SOFTWARE PROTOTYPING

The concept of a prototype is well known from the industrial production world. For example, before an automobile is mass-produced, a prototype model is developed. The prototypers are interested in testing the functionality and performance of the proposed vehicle, including its user interfaces, appearance, handling, and adherence to standards. The company builds the prototype to help ensure a satisfactory product before tooling for mass production.

In software development, the notion of mass production is missing; but the initial development cost of a software product is analogous to the cost of tooling a production line. In the software industry it is also wise to ensure that the product is satisfactory before committing the resources of an expensive development effort. Moreover, managing the development of software products has proven remarkably difficult. The history of large-scale software developments is littered with ex-

amples of large cost overruns, missed deadlines, and project failures.

Software prototyping constitutes one among a number of methods addressing the lack of predictability of large-scale software development. Rapid software prototyping has become the generic term for different techniques involving the production and use of prototypes in the software development process. The term *prototyper* refers to the team of specialists who are charged with constructing a prototype. The term *user* refers to the customer who will eventually use the system being prototyped; the user is the domain expert who is often best qualified to evaluate the prototype.

SUMMARY OF BENEFITS AND RISKS

In the proper circumstances, the use of software prototyping can provide significant benefits to a software development project. A survey of case studies (1) indicates that projects which use evolutionary prototyping can expect reduced development times of up to 70%. To realize the benefits, management must understand the prototyping life cycle and, in particular, be aware of the differences between an evolutionary and an exploratory prototype (see below). The prototyper must understand the application domain and be adept at managing reactions to a working prototype.

- Reduction of development risk
- Reduction of development time
- Improved development process visibility
- Increased customer and user participation

There are also some risks when applying prototyping methods resulting in:

- Unrealistic development schedules and budgets
- Unrealistic user expectations
- Poor design
- Poor long-term maintainability

THE GOALS OF SOFTWARE PROTOTYPING

A prototype is an artifact that is relatively inexpensive to build and enables the evaluation of some hypotheses about a planned software product. In addition, it is desirable that the artifact be executable, meaning that it is possible to subject it to operational scenarios in some automated fashion. See Ref. 2 for a complete discussion of these concepts.

The purpose of prototyping is to limit the risks associated with a development step. A prototype should be developed with a particular hypothesis in mind and should be constructed to check that hypothesis. For instance, the hypothesis may address system functionality (would the user be satisfied with a certain set of functions?), usability issues (which of two user interface designs is better?), algorithmic correctness (will a particular algorithm correctly evaluate the queuing load on a network node?) or resource allocation (will a given distribution of a database handle the expected query load?).

Prototyping is an important tool for risk assessment, whether the software engineering process being used is the traditional waterfall, the spiral model, an object-oriented process, or an evolutionary process. In any case, the application of prototyping, including tools and methods and the prototyping goals, is profoundly affected by the choice of software process model. Although prototyping is useful in all phases of the waterfall and spiral process models, its application is considered particularly appropriate as part of the early requirements specification and system design phases.

A substantial proportion, typically more than 50%, of faults discovered during software development can be traced back to mistakes made in the requirements specification or in the system design. Many of these faults are detected during or even after system test. The later an error is discovered, the more expensive it is to correct. According to (3), leaving a requirements fault undetected until implementation can be 10 times more expensive than correcting it immediately. Leaving it undetected until *deployment* can be 50 to 200 times as costly. Consequently, there is a strong need to improve the quality of the requirements engineering process; this is a major focus of prototyping methods and technology development.

Software development is a fractal process. The notions of requirements, design, implementation, and deployment are replicated at many levels. It is well established that early fault detection is quite cost-effective. Whenever requirements engineering is appropriate (e.g., overall system functionality, data modeling, component design), prototyping is a possible requirements exploration and validation tool. Effective use of prototypes will find faults early with a corresponding decrease in project costs.

THE PROTOTYPING PROCESS

Regardless of where prototyping is applied in the software development process, the prototyping process should follow a process similar to the following:

1. Identify the hypothesis to be validated.
2. Construct the prototype.
3. Construct the operational environment.
4. Construct validation scenarios.
5. Extract information from the scenario enactments.
6. Use this information in the development process.

Whether the development then proceeds by refining the prototype or by constructing a completely new product distinguishes the two broad categories of software prototyping methods: *exploratory prototyping* and *evolutionary prototyping*.

Exploratory Prototyping

An exploratory prototype is created with the explicit understanding that it will be discarded. As Frederick Brooks (4) observed regarding software development, "plan to throw one [software system] away; you will, anyhow."

To create a prototype, a subset of expected system characteristics is identified, implemented relatively quickly, and investigated. Exploratory prototypes are used to investigate a variety of subsets offering different perspectives on the planned system. Some common examples of prototype usage include the following: (1) Users react to proposed user inter-

face designs, (2) system architects create functional prototypes using extant components, and (3) programmers explore algorithm designs using experimental code.

