Patchwork Prototyping: A Rapid Prototyping Technique That Harnesses the Power of Open-Source Software

[This is an early draft version of a book chapter that was later published. It is not the same text as was published in the final form.]

M. Cameron Jones
University of Illinois at Urbana-Champaign
Graduate School of Library and Information Science
501 E. Daniel St. Champaign, IL 61820
Phone: 217-721-0658 (m)
Fax: 217-244-3302
Email: mjones2@uiuc.edu

Ingbert R. Floyd
University of Illinois at Urbana-Champaign
Graduate School of Library and Information Science
501 E. Daniel St. Champaign, IL 61820
Phone: 217-721-3171 (m)
Fax: 217-244-3302
Email: ifloyd2@uiuc.edu

Michael B. Twidale
University of Illinois at Urbana-Champaign
Graduate School of Library and Information Science
501 E. Daniel St. Champaign, IL 61820
Phone: 217-265-0510 (o)
Fax: 217-244-3302
Email: twidale@uiuc.edu

ABSTRACT

This chapter explores the concept of patchwork prototyping - the combining of open source software applications to rapidly create a rudimentary but fully functional prototype that can be used and hence evaluated in real life situations. The use of a working prototype enables the capture of more realistic and informed requirements than traditional methods that rely on users trying to imagine how they might use the envisaged system in their work, and even more problematic, how that system in use may change how they work. Experiences with the use of the method in the development of two different collaborative applications are described. Patchwork prototyping is compared and contrasted with other prototyping methods including paper prototyping and the use of commercial off the shelf software.

INTRODUCTION

The potential for innovation with open-source software (OSS) is unlimited. Like any entity in the world, OSS will inevitably be affected by its context in the world. As it migrates from one context to another, it will be appropriated by different users in different ways, possibly in ways in which the original stakeholders never expected. Thus, innovation is not only present during design and development, but also during use (Thomke & von Hippel, 2002). In this chapter, we explore an emerging innovation-through-use: a rapid prototyping-based approach to requirements gathering using OSS. We call this approach "patchwork prototyping" because it involves patching together open-source applications as a means of creating high-fidelity prototypes. Patchwork prototyping combines the speed and low cost of paper prototypes, the breadth of horizontal prototypes, and the depth and high-functionality of vertical, high-fidelity prototypes. Such a prototype is necessarily crude as it is composed of stand-alone applications stitched together with visible seams. However, it is still extremely useful in eliciting requirements in ill-defined design contexts, because of the robust and feature-rich nature of the component OSS applications.

One such design context is the development of systems for collaborative interaction, like cybercollaboratories. The authors have been involved in several such research projects, developing cyberinfrastructure to support various communities, including communities of learners, educators, humanists, scientists, and engineers. Designing and developing such systems, however, is a significant challenge; as Finholt (2002) noted, collaboratory development must overcome the "enormous difficulties of supporting complex group work in virtual settings" (p. 93). Despite many past attempts to build collaborative environments for scientists (see Finholt, 2002 for a list of collaboratory projects), little seems to have been learned about their effective design, and such environments are notorious for their failure (Grudin, 1988; Star & Ruhleder, 1996). Thus, the focus of this chapter is on a method of effective design through a form of rapid, iterative prototyping and evaluation.

Patchwork prototyping was developed from our experiences working on cybercollaboratory projects. It is an emergent practice we found being independently redeveloped in several projects; thus we see it as an effective, *ad hoc* behavior worthy of study, documentation, and formalization. Patchwork prototyping is fundamentally a user-driven process. In all of the cases where we saw it emerge, the projects were driven by user groups and communities eager to harness computational

power to enhance their current activities or enable future activities, and the developers of the prototypes had no pretence of knowing what the users might need *a priori*. As a result, patchwork prototyping's success hinges on three critical components:

- Rapid iteration of high-fidelity prototypes;
- Incorporation of the prototypes by the end-users into their daily work activities;
- Extensive collection of feedback facilitated by an insider to the user community.

In this chapter, we focus on how the method worked from the developers' point of view. It is from this perspective that the advantages of using OSS software are most striking. However, one should bear in mind that the method is not just a software development method, but also a sociotechnical systems (Trist, 1981) development method: the social structures, workflows, and culture of the groups will be co-evolving in concert with the software prototype.

