It is this balance between automation and user involvement [34, 4] that is not optimal in existing automated software refactoring tools.

In order to appropriately allocate the sub-tasks of software refactoring between computers (automated) and users (manual), we have used Fitts' MABA-MABA List [6] (as shown in Figure 2), otherwise known and described as Fitts' law of task allocation [26]. Fitts' List identifies which tasks can be better completed by men (i.e. users), and which by machines. The items from Fitts' List most relevant to software refactoring are for men *Reasoning inductively* and *Exercising judgement*, and for machines *Reasoning deductively*. *Inductive reasoning* is defined as reasoning from detailed facts to general principles (i.e. generalisation), whereas *Deductive reasoning* is defined as reasoning from the general to the particular.

The areas in which judgement must be applied within the software refactoring process is the decision on what refactoring transformations will be applied to remedy an identified code-smell instance. The subjective nature of the goal of refactoring (improving a software system whilst maintaining behaviour) is the reason for this. Due to this need for human judgement, only the proposal of the available options may be automated.

Code-smells include ones that can be detected automatically through program analysis techniques, but also ones that are based on loose heuristics that are difficult to detect automatically [28]. An example of a code-smell that

Men [sic] are better at

- Detecting small amounts of visual, auditory or chemical energy
- Perceiving patterns of light or sound
- Improvising and using flexible procedures
- Storing information for long periods of time, and recalling appropriate parts
- Reasoning inductively
- Exercising judgement

Machines are better at

- Responding quickly to control signals
- Applying great force smoothly and precisely
- Storing information briefly and erasing it completely
- Reasoning deductively

Figure 2. The Fitts' MABA-MABA List

utilises deductive reasoning is the 'duplicate code' code-smell [9] which can be automatically detected when a threshold for the percentage similarity between code specimens is provided. The 'long method', and 'large class' code-smells, which can be automatically detected using software metrics where appropriate thresholds have been identified, are other examples of code-smells that can be computed with deductive reasoning. An example of a code-smell that cannot be computed with deductive reasoning is the 'speculative generality' code-smell, which exists in a class that includes extra functionality. As 'extra functionality' cannot be quantified, inductive reasoning must be used. The proposal and selection of appropriate refactoring transformations to remedy code-smells similarly requires both inductive and deductive reasoning.

It is important to note that for the code-smell and remedy proposal stages of refactoring, the mixture of inductive and deductive reasoning required indicates that these stages should not be fully automated. However, these stages can be partially automated. This partial automation will assist the user by lowering the effort needed to complete refactoring whilst still allowing user input and maintaining the user's understanding of the code they are refactoring. Using a previous evaluation of refactoring tools, our analysis of the process of refactoring using Fitts' List, and Sheridan's eight levels of automation, the graph in Figure 3 was developed. In the graph, the Fitts' List ideal level of automation is compared to the fully manual refactoring process described by Fowler [9] and fully-automated, autonomous refactoring support such as Guru [19]. Full automation (i.e. level 8) is not optimal for the task of refactoring due to the need to involve the user. The Sheridan levels of 6-6-2-4 for the ideal, reflect a tool automatically

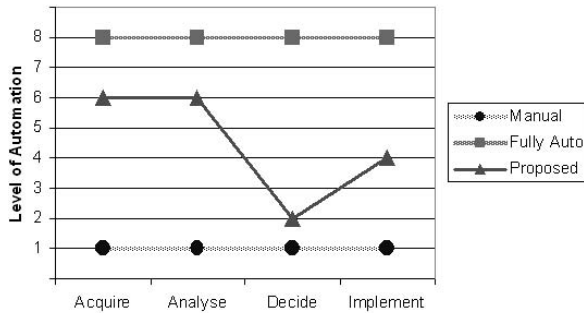


Figure 3. Refactoring automation levels

performing code smell detection and proposing applicable transformations to the user, who then selects which transformations to perform (i.e. the user approves a given choice, c.f. sheridan level 4) and the tool automatically performs the transformation. This balances automation with user involvement to minimise user workload, and maximise the user’s ability to maintain an understanding of the system.