Requirements Feedback. An exploratory prototype is used to increase understanding of a planned system and its operational context. It is a learning tool for the users and the software development team. The result of a successful exploratory prototype is intended to be captured as an improved set of requirements.

Domain Exploration. Exploratory prototyping is particularly useful for a project that lacks extant traditions for domain and systems structures. Used by the planned system users, an exploratory prototype is an excellent tool for eliciting requirements. Many studies indicate that use of prototype results in increased user participation in the requirements definition and consequent improved likelihood of acceptance of the final product. (See Ref. 1 for a summary of many projects studied.) Expert systems and rapid application development (RAD) techniques are commonly used tools for domain exploration; these will be discussed in more depth later.

Exploratory prototyping is also useful in examining the planned product's software architecture and in investigating the choices in resource allocation, overall system structure, and the architectural design for the product's operational characteristics. In these instances, the distinction between the notions of prototyping, design, and modeling can become fuzzy.

Evolutionary Prototyping

In evolutionary prototyping, a prototype evolves gradually until the finished system is deployed. In its ultimate form there is no maintenance phase; the life of a system is considered a series of prototypes until its final decommissioning. This perspective on software development coincides with the observation that the distinction between development and maintenance is rather arbitrary. The distinction between a prototype and the formal product deployment may become a matter of packaging and marketing rather than one of system differentiation.

The shift in emphasis—from discarding a prototype to deploying it—results in substantively different concerns regarding the prototyping process itself. The most profound difference is that exploratory prototyping is one among many tools that can be applied to requirements validation, whereas evolutionary prototyping is a defining part of the development process.

Prototype Requirements

Prototyping goals imply that a prototype must be inexpensive to develop and available early in the development process. Furthermore, it should extract information by executing the validation scenarios, and the information extracted should be readily transformed into feedback for the next development step.

Rapidity

Prototypes are created for learning and experimentation, and they must be simple to construct and easy to modify, even to the point of being able to make modifications interactively while experimenting.

A prototype is a form of abstraction of the system under consideration. This grants the prototyper the luxury of ignoring minor or troublesome features of the system being prototyped. In particular, a prototype often ignores the handling of exceptional circumstances (5).

In the case of an exploratory prototype, the prototyper need not be concerned with documentation, performance, or maintainability. The prototyper should adopt the appropriate set of tools for the hypothesis being tested [e.g., a rapid application environment for functionality and a user interface editor for graphical user interface (GUI) design]. Such choices will usually be at the cost of overall system quality but will not matter since the exploratory prototype will be thrown away.

Information Extraction

The operating environment of a prototype needs to maximize the information extractable from each validation scenario. Moreover, it must be clear from prototype usage what information is significant and what is incidental.

A challenge when designing a prototype is to manage the information flow. On one hand, it is desirable to obtain a large quantity of information for analysis. It is in the nature of experiments, including prototyping, to seek the unexpected. When the unexpected occurs, it is important to be able to analyze the cause and significance of the observed behavior. On the other hand, there is a danger of becoming overwhelmed by the data, thereby missing significant information. The successful prototyping environment provides the development team with tools for information filtering and extraction (6).

RISKS IN PROTOTYPING

Any engineering activity, including prototyping, involves trade-offs and risks. The prototyping risks can be characterized as either *technology-based* or *managerial*.

Technology Risks

Degeneration of System Structure. To begin an evolutionary prototype, one does not need a complete requirements specification nor a complete design. It is normal for requirements to change frequently during the development process and for the prototype to change in response. In the hands of unskilled developers, the prototype's overall structure can quickly degenerate as changes are introduced. Some of the case studies in Ref. 1 report such degeneration.

User-Centric Development. The choice of system parts to prototype is commonly driven by the need for risk assessment. A project builds prototypes when there is a substantive risk of user rejection; GUI design is a good example since it is so visible to the users. Such a user-centric focus may result in a skewed perspective on the importance of certain features. It can result in design decisions being driven by the choice of surface features and the ease whereby features can be added to a prototype.

Management Risks

The use of prototypes can result in unrealistic user or customer expectations. Since a prototype is constructed in a relatively short time, the user may not understand why it is necessary to provide further funding in order to move the system from its prototyping (but seemingly functional) stage to a deliverable state.

Feature Overload. A related issue is the ease whereby users can add features to a prototype. The result can be a system suffering from severe feature overload, because features tried out by distinct user communities are combined in the deployed system.

Quality Cannot Be Retrofitted. An exploratory prototype is created with little concern for long-term sustainability. The result of forcing the further development of such a prototype into a deployed product is likely to be overwhelming maintenance cost, possibly causing an early retirement or full reengineering. A related problem arises because an exploratory prototype is often constructed using special-purpose tools, environments, and languages. If the prototype is transformed into a deployable product, it will be burdened with its development environment to the detriment of execution efficiency, platform independence, and maintainability.

