The Role of Functional Prototyping in the Development Process

by Jim Conallen

Editor's Note: Each month, we will feature one or two articles from the Rational Developer Network, just to give you a sense of the content you can find there. If you have a current Rational Support contract, you should join the Rational Developer Network now!

Abstract

This paper describes the benefits of high fidelity prototyping to ensure success of UI intensive, data oriented applications, and introduces a new generation of visual, declarative prototyping technologies for visualization and simulation of application requirements.

Addressed primarily to system analysts, these tools are designed to enhance the process of elicitation, validation, documentation, and formalization of system requirements, and to enable more successful communication with stakeholders. These tools offer high functionality without programming, using declarative, logical implementation to simulate application interface and behavior.

The paper shows how simulation modeling with a functional prototype can be integrated with formal modeling techniques: application entities, navigational flows, user interactions, and views, as implemented in the prototype, can be transferred to UML models and used to jumpstart application user interface, architecture and database design.

The Importance of Prototyping to System Success

Well-understood systems are all alike in their explicit knowledge of system requirements; every implicit requirement must be uniquely discovered.
Vendors, trade literature, and university course material today usually select examples where the domain knowledge is well known. This often gives us a false sense of what to expect when developing systems. When engineering students are taught rudimentary system analysis using the classic order entry or bookstore example, which are understood almost intuitively, it leads students to think that the analysis of a system is extraneous and that the "real work" starts with design.

However, simple examples are used as teaching tools, to enable understanding of the analysis process. In real projects, there will always be percentage of requirements that need to be discovered, understood, communicated, documented, and validated. Moreover, even when a requirement is implicit or obvious, prototyping can be helpful in defining the user interface and making decisions regarding the application architecture.

**Prototyping as a Key For Successful System Analysis**

The success or failure of a project lies not with the elegance of its code but in its analysis of the system and end user requirements. System analysts have the difficult job of eliciting knowledge from stakeholders and domain experts, and communicating this knowledge to many different parts of the application development team. Therefore, what system analysts need is a way to communicate requirements. The most appropriate method for communicating requirements with stakeholders and domain experts is through prototyping.

Prototyping is a means of communication. More importantly, it is an additional means of communication, designed to provide an additional dimension to application analysis and design: visualization and simulation of use cases and requirements documentation. Prototyping does not replace use cases, requirements documentation and the like; instead, prototyping is meant to communicate them more efficiently.

Static, throwaway prototypes are beyond the scope of this article, as we focus on high fidelity, functional prototypes that simulate both user interface and application behavior.

Visualization of application features and requirements and simulation of application behavior enables both buy-in and feedback. The result is well-grounded validation, more complete grasp of project complexity and scope, and insurance that the application is one that will successfully answer user needs. High fidelity, functional prototyping is the means to that end.

If prototyping is generally advantageous to successful system design, it is essential in the analysis and design of Web-based, data oriented applications, where standards are still in infancy. A Web application cannot be designed using only high-level, abstract definitions; its user experience cannot be an afterthought. Elicitation, visualization, simulation, and validation of Web application requirements beg for the use of high fidelity, functional prototyping.

**The Problems with Current Prototyping Methods**

According to RUP definitions, there are many legitimate forms of prototyping and the word "prototype" is associated with a wide range of tools: PowerPoint,
Word, Visio, and Visual Basic are all accepted prototyping tools.

The reason for such a variety of prototypes is that none of the available tools are dedicated prototyping tools: they are not intended for system analysts and they are not designed for eliciting and visualizing system requirements. Using existing tools to prototype an application rivals application development instead of accelerating the process.

A functional prototype can effectively be used for elicitation, understanding, communication, documentation, validation, and formalization of system requirements. To be effective as a mainstream application analysis and design tool, the emphasis must remain on using the prototype to further application analysis and design, and not prototype development.

**More Effective Prototyping**

To be useful, application prototypes must be both highly functional and easily implemented by analysts and domain experts who need to express use cases and requirements in a language that can be understood down the line.