5 Usability guidelines

Though popularised by people such as Nielsen [21] and Shneiderman [27] in the 1990s, collections of general heuristics, rules and guidelines for user interfaces can be found dating back to the early 1970s [11]. To study these collections, we collated a number of representative lists (spanning from 1971 to 2004) into a single list. During the collation of this list, it was noted that although no one set contained a complete set of guidelines, the guidelines in the individual sets fitted into several broad categories. We have used these categories to group like guidelines. Our categories are: Consistency, Errors, Information Processing, User Experience, Design for the User(s), User Control, Goal Assessment, and Ease of Use. Table 1 presents the origin of the 11 sets of guidelines, the original number of guidelines in each set, the number from each original set in each category, and the number of distilled guidelines for each category.

To remove duplication within the identified categories, repeated and related concepts were distilled into a single guideline for that concept. Examples of guidelines from different collections that address the same concept are: Error prevention [21] and Offer error prevention and simple error handling [27]. These two guidelines have been encapsulated in Assist the user to prevent errors (through feedback, constrained interface, use of redundancy), in the Error category. Table 1 also provides the number of distilled guidelines to illustrate the result of the distillation

process. The distilled guidelines are presented in full in Figure 4.

6 Additional guidelines for software refactoring

When studying existing user interface design guidelines, we found that for software refactoring, the distilled set of guidelines was incomplete. In general, the goal of using a computer-based system is to allow the user to complete a specified task more efficiently and to a higher standard. For simpler tasks this is relatively straightforward and usability requirements can be developed from the existing guidelines. However, for more complex tasks, such as software refactoring, these guidelines are insufficient, as they do not take into account balancing automation with user understanding and involvement as was discussed in Section 4. As such, we have extended the existing guidelines with six new guidelines to augment the existing list for use in designing interfaces for software refactoring tool support. These new guidelines were developed using Fitts’ List [6] for task allocation and automation for situation awareness theory [34, 4, 5].

The guidelines generated from Fitts’ List are: Automate error-prone tasks/sub-tasks (E7), Automate tedious/repetitive/time-consuming tasks/sub-tasks (EU5), and Automate mundane/computable tasks/sub-tasks (DU3). The additional guidelines that have arisen from situation awareness and automation literature are: Assist the user to maintain a mental model of the structure of the application system/data/task (IP6), Maximise the user’s understanding of the application system/data/task at the required levels of detail (IP7), and Provide feedback/assessment/diagnostics to allow the user to evaluate the application system/data/task (GA5). These additional guidelines appear as grey items in Figure 4.

7 Usability requirements for software refactoring

To demonstrate the use of the guidelines, and to achieve our goal of developing usability requirements for software refactoring tool support, we applied the full set of guidelines (both distilled and additional guidelines) to the task of software refactoring. The process used to develop the requirements from the guidelines via refinement is based upon the model of Pierotti [24, 25]. Following the derivation and development of the requirements, any redundant or repeated requirements were merged, with the guidelines that generated the merged requirements noted to maintain traceability for each requirement. Note that the removal

Consistency

1. Ensure things that look the same act the same and things that look different act different.
2. Be consistent with any interface standards (either explicit or implicit) for the domain/environment.

Errors

1. Assist the user to prevent errors (through feedback, constrained interface, use of redundancy).
2. Be tolerant of errors.
3. Provide understandable, polite, meaningful, informative error messages.
4. Provide a strategy to recover from errors.
5. Permit reversal of actions/ability to restart.
6. Allow the user to finish their entry/action before requiring errors to be fixed. Do not interrupt the task being completed.
7. Automate error-prone tasks/sub-tasks.

User Experience

1. Make interfaces minimal, simple to understand, organised, without redundancy, socially relevant (especially for communication) and aesthetically pleasing.
2. Provide the information, or access to the information, needed for a decision when/where the decision is needed.
3. Use the fewest number of steps/screens/actions to achieve the user's goal.