Not Throwing Away a Throwaway. Customers will often be tempted to view the successful exploratory prototype as a satisfactory, deliverable system. This is particularly a danger when exploring user interface issues or when the prototype is created using a rapid development toolset. Such prototypes exhibit major features expected of the product, and the shortcuts and support structures enabling its rapid development may not be apparent to the nontechnical person.

Risk/Reward Evaluation

Overall, prototyping technologies exhibit an excellent risk/reward ratio. The primary risks are managerial rather than technical. When well understood and properly applied, prototyping reduces both schedule and feature risks of system development. See Ref. 5 for a discussion of these trade-offs.

PRESENTATION, FUNCTIONALITY, AND STRUCTURE

It is useful to distinguish three important prototyping domains, each with its own set of traditions, methods, and tool support:

- *Validation of System Presentation.* This is the domain of user interface design. The objective is to evaluate the quality of the user interface with respect to the various user communities that will interact with the system, and to provide early corrective feedback in order to ensure system usability. The validation of system presentation is intimately related to the validation of system functionality.
- *Validation of System Functionality.* This is the domain of functional specifications and addresses the underlying computational functionality that the user interface exercises.
- *Validation of System Structure.* This is the domain of system architectures and addresses the overall internal structures of a system as well as the performance issues related to those structures.

VALIDATION OF SYSTEM PRESENTATION

User interface prototyping focuses on the portion of a software system that interacts with the end user. GUIs are the norm in interactive systems; tools and techniques for user interface prototyping exist primarily for the development of GUIs with little consideration given to textual user interfaces.

The objective of a user interface prototype is to explore the usability and functionality of the system as presented to its potential users. In the narrowest sense, such a prototype focuses on the detailed design of data presentation and user interaction. It concentrates on the control devices (e.g., point-and-click, the locations of input fields, and the GUI button design), the presentation of information (e.g., the use of graphs and color to present information from data repositories), and the user's progression from screen to screen.

As user interfaces have become more sophisticated, the development cost has risen markedly (7). The high cost of interface development makes the use of prototypes increasingly important. Equally important is the need to involve users in the early stages of project development. An interface prototype is a tangible artifact which stimulates the requirements discussion between software developers and users. It is much too expensive to entertain such discussions after the product GUI is implemented.

Prototyped GUIs enable earlier and different feedback from the users than requirements presented in text. Presenting the users with a seemingly working system is more likely to identify missing functionality and requirement misconceptions. In the broad sense, a user interface prototype is employed to understand system requirements. The essential question to be answered is, Will the planned system offer the users satisfactory functionality?

Active user participation in the software development process is called user-centered design (8). With this development style, users and prototypers collaborate on constructing a prototype GUI. When users are satisfied that the interface meets their needs, the prototype is translated into a production GUI that becomes part of the software system. The effort required to translate from prototype to production interface depends on whether a prototyping tool supports exploratory versus evolutionary development, and it will be discussed below.

Participants involved in interface prototyping may include graphic artists, human factors specialists, the software specialists, and the users. Graphics artists are consulted on matters of aesthetics and interface layout. Human factors specialists address issues of interface usability and ergonomics.

TOOLS AND METHODS FOR PROTOTYPING SYSTEM PRESENTATION

Interface Storyboarding

The initial phase of interface prototyping typically presents interface sketches to the users. This phase of prototype development is called *storyboarding.* The term is borrowed from

the film industry, where it describes the process used to sketch the scenes of a motion picture in preparation for actual filming. In the same way, storyboards show users how the GUI will appear in the final product. Initial storyboarding may not involve the use of computer-based tools. Prototypers may draw sketches by hand; after initial discussion, the sketches are then prepared on the computer.

A general-purpose drawing editor is a useful tool for the preparation of computer-based storyboards. A drawing editor is simple to use and flexible in the interface sketches that can be drawn. Another advantage of preparing storyboards with a general-purpose editor is platform independence. Frequently, special-purpose prototyping tools are designed to run on a specific platform that often limits the range of GUI styles that can be storyboarded.

A disadvantage of platform-independent storyboards is the difficulty of evolving to the production interface. Most production GUI libraries do not support general drawing as the basis for interface development. Rather, GUI libraries provide a fixed set of interface components in a limited set of styles. It is likely that storyboarding using a drawing editor, which lacks specific prototyping capabilities, will make it more difficult to evolve the GUI into a production interface.

Tools designed specifically for interface prototyping add some or all of the following features to a drawing editor:

- Presentation management, to support viewing storyboards in useful orders, accompanied with explanatory remarks
- Screen sensitization, whereby specified areas of an interface screen can be defined as "hot spots" that are sensitive to user input
- Scripting, whereby prototypical interface behavior is defined using a programming language
- GUI library access, so that elements of a production GUI can be added to prototype screens to augment hand-drawn elements.