**Introducing a Dedicated Prototyping Tool**

This in mind, we can envision a dedicated prototyping tool, designed specifically for application analysis and design:

- Designed for system analysts, without demanding programming skills
- Capable of simulating both user interface and application behavior
- Flexible enough to work top down or bottom up and to accompany the iterative elicitation and design processes
- Sharable with cross-platform viewing capabilities

Breakthrough technologies in the field have made live, fully functional prototypes a reality. New visual prototyping development technologies enable "instant prototyping", allowing system analysts to model system user interface and behavior, without writing code.

The solution lies in visual, high-level, declarative based definitions of the prototype interface and behavior.

**Functional Prototype vs. Target Application**

At first it would seem that such an ideal prototype would blur the lines between prototype and application. After all, if a prototype can simulate the application, why shouldn't it go the next step and "grow up" to be the application? And how can we draw the line between the capabilities of functional prototype and its target application?

To simulate the user experience, the prototype should be able to model the application screens and behavior using live or trial data, so it must share a common façade with the target application. However, prototype
Implementation is completely independent from application implementation. The usefulness of a prototype is in its ability to communicate application requirements, not in its ability to generate code directly. Ideally, a functional prototype would use some sort of effortless implementation mechanism that is not resource intensive, that doesn't require coding and that doesn't necessitate writing the application twice.

The solution lies in a universal, definitions driven engine, able to simulate both interface and behavior of database oriented, user interface intensive applications.

**The Scope of Prototyping**

In terms of output, we want a functional prototype that mirrors the application look and feel and simulates application behavior. In terms of implementation, we want a prototype development environment that is effortless-high level, flexible, able to utilize live data and existing services, easily shared-all without programming.

The functional prototype provides both visualization of application entities-business objects and their attributes, and simulation of their behavior via connection to appropriate UI entities. Thus, the prototype helps the system analyst to define all essential application entities and components - use cases, business objects, screens, interactions, and flows.

**Integration of Live Data and Services**

Ideally the prototype should be able to model user interface, system entities, attributes, and behavior on live data. However, since real data is not normally available in the requirements elicitation stage, live data must be provided by built-in trial database sources. As application resources - data and services - become available, the prototype must be able to integrate them. Thus the prototype becomes increasingly representative and understandable by stakeholders.

Integration of live data and services provides an additional dimension of high fidelity: users are able to experience more exact application behavior aspects based on real data. While this is important in all the stages of application analysis, in our opinion, the importance of working with live data is maximized in the realization of test cases.

**Sharing Capabilities**

The prototype must include a mechanism for sharing and eliciting feedback from remote participants, and should include a publishing feature that allows easy access to the prototype over the Web via standard browsing tools. Remote participants must be able to experience the prototype as it was designed, and to provide online feedback.

**Automatic Documentation**

In addition, the prototype should be able to automatically generate documentation of requirements as implemented in the prototype, and should
include a mechanism to ensure updating of requirements documentation.

**Functional Prototyping in Requirements Elicitation Process**

Requirement elicitation is an iterative process that involves a number of players, each responsible for well-defined activities and deliveries. Adding a functional prototyping to the requirements elicitation process neither reduces the number of players involved, nor rearranges their responsibilities. Instead, the role of the functional prototype is to help the players ensure that more of the system is understood at each step of the process.

Using a visual prototyping development helps the system analyst communicate more freely with system stakeholders. In addition to documents and use cases, the analyst or requirements specifier can express requirements visually to the user experience designer. The user experience model can be generated directly from the functional prototype, and the prototype can be changed based on input from the user experience designer.

Functional prototypes also allow domain experts WYSIWYG, real time access to published prototypes, ensuring that the system analyst or requirements specifier can integrate feedback from remote participants as well.

![Figure 1: Prototype-centric View of the Requirements Elicitation Process](image)

**Hands-on Prototyping**

To proof the concept discussed above, we'll use easyPilot, a package of analysis and design tools for rapid modeling of enterprise application requirements. These tools model application requirements and specifications as functional prototypes that enable early testing and validation of application usability, functionality, and flow (for more information see [http://www.easybase.com](http://www.easybase.com)).

