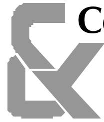

Constantine & Lockwood, Ltd.

Rapid Abstract Prototyping

Larry Constantine
Director of Research & Development | Constantine & Lockwood, Ltd.

Abstract. Abstract prototypes are an intermediate form that can speed the user interface design process and improve the quality of the results. Abstract prototypes help bridge the conceptual gap between a task model and a representational paper prototype for a user interface design. This short article introduces abstract prototypes in the form of simple content inventories and content navigation maps. An example illustrates the process of using abstract prototypes and suggests the possible advantages over other prototyping techniques.

Keywords: abstract prototypes, content models, task-driven design, usage-centered design, user interface design, content navigation, paper prototypes

Learn more about abstract modeling and usage-centered design at http://www.forUse.com.

The programmer smiled proudly as he handed us a floppy disk. It was the morning of the third day in a week-long class on usage-centered design, and while the rest of the class was working on design models and paper prototypes, this enterprising developer had hacked up a visual prototype in Delphi on his laptop. We loaded the floppy and launched the application. Compared to the collections of Post-it™ notes and the rough pencil sketches scattered over table-tops around the room, his prototype looked a lot prettier and a lot more like a real program. That, precisely, was the problem. Its good looks hid numerous design problems that would have been easier to spot and to avoid if the prototype had not looked quite so real, quite so concrete.

Abstract prototyping is a way to avoid the seduction of attractive prototypes that disguise weak designs. Using sticky notes or other simple tricks, abstract prototypes can represent the contents of a user interface without showing what it looks like. Lucy Lockwood and I developed this design technique to bridge between user problems and visual prototypes. By making better use of modern visual development tools, abstract prototyping can speed and simplify the design of highly usable systems and help you produce improved, more innovative software products.

Revised and corrected version of an article originally published in *Software Development,* October 1998. The author may be contacted at Constantine & Lockwood, Ltd., 58 Kathleen Circle, Rowley, MA 01969; tel: 1 (978) 948 5012; fax: 1 (978) 948 5036; email: LConstantine@compuserve.com
© 1998, 2002 L. L. Constantine

Trapped in Concrete

With the advent of modern visual development tools, such as, Delphi, Visual Studio, JBuilder, and Visual Age, constructing graphical user interfaces has become so easy that there hardly seems to be any need for design at all. Just open a new form, throw on a data-aware display grid, drag-and-drop a menu bar plus a few widgets from a tool palette, and without so much as a pause for a sip of coffee, you have the user interface all laid out. And it looks pretty good, too!

On the plus side, visual development tools facilitate visual thinking, making it easy to lay out user interfaces merely by fiddling around and eyeballing the results. No one who has used such systems needs to be told that they are fast, powerful, and easy to use. However, their very simplicity can encourage rapid and undisciplined assembly of user interfaces. Forms, windows, or dialogue boxes are so easily created that interaction contexts can multiply like tribbles. Furthermore, the assortment of GUI widgets standing ready for instant instantiation encourages the proliferation of features. The result can be bloated interfaces, stuffed with features, that take the user on a merry chase from dialogue to dialogue.

The power of the tools reinforces the practice of just throwing something together in the hope of fixing it later. But if there is not enough time to think it through and do it right now, there probably won't be enough time later. Usually all that is accomplished later is a bit of aesthetic polish, straightening out the alignment of some of the widgets, adding a glyph here or there, and re-ordering a few of the controls.

Because visual development tools can make even careless design look good, those aspects of user interfaces that most profoundly affect usability may end up being neglected. What capabilities must be present within the user interface for it to solve the users' problems? How should they be organized into handy collections? How should the work flow within and between the various parts of the interface? These are all questions that are most easily answered with abstract models.

Why Model?

As a breed, visual developers do not spend a lot of time building models. Like the participant in our design training, many would probably argue that it's quicker and easier just to start programming.

We use models in our work precisely because they actually speed up development and help us get to a superior solution more quickly. Good models clarify design issues and highlight tradeoffs, so decisions can be resolved rapidly. Perhaps most importantly, models help us to know what to build., which really speeds things up because any time spent building the wrong system or creating unnecessary features is time completely wasted. Even under the tightest deadlines, the most effective developers construct analysis and design models because they know that modeling actually saves time [Constantine, 1995b; 2000]. In fact, modeling may become even more vital as the time-scale of development is compressed [Thomas, 1998].