In many prototyping tools, storyboard screens can be organized into navigable collections. Two common organization metaphors are *stacks of cards* or *books of pages*. Navigation commands include moving forward, backward, to the front, to the back, and chronologically through most recently visited screens.

Prototypers can create typical *scenarios* that guide a user through a storyboard collection. Scenarios can be displayed as sequential slide shows, whereby a user presses buttons to move between screens. Advanced scenarios allow user input that simulates interactions that will be carried out on the actual production interface. For example, instead of pressing the forward arrow to move to the next screen, the user could press an actual interface button or type some text into a dialogue box. In response, the scenario will display a screen that represents how the system would respond. An example of the initial screen for a scenario-style prototype is shown in Fig. 1. The complete scenario shows a series of screen storyboards, augmented with remarks about the scenario's progress and

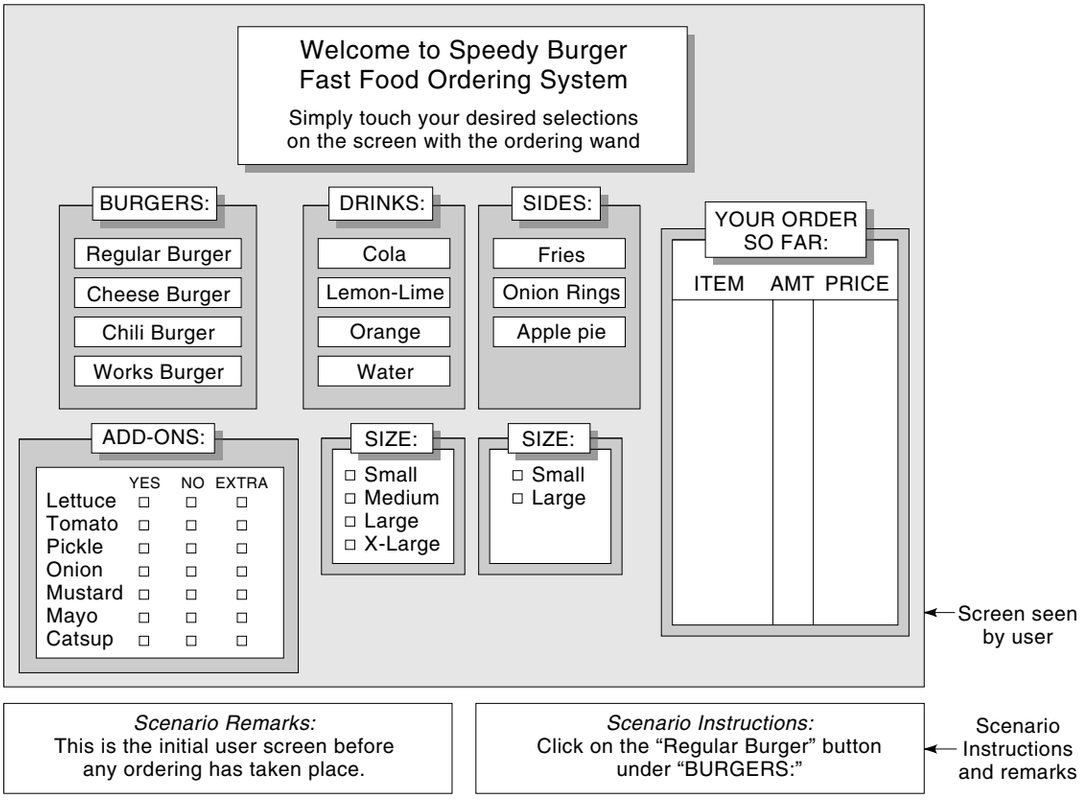

Figure 1. Sample screen of a user interface prototype showing interface contents plus explanatory remarks for the prototype viewer.

instructions showing the user how to interact with the screens.

Prototypes that allow the user to interact directly with screen elements are developed using *screen sensitization.* Such an element is sensitive to user input, such as a mouse click or key press. The elemental form of sensitization links a user action with a predefined screen. For example, in the scenario screen of Fig. 1, the developer has linked a mouse click on the box labeled "Regular Burger" to a screen that shows a regular hamburger added to "Your Order So Far." This example is *single-threaded;* there is only one sequence the user can follow in the prototyped scenario. Single threading forces the user to interact with a single screen object. Multithreaded scenarios are created by allowing the user to interact with multiple objects on a single screen. Multithreaded scenarios are inherently more difficult to manage and may result in user confusion if not carefully designed.