We'll use easyPilot to demonstrate the advantages of a dedicated prototyping tool with loads of built-in functionality and almost transparent implementation.

For our purposes, the prime advantage of easyPilot is its declarative design
Initial Elicitation

The following example demonstrates a fictitious requirements elicitation cycle, with the added advantage of a dedicated prototyping tool. For simplicity's sake, let's go thru the motions of eliciting requirements for an order-entry system. An order-entry is a good example of an intuitive system with many implied requirements.

As always, understanding is the first step. We need to determine what are the main business entities, their attributes, and the logical relationships between them. In a dedicated prototyping environment, these use cases and flows are expressed as persistent entities, screens, interactions and navigation.

From the RUP perspective, initial elicitation belongs to the Inception phase, which deals with establishing project scope, estimating the overall cost, schedule, and potential risks, as well as determining the critical uses cases and scenarios.

Almost immediately, we understand that there are at least two main business entities: customer and order. We need a way to organize the customers, a way to represent the orders for particular customer, and to define a flow allowing customers to build the order.

This in hand, we have enough information to begin building the functional prototype.

- The main business entities and their relationships are defined in the prototype as logical tables and associations, using high-level, logical definitions only
- Screens are defined visually, via drag and drop and predefined form templates
- Interactions and navigation are defined by applying customizable, built-in functionality to action buttons

In each iteration, we can add new artifacts and add improve definitions of existing elements.

Since the functional prototype simulates application behavior and not just its user interface, system analysts, stakeholders and domain experts are able to test application functionality at each iteration. Built-in functionality allows users to perform a variety of data manipulations, for instance, to add, delete, browse, query, and sort data.

Here is the output of two separate iterations in the requirements elicitation process. They show the same screen at different stages in the cycle.
Prototyping Step-by-step

Prototyping Use Cases

The purpose of the prototype is to visualize use case flows of events. The use case description can be used as a script for the system analyst to follow when walking through the prototype with stakeholders, confirming that the project team's understanding of the application behavior is correct.

Since the prototype is used to illustrate and enact application use cases, and
not to replace them, the prototype must be able to integrate with existing CASE tools. Optimally, formal modeling techniques should be able to make use of the prototype implementation, generating artifacts from the prototype and using the prototype to update the use cases automatically.

**Prototyping Objects**

Because identification of the main business entities and relationships is an iterative process, the prototype must allow changes to the definitions as the system grows and as requirements evolve. The system analyst should be able to focus on informational attributes, without having to define indexes, keys, or deal with database normalization.

The prototype must be able to use trial data, and live data when it becomes available. This can be accomplished on a logical level using built-in database source and related database functionality, allowing the system analyst to make high-level definitions about entities and their relationships.

**Prototyping Views**

When prototyping application requirements, the emphasis must always be on the results of the prototype output and not on its development. The prototype must allow the system analyst to concentrate on application logic, functionality and flow, without having to worry about data schema or binding user interface elements with data entities.

Ideally, a dedicated prototyping tool would allow visual, drag & drop implementation of user interface and application forms, top down or bottom up design, and infinite changes to the prototype as application requirements evolve.

![Figure 3: Prototyping, Advanced Design](image-url)
Prototyping Interactions and Flows

To prototype navigation paths and complete the logic of the application under analysis, prototype forms must use a linking mechanism that allows realization of use cases and application flows. Low-level functionality behind each script can be built-in and managed automatically.

Publishing

The prototype should include a publishing mechanism that allows remote participants to experience the prototype functionality and to provide online feedback. Ideally, a published prototype would retain its functionality and viewable using standard Web browsers.

![Figure 4: Web Publishing](image)

Documenting

Since the prototype represents the most up-to-date vision of the application under analysis, there should be some method of integrating requirements, as implemented in the prototype, with existing requirements documentation.
Integration with Formal Modeling

Top down, abstract, structural modeling is one way to describe an application, while the prototype implements simulation modeling, describing the same application visually. The two methods compliment each other. Thus, the most effective way to model application requirements makes use of both kinds of modeling.