Models also help us to deliver better, more robust systems. One would hardly want to fly in a commercial aircraft that had been built without benefit of wind-tunnel tests of scale models or simulations using computer models. As in other fields, models help software developers to document the results of discussions and to communicate their understanding of the issues and answers to others: to other members of the development team, to clients, to end users [Constantine, 1994].

We devised abstract prototyping because we found that the sooner developers started drawing realistic pictures or positioning real widgets, the longer it took them to converge on a good design. Abstract models are always much simpler than the real thing. They help us avoid being misled or confused by messy and distracting detail, making it easier for us to focus on the whole, on the broad picture and overall issues. Concrete tools tend to lead to concrete thinking.

They draw us into thinking in conventional and familiar terms and hinder us from thinking creatively. Abstract models invite innovation, encouraging us to fill in the details more creatively [Constantine, 1996].

Content Modeling

We do abstract prototyping using content models. A content model represents the contents of a user interface—what it contains—without regard for the details of what the user interface looks like or how it behaves. In a content model, a user interface is a collection of materials, tools, and working spaces, all of which are described in terms of the functions they serve, their purposes, or how they are used. Materials are simply the stuff that users are interested in seeing or manipulating, tools are what enables users to do things with materials, and working spaces are the various parts of the user interface that combine the tools and materials together into useful collections.

It is not hard to see that tools, materials, and working spaces can be used to describe a lot more than just software user interfaces. Most tasks are carried out in particular places using appropriate collections of tools and materials to complete the work. A machine shop and a kitchen are laid out differently to suit different tasks. They contain different equipment and supplies, which are stored and organized in ways suited to their distinct uses. Before baking a lemon-poppy-seed cake, I will assemble a different set of utensils and a different set of ingredients than were I to be cooking up a potato curry.

Good user interface design, then, is a matter of assuring that all the tools and materials needed by the user are found on the user interface and are assembled into collections that are convenient for the tasks to be performed with the software. That is what abstract prototyping helps us do.

Low-Tech, Low-Fi

We take a decidedly low-tech approach to abstract prototyping. In fact, in our experience, better designs result from more primitive, and therefore more abstract, modeling tools.

We use sheets of paper to represent the working spaces (or interaction contexts, as they are sometimes called) within the user interface, and we fill them with Post-it™ notes to represent the needed tools and materials: hot colors for tools and cool ones for materials. Sticky notes have a lot of advantages for this purpose. They are easily rearranged, so a variety of radically different organizations can be tried out remarkably quickly. Sticky notes also do not look much like real GUI widgets. We can invent or invoke anything we can describe or define in words without worrying about what it will look like or how we will program it. Some human-computer interaction specialists refer to this sort of a model as a “low-fidelity prototype,” because it does not look much like a real user interface, but we think this misses the point: a content model is really a high-fidelity abstract model.

Let’s illustrate content modeling with a highly simplified example. Imagine we are designing a tool to be used by software developers for tracking usability problems and user interface defects in the software they produce. We want a simple way to mark the location of each problem on the user interface design, to identify the type of problem, to annotate it with a brief description if needed, and to indicate the seriousness of the defect. For instance, we might want to mark a tool-button and note that it has an obscure icon, a moderately serious problem. We know from experience that usability problems fall into predictable categories, which we do not want to have to keep re-entering. Predefined problem types will also make it easier to track our performance over time, learning what mistakes to avoid. We will also want to track the status of identified defects along with the responsible developer.

Starting Points

How do we start the content model? Programmers who are most comfortable thinking in terms of procedures or methods may be inclined to start thinking about the tools or active features of the user interface, but we get better results by figuring out first how to meet the information needs of users. Once the data and information requirements are met, the tools and functions needed to manipulate or transform these materials are easily identified. Others who have used similar approaches also find advantages in giving first attention to the data or information required within the user interface [Lauesen and Harning 1993].

We usually begin by asking simple questions: What information will the user need to create, examine, or change? What information will the user need to present or offer to the system? What information will the user require from the system? What views, perspectives, or forms of presentation will be of value to users? What conditions or events will the user need to convey to the system, and about what conditions or events will the user need to be informed?