Ease of Use

1. Make the system flexible.
2. Make the system simple to use.
3. Make the system efficient to use.
4. Make the system enjoyable to use.
5. Automate tedious/repetitive/time-consuming tasks/sub-tasks.

Design for the User

1. Define the user and match the system to the user.
2. Use the user's mental model and language (avoid codes).
3. Automate mundane/computable tasks/sub-tasks

Information Processing

1. Assist the user to understand the system.
2. Minimise memorisation (i.e. reduce short-term memory load), through use of selection rather than entry, names and not numbers, predictable behaviour and access to required data at decision points.
3. Make commands and system responses self-explanatory.
4. Use abstraction or layered approaches to assist understanding.
5. Provide help and documentation, including tutorials and diagnostic tools.
6. Assist the user to maintain a mental model of the structure of the application system/data/task.
7. Maximise the user's understanding of the application system/data/task at the required levels of detail.

User Control

1. Adapt to the user's ability, allow experienced users to use shortcuts/personalise the system, and use multiple entry formats or styles.
2. Put the user in control of the system, ensure that they feel in control and can achieve what they want to achieve. Allow users to control level of detail, error messages and the choice of system style.

Goal assessment

1. Ensure the user always knows what is happening. Respond quickly, meaningfully, informatively, consistently and cleanly to user requests and actions.
2. Make it easy for the user to find out what to do next.
3. Make clear the cause of every system action or response.
4. Provide an action/response for every possible type of user input/action.
5. Provide feedback/assessment/diagnostics to allow the user to evaluate the application system/data/tasks.

Figure 4. User Interface Guidelines

<i>Original Set</i>	C	E	IP	UX	DU	UC	GA	EU	<i>Total</i>
Nielsen [21]	1	2	2	1	1	1	1	1	10
Shneiderman [27]	1	2	1	1	0	2	1	0	8
Lund [16]	4	6	4	8	3	3	2	4	34
Hansen [11]	0	6	5	0	1	5	0	0	17
Cheriton [2]	0	4	1	1	0	2	1	0	9
Gaines and Facey [10]	1	0	2	0	1	1	3	0	8
Kennedy [14]	0	3	0	3	1	4	0	1	12
Pew and Rollins [23]	0	1	0	0	3	0	1	1	6
Wasserman [33]	0	0	1	1	0	2	0	1	5
Turoff, Whitescarver and Hiltz [32]	0	2	1	0	0	1	0	2	6
Hedberg and Perry [12]	0	1	0	0	1	1	0	2	5
Total	7	27	17	15	11	22	9	12	120
Distilled	2	7	5	3	2	2	4	4	29

Table 1. Number of initial and distilled guidelines. Key: Consistency (C), Errors (E), Information Processing (IP), User Experience (UX), Design for the User(s) (DU), User Control (UC), Goal Assessment (GA), Ease of Use (EU).

of redundant or repeated requirements meant that all requirements derived from the Ease of Use (EU) guidelines have been registered under other categories due to overlap of generated requirements.

7.1 Usability requirements analysis

Refining usability guidelines into requirements resulted in an initial set of 81 usability requirements. We have selected three requirements for discussion. The full set of usability requirements is available in the Appendix of the online version of this paper².

Requirement 1: Automate code-smell identification

This requirement originates from the additional error guideline: Automate error-prone tasks/sub-tasks (E8), the additional Design for User(s) guideline: Automate mundane/computable tasks/sub-tasks (DU3), and the additional Ease of Use guideline: Automate tedious/repetitive/time-consuming tasks/sub-tasks (EU5). Code-smell identification challenges the user to synthesise and analyse a code base to identify potential undesirable trends (code-smells). It is the task of synthesis and analysis that can be error-prone, computable and tedious/repetitive/time-consuming.

Requirement 2: Allow rules for detection of code-smells to be customised (added, removed and modified) by the user

This requirement was derived from the distilled User control guidelines: Put the user in control of the system, ensure that they feel in control and can achieve what they want to achieve. Allow users to control level

of detail, error messages and the choice of system style (UC2). Allowing the user to define, modify and remove rules for how code-smells are detected, supports the definition of newly discovered code-smells, but also allows the user to control what is considered a code-smell, supporting differences in personal preference as well as corporate standards.