At any time during the application development process, the functional prototype represents a snapshot of requirements that can be converted into UML model. easyPilot fully supports such a conversion: intelligent forward and reverse engineering options enable exchange between prototype and UML model, including a mechanism for synchronization of changes made in both
High-level analysis models describe the application under construction using predefined stereotypes: boundaries, controls and entities, while design models use a set of stereotypes and tagged values designed to enable formal representation of functional UI entities.

**How to use Functional Prototypes as Design and Architectural Requirements**

When you first look at, and interact with the functional prototype there is the danger that the stakeholders will think that all the work is done and that the application is ready for use. In fact if the system analysts did their job well enough it would be very difficult to distinguish the prototype from a real implementation of the target system. What is not apparent is that the underlying architecture supporting the prototype may not meet the needs of rest of the requirements of the system.

The functional prototype addresses the functionality of the target application - not the entirety of the system. In addition to the functional requirements there are several categories of non-functional requirements; performance, scalability, deployment, security, interoperability, etc. that needs to be addressed in order for the system to be considered a success. It is not the job of the analyst developing the use cases or functional prototypes to consider these other non-functional requirements at any significant level of detail, their job is to ensure that the functional requirements are captured.

In the remainder of this article we show how the UML model, as produced by the prototype, itself a model artifact, can be used to generate additional UML artifacts to drive user experience development, system architecture, and database architecture.
User Experience Development

The user experience team is responsible for producing the actual user interface on top of the defined architecture for the application. This means that the UX team has to make the system look like the prototype but run on top of the developing architecture. It is also likely that some members of this team worked to develop the prototype, since the prototyping process often requires some user interface design skill.

A new type of UML model (insert reference to UX RUP Plugin) has recently been introduced to help bridge the gap between the development of the user experience and the engineering aspects of a Web application. The UX model is a UML model that defines a view of a system in terms of screens and navigational flow. It does not attempt to model any of the underlying business or data tiers found in most Web applications. However, it does capture and define the screens the user sees. Their content and the expected pathways that the user can use to navigate through them.

The UX model is can be derived very easily from the prototype model. In the prototype "UIForm" elements clearly map to UX model "screen" elements. The display fields defined in the "UIForm" classes map to the attributes in the UX model "screen" class. The input fields defined in the "UIForm" class map to attributes in the "input form" classes of the UX model. In general there is a high degree of correspondence between the prototype and the UX model. However the two are not equivalent.

![Figure 7: User Experience Model Traces to the Prototype Model.](image)

The prototype models functional requirements, with details regarding the underlying data structure and system controls. The UX model models the same
requirements from the user interface point of view, with a focus on system architecture.

Before implementing the actual interface (i.e. authoring the HTML and producing the artwork) certain architectural decisions need to be made. The most critical being how the non-standard user interface elements are to be implemented. Most Web applications these days make use of user interface elements such as tree controls or data grids, however there are many different ways to implement these. The architecture can specify applets, embedded controls, JavaScript, or even mimic the controls with a series of Web pages and artwork. The UX team must work within the architectural framework that was defined for the application.

**Defining the Architecture**

The responsibility of ensuring that the non-functional requirements are met in the system is given to the architect and design teams. In a Web centric architecture some of the architecture is already defined. The fact that the architecture must be Web centric means that there are a number of screens to be defined each with a certain amount of dynamic content (i.e. data from a database) and user interaction (browsing and selecting data). In Web architectures the connection between these screens is of importance as it represents the navigational paths through the system that accomplish actual business objectives.

The architecture also defines how screens are produced (ASPX, JSP, etc.) and how the elements of the business logic of the system are assembled and invoked. In short the architecture in a Web application defines the application in many critical ways. This is, however, not an article on architectures in general but rather how one can use the byproducts of a functional prototype to help jumpstart and drive the architecture.

When we examine what the functional prototype has to offer us from an architect's view we can see that it provides:

- A list of candidate screens and their navigational paths
- The user interface elements in each screen
- Actions that a user can invoke
- A set of entities in database tables
- A set of views of the database for particular screens

All of these are significant to the architect and design team, however the user interface elements are of particular interest to the UX team while the data schema elements are of particular interest to the database team.