Readers familiar with use cases will recognize such questions, which are much the same questions that are asked in identifying use cases. This is no accident. One goal of good user interface design is to provide simple and direct support of the tasks of interest to users. In usage-centered design, the interface content model is derived from essential use cases (see sidebar), which makes the process more straightforward and favors clean, simple designs. However, even if you are not working from use cases, thinking in terms of basic user needs can still be effective.

ABSTRACT CASES

Usage-centered design employs a simplified, abstract form of use cases known as essential use cases [Constantine, 1995a]. Rather than being cast in the form of concrete and specific user actions and system responses, as are the more traditional or concrete use cases, essential use cases are expressed in terms of user intentions and system responsibilities. For example, for recordingUsabilityDefect, the essential use case might have a narrative like this:

recordingUsabilityDefect

USER INTENTION	SYSTEM RESPONSIBILITY
optionally select interface context	
	show interface context
indicate defect location	
	show marked interface context suggest defect types
pick type of defect optionally describe note defect severity	
	show entered details
confirm	
	remember defect with details

In so far as practical, essential models are technology and implementation independent. In the essential use case above, no mention is made of how information is displayed or how selections are made, for example. Such use cases provide a smooth transition into abstract prototyping using content models.

We start by labeling a piece of paper as the current working space. The working space could become an application window, a dialogue box, a child window, or even just one page in a tabbed dialogue, but we defer the choice until later. At this point the main issue is what spaces will contain what tools and materials. The name we give to a working space usually reflects what users will do in that part of the user interface. Let's call our current working space, "Recording Usability Problems."

To record usability problems, the user will need some kind of picture of each part of the user interface, each interaction context within the design being tracked. Since a complete user interface design typically consists of a collection of dialogues, forms, and other contexts, the user will need access to the entire collection. So, we label some sticky notes for these

materials—an **interfaceContextHolder** to hold an image, an **interfaceContextName**, and an **interfaceContextCollection**—and slap them onto the paper. To ensure standard descriptions of usability problems, we add a collection of **usabilityDefectTypes** from which the user can select. Since a specific usability problem may be thought of as consisting of a type, a description, and a degree of severity, we can simplify the picture by combining these closely related ideas into a single material called the **usabilityProblem**.

Materials need to be supported with appropriate tools. To identify the necessary tools, we ask questions like: How will users need to operate upon or change the information of interest on the user interface? What capabilities or facilities will need to be supplied by the user interface for users to be able to accomplish their tasks?

For the usability problem tracker, we see the need to be able to switch images of interaction contexts, so we add a **contextNavigator** tool to page forward and back through a collection of images. If we have thought carefully about how the defect tracker will be used, we will realize that the user will sometimes need to switch directly from one image to another, so we add an **interfaceContextSelector** tool. Since we will need a means for marking the location of problems or defects, we also add a **defectLocationMarker** tool. As we add abstract components we might have some ideas about their implemented form; these we can write directly on the sticky notes, not as final decisions but as possible suggestions.

At this stage, we might have something like the working space illustrated in Figure 1. As shown, it can be useful to group tools and materials together to indicate their relationships or to suggest similarities.