REQUIREMENTS GATHERING IN COLLABORATIVE SOFTWARE DESIGN

Software engineering methods attempt to make software development resemble other engineering and manufacturing processes by making the process more predictable and consistent. However, software cannot always be 'engineered', especially web-based applications (Pressman et al., 1998). Even when application development follows the practices of software engineering, it is possible to produce applications that fail to be used or adopted (Grudin, 1988; Star & Ruhleder, 1996). A major source of these problems is undetected failure in the initial step in building the system: the requirements gathering phase. This is the most difficult and important process in the entire engineering lifecycle (Brooks, 1975/1995, p. 199).

In designing systems to support collaborative interaction, developers are faced with several complex challenges. First, the community of users for which the cyberinfrastructure is being developed may not yet exist, and cannot be observed to see how they interact. In fact, there is often a technological deterministic expectation that the computational infrastructure being created will cause a community to come into existence. Even in the case where there is a community to study, many of the activities expected to occur as part of the collaboration are not currently being practiced because the tools to support the activities do not yet exist. As a result, developers gain little understanding about how the users will be interacting with each other, or what they will be accomplishing, aside from some general expectations that are often unrealistic.

Gathering requirements in such an environment is a highly equivocal task. Where uncertainty is characterized by a lack of information which can remedied by researching an answer, collecting data, or asking an expert, equivocal tasks are those in which "an information stimulus may have several interpretations. New data may be confusing, and may even increase uncertainty." (Daft & Lengel, 1986, p. 554). Requirements gathering is one such situation, where the developers cannot articulate what information is missing, let alone how to set about obtaining it. The only resolution in equivocal situations is for the developers to "... enact a solution. [Developers] reduce equivocality by defining or creating an answer rather than by learning the answer from the collection of additional data." (Daft & Lengel, p. 554). As Daft & Macintosh (1981) demonstrate, tasks with high equivocality are unanalyzable (or rather, have low analyzability: Lim & Benbasat, 2000), which means that people involved in the task have difficulty determining such things as alternative courses of action, costs, benefits, and outcomes.

RAPID PROTOTYPING

Rapid prototyping is a method for requirements gathering which has been designed both to improve communication between developers and users, and to help developers figure out the usefulness or consequences of particular designs before having built the entire system. The goal of rapid prototyping is to create a series of iterative mockups to explore the design space, facilitate creativity, and to get feedback regarding the value of design ideas before spending significant time and money implementing a fully functional system (Nielsen, 1993). There are several dimensions to prototypes. One dimension is the range from low-fidelity to high-fidelity prototypes (see table 1; Rudd, Stern, & Isensee, 1996). Low-fidelity prototypes have the advantages of being fast and cheap to develop and iterate. However, they are only able to garner a narrow range of insights. Perhaps the most popular low-fidelity prototyping technique is paper prototyping (Rettig, 1994). Paper prototypes are very fast and very cheap to produce. They can also generate a lot of information about how a system should be designed, what features would be helpful, and how those features should be presented to the users. However, paper prototypes do not allow developers to observe any real-world uses of the system, or understand complex interactions between various components and between the user and the system. Also, they do not help developers understand the details of the code needed to realize the system being prototyped.

High-fidelity prototypes, on the other hand, can simulate real functionality. They are usually computer programs themselves which are developed in rapid development environments (Visual Basic, Smalltalk, etc.) or with prototyping toolkits (CASE, I-CASE, etc). In either case, these prototypes, while allowing programmers to observe more complex interactions with users and to gain understandings about the underlying implementation of the system, are comparatively slow and expensive to produce and iterate (Rudd et al., 1996). These costs can be offset somewhat by incorporating these prototypes into the development of the final system itself as advocated by RAD (rapid application development) (Martin, 1991). However, critics of RAD methods are quick to point out the limited scalability of software built using source code from prototypes (Beynon-Davies, Carne, Mackay, & Tudhope, 1999). Typically low-fidelity and high-fidelity prototypes are used in succession, with developers increasing the fidelity of the prototypes as they develop the specifications. Due to their high cost, high-fidelity prototypes may only be built for a select number of designs generated by low-fidelity prototyping, which precludes the generation of a series of disposable high-fidelity proofs of concepts to test out alternative design ideas.

