Low Fidelity User Interface Prototypes as Agile Refactoring Tools

Michael Wainer
Department of Computer Science
Southern Illinois University Carbondale
Carbondale, IL 62901

Abstract - Agile development relies on refactoring and iterations to continually refine working code to meet requirements. The goal of refactoring is to produce code which is easy to understand and maintain. Refactoring user interface (UI) code is subject to ordinary refactorings plus others specific to this area. Even as refactoring tools grow more powerful, most programmers do not fully take advantage of automated support for many of the more complicated refactorings. As an alternative to automating source code refactoring tools with more and more sophisticated code transformations, our proposal brings in a more human element considering low fidelity prototypes (screen sketches etc.) as high-level refactoring tools. A beneficial side-effect may also be improved understanding of refactoring and earlier clarifications on important design issues. Suggestions are made for how low fidelity prototypes, already proven so useful in interaction design, can be used to get an earlier start on refactoring as well.

Keywords: refactoring, prototyping, HCI, agile, tools

1 Introduction

Users have high expectations for their software and in markets where users have a choice, careful consideration of the user interaction design, including the graphical user interface (GUI), is a must. Developing highly usable software requires a considerable effort: practically universal recommendations are to involve users early and often, iterate, and evaluate designs using prototypes.

When developing such software in an agile manner, the use of low fidelity prototypes such as screen sketches and paper prototypes is extremely valuable. Such prototypes are effective not only for customer design evaluations but also as a way of specifying and communicating design ideas among the team. They are fast, cheap and easy to produce and modify. Agile interaction design can be coupled with agile software development by using an interaction design track in parallel with the development track. This dual track approach is discussed in section 2.

A common approach in coding GUI interfaces is to separate the internal data and logic (the business logic) from the particulars of how it is visualized and interacted with. Section 3 explores possible implementation strategies for coding GUIs including an introduction to the Model-View-Controller paradigm. Most interfaces use many components (widgets) which must be properly organized within the application's windows. Developers may be assisted in these tasks by ever more capable visual interface builder tools that are often part of the development environment.

Agile processes produce code which evolves through many iterations. Rather then spending a large amount of upfront time deriving a detailed and unchanging specification, change is recognized as a certainty so design work will continue throughout the project. To make this feasible the code must adapt; not only to add new features but also to stay easy to understand and maintain. Thus refactoring, improving and clarifying the evolving code (without changing its functionality) has become an important agile practice. Refactoring and the related idea of prefactoring with a special emphasis on interface code are taken up in sections 4 and 5.

Given the importance of refactoring when developing an application in an agile manner, section 6 looks at how low fidelity prototypes can be used as refactoring tools. As such prototypes are already available for communicating the interaction design, we show that they can also allow for rapid communication and exchange of ideas regarding refactoring options. An example of this is illustrated by following a sample case introduced in earlier sections. Finally, section 7 summarizes and concludes this paper and gives a look towards future work.

2 Agile Development

We assume an agile development methodology [1] for both GUI and application development as a whole. Agile development expects change and therefore does not put much effort into big upfront design. Instead, it is recognized that the code will have to evolve. Agile development will also favor using lightweight modeling artifacts and documentation as the main emphasis is on working code.
While agile development does not favor big upfront planning, some form of planning is often appropriate. During an iteration, developers will be implementing code derived, in part, from an earlier iteration's interaction design work. As shown in Figure 1, it has been found useful to consider dual tracks of development: one for the interaction design and one for application development [2, 3]. Rather than striving for a complete and detailed interaction design, the interface is designed in enough detail for just the features present in the current development iteration. The interaction design team (which may overlap with the developers) works with prototypes and/or existing code from previous iterations to create a design for the next iteration. Designs are expressed to the developers in an agile manner – often as paper or low fidelity prototypes [2]. An iteration 0 may sometimes be used for initial setup and exploration of internal details which have little dependence on the user interface.

Paper prototypes, screen sketches and navigation flow diagrams have proven to be useful artifacts to communicate the specification for the user interface of applications. These artifacts may be created by hand or by various drawing or modeling applications. They are generally not used to directly generate code but to communicate the design amongst members of the team. Paper prototypes can simulate much of the behavior of a system before any coding. The sketchy nature of low fidelity artifacts invites more appropriate user feedback on early design aspects.