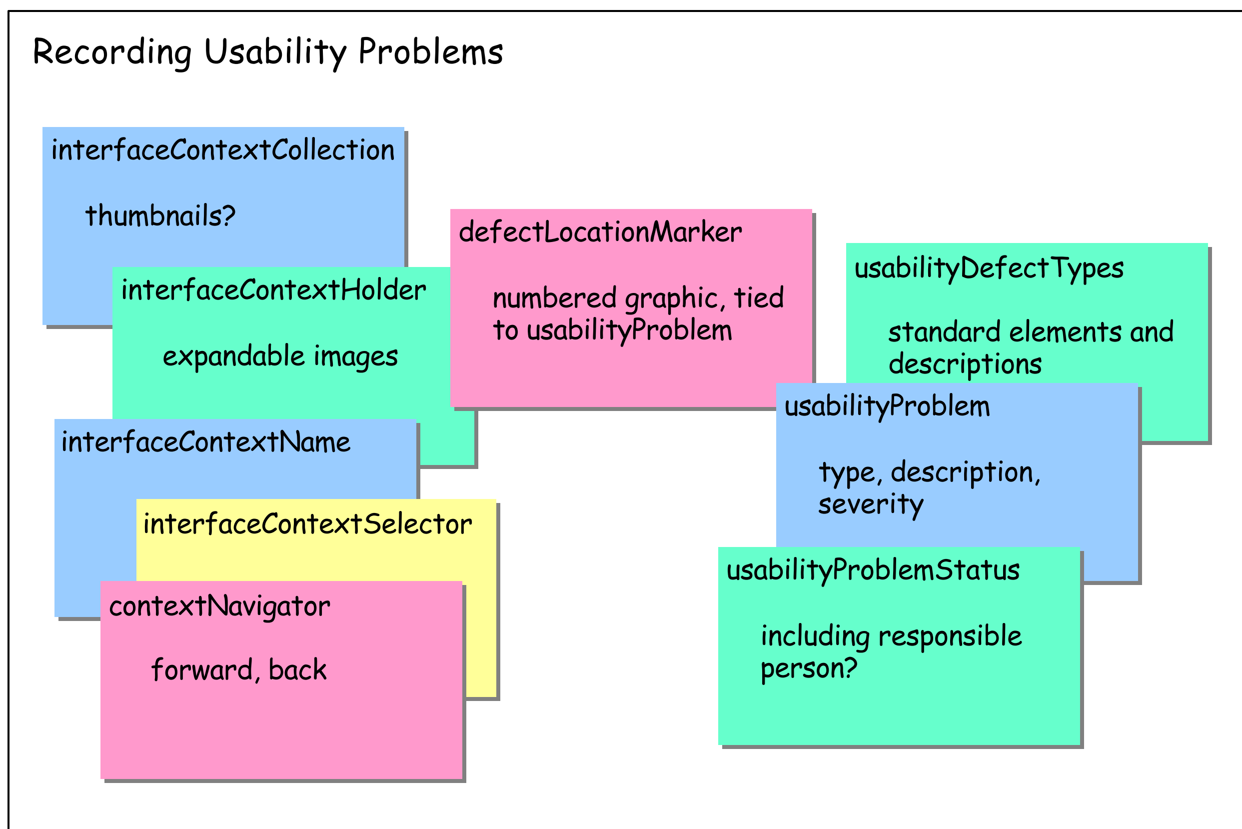

Figure 1 – Working space in progress.

Is this enough to solve the problem of recording usability defects? It is hard to be certain without knowing all the things the user will want to do. Certainly for a complete, practical

system we would need more. We would need some housekeeping or administrative tools, such as for saving an annotated collection of interface images. In addition, the user would need to see and print lists of all the defects in a project, summarized by interaction context, by defect type, by status, and by responsible person, for example

If we started creating a new working space for "Reviewing Usability Defects by Interaction Context," we would quickly find ourselves filling it up with almost all the same tools and materials as the "Recording Usability Problems" working space. We would not need a **usabilityProblemTypes**, but we will need an entire list of problem descriptions and a whole set of problem markers. In content modeling, whenever we see two working spaces with substantially the same contents, we consider combining them if it will simplify the user interface. In this case, merging the two working spaces into one probably has more advantages than disadvantages.

Fairly quickly, we will have identified a number of distinct but interrelated working spaces or interaction contexts. How these are interconnected is an important part of the overall organization—the architecture—of the user interface. A complete content model really consists of two parts: the content model itself, and a navigation map that shows how all the different interaction contexts are interconnected so that the user can navigate among them. In practice, we develop these two views of the interface contents in tandem.

A simple and flexible notation suffices for the navigation map. (See Figure 2.) The navigation map allows us to look at the complete organization of even very complex user interfaces with many screens or dialogues. Of course, the usability defect tracker is not very complex; Figure 3 represents one possible navigation map for it.