Table 1 Advantages and disadvantages of low and high-fidelity prototyping (Rudd, Stern, & Isensee, 1996, p. 80)

	Advantages	Disadvantages
PrototypesLow-Fidelity	• Lower development cost • Can create many alternatives quickly • Evaluate multiple design concepts • Useful communication device • Address screen layout issues • Useful for identifying market requirements • Proof-of-concept	• Limited error checking • Poor detailed specification to code to • Facilitator-driven • Limited utility after requirements established • Limited usefulness for usability tests • Navigational and flow limitations • Weak at uncovering functionality and integration related issues
PrototypesFidelityHigh-	• Complete functionality • Fully interactive • User-driven • Clearly defines navigational scheme • Use for exploration and test • Look and feel of final product • Serves as a living specification • Marketing and sales tool	• More expensive to develop • Time-consuming to create • Inefficient for proof-of-concept designs • Not effective for requirements gathering

Another dimension to be considered in the prototyping discussion is scope. Software can be viewed as consisting of a number of layers, from the user interface, to the base layer which interacts with the underlying operating system or platform. Horizontal prototypes encompass a wide scope, spanning the breadth of a system but only within a particular layer (usually the user-interface). Users can get a sense of the range of the system's available functions; however, the functionality is extremely limited. This can help both the user and the programmer understand the breadth of the system, without plumbing its depths. Vertical prototypes on the other hand, take a narrow slice of the system's functionality and explore it in depth through all layers. This allows users to interact with a particular piece of the system, and gives the programmer a detailed understanding of the subtle issues involved in its implementation (Floyd, 1984; Nielsen, 1993).

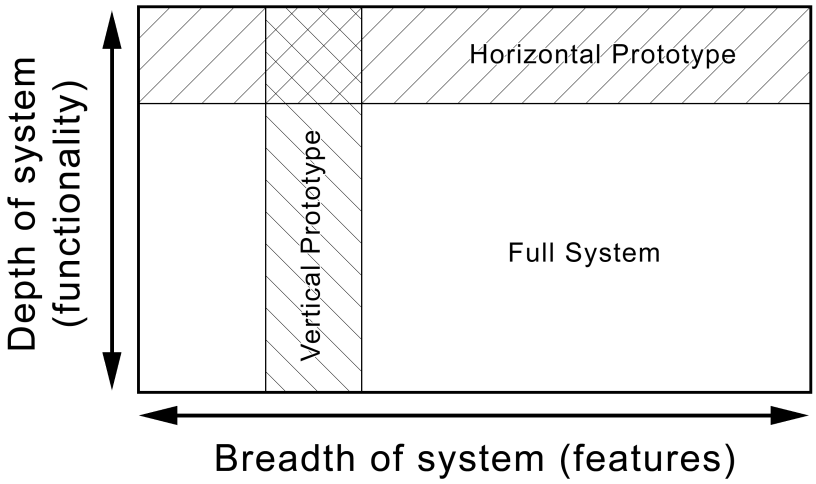

Figure 1 Horizontal and vertical prototypes (Nielsen, 1993, p. 94)

The high equivocality present when designing collaborative systems makes it difficult to apply rapid prototyping techniques effectively. Because users may not be able to articulate what they want or need, it helps to be able to collaboratively interact with high-fidelity systems in order to test them in real world situations and see what requirements emerge. Without such an experience, it is unlikely that any feedback the developers get from the users, either through direct communication or observation will be useful. Thus, low-fidelity prototypes are limited in their power to elicit requirements as the users have difficulty imagining how the system the prototypes represent will work, what it could do for them, or how they might use it. Also, since the majority of tasks involved in collaboration are quite complex, and require multiple kinds of functionality to complete, the users need to be able to interact with the system as a whole, and with considerable depth of implementation, thus requiring a prototype that is both horizontal *and* vertical.

The economics of developing high-fidelity prototypes which are both horizontal and vertical in scope, however, are problematic. Even if the developers were to build a series of high-fidelity, vertical prototypes, they would end up having built the equivalent of an entire system from scratch just to have a functionally sufficient prototype. Not only would it be expensive and time-consuming, but the functionality and robustness would be minimal at best. Also, it is likely that the work would need to be discarded and replaced with something new, since it is unlikely that the design would be correct on the first, second, or even third try. Thus, the typical methods of prototyping are not sufficient, either because developing all the code would be too expensive, or the prototypes which are developed do not have high enough fidelity.

The proliferation of production-scale OSS systems has created a vast field of growing, reliable, usable, and feature-rich programs, a large number of which support aspects of web-based collaboration. These programs can be easily stitched together because the code is open and modifiable. Furthermore, they can be treated as disposable since one application can easily be discarded and replaced with another. This presents an opportunity for developers to rapidly build and evaluate a high-fidelity prototype of a collaborative environment comprising a patchwork of

multiple open-source applications. Such a prototype spans the breadth of a horizontal prototype and the depth of a vertical prototype within a single system.