To illustrate more specifically, we use an example of a small Java Swing application, PolygonMaker, which is to be developed in an agile iterative manner. PolygonMaker is an application to assist developers in generating the source code required to specify a polygon for various systems or languages. The user is able to specify the polygon points and initiate a process to generate the source representing the specified polygon. Initial choices might be as a Java2D Shape, POVRay (a ray-tracing package) prism object, etc. A screen sketch of PolygonMaker's main window is shown in Figure 2.

The user specifies new points in the entry field at the lower left and the polygon is visualized as the table of points on the left and the diagram on the right. The diagram shows the last point entered with emphasis and connects it to the first point with a dashed line to illustrate the resulting polygon. The design is expected to change as the application evolves. The user interface aspects may be under pressure to change in different ways than the underlying application's algorithms and data structures (its "business logic"). The next section discusses a coding approach commonly used to combat this problem as well as the help that can be offered by visual interface construction tools.

3 Implementation Strategy

A commonly accepted principle of good object oriented design is the “Single Responsibility Principle” [1] which seeks to limit the pressures of change exerted upon objects by having objects focus upon a single facet. In this way, they will only have to be modified in response to changes impacting that facet and are kept rather isolated from other possible changes. Like many design principles, while the basic concept seems simple, in practice it can be difficult to apply. In terms of software supporting graphical user interfaces, the Model-View-Controller(MVC) paradigm is commonly used to separate concerns (see Figure 3).

In the Model-View-Controller(MVC) the internal core logic of application entities (business logic) is kept separate from the user interface. Business rules and interface concerns change for different reasons and separating these concerns helps to isolate changes in one from affecting the other. Having a separable model also makes it possible to develop and test the model independently or in parallel with
the GUI. Test Driven Development, a common practice in agile development, incrementally builds a suite of tests which drive and direct the development of the model.

Since constructing a GUI involves the specification and creation of its visual aspects, it is only natural that visually based tools can be very helpful. In this paper, WindowBuilder a visual construction tool for Java Swing interfaces (and other systems as well) is taken as representative of these utilities. These tools usually provide similar interactive design spaces. A design space visually shows the interface under construction and allows changes to be made by dragging components from a palette. A structure view shows the containment hierarchy of the components. Selected components are highlighted in both the design and structure views. A selected component also has properties that are available for inspection and modification in a properties view.

Figure 4 shows a screenshot of WindowBuilder working on another screen for the PolygonMaker example application. WindowBuilder, now freely available as a plugin for the Eclipse IDE is highly capable and recently acquired and released as open source by Google. While such tools reduce the tedium of coding many details of a GUI, they have drawbacks as well. From unexpected widget sizes and layout issues, to failures and delays in parsing, it still can be far easier to sketch a screenshot by hand (especially if custom graphics components are required). Thus even though visual build tools are increasingly powerful and useful in building code, they have not eliminated the need for simple low-fidelity prototypes for early design.

It is also interesting to note that the code generated by GUI builders is often quite different from what would typically be constructed manually. Developers may have some control over aspects of the code generated but in general, machine generated code will use uninformative names and create fewer but lengthier methods. The code tends to be easier for the tool to generate and read than it is for humans. This may become an issue especially in the context of an it-
erative development process in which the code is continually revised. In effect, reverse engineering will be needed to interpret and adapt the code generated by the visual build tool[5].

4 Refactoring

Over time as code evolves it often begins to diminish in clarity and quality: methods grow too long, logic gets more convoluted, the original names, data structures and algorithms may become less appropriate. These problem areas in the code are referred to as smells. Many specific code smells have been cataloged along with suggested refactorings to remove them. Refactorings are transformations of the code with the goal of making the code easier to understand and maintain [6]. Specifically, refactorings, while they may be applied to improve design in preparation for new features, do not themselves aim to implement new features. Since agile development methods use continuous design, rather than big upfront design, refactoring is essential to maintain the quality of the code as it evolves.

A very frequent refactoring activity, rename, is supported both in visual construction tools as well as through source code editors. Good names can significantly improve the ability of the code to communicate its intent. Naming is so important that Martin in “Clean Code”[7] provides seven heuristics devoted to just that consideration alone.

Another frequently used refactoring activity is extract. It is often used to reduce the number of lines in methods having the “long method smell”. A method is reduced in size by extracting some of its lines into a new method. The original method then refers to the newly extracted method. Once again naming is important as the new method should have a name which makes its purpose clear. A call to a clearly named purpose often eliminates the need to comment a sequence of statements.