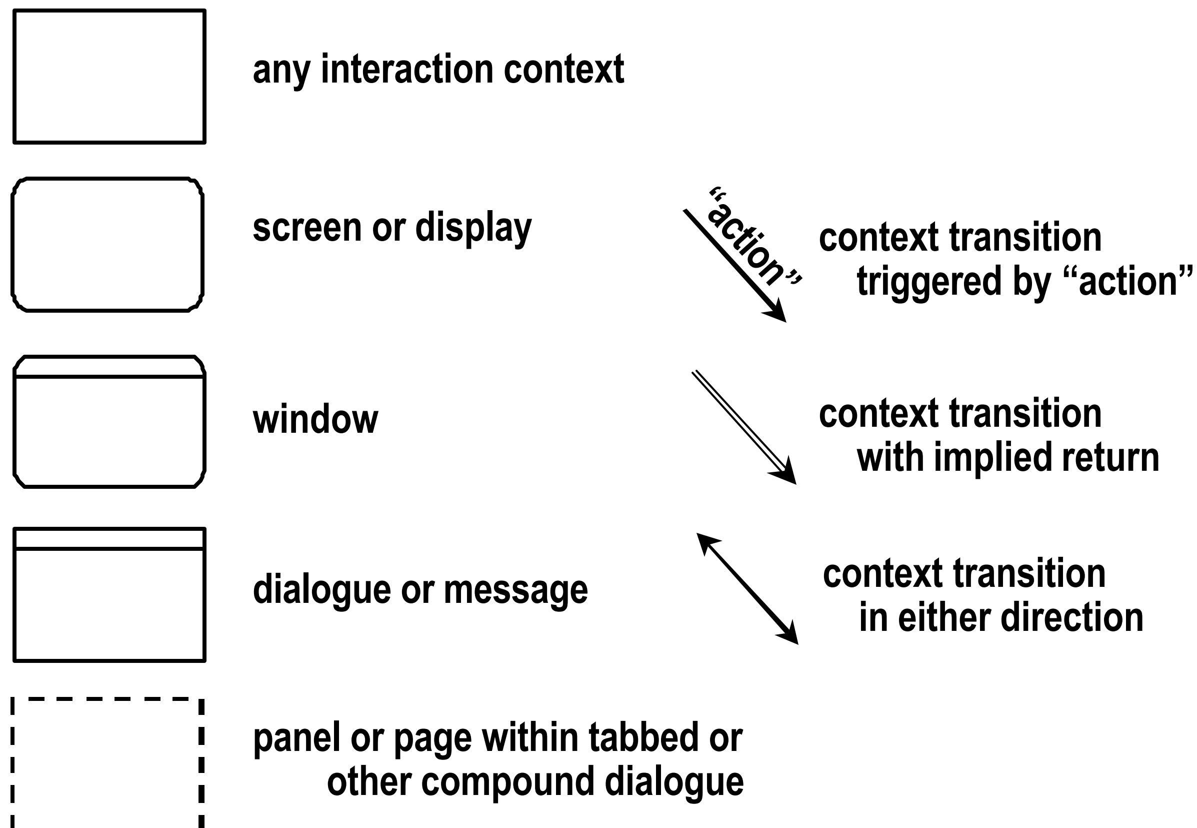

Figure 2 – Notation for navigation.