The design and architecture team however are less interested in the details of the user interface elements and database column than the screens and navigation routes themselves. The architect is responsible for determining exactly how the screens are to be built and how they are to connect to the middle and/or data tiers of the application.
The architect begins with the UML model produced by the prototyping tool (Figure 8). In this diagram the "UIForm" Product Lookup is modeled as a class that contains a number of user interface elements; one "UIGrid" stereotyped class and seven "UI Button" stereotyped classes. The grid element specifies four attributes, each of which represents a named piece of dynamic content. Attributes in a grid control represent columns of data that either come directly from a database table or are produced by some middle tier logic. The form in this diagram does not define any "UIField" elements outside of the grid, but it could have.

![Figure 8: Class Diagram from Prototype Model](image)

Figure 9 we see how a screen in the UX model is mapped to the form defined in the prototype model. In this diagram the "screen" Product Lookup has a "trace" stereotyped dependency on the prototype model's Product Lookup form. The screen also contains the class Product. The class Product represents the information that is found in one line of the data grid.

![Figure 9: UX Model Maps to the Prototype Model](image)

In addition to the fields containing the application's dynamic content the prototype also indicates other elements like images and field labels. These
elements are not architecturally significant to the design team, but they are to the UX team.

The prototype model also defines buttons. These buttons represent both user interface elements and actions that the user can invoke in this screen. The rules of the architecture determine if these will be implemented with simple hyperlinks, HTML button elements or some other mechanism. In many cases these buttons will result in a page change (new screen or UI form). In general the UX model has a close correspondence to the prototype model. "UIForm" stereotyped classes in the prototype model map well to the "screen" stereotyped classes in the UX model. The "UIField" stereotyped attributes in the prototype model map well to attributes in the UX model that represent dynamic content or form input controls. This correspondence is in fact so close that some teams may decide to forgo the creation of a formal UX model and use the prototype model as the user experience model. Although this might work small applications larger applications might be better off maintaining separate models. The prototype is a model of the functional requirements of the system. The UML model that it produces is just another abstraction of that model expressed in UML. The UX model on the other hand doesn't model the functional requirements but the architecturally significant elements of the user experience. The UX model doesn't identify images or labels, but rather focuses on dynamic content and screen definitions. The prototype model and the UX model, although semantically close to each other do have different goals.

There are many architectural approaches that can be taken when developing client server business applications. The design model is where the first appearance of a concrete architecture can be seen. The software architecture document (SAD) defines the rules by which the application is assembled. It defines types of technologies and design elements that are allowed to be used. Each of these elements should either directly or indirectly map to a requirement of the application. Since the prototype model is an expression of the application's requirements in UML and makes for a convenient place to establish a design / requirements mapping.

Figure 10: Shows how we can take a simple mapping approach, using Rose Web Modeler to begin the process of modeling ASP or JSP pages. The diagram shows how UI forms map roughly to "server page" stereotyped classes.. The prototype model contains a lot of detail about the forms, and it is up to the implementer to ensure that all the data specified by the prototype forms are made available to the team responsible for implementing the scripted pages and HTML.
A detailed analysis of the prototype might lead the architect to adopt a compartment based Web page assembly mechanism. In this situation each Web page is logically divided into regions and built by separate JSPs or ASCXs. This has the advantage of being able to reuse tightly focused page creation elements across the application, and provides a single source for updating them. In this situation "server page" still trace to UIForms in the prototype model, however instead of 1:1 mapping there is a m:n mapping.

There are a wide variety of possible architectures and mappings between the prototype and the design model. There is no one correct architecture for all applications. The important point is not which mapping is correct but rather there is a mapping between the prototype and the design. The prototype, representing functional requirements as expressed in UML, is an easy, efficient way to connect the design and implementation of the application to the requirements that drove it.

One of the most beneficial products of the prototyping tool is the behavioral diagram. Since the prototype is fully functional it has embedded in it the ability to exhibit the required behavior. This can translate into simple sequence diagrams:

**Figure 10: Design Model Mapping to Prototype Model.**
These diagrams express the required behavior of the system in terms of the elements of the prototype. The general approach in the prototype is for each UI form to have its own controller, which is responsible for interacting with the data tier and producing the response UI form. This is the general approach many take when first analyzing and evolving use cases in the system, and serves as just a start. It is very likely that as the design evolves this simplistic approach will evolve into a more suitable one.