ORIGINS AND EXAMPLES OF PATCHWORK PROTOTYPING

Patchwork prototyping is a rapid prototyping approach to requirements gathering which was emergent from practice rather than designed *a priori*. We have been involved with several groups which were developing cyberinfrastructure to support collaboration, and in each group we observed *ad hoc* prototyping and development strategies which were remarkably similar and which developed entirely independent of each other. Upon making these observations, we realized that there was a core process at work in each of these projects which could be abstracted out and described as a general approach to requirements gathering for developing cyberinfrastructure. Because patchwork prototyping evolved from practice, however, we believe that it will be much easier to understand our formal description of the approach after we describe some of the relevant details of our experiences. In this section we describe two projects with which we were involved, and the relevant dynamics of each project; in the following section we describe the patchwork prototyping approach more abstractly.

Project Alpha: Building a Cybercollaboratory for Environmental Engineers

Project Alpha (a pseudonym used to preserve anonymity) was devoted to building a cybercollaboratory for environmental engineers. At the beginning, the project was intended to be a requirements gathering project, and the goal was to build a functional prototype of the cyberinfrastructure which would be presented to the granting agency as part of a larger proposal. The effort was a success and now, more than a year after the project began, the prototype is being converted into a production-scale system. The cybercollaboratory prototypes were largely designed and built over a period of six months by a team of two developers, with significant contribution to the design by a team of around twelve to thirteen other researchers (these researchers, plus the two developers, we call the design team), and some minor programming contributions by undergraduates employed by the project. By the end of the prototyping phase, there was a community of users, which included 60-70 active users out of approximately 200 registered users, ten of which comprised a core group of vocal users who provided significant feedback on the design

The project Alpha prototype was constructed on the Liferay portal server framework. In addition to using existing portlets, the developers also wrapped other OSS applications in portlet interfaces, enabling their rapid integration into the prototype. A number of different OSS applications were used, including: the Heritrix web crawler, the Lucene search engine, and the MediaWiki wiki system. Other applications were similarly integrated, but were not necessarily publicly available OSS. Some were in-house applications developed by other projects, for which the developers had source code. These applications were used to prototype data-mining and knowledge management functionality in the cybercollaboratory.

The general process by which these tools were incorporated was very *ad hoc*. The development team might decide on prototyping a particular function, or the programmers might get some idea for a 'cool feature' and would set about integrating the feature into the system. This approach had

several unexpected benefits. First, minimal time was spent building portlets, so that when a version of the prototype was presented to the design team, minimal effort was lost when particular features or portlets were rejected as being unsuitable. Second, it allowed the design team to choose between several different portlets which had essentially the same function but different interfaces (i.e., were optimized for different types of use). Third, it allowed the developers to easily switch features off when the interface for a portlet was too complex, or turn them back on if they were requested by either the design team or the active users. Fourth, the development community and the associated forums, mailing-lists and websites, surrounding the OSS applications which were integrated into the prototype served as excellent technical support (Lakhani & von Hippel, 2002).

The fact that the prototype was fully functional was critical to its success in eliciting requirements. By using the prototypes over a period of six months, the users were able to incorporate them into their day-to-day work practices. This allowed them to evaluate the utility of the tool in various contexts of actual use. Without functionality, the developers feel that it would have been impossible to effectively gather requirements. However, it was also vital that the users communicate their experiences to the developers, both formally and informally. To this end, the developers conducted several surveys of the users, asking them about the prototype and features they found useful. The developers also used the prototype itself to solicit feedback. On the front page of the prototype was a poll asking users to vote for the features they liked the most. Additionally, on every page of the prototype was a feedback form that allowed users to send quick notes about the system as they experienced it. The users also communicated with the developers via informal means such as email and face-to-face meetings. However, the most important method of obtaining feedback was that one of the PIs in the project acted as an intermediary, actively soliciting feedback from users as an insider to the community of environmental engineers. The PI's position allowed them to receive more feedback of higher quality and honesty than the developers would have been able to collect on their own.

To illustrate the process in more detail, we describe how one particular piece of OSS was integrated with the cybercollaboratory. The developers wanted to allow users to be able to collaboratively edit documents in the system. The Liferay suite had a wiki system available which the programmers enabled; however, users found that tool to be too difficult to use, partly because of the unintuitive markup syntax of the particular wiki used, and partly because they had no tasks which clearly lent themselves to the use of such a tool. Later during the prototyping phase, some members of the design team wanted to demonstrate the usefulness of scenarios and personas in facilitating requirements gathering, and from prior experience suggested the use of a wiki. In response to this request and the prior difficulties in using the bundled tool, the developers installed MediaWiki on the server, and added a link from the cybercollaboratory's menu next to the existing wiki tool pointing to the MediaWiki installation. No time was spent trying to integrate the Liferay and MediaWiki systems; each application had separate interfaces and user accounts.