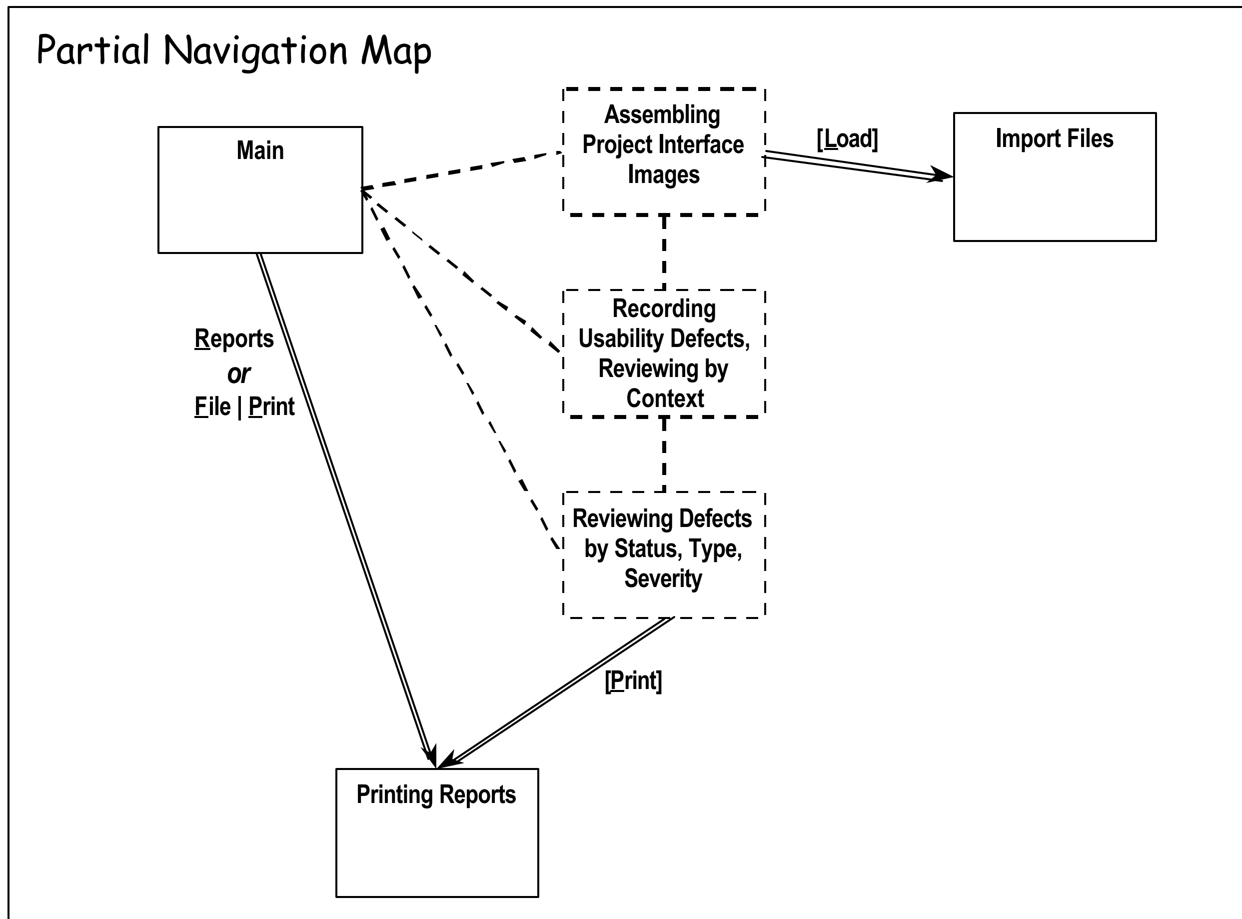

Figure 3 – Navigation map.

Down to Specifics

For most designers, translating an abstract prototype into a visual design or a working prototype is where the real fun begins. Typically, we start with a somewhat naïve transliteration: one abstract component implemented as a single standard GUI widget. For example, the **`interfaceContextHolder`** could become a simple bitmap display that expands to full screen on double-click.

Often, however, there are opportunities for creative combinations that save screen real estate or simplify operation. For example, the **`interfaceContextName`**, **`interfaceContextSelector`** could be implemented as separate widgets, but a single drop-down combo box would cover both functions. Alternatively, the forward-and-back functions of the **`contextNavigator`** could be combined with **`interfaceContextName`** using a spin box. In fact, all three capabilities might be realized with the single custom widget shown in Figure 4. Although non-standard, carefully crafted components based on simple visual and behavioral theme-and-variation can be easy for users to figure out and can significantly improve usability.