Requirement 3: Allow suggested changes (to the system being refactored) to be viewed, changed, or cancelled prior to transformation

Based upon the need for the user to maintain a mental model and understanding of the system being refactored, this requirement is derived from the Information Processing guidelines: Assist the user to maintain a mental model of the structure of the application system/data/task (IP6), and Maximise the user's understanding of the application system/data/task at the various required levels of detail (IP7); as well as the Goal Assessment guideline: Provide feedback/assessment/ diagnostics to allow the user to evaluate the application system/data/task (GA5).

This requirement ensures that a refactoring tool will keep the user aware of changes that will be made to their code so that they can adjust their mental model/understanding of the system accordingly.

7.2 Discussion

Overall, the use of the combined set of both the distilled and additional guidelines has yielded a more complete set of user requirements for software refactoring tools, than if only the distilled guidelines were used. The additional guidelines we have developed and applied provide the added specificity required for the domain of software

²<http://www.itee.uq.edu.au/~erica>

refactoring. This added specificity ensures that development of applications such as software refactoring tools include much needed usability support for error-prone, time-consuming, mundane and repetitive tasks; suggesting the automation of these concepts in accordance with Fitts' List.

8 Usability analysis of refactoring tools

To analyse the state-of-the-art for usability of automated refactoring tools, we selected four refactoring tools. These tools have been analysed using a small case study to identify compliance with the 81 usability requirements. The refactoring tools studied are: Eclipse 3.2 [7], Condenser 1.05 [18], RefactorIT 2.5.1 [29], and Eclipse 3.2 with the Simian UI 2.2.12 [3] plugin. These tools were selected as representative of available automated refactoring tools, with the inclusion of Eclipse and RefactorIT due to their reputation as premiere refactoring transformation tools [8], and Condenser (command-line) and Simian (GUI) as representative of code-smell detection tools.

The process for the evaluation was to use each tool to refactor a Java program that implements a library system (also used in [17]). The system has 12 initial classes with 319 LOC, which is refactored into 15 classes with 250 lines of code. After the case study, a checklist of the requirements is completed using a three-point scale (0, 0.5, 1) for the level of compliance with the usability requirement. A summary of the results of the evaluation by category is presented in Table 2. The full results of the usability evaluation are available online as an appendix to this paper³.

8.1 Discussion

Overall the requirements that the tools performed most poorly on are those related to user control, i.e. the ability to define new, and modify or delete existing code-smell, smell-to-transformation mappings and transformation definitions; and those related to automation of code-smell detection and remedy proposal. Providing feedback about code-smell instances to the user within the user's regular development view was also an issue in which further advances can be made. Finally, user control of the levels of automatic investigation, i.e., whether refactoring tools act reactively (i.e. only when the user instructs) or actively (such as with incremental compilation) was not supported by any of the studied tools.

The results of the usability requirements evaluation can also be summarised using Sheridan's levels of automation

presented in Section 4. The graph in Figure 5 shows the levels of automation for the four refactoring tools studied compared to the proposed ideal level presented in Section 4. From this graph it can be seen that none of the studied tools provided the ideal level of automation. The tools exhibiting automation closest to the ideal are Eclipse with the Simian UI plugin and Condenser, however these tools still over-automated and under-automated key parts of the refactoring process.