Storyboarded scenarios are fundamentally no more than "storefronts" for a software product. To use another analogy from film production, a storyboarded scenario is like the Western town constructed with storefronts that appear authentic but are actually propped up with supports. These props appear real in a movie but do not function as real buildings. In a similar way, a storefront GUI does not function as a real program; no actual computation is performed in response to user interactions. Rather, the entire set of screen outcomes is predefined; the order in which the screens are presented can vary but there is no functionality. To provide real functionality, the prototyper must use *interface scripting.*

Interface Scripting

In Fig. 1, the response to a user click on the "RegularBurger" button is a transition from the first storyboarded screen to the following screen. In an unscripted prototype such as this, the only means to depict interface behavior is to transition from one predefined screen to another. When scripting is employed, the prototyper may add specialized functionality that is invoked in response to user-initiated events. This functionality can perform computation that approaches that of a production system. For example, the following script could be attached to the "Regular Burger" button in Fig. 1:

```
on MouseUpEvent
  put ''1 Burger $1.89'' + newline
  after the last line of screen field OrderSoFar
  highlight the last line of screen field
    OrderSoFar
end MouseUpEvent
```

This script is written in an English-like programming language similar to the language of a typical prototyping tool. The result of the script is to append a string into the "Order So Far" area of the interface display.

In general, prototyping scripting languages have the functionality of conventional programming languages, plus specialized features to deal conveniently with objects displayed in a prototype interface. For example, the above script uses the following scripting language features:

- Connecting to a specific user-initiated event with the "on MouseUpEvent" construct
- Accessing an object in the prototype screen with the "last line of screenField OrderSoFar" construct
- Performing an update to the interface screen with the "highlight last line" construct

When a script is added to a prototype, a fundamental change results in the nature of the prototype. In the unscripted version, there is a single predefined screen that displays the response to the "RegularBurger" button. The behavior of pressing this button is shown in the transition from the screen with an empty "Order So Far" to the screen with exactly one regular burger added to the "Order So Far." In the scripted prototype, only one copy of the interface screen is needed to show the effects of pressing "Regular Burger." The prototypical behavior of the button press is shown by changing the contents of the "Order So Far" field on the screen. Furthermore, each button press results in an updated display; another regular burger is added to the order. In this way, the scripted prototype depicts functionality in a more general way than the unscripted version.

This script illustrates how a prototype can provide reduced functionality when compared to the production system. In the production version of the Speedy Burger ordering system, the "Order So Far" display would be updated—not by appending a new line for every burger ordered, but by incrementing a regular burger order counter. To do this, the script for the "Regular Burger" button would be enhanced to perform some arithmetic to increment the burger count. In addition, the script must employ a conditional computation to respond correctly on the first button press as compared to subsequent presses. There is a obvious trade-off between prototype complexity versus prototype development time. The level of prototype scripting must be determined by the members of the development team based on their needs. In general, when the prototype conveys system functionality to the end user in a clear manner, the prototype is complete. At that point, the process of evolving the interface prototype to a production GUI can begin.

GUI Development

The transition from a scripted prototype to a production GUI can be seamless if the prototyper uses a tool that supports evolutionary development. Such a tool must provide two capabilities to the prototyper:

- The ability to place operational GUI elements onto a prototype screen
- A general scripting language that supports production software development

From a technical standpoint, a fundamental distinction can be made between the behavior of a scripted storyboard versus a production GUI. Specifically, a production GUI is more than a picture with selected areas sensitized. A true GUI is a collection of interactive elements (called *widgets*) that have their own well-defined, intricate behaviors. Examples of common GUI widgets include push buttons, pull-down menus, and scrollable text boxes. In Fig. 1, the "RegularBurger" button is drawn as a shaded rectangular graphic, not as push-button widget. The sensitization of the rectangle is performed after the fact, as a convenient way to support user interaction. In a

production GUI, a push-button widget replaces the sensitized rectangle. Among the distinctions between a sensitized graphic and a GUI widget are the following:

- A widget is designed to connect to a compiled function that is part of a production software system, whereas a sensitized graphic connects with an interpreted script.
- A widget has built-in behaviors that are more sophisticated than a sensitized graphic; for example, a push-button widget responds to a mouse-down event by highlighting itself, and it responds to a mouse-up event by unhighlighting.

In most evolutionary prototyping tools, there is no automatic way to convert from a sensitized graphic to a corresponding widget. Instead, the initial prototype storyboards are not drawn with graphical shapes, but with actual widgets that can be graphically manipulated on a prototype screen. Tools that support such a development style are called *interface builders*.

Many interface builders are less flexible than full-feature prototyping tools. For example, most interface builders do not include general-purpose drawing functions and supply only a fixed set of predefined widgets. If a desired GUI element is not available, the behavior may be difficult to define, potentially requiring substantial program development. Such work defeats the spirit of rapid interface prototyping.

The combination of the rapid interface prototyping plus support for evolutionary development is emerging in the development environments for some programming languages, such as Visual Basic and Java. Such tools have the potential to provide a full set of predefined widgets in addition to the features of prototyping tools that make interface development rapid and convenient.