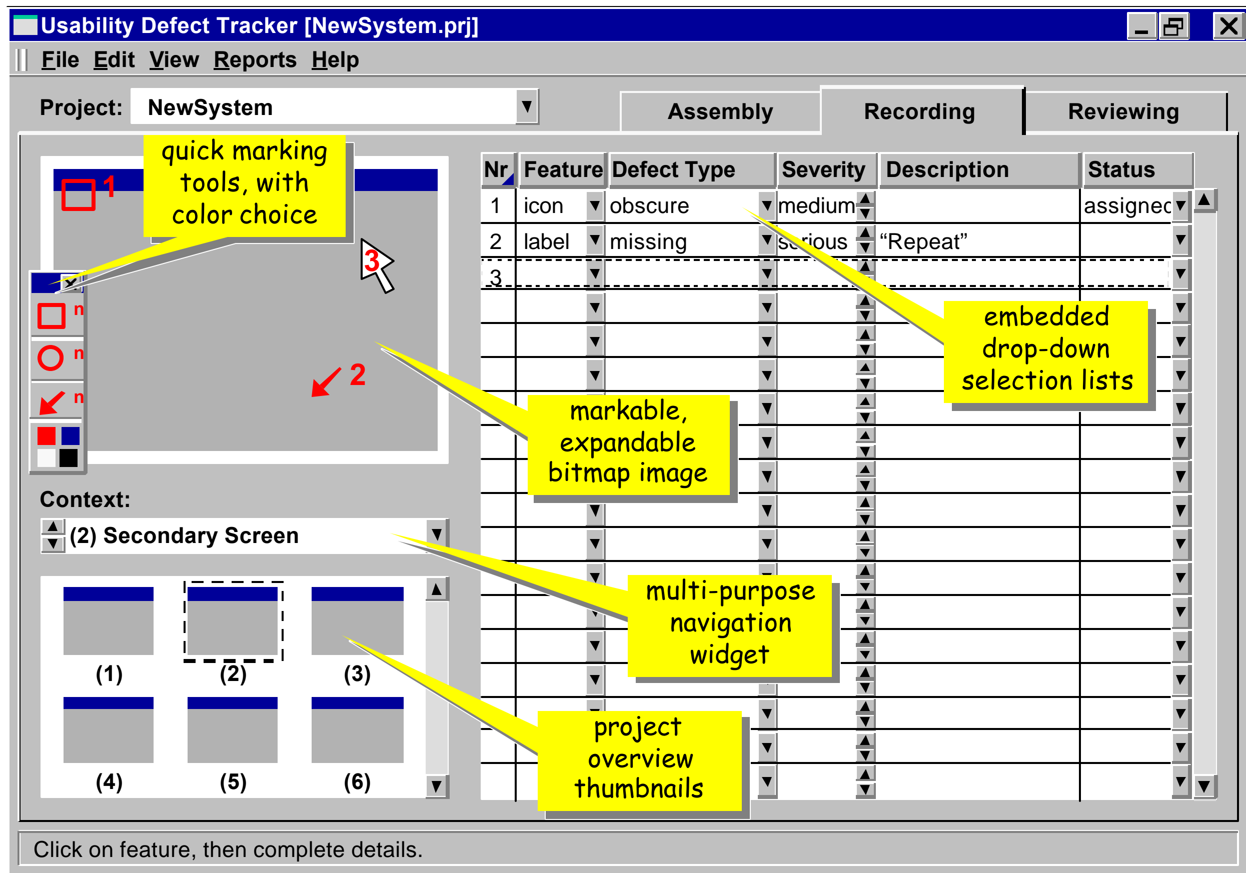

Figure 4 – Concrete implementation model.

The visual prototype in Figure 4, one of many possible designs, incorporates a number of these "creative translations." The **`defectLocationMarker`** becomes a floating tool palette; the **`usabilityProblemTypes`** and **`usabilityProblemList`** have been combined using embedded drop-down lists. Usability inspections and testing can no doubt further improve on this initial visual design.

Prototype Payoff

Abstract prototyping helps you to figure out quickly what you are building before you start positioning and linking GUI widgets with a visual development tool. In our experience you get superior user interfaces in fewer iterations: designs that are simultaneously more innovative and a better match to user needs. The small investment in constructing a content model can lead to big-time savings over the entire project. Visual programming goes even faster when you already know what screens, windows, and dialogues need to be created and what needs to go into each one. Remember, too, all the time you spend building the wrong prototype is wasted anyway!

References

Constantine, L. (1994) "Modeling Matters" *Software Development, 2,* (2) February. Reprinted in L. Constantine, *The Peopleware Papers* (Prentice Hall, 2001).

Constantine, L. (1995a) "Under Pressure" *Software Development, 3* (10), October. Reprinted in L. Constantine, *The Peopleware Papers* (Prentice Hall, 2001).

Constantine, L. L. (1995b) "Essential Modeling: Use Cases for User Interfaces," *ACM Interactions, 2* (2): 34-46.

Constantine, L. (1996) "Abstract Objects," *Object Magazine, 6,* (10) December. Reprinted in L. Constantine, *The Peopleware Papers* (Prentice Hall, 2001).

Constantine, L. (2000) "Cutting Corners: Shortcuts in Model-Driven Web Design." The Management Forum, *Software Development*, February 2000. Corrected reprint in L. Constantine, ed., Beyond Chaos: The Expert Edge in Managing Software Development (Addison-Wesley, Boston, 2001).

Lauesen, S., and Harning, M. B. (1993) "Dialogue Design Through Modified Dataflow and Data Modeling." In Grechenig, T., & Tscheligi, M. (eds.) *Human-Computer Interaction, VCHCI'93.* Springer-Verlag.

Thomas, D. (1998) "Web-Time Development," *Software Development, 6* (10), October. Reprinted in L. Constantine, ed., *Beyond Chaos: The Expert Edge in Managing Software Development* (Addison-Wesley, Boston, 2001).

Learn more about abstract modeling and usage-centered design at http://www.forUse.com.