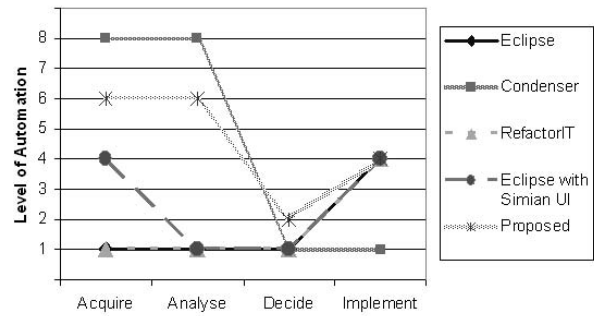


Figure 5. Automation levels for refactoring tools

The purpose of this evaluation was not to find particular faults with the tools studied but rather to identify areas in which the tools consistently do not fulfil usability requirements. From the results of the usability requirements analysis of existing tools, we are going to focus on 21 requirements in a prototype refactoring tool. The prototype will be developed as a plugin to the Eclipse software development environment [7], so that it may be used in conjunction with other plugins such as Simian UI. As our prototype is planned to be an incremental improvement on support provided by existing refactoring tools, a subset of the requirements that were not supported was selected for study and implementation. These requirements have been chosen from those that apply to code smell detection and transformation proposal, with a particular focus on integration and automation of these stages. The full set of selected usability requirements is available in the Appendix of the online version of this paper³.

The set of selected usability requirements includes:

- Automate code smell detection.
- Automate the proposal of transformation to remedy identified code smells.
- Integrate the stages, i.e. remove the need for the user to pass output from one stage into the next.

³<http://www.itee.uq.edu.au/erica>

Refactoring tool	C	E	IP	UX	DU	UC	GA	Total
Eclipse 3.2	4	8.5	9	7	7.5	6	3.5	45.5
Condenser 1.05	0.5	5	1	3	2.5	1.5	3	16.5
Refactor IT 2.5.1	4	10.5	7	9	7	4	6	47.5
Eclipse 3.2 with Simian 2.2.12	4	12	9	9	7.5	7	6	54
Total requirements by category	4	15	10	10	9	25	8	81

Table 2. Refactoring tool evaluation: requirements score by category

- Provide incremental exception (code smell) checking with inline, non-disruptive feedback.
- Support the whole refactoring process.
- Allow the user to chose if detection is constantly running (real-time, incremental), at selected intervals or only when activated.

9 Conclusion

This paper has presented an initial usability study of software refactoring tools. ISO 9241–11 has been used to specify the task of software refactoring, including the users, operation environment and the component sub-tasks. Additionally, we have analysed the level of automation and the allocation of tasks between the user and the computer, to propose an optimum level of automation for software refactoring tool support. To further study the general usability of refactoring tools, we have gathered and distilled 120 usability guidelines from 11 different sources, into a list of 29 usability guidelines. Six additional guidelines were developed to complete this list based upon Fitts' List for task allocation, and literature from the area of situation awareness and automation. Using a set of 81 usability requirements derived from the complete set of guidelines, four refactoring tools were evaluated. From these evaluations, it is clear that work to automate the areas of code-smell detection and proposal of refactoring transformations is needed to improve the usability of software refactoring tools.

In future work we will build a prototype refactoring support tool implementing the semi-automated approach to software refactoring outlined in this paper, based upon the outlined subset of usability requirements. This refactoring support will address usability requirements that are not being met by existing refactoring tool support. Being careful to balance automation with user interaction, the prototype 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. In order to determine the effectiveness of this approach in improving the usability of software refactoring tools, this prototype will also be evaluated.

Acknowledgements

The authors wish to thank Alexandra Wee 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 Distributed Systems Technology (DSTC) through the Australian Federal Government's CRC Programme (Department of Education, Science and Training).

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

References

- [1] 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.
- [2] D. Cheriton. Man-machine interface design for timesharing systems. In *Proceedings of the Annual Conference ACM 76*, pages 362–366. ACM Press, 1976.
- [3] Red Hill Consulting. Simian UI 2.2.12. <http://condenser.sourceforge.net/>, Accessed October, 2006.
- [4] M. Endsley. Automation and situation awareness. In R. Parasuraman and M. Mouloua, editors, *Automation and human performance: Theory and applications*, pages 163–181, 1996.
- [5] M. Endsley. Theoretical underpinnings of situation awareness: a critical review. In M. R. Endsley and D. J. Garland, editors, *Situation Awareness and management*, 2000.
- [6] P. Fitts. Human engineering for an effective air navigation and traffic control system. Technical report, Ohio state University Foundation Report, Columbus, OH, 1951.
- [7] Eclipse Foundation. Eclipse 3.2. <http://www.eclipse.org/>, Accessed September, 2006.
- [8] M. Fowler. Refactoring home page. <http://refactoring.com/>, Accessed October 2005.
- [9] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [10] B. Gaines and P. Facey. Some experience in interactive system development and application. In *Proceedings of IEEE 63*, volume 6, pages 894–911. IEEE Press, 1975.