As refactoring becomes more widely practiced and understood, IDEs are increasing support for automated refactorings. Eclipse[8] provides a refactoring menu with many of the refactorings from Fowler[6]. Research is underway to study more about how developers use refactoring tools in their work [9-11]. While IDEs like Eclipse now provide many refactoring options, it has been difficult to determine exactly how refactoring is used in practice[10]. Some studies try to determine refactoring histories by examining repository data while others have used instrumented IDEs to collect and report various types of information. Experiments and interviews with developers are another approach. Overall it seems as though only a handful of the automated refactoring features are widely used. Some attribute this, in part, to the difficulty (poor usability) of using refactoring tools[10-11].

It is logical to assume that certain types of refactorings might be associated with specific types of code. Indeed, GUI code analyzed by participants in Mäntylä and Lasseni-us'[9] refactoring study was noticed to tend toward typical characteristic smells. Furthermore, Marinilli[12] suggests several specific refactorings that are applicable to GUI code. Visual interface construction tools will often directly support some forms of refactorings (even as the code they generate may necessitate others [5]). The complexity of supporting a modern interface suggest a multitude of refactorings which may be possible including: switching layout managers, obtaining text and image content from a resource loading system, internals of event handling, undo and logging. Aside from pure UI issues, the model, or what the application uses as its internal model(s) might also be refactored.

5 Refactoring Early: Prefactoring

Recognizing that code will need to change and that there is value in refactoring, can the practice be advanced by utilizing the knowledge gained from previous software development experiences? That is the idea behind Prefactoring as put forward by Pugh[13]. Some of the main concepts emphasized in prefactoring are listed below:

Interface: more focus on what components should do rather than on how.
Abstraction: express ideas with pseudocode.
Separation of Concerns: responsibilities are split between classes, methods etc.
Readability: code should clearly communicate what it does.

There is considerable overlap between these concepts and those supporting good interface design and implementation. Abstraction and interface manifest themselves in the ideals of expressing interaction design with low fidelity prototypes: offering a high-level way to visually explore what an application should do. Parameterization permits an abstraction to consider a generalization of items rather than each individual instance separately. Separation of concerns mirrors the MVC paradigm. Readability relates to choosing meaningful names and appropriate chunking of ideas. Grouping related interface items together often makes sense for the user (logical groupings, consistency) as well as for the implementation (readability, reusability, reduced duplication).

6 Low Fidelity Prototypes: Refactoring Tools

As software developers prepare to start a new iteration, assume they have available (or can quickly construct) a low fidelity prototype. Consider how that prototype could be used for refactoring (or prefactoring). While not meant to
be an exhaustive list, for clarity, we will focus on refactorings concerned with *naming*, *extraction* and *parameterization*. These refactorings and how they may be explored with low fidelity prototypes are now considered in more detail.

**name/rename**: interface items may have names visible to the user (labels, titles etc.) as well as internal names. Especially objects that are to be referenced at larger scopes, should have clear names. Naming and renaming can iterate quickly and using the prototype it is easy to involve interaction designers and/or customers in the discussion about appropriate names. Good names are extremely important for creating highly readable code.

**extract**: a window or dialog may contain many components or areas devoted to different types of information or interaction. The implementation might specifically use separate methods extracted to group together code devoted toward different aspects. Extract may also be useful for breaking down event handling routines. New classes may also be extracted.

**parameterize**: a window, pane or other object or method may be representative of a family of similar objects. Rather than create each item with distinct code, merge implementations and use parameters to specialize. This can reduce instances of duplicate code.

Recall the MVC paradigm discussed earlier. View components may be the first to mind when examining a prototype. In Figure 5, view aspects are clearly the polygon graph and its table of points as well as the view of the generated source code. The graph is likely to be a custom component but it also brings to mind the internal polygon model and what information it contains. A basic polygon model may be composed of a list of vertices but to support graphical editing the application model may add a “selected vertex” data structure as well. In this application, the model is supplemented by additional converter objects to generate source code conversions for polygons. These model classes (or perhaps interfaces), objects and methods all require good names. Models can be tested without the GUI so initial ideas regarding names, method parameters etc. may already exist.