VALIDATION OF SYSTEM FUNCTIONALITY

By broadening the scope of experiments from user interfaces, prototypes can be constructed that include some functionality and support the prototyped GUIs with prototype implementations. The exploratory programming techniques developed by the artificial intelligence community have a long history of addressing this prototyping domain.

The object-oriented community's emphasis on reuse through the use of rapid application development (RAD) environments (focusing on programs), component reuse (focusing on experimental architectures), and so-called "middleware" technologies (which are likely to introduce permanent system architecture features) has also worked well as a basis for software prototyping.

The developers may ignore a number of long-term issues when focusing an exploratory prototype on a requirements set. There is an expectation that system construction will be easier and swifter when ignoring efficiency or other secondary concerns. Expert systems and formal specification methods represent the highest abstraction in this regard. Common to both is the capture of system requirements as sets of rules or logical assertions, which become subject to analysis or execution.

Expert Systems

Expert systems development processes evolved to create software from inadequately articulated knowledge by transforming such knowledge into enactable code. In the domain of artificial intelligence software, expert system methods are often employed to create fully functional products. Expert system methods can also be used for prototyping, even if the ultimate system implementation is based on explicit coding. The most important attributes of an expert system relevant to prototyping are the rapidity with which a system can be developed and the rule-based representation that permits a prototype to be defined in higher-level terms than with a conventional procedural programming language.

LISP Systems

Another set of prototyping tools from the artificial intelligence community is that of functional programming, specifically the LISP-based systems. LISP has proven useful for the creation of exploratory prototypes. The interpretative nature of LISP, together with its weak typing, makes it particularly useful for prototyping. Being interpreted, the code–compile–run cycle becomes irrelevant because code can be modified incrementally. Being weakly typed, the system becomes quite forgiving of coding changes since no time will be spent hunting for type mismatches. The former is less of a concern today; the latter may be argued as being a detriment to prototyping since weak typing may preclude the early detection of serious type problems. Polymorphic and subtyping constructs offer most of the convenience of weak typing without sacrificing the safety of strong typing.

Rapid Application Development Environments

Another reason for the continuing popularity of LISP-based systems is the availability of excellent development tools and a rich base of extant system components. This is also the strength of object-oriented RAD systems. RAD systems provide developers with a large set of definitions and components, a set of reuse mechanisms (e.g., subtyping, inheritance, and polymorphism), and support for a rapid code–compile–test cycle. RAD tools transcend single-file compilation to support the notions of software projects, bringing together within a single framework the code, the user interface, and the supporting libraries of predefined modules. They often supply the prototyper with graphics-based tools for interface design. RAD tools also offer module interconnection formalisms, which enable rapid software construction from reusable components.

Compound Reuse

The prototyping techniques described thus far generally focus on defining specific details of prototype behavior. Such techniques can be characterized as *small-grain* prototyping. An alternative *large-grain* prototyping technique is based on the reuse of existing software components with predefined behavior. In this approach, a prototype is assembled from a collection of existing components retrieved from a well-organized component library. Until recently, the software industry has employed a number of proprietary technologies for compound reuse based on specific, proprietary interface development environments (IDEs). The software industry has adopted stan-

dards on component definition for reusability that show much promise for the development community. Examples of such standards are CORBA and Sun Microsystem's JavaBeans.

Formal Methods

Formal methods are a set of development technologies with certain similarities to expert systems. As with expert systems, formal methods can be used for full-scale system development, as well as for prototyping. Formal methods and expert systems share the attribute of high-level software representation that makes both methods suitable for prototype development. Also, some formal methods use rule-based techniques similar to the techniques used in expert systems. The result of using a formal method is a set of formal descriptions of the planned system, expressed in a formal logic, such as of universal algebra (9), first-order logic (10), or modal logic (11). Unlike the other methods, formal specifications are often not intended to be executable or to synthesize executable code. Instead, the focus is on analyzing the specification using verifiers or consistency checkers, checking whether the system as described by the specification satisfies the requirements as perceived by the system users. See FORMAL SPECIFICATION OF SOFTWARE for further information.

RISKS IN USING PROTOTYPING IN VALIDATING SYSTEM FUNCTIONALITY

A functioning system can be quite seductive. If a prototype is the primary repository for capturing the functionality of a proposed system, it is likely there will be functional deficiencies. An experiential approach to system design is prone to get caught up by what *is* rather than identifying what *should be.* The use of prototyping must be accompanied by a separate means for capturing the results of the prototyping experience—either as systems documentation (evolutionary prototyping) or as an improved requirements document (exploratory prototyping).

Since a prototype represents an abstraction, the developer is obliged to identify explicitly what carries over when transforming the prototype into a production system. One of the potential risks of prototyping is that the rapid creation of the abstract prototype makes the transition of functionality difficult. For instance, the use of weak typing in a prototype may make the transformation to production code, which may require strong typing, difficult. As has been pointed out earlier, quality cannot be retrofitted. If quality is one of the aspects of system construction that has been abstracted away in a prototype, then the prototype should be of the exploratory variety.