The design team can use these diagrams as a starting point to express the same scenarios in terms of design elements.
Of course, while this may or may not be the appropriate architecture for any specific application, it does provide a starting point. As the architecture is refined the details and specifics in the design and implementation will change. The prototype can be updated to reflect changes to the design model, to the architecture or to reflect new understanding of functional requirements. Following "trace" relationships to the design and implementation elements will quickly identify the areas of the application that must be evaluated when assessing the impact of the requested change.

**Database Development**

The database development team is responsible for the logical data model, and for the development of the actual physical database. In a real life application the database may include a number of triggers, stored procedures and constraints that help maintain the integrity of the database. The system analyst developing the prototype may or may not have had a working database to begin with when creating the prototype. As a result tables may have been added to in order to get the prototype to work.

The target application’s database may need to be structured very differently to meet the needs of rest of the system’s requirements. For example one common scalability strategy is to implement the data tier with multiple front end databases all acting as read only data sources (i.e. only select statements), with one central database on the back end dedicated to accepting all update requests, that are later replicated to the front end databases in a timely fashion. For applications that expect the majority of their request to be SELECT statements this provides a good solution that can scale up to millions of simultaneous users.

Functional prototypes are not intended to scale to that number of users, and therefore don’t need such a sophisticated data tier. So the job of the database team is to use the information captured in the prototype, to examine
requirements in light of the constraints imposed by the architecture to design and produce an appropriate database tier for the application.

If we look at our online store example, and the UML produced by the prototyping tool we can see a simple expression of database tables and relationships. The prototyping tool placed certain constraints on the use of primary and foreign keys that may or may not be suitable for the target application. It is up to the database team to use this preliminary schema as the basis for the eventual schema.

![Diagram of Customers and Purchasing Contacts UML model](image)

**Figure 13: Elements of the Prototype’s Schema Expressed in UML**

Using Rose we can quickly transform this prototype of the schema into an actual physical schema associated with a physical database type (i.e. SQL Server 200).

![Diagram showing transformation from prototype to physical schema](image)

**Figure 14: UML Data Model Created Directly from the Prototype Schema.**

This data model becomes the foundation for the actual database in the target application. The database team may need to re-work the schema. The model will however maintain the tractability links between the data schema as it appears in the context of the prototype, functional requirements, and the
Using Functional Prototypes as Test Artifacts

While design and implementation of the target application is underway, the testing team can begin the process of developing test cases and test scripts. Giving an accurate functional prototype to a test designer can be like winning the lottery. Most test teams have to wait until the implementation team has developed enough of the application before they can even try out their scripts. With a functional prototype, scripts can be developed and debugged before the first versions of the actual application are available to the team.

Of course some changes are to be expected. The functional prototype will use one mechanism for its user interface elements (in the case of easyPilot it is with applets), but the target application may employ a different mechanism (discrete tables across multiple Web pages, COM objects, JavaScripting, etc.). The look should be similar but the test team should be prepared to react to changes if needed.

Conclusions

Functional prototyping tools for visualization and simulation of requirements can and should be integrated as a primary means of elicitation, understanding, communication, validation, documentation and formalization of system requirements.

Functional prototyping is an effective tool for teams building business oriented applications, especially for new systems with primarily non-technical users. The requirements for these types of system are often very vague at first and change rapidly early on in the requirements gathering phase. Functional prototypes help give the stakeholders and future users an early view of the
system that can be very easily understood.

Good prototyping tools, like easyPilot, go one step further. By producing UML models directly from the prototype the development team has an easy source of reference and traceability from which to start the process of developing the right solution with the right architecture.

Notes

1 See RUP: Phase: Inception

2 In the RUP these non-functional requirements are collectively called "Supplementary Specifications."

3 In the case of our on line store any other type of architecture is just not practical.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!

Copyright Rational Software 2002 | Privacy/Legal Information