The controller coordinates the events and provides buttons like “Add”, “Copy to Clipboard”, “Convert” etc. In some cases, controls may be enabled/disabled based upon the state of the model (i.e. “Convert” should not be enabled if at least 3 points haven’t been defined). Since such controls need to be referenced to update their status, they require special naming considerations. An alternative implementation strategy might use Swing Actions but again the actions should be carefully named.

To illustrate more specifically, the “Convert” dialog (bottom left of Figure 5) is opened once the user has specified a
polygon and clicked the Convert button. It is shown for a POVray conversion but no doubt it will be very similar to the conversion dialogs for other source code options. Rather than duplicate essentially the same thing (removing duplication is a common refactoring goal), the dialog can be extracted into a new class and its constructor parameterized. The orange dashed ellipses indicate possible parameter items – the items are also annotated for naming. Deriving good names naturally leads to thoughts about what the data represents and, if it is dynamic, how it changes. Here the data could be passed in directly as a parameter or be generated by other objects (polygon and converter objects).

Of course, once implemented, items may continue to change and new prototype sketches generated by the I.D. team reveal how. Figure 6 shows both the implementation of the conversion dialog along with the design sketch for the next iteration. Screenshots of the working software (and other design diagrams, uml etc.) can be used along side the prototype, to plan the refactoring and to prepare for design changes and new features. Prototype sketches can be scanned and overlaid with notes regarding the current area of focus (refactoring overlays). Figure 6 concerns moving the ConversionDialog into a section of the main window. The notes show a refactoring plan to extract the content creation aspect of the dialog to a method that instead creates a pane. The pane can then be used in the main window. To work with all different types of conversions, parameterization for different type names and source code is maintained. Since the pane will remain open, an update method is also proposed.

Notice that extraction of the code requires new names, decisions of what parameters to use, as well as a determination of where the code should go. Renaming or eliminating the original class is a possibility. Even though the existing code will ultimately be refactored using the refactoring tools of the IDE (and/or manual code refactoring), it can still be helpful to work out a refactoring plan by beginning to refactor using the low fidelity prototype. This can provide a more fluid environment for thought and discussion with a greater sense of context as the existing code is prepared for modification to fit into the new design. A rapid exchange of ideas over the low fidelity prototype may serve to explore and eliminate some refactorings before making any changes to the actual source code.

As new features are considered, the existing code modifications move beyond refactoring but the issues of renaming, extraction etc. become immediately apparent and the process repeats. Prototypes with their refactoring overlays may be saved (scanned and checked into the team repository) for review during retrospectives or whenever needed.

Figure 6: Further design iteration moves the conversion dialog information into a section of the main window.
7 Conclusion

Much effort has been placed into making powerful automated refactoring tools available to developers. Research has shown that these tools do not seem to be used as often as might be expected. Some tools suffer from usability issues and there may also be a lack of awareness of complex refactorings among developers. Rather than focusing exclusively on building more powerful automation into refactoring tools, this paper proposes a contrary idea; expand refactoring tools to the more human side by overlaying refactorings onto low fidelity prototypes.

A criticism of prefactoring, and perhaps this method, is that we may fall into the trap of too much upfront planning. That is obviously not the intent. We merely wish to introduce “prototypes as refactoring tools” as one more tool to be used where appropriate. As is the case with simple low fidelity prototypes, they provide a fast and flexible way to consider many ideas quickly and in a way that serves to foster communication among team members. Much of maintaining good clean code involves naming and communicating intent. Informal, light and fast, refactoring with prototypes allows for quick iterations over naming and helps to raise important ideas about what aspects might be extracted and parameterized.

To quote from the Agile manifesto[14]: “Individuals and interactions over processes and tools”. Teammates working together around a whiteboard or table on the initial refactoring for an iteration using prototypes promotes this idea. While there is certainly value in automated refactoring tools, ultimately there is a very human side to crafting code which effectively expresses its intent. Ideally developers will have a broad spectrum of tools to help them succeed in this challenging endeavor.

We plan to continue exploring these ideas. It is expected that refactoring using prototypes will also aid in brainstorming with the team (and customer) and help in story estimation and splitting. Refactoring discussions using prototypes may also raise awareness of refactoring options in general and result in more use of automated refactoring tools as well. We are hoping to collect data on these anticipated effects as well as to determine what if any new notations might be appropriate for annotating refactoring thoughts onto screen sketches etc.

8 References


All web sources were accessed March 2011.