Similarly, it can be difficult to transition to production code a prototype that has been developed using a RAD tool. The efficiency of the tool is often a result of its intimate interaction with the system being created, with subsequent problems when the system is to be extricated from the development environment. The obvious example is when the prototype is interpreted; the interpreter is properly regarded as part of the system with the implications on execution efficiency and platform independence. More insidious may be the RAD tool that generates the code of the prototype. Though nominally independent of the development environment, the code generated by the tool may itself be unmaintainable in the absence of the development environment, thereby forcing future developers to use the same tool set.

Common to all these risks is the seduction of the developer into making more of a prototype than initially warranted and expected.

Though these risks are significant, they should be considered management risks rather than problems intrinsic to prototyping. When properly used as a device for systems experimentation and risk reduction, prototyping is a valuable tool for exploring system functionality.

VALIDATION OF SYSTEM STRUCTURE

Most development organizations clearly understand the reasons for prototyping the graphical user interface of a system under development. It is far less common to consider prototyping the architectural structures of a new system; in fact, many organizations are still struggling with their processes for building the actual system structures and have never considered prototyping.

There are several good reasons for building prototypes to study system structures:

- *To Understand Performance Issues.* The performance of a software system is often directly related to the system structure design choices. Prototyping can assist developers in understanding these performance relationships.
- *To Consider the Trade-Offs Among Alternative Designs.* Developers often make significant design decisions in an ad hoc manner. They may have become aware of a particular system structure in a course or on a previous project. Perhaps they recently read a journal article. Prototyping can provide an inexpensive tool for studying alternatives.
- *To Study the Feasibility of a Particular Design.* A design mistake discovered later in the development cycle can be very expensive. Prototyping can help prevent such mistakes.

It is useful to consider system structures from two points of view. *Static structures* are embodied physically in the software; they are abstractions created by the developer to understand and simplify the software structure. *Resource considerations* are issues, such as memory size or performance, that arise when the program is executing. System structure decisions often merit consideration for prototyping.

Static Structures

There are good reasons for designing software structures at different abstraction levels. Each level serves a purpose for different phases of the life cycle. Design hypotheses, suitable for prototyping, are made at each level. For example, the decision to use a client–server software architecture might be made early in a product life cycle by the development manager. The decision to design a particular user interface module might be taken by a senior designer in the design phase. A programmer might decide, late in the project's cycle, to use a linked-list data structure. A common static structure taxonomy, from the lowest to the highest abstraction level, is discussed in the following sections.

Code and Data Structures. These are the structures created in some programming language. A typical decision, taken by a programmer, would be the choice of a particular data structure (e.g., array) to contain the data for an abstraction (e.g., set).

Module Design. A module is a separately compilable unit that contains a set of related functions or classes. A typical software design comprises a set of modules combined to form the high-level design of a software system. The module interface forms the abstraction that defines how the module is viewed by a software developer. A typical decision might be to create a particular class (e.g., set) to represent a needed abstraction.

Software Architecture. A software architecture describes the decisions taken concerning the interrelationship between modules. Typically, there is a pattern in such relationships; some common patterns are client-server, object-oriented, and process control. The software developer, early in the life cycle, would take the decision to use a particular architecture. See (12) for a description of software architecture concepts.

Run-Time Considerations

There are a number of run-time considerations suitable for prototyping by system developers. Paramount among these is performance; failure to meet performance goals is a common problem. And it is quite difficult to improve performance after a system is implemented. As performance is quite visible to the users, it is an excellent candidate for prototyping to minimize risk.

Projects using the evolutionary prototyping process are especially vulnerable to performance risks because a working prototyping is a seductive artifact. Users observe a working artifact that has been constructed quickly at minimal cost; it is easy to overlook that other users will be expecting a specified performance level.

Another run-time consideration is resource utilization. Some examples are as follows:

- How much memory or disk space will the system require?
- Is the bandwidth of channel *X* sufficient?
- What is the expected response time for operation *Y*?

Prototypers check resource consumption for the same reasons they check performance; they want to minimize risks when the system is deployed. The risks can be significant, especially in embedded systems. If, for example, the software for a pacemaker is constrained to a specified memory size, it will likely be impossible to add more memory. In such a scenario, the effect could be product failure.

TOOLS AND METHODS FOR PROTOTYPING SYSTEM STRUCTURES

There exists a variety of techniques for checking hypotheses about system structures. In some cases, these techniques may not be commonly considered as prototyping.

Architectural Languages

Architectural languages exist to model structural designs. Such languages may be textual or graphical. Some languages have a rich semantic content and allow the prototyper to generate code or execute simulators. There are also architectural languages that can be interpreted symbolically.

