Dr. Jones: A Software Design Explorer's Crystal Ball

Mark A. Foltz

The Problem: Most of software design is redesign. Redesign happens when the programmer finds a better solution for the initial design problem or the problem itself changes. Yet there is a dearth of effective tools for effective software redesign – it proceeds primarily by planning with pen and paper, away from the computer where tools could help the programmer the most. To address this problem, I am developing Dr. Jones, an assistant that helps the programmer improve the design of Java programs.

Dr. Jones diagrams the design of a Java program and allows the user to improve its design by applying refactorings, localized patterns of structural change in object-oriented programs that improve their design, without changing their visible behavior [1]. Dr. Jones records the programmer's intended refactorings and updates the diagram with the resulting (presumably improved) design. It will support the kind of design exploration a programmer would ordinarily do with pen and paper, while also providing a "crystal ball:" the ability to see design evolve. This facilitates exploring the design space of the program, enabling the programmer to choose the best alternative design.

Motivation: Effective software redesign is crucial step in reducing the cost of software maintenance. But planning such redesign is presently a cumbersome manual process. The first step – getting a clear picture of the current design – typically requires a programmer to manually reverse engineer the program into hand-drawn diagrams. Next, problems and necessary refactorings are noted on these diagrams. Finally, the programmer returns to the source and implements the refactorings using the diagram as a guide. If further refactoring is needed – or the refactoring plan has to be rethought – this process must begin from scratch.

Tools like Dr. Jones have immense potential for streamlining this process. Diagrams that accurately reflect the current program design can be generated automatically by analyzing source code. Refactoring proceeds by interacting with the diagram, where cumulative impact can be measured, alternatives explored, and plans recorded, without the need for manual redrawing.

Previous Work: Nearly all existing refactoring systems are based on the metaphor of *source code transformation* – a refactoring immediately alters the source text of the program [2]. This limits the programmer to refactorings that can be done safely and automatically. Partially automating source-level refactorings doesn't help, because it forces the programmer to make detailed programming decisions when she would rather be thinking about design.

On the other hand, software diagramming tools produce diagrams that help the programmer understand existing programs, without the ability to redesign them [3]. These tools, though useful, aren't focused on a specific task. Thus, they often produce overly complicated diagrams that contain too much information, or choose to elide information that might be crucial for design decisions.

Dr. Jones addresses these limitations. Because it refactors design, not source, it can provide a wider refactoring vocabulary to the programmer. And because refactorings are local transformations on program structure – like moving a method from one class to another – it can produce more focused and more relevant diagrams that general visualization tools.

Approach: The example in Figure 1 illustrates my approach. At the top is part of a UML-style diagram of a simple calendar application, which Dr. Jones can produce by reverse engineering the program's source. Dr. Jones' user chooses to change the design by "pulling up" the getTodoText() and getApptText() methods (which duplicate functionality) into a getItemText() method in the superclass. At the bottom of the figure, Dr. Jones has transformed the design to reflect this refactoring, and notes that the todoText and apptText fields (in italics) may need to be pulled up as well, since they were used in the corresponding original get methods[1].

The user may then choose to pursue this refactoring immediately, or explore additional design changes using other refactorings. Refactorings that Dr. Jones suggests are kept as "to-dos" so the programmer can revisit them at a later time. The outcome of a session with Dr. Jones is a list of the chosen

[1] This functionality is not yet implemented in the Dr. Jones prototype.

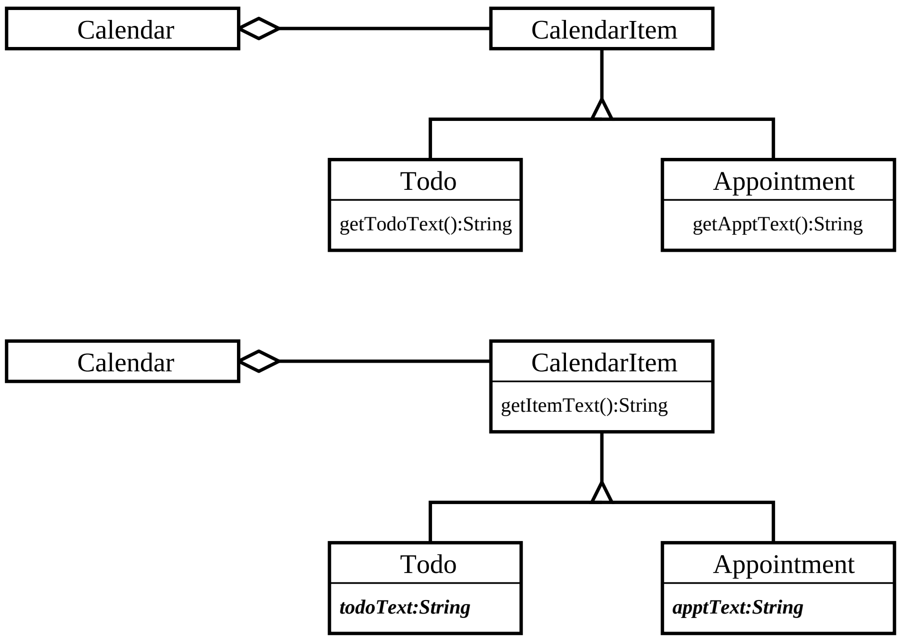

Figure 1: A refactoring interaction with Dr. Jones.

refactorings and their affected source code elements, so the programmer can later return to the source code to implement them.

Impact: Two long-term goals of software engineering research are software reuse and design rationale capture. The two are closely related, as good design rationale gives programmers valuable information on how to reuse software most effectively. Dr. Jones moves us closer to both of these goals. With a redesign tool, programmers will be more willing to adapt existing software instead of starting from scratch. And, by bringing the redesign process from pen-and-paper to the computer, it can serve as a platform for capturing valuable (re)design rationale.

Future Work: Dr. Jones faces two key challenges. First, it needs a refactoring vocabulary that makes sense to the programmer as well as the tool. I am developing a taxonomy of known refactorings around verbs like RENAME, MOVE, and ABSTRACT. With this taxonomy, the programmer can describe her intentions to Dr. Jones, and Dr. Jones can interpret them to produce the intended hypothesized design. The second challenge is to convey the outcome of these refactorings to the programmer with concise and meaningful diagrams. I am also developing a focus tracking mechanism that will follow the programmer's local refactoring context to render the most relevant parts of the program design.

Once these challenges are addressed, Dr. Jones could be extended to handle larger scale refactorings like design patterns, implement the refactoring plans it produces (under programmer guidance), or critique designs and suggest refactorings (using appropriate domain knowledge). Also, I intend to explore more natural interfaces to Dr. Jones, because programmers should be able to converse with their software design tools as naturally as with each other.

Research Support: This research is supported by MIT Project Oxygen.

References:

[1] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Object Technology Series. Addison-Wesley, Reading, MA, USA, 1999. With contributions by Kent Beck, John Brandt, William Opdyke, and Don Roberts.

[2] Don Roberts, John Brandt, and Ralph Johnson. A refactoring tool for Smalltalk. In *Proc. Theory and Practice of Object Systems*, 1997.

[3] Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Müller. Rigi: A visualization environment for reverse engineering. In *Proceedings of the 1997 International Conference on Software Engineering*, Boston, MA, USA, May 1997.