- [11] W. Hansen. User engineering principles for interactive systems. In *Proceedings of AFIPS National Conference Proceedings*, volume 39, pages 523–532. AFIPS Press, 1971.
- [12] J. Hedberg and N. Perry. Human-computer interaction and CAI: A review and research prospectus. *Australian Journal of Educational Technology*, 1(1):12–20, 1985.
- [13] International Standards Organisation & International Electrotechnical Commission, Geneva, Switzerland. *International Standard ISO 9241-11 Ergonomic requirements for office work with visual display terminals (VDTs) Part 11: Guidance on Usability*, 1998.
- [14] T. Kennedy. The design of interactive procedures for man-machine communication. *International Journal of Man-Machine Studies*, 6:309–334, 1974.
- [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] A. Lund. Expert ratings of usability maxims. *Ergonomics in Design*, 5(3):15–20, July 1997.
- [17] E. Mealy and P. Strooper. Evaluating software refactoring tool support. In *Proceedings of the Australian Software Engineering Conference (ASWEC), 18th - 21st April 2006, Sydney, Australia*, pages 331–340. IEEE Press, 2006.
- [18] I. Moore. Condenser 1.05. <http://condenser.sourceforge.net/>, Accessed September, 2006.
- [19] I. Moore. Guru - a tool for automatic restructuring of Self inheritance hierarchies. In *Technology of Object-Oriented Language Systems (TOOLS)*, volume 17, pages 267–275. Prentice-Hall, 1995.
- [20] Emerson Murphy-Hill. Improving usability of refactoring tools. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 746–747. ACM Press, 2006.
- [21] J. Neilsen. Ten usability heuristics. http://www.useit.com/papers/heuristic/heuristic_list.html, Accessed June 2006.
- [22] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Department of Computer Science, 1992.
- [23] R. Pew and A. Rollins. *Dialog specification procedures*. Bolt Beranek and Newman, rev edition, 1975.
- [24] Denise Pierotti. Usability techniques: Heuristic evaluation - a system checklist. <http://www.stcsig.org/usability/topics/articles/he-checklist.html> Accessed 24 October, 2006.
- [25] Denise Pierotti. Usability techniques: Heuristic evaluation activities. <http://www.stcsig.org/usability/topics/articles/he-activities.html> Accessed 24 October, 2006.
- [26] T. Sheridan. Function allocation: Algorithm, alchemy or apostasy? *International Journal of Human-Computer Studies*, 52(2):203–216, 2000.
- [27] B. Shneiderman. *Designing the user interface. Strategies for effective human-computer interaction*. Addison-Wesley, 3rd edition, 1998.
- [28] S. Slinger. *Code Smell Detection in Eclipse*. PhD thesis, Faculty of Electrical Engineering, Mathematics and Computer Science, 2005.
- [29] Aqrif Software. RefactorIT 2.5. <http://www.refactorit.com/>, Accessed September, 2006.
- [30] M. Toleman and J. Welsh. Systematic evaluation of design choices for software development tools. *Software – Concepts and Tools*, 19:109–121, 1998.
- [31] 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.
- [32] M. Turoff, J. Whitescarver, and S. Hiltz. The human-machine interface in a computerized conferencing environment. In *Proc. IEEE Int. Conf. on Systems, Man, and Cybernetics*, pages 145–157. IEEE Press, 1977.
- [33] A. Wasserman. The design of idiot proof interactive programs. In *AFIPS National Computer Conference Proceedings*, volume 42, pages 34–38. AFIPS Press, 1973.
- [34] C. Wickens, S. Gordon, and Y. Liu. *An Introduction to Human Factors Engineering*. Addison-Wesley Longman, 1998.