Some architectural languages have little or no semantic content and serve only as a notation for the system architecture. Use of the latter type should be thought of as design work rather than prototyping.

Code Reuse. Structural hypotheses can be tested by constructing subsystems with extant components such as class libraries, general-purpose functions, or GUI modules. Code reuse is a useful technique for both throwaway and evolutionary prototyping.

Reusable components are widely available as commercial products; many vendors allow short trials of a product for throwaway prototyping purposes. There are thousands of components available on the Internet at no charge or for a nominal shareware charge. See SOFTWARE REUSABILITY.

Data Modeling

A data model is a logical representation of the data to be used in the application. The term *data modeling* is usually associated with applications that access their data using a database management system (DBMS). It is quite common to build a prototype data model during the early stages of the product life cycle for several reasons:

- A prototype data model is an excellent medium for providing feedback to users; building a data model is an obvious task during the analysis phase.
- Data modeling decisions are significant. The data model contains all the data fields that will be provided to the user. The model can drastically affect update and query performance.
- It is easy to alter the data model early in the life cycle; it can be quite difficult once a database is populated.

There are a large number of methods and tools available for data modeling. The formal foundations are well established and widely taught. Every DBMS vendor provides tools for building a prototype database; for large projects, the vendors will sometimes contribute analysts to assist the data modeling effort. Such assistance is part of their marketing effort.

Simulation

A software simulation is created to mimic one or more product behaviors. Behaviors such as user response time, transaction rates, and disk usage are commonly simulated. For example, the developer may wish to determine the CPU power needed to reach a specified transaction rate. A simulation can answer such a question.

An exploratory prototype can also be thought of as a simulation of the product's functional behavior; users and developers evaluate that behavior when using the prototype. In contrast, an evolutionary prototype is not a simulation because it is, in fact, the actual product.

To be useful, a simulation must be validated. Some common methods to validate simulations are:

- *Analytical Model.* For some types of systems, it is possible to construct an analytical model. For example, the throughput of a transaction processing system can be modeled using queuing theory. The analytical model provides an alternative method of computing throughput and can validate the prototype simulator.
- *Domain Experience.* If the simulator has been used and validated for similar problems in the same application domain, it is reasonable to assume that the simulation is valid. For example, the companies that simulate mainframe configurations have used their tools numerous times; the simulations have been validated through successful usage.
- *Use.* An exploratory prototype is validated through use; users and developers commit resources to the validation process.

  Run-time simulations can sometimes be validated by using the simulation as part of a larger system. For example, suppose a simulation is created to model telephone traffic for a large city. It may be feasible to connect the simulation to a working telephone switch and use the telephone switch to validate the simulation.

BIBLIOGRAPHY

1. V. S. Gordon and J. M. Bieman, Rapid prototyping: lessons learned, *IEEE Softw.,* **12** (1): 85–95, 1995.
2. D. P. Wood and K. C. Kang, *A classification and bibliography of software prototyping,* CMU/SEI-92-TR-13, Pittsburgh, PA: Software Engineering Institute, Carnegie-Mellon University, 1992.
3. A. M. Davis and M. D. Weidner, *Software Requirements: Objects, Functions, and States,* Englewood Cliffs, NJ: Prentice-Hall, 1993.
4. F. P. Brooks, *The Mythical Man-Month,* 2nd ed., Reading, MA; Addison-Wesley, 1995.
5. S. McConell, *Rapid Development,* Redmond, WA: Microsoft Press, 1996, pp. 433–444, 569–574.
6. Luqi and R. Yeh, Rapid prototyping in software development, in J. J. Marciniak (ed.), *Encyclopedia of Software Engineering,* New York: Wiley, 1994, pp. 978–984.
7. B. A. Meyers and M. B. Rosson, Survey on user interface programming, *Human Factors in Computing Systems, SIGCHI Proc.,* 1992, pp. 195–202.
8. D. A. Norman and F. W. Draper, *User-Centered Systems Design,* Hillsdale, NJ: Erlbaum, 1986.
9. M. Wirsing, Algebraic specification, in J. van Leuwen (ed.), *Handbook of Theoretical Computer Science,* Cambridge, MA: MIT Press, 1990.
10. B. Potter, J. Sinclair, and D. Till, *Introduction to Formal Specification and Z,* Upper Saddle River, NJ: Prentice-Hall, 1996.
11. T. Bolognesi, J. van de Lagemaat, and C. Vissers, *Lotosphere: Software Development with Lotos,* Norwell, MA: Kluwer, 1995.
12. M. Shaw and D. Garlan, *Software Architectures,* Upper Saddle River, NJ: Prentice-Hall, 1996.

SIGURD MELDAL
GENE L. FISHER
DANIEL J. STEARNS
California Polytechnic State University
PETER C. ÖLVECZKY
University of Bergen