One benefit of using the MediaWiki system was that it allows people to use the system without logging in, thereby mitigating the need to integrate authentication mechanisms. Users found the MediaWiki system easier to learn and use, and began using it exclusively over the in-built Liferay wiki. The developers then decided to embed the MediaWiki interface in the rest of the cybercollaboratory and wrote a simple portlet that generates an HTML IFRAME to wrap the MediaWiki interface. Each step of integrating the MediaWiki installation took only minimal effort

on the part of the developers (sometimes literally only a matter of minutes) and generated insights about the role and design of a collaborative editing tool in the cybercollaboratory. Among the design insights gained by the developers are that the tool should be easy to use with a simple syntax for editing. Also, the tool should support alternate views of the data; offering a unified view of all documents either uploaded to the site's document repository or created and edited on the wiki. The users were able to see how this tool could benefit their jobs and that shaped the requirements of the tool. As a result of this process, the project is currently implementing a new collaborative editing component. This component will have features like integrated authentication; group- and project-based access control; integration with other features (e.g., project views, and wiki-linking). Additionally, the new collaborative writing component will deprecate redundant and confusing features like in-wiki file uploads.

Project Beta: Building Collaborative Tools to Support Inquiry-Based Learning

Project Beta is an ongoing research project aimed at designing and building web-based tools to support processes of inquiry as described by John Dewey (Bishop et al., 2004). Initiated in 1997, the project has embraced a long-term perspective on the design process and produced a series of prototypes which support inquiry-based teaching and learning. In 2003 the project began exploring the development of tools to support collaborative inquiry within groups and communities. The current prototype is the third major revision of the collaborative cyberinfrastructure, with countless minor revisions on-going. Throughout the project's lifespan several generations of programmers have joined and left the development team. For a thirty month stretch the majority of programming was sustained by a single graduate student programmer. Between four and eight other researchers filled out the design team.

The prototypes are available for anyone to use, and the source code is also distributed under a Creative Commons license. To date, the prototypes have been used to support a large number of communities of users ranging from water quality engineers, to volunteers in a Puerto Rican community library in Chicago, from researchers studying the honeybee genome, to undergraduates in the social sciences. There are numerous other groups using the system for any number of purposes. Given this scenario, it is practically impossible to design for the user community or any intended use.

The prototypes were developed in the PHP programming language on an open-source platform consisting of Apache, MySQL, and RedHat Linux. In contrast to Project Alpha where the developers initially did very little programming and primarily used readily available tools, the developers of Project Beta spent considerable effort building an infrastructure from scratch, in part because the developers were initially unaware of relevant OSS. However, as the project progressed several open-source tools were incorporated into the prototypes including the JavaScript-based rich-text editors FCKEditor and TinyMCE, phpBB bulletin board system, and MediaWiki.

To demonstrate the process in more detail, we describe how one particular piece of OSS was integrated with the prototypes. In the earliest version of the cyberinfrastructure, users expressed an interest in having a bulletin board system. The developers selected the phpBB system and manually installed copies of phpBB for each community that wanted a bulletin board; the bulletin board was simply hyperlinked from the community's homepage. In the next iteration of the

prototype, the phpBB system was modified to be more integrated with the rest of the prototype. Users could now install a bulletin board themselves, without involving the developers, by clicking a button on the interface. Furthermore, the authentication and account management of the bulletin board was integrated with the rest of the prototype, eliminating the need for users to log in twice. However, the full features of phpBB were more than the users needed. They primarily made use of the basic post/reply functions and the threaded conversation structure. Users indicated that the overall organization of the board system into topics, threads, and posts made sense to them. In the most recent major revision of the prototype, the phpBB system was replaced by a simpler, more integrated home-made bulletin board prototype which supported these basic features. Had the development progressed in the opposite order (i.e., building the simple prototype first, then adding features) it is possible that developers could have wasted valuable time and energy prototyping features which would only be discarded later for lack of use.

GENERALIZED APPROACH TO PATCHWORK PROTOTYPING

Based on the experiences described above, we have outlined a general approach to building patchwork prototypes using OSS. While our experience has been primarily with web-based tools, and this process has been defined with such tools in mind, it is likely that a similar approach could be taken with prototyping any kind of software. Like other prototyping methods, this is designed to be iterated, with the knowledge and experience gained from one step feeding into the next. The approach entails the following five stages:

1. Make an educated guess about what the target system might look like;
2. Select tools which support some aspect of the desired functionality;
3. Integrate the tools into a rough composite;
4. Deploy the prototype and solicit feedback from users;
5. Reflect on the experience of building the prototype and the feedback given by users, and repeat.

For the most part, these steps are relatively straight-forward. Making the first educated guess about what the target system might look like can be the hardest step in this process, because it requires the design team to synthesize their collective knowledge and understanding of the problem into a coherent design. In this first iteration of the process, it is often helpful to use paper prototypes and scenarios, but their function is primarily to serve as communications devices and brainstorming aids. The high equivocality of the situation almost guarantees, however, that whatever design they produce will be insufficient. This is not a failure. It is an expected part of the process, and the design will be improved on subsequent iterations. The important thing is to have a starting point which can be made concrete, and not to spend too much time brainstorm ideas. It is essential not to become bogged down in controversies about how the software “ought” to look, but rather to put together a prototype and test it out with users in their everyday environments and let the users figure out what works, what does not, and what is missing.

Selection and Integration of Tools: The Benefits of Using Open-Source Software

There are several important considerations to keep in mind when selecting the tools. On first glance, patchwork prototyping as a method does not *require* OSS; the same general process could theoretically be followed by using software that provides APIs, or by creating prototypes through adapting methodologies for creating production scale software systems such as COTS (Commercial Off-The-Shelf) integration (Boehm & Abts, 1999). However, using OSS confers several important advantages; in fact, we believe that patchwork prototyping is only now emerging as a design practice because of the recent availability of a significant number of mature, production-scale OSS systems.

Without access to source code, developers are limited in how well they can patch together different modules, the features they can enable or disable, their ability to visually integrate the module with the rest of the system, and their ability to understand the underlying complexity of the code needed to construct such systems on a production-scale. High-profile OSS is often of high quality, which means that difficult design decisions have already been made. Given that it is built from the collective experiences of many programmers, less effective designs have already been tried and discarded. In fact, by using and delving into the human-readable (compared to that generated by CASE tools, for example), open-source code, the developers can get a grounded understanding of how particular features can be implemented, which can enable them to better estimate development time and costs. High-profile OSS is often of high quality, which means that difficult design decisions have already been made, and it is built from experience: less effective designs have already been tried and discarded.

The web-based nature of patchwork prototypes affords several ways of integrating the selected software into the prototype, ranging from shallow to deep. Shallow integration consists of either wrapping the tools in an HTML frame to provide a consistent navigation menu between the tools, or customizing the HTML interfaces of the tools themselves to add hyperlinks. Most open-source web applications use HTML templates, cascading style sheets, and other interface customization features, which make adding or removing hyperlinks and changing the look-and-feel very easy. The advantage of shallow integration is the ease and speed with which the developer is able to cobble together a prototype. A significant drawback to shallow integration is that each application remains independent.

Deeper integration usually requires writing some code, or modifying existing source code. This may include using components or modules written for the extension mechanisms designed into the application or other modifications made to the application's source code. If the developers cannot find precisely what they are looking for, they can fashion the code they need by copying and modifying similar extension code; or, in the worst case, the developers will need to write new code to facilitate the integration. However, the amount of code needed is very little in comparison to the amount of code that would have been required of the developers building a prototype from scratch.

For any prototyping effort to be worthwhile, the costs of creating the prototypes must be minimal. OSS systems tend to be fully implemented, stand-alone applications with many features and capabilities which provide a wealth of options to play with when prototyping to elicit requirements. The minimal effort required to add features allows the programmers to treat the features as disposable, because little effort was needed to implement them, so little effort is wasted when they are switched off or discarded. That most OSS is free is also important, both for

budgetary reasons and because the developers can avoid complicated licensing negotiations. Additionally, most OSS has a very active development community behind it with members who are often eager to answer the developer's questions in considerable depth, and do so for free, unlike the expensive technical support which is available for commercial products. All of this facilitates the requirements gathering process, because iterations of the prototype can be rapidly created, with high functionality, at minimal cost and with minimal effort and emotional investment by the developers.

Deployment, Reflection, and Iteration

Making an educated guess about what the target system might look like can be the hardest step in this process, because it requires the design team to synthesize their collective knowledge and understanding of the problem into a coherent design, yet the high equivocality of the situation almost guarantees that whatever design they produce will be insufficient. However, this is not a failure. It is an expected part of the process, and the design will be improved on subsequent iterations. The important thing is to have a starting point which can be made concrete.

There are several important considerations to keep in mind when selecting the tools. Access to the source code and freedom to modify it is essential. Without such access, the developers are limited in how well they can patch different modules together, in what features they can enable or disable, in how they create a visual integration with the rest of the system, and in their ability to understand the underlying complexity of the code which they are integrating and will likely have to rewrite themselves for the production scale version. In fact, by using and delving into the open-source code, the developers can often get a good feel for how complicated it will be to implement a particular feature robustly, and can make better estimates as to how long it will take and how expensive it will be to implement a particular feature.

The web-based nature of patchwork prototypes also lends them several advantages. The prototypes are not platform dependant, therefore one prototype can be used for everybody, no matter what operating system they happen to be using (Mac, PC, Linux, etc.). Additionally, browsers and web standards come with some built-in support for handling various display and interface issues, such as support for multiple languages (e.g., UTF-8 character encodings, support for left-to-right and right-to-left scripts, etc.).

Finally, it affords several ways of integrating the selected software into the prototype, ranging from shallow to deep. Shallow integration consists of either wrapping the tools in an HTML frame to provide a consistent navigation menu between the tools, or customizing the HTML interfaces of the tools themselves to add hyperlinks. Most open-source web applications have easily editable interface templates and other customization features, such as HTML templates and cascading style sheets, which make adding or removing hyperlinks and changing the look-and-feel very easy. The advantage of shallow integration is the ease and speed with which the developer is able to cobble together a prototype. A significant drawback to shallow integration is that each application remains independent.

Deeper integration usually requires writing some code, or modifying existing source code. The developers search for existing source code which has been written by others in the open-source

community. This code might be components or modules written for the extension mechanisms designed into the application or other modifications made to the application's source code. If the developers cannot find precisely what they are looking for, they can fashion the code they need by copying and modifying similar extension code; or, in the worst case, the developers will need to write new code to facilitate the integration. However, the amount of code needed is very little in comparison to the amount of code that would have been required of the developers building a prototype from scratch.

The minimal effort required to add features allows the programmers to treat the features as disposable, because little effort was needed to implement them, so little effort is wasted when they are switched off or discarded. This facilitates the requirements gathering process, because iterations of the prototype can be rapidly created, with high functionality, at low cost, and with minimal investment by the developers.

During the deployment of the prototype, future users integrate the cyberinfrastructure into their work practices for an extended period of time and explore what they can do with it collaboratively. The collection of feedback on user experiences allows requirements gathering which is not purely need-based, but also opportunity- and creativity-based. By seeing a high-fidelity prototype of the entire system, users can develop new ideas of how to utilize features which go beyond their intended use, and conceptualize new ways of accomplishing their work. In addition, users will become aware of gaps in functionality which need to be filled, and can explain them in a manner that is more concrete and accessible to the developers.

When reflecting on the collected feedback, however, the design team must realize that the prototype does not simply elicit technical requirements; it elicits requirements for the collaborative sociotechnical system as a whole. The existence of the prototype creates a technological infrastructure which influences the negotiation of the social practices being developed by the users via the activities the infrastructure affords and constrains (Kling, 2000). The design team must be aware of how various features affect the development of social practice, and must make explicit the type of interactions which are required but are not currently realized. By allowing the users to interact with the prototypes for extended periods, collecting feedback on their experiences, and paying attention to the social consequences of the cyberinfrastructure, a richer understanding of the sociotechnical system as a whole can emerge. Thus, reflection is a process of attending to the consequences of the design for the broader sociotechnical system, and integrating those consequences into a holistic understanding of how the system is evolving.

Iteration is essential to the rapid prototyping approach. First, iteration allows for the exploration of more features and alternatives. This can uncover overlooked aspects of the system which might be of use. This can also reinforce the importance or necessity of particular features or requirements. Furthermore, iteration provides the users with a constant flow of new design possibilities which prevents them from becoming overly attached to any single design giving them the freedom to criticize particular instances of the prototype. Ultimately, it is impossible to reach complete understanding of the system given its evolving nature. However, by iterating the prototyping process, the design space may narrow, identifying a set of key requirements. At this point the design is not complete, but work on a flexible production-scale system can begin, and further exploration of the design space can be continued within that system.

Table 2. Comparison of patchwork prototyping with other methods.

Paper Prototyping	Patchwork Prototyping	COTS/API Prototyping
Speed		
Can iterate a prototype multiple times in an afternoon	Can iterate a prototype in less than a week	Can take weeks or months to iterate a prototype
Monetary Costs		
Cost of office supplies	Free, or minimal cost of licenses if in business setting	Purchasing and licensing software can be expensive
Availability of Materials		
Usually already lying around	Large number of high quality OSS available for free download	Not all commercial systems have APIs
Functionality		
Non-functional	High	High
Accessibility		
Anyone can prototype systems using paper, including non-technical end-users	Requires skilled programmers to create patchwork prototypes	Requires skilled programmers to integrate commercial software
Interface		
Not polished, but can provide a consistent, and/or innovative interface concept for consideration	Not renowned for excellent usability. Assembled components may be inconsistent	Individual elements may be high quality and familiar. Assembled components may be inconsistent
Flexibility		
High – can do anything with paper	High – can modify source to create any desired functionality	Low – are restricted to what the API allows, which may be limited
Disposability		
High – little investment of time, money, emotions	High – little investment of time, money, emotions	Low – significant effort and money can result in high emotional investment
User Attachment		
Low – users can see it is rough and non-functional	Med to High – upon using it, can get attached to the system, unless iterated rapidly	High – cannot be iterated fast enough to avoid attachment

STRENGTHS & LIMITATIONS

Patchwork prototyping addresses two major problems that designers face when building new sociotechnical systems. First, it allows the design team to get feedback on the prototype's use in real-world situations. Users interact with the system in their daily activities which focuses their feedback around task-related problems. In Project Alpha, when members of the design team started using the prototype, the feedback changed from general praise or criticism of the appearance of the interface, to more detailed explanations of how particular functionality aided or inhibited task performance. Second, it reduces the equivocality of the design space. By creating a functional

prototype, discussions change from being highly suppositional, to being about concrete actions, or concrete functionality.

Integration into the real-world context is markedly different from other prototyping and requirements capture methods. Paper prototypes are typically given to users in a laboratory setting (Nielsen, 1993), thus all the tasks are artificial. While this can give developers important design insights, the drawback is that prototypes can end up optimized for artificial tasks, and not for real-world use. More expensive methods such as participatory design (Ehn & Kyng, 1991) and ethnography (Crabtree, Nichols, O'Brien, Rouncefield, & Twidale, 2000) try to incorporate real-world use into the design process; the former by bringing users into the design team, the latter by observing users in their natural work environment. However, when the technology that these methods were used to design is introduced, it inevitably changes the practices and social structures present in the work environment, often in a way that cannot be predicted. Patchwork prototyping overcomes these limitations by being cheap and by providing real-time feedback on both users' problems with the software, and the effects the software is having on the broader work context.

The advantages of patchwork prototyping can be seen when comparing it to other prototyping techniques. In Table 2 we compare it to paper prototyping and to prototyping using commercial off the shelf (COTS) software. The advantages of patchwork prototyping are that it has many of the benefits of paper prototyping, including low cost and ready availability of materials, yet provides the high functionality of COTS/API prototyping, and the effort needed to create the prototypes and the length of the iteration cycles lies somewhere in between. Thus, while we see the method as being yet another tool for developers and designers to have in their tool-box, in many ways, it combines the best of both worlds.

The patchwork prototyping approach is not without limitations, however. Despite our hope that the visibility of the seams between the applications would be interpreted by the users as an indication that the prototype is a work-in-progress, our experiences seem to indicate that the users still view it as a finished product due to the fact that it has real functionality. It is possible that such interpretations can be overcome through social means, by emphasizing the fact that the system is a prototype to all users who are encouraged to test it. However, since none of the projects we participated in did this, we have no idea whether or not that would be sufficient. One thing that is clear, however, is that visual coherence between applications greatly facilitates the ease of use, and positive perceptions of the system as a whole. In fact, in project Alpha it was realized that users need different views of the component modules and features depending on the context in which they access the applications, and in some of those views the distinctions between modules must be totally erased.

Patchwork prototyping requires highly skilled programmers to be implemented effectively. Programmers must have significant experience within the development environment in which the OSS applications are coded; otherwise, they will spend too much time reading code and learning the environment, and the speed of implementation will not be as fast. Also, OSS can have security vulnerabilities which can compromise the server on which they are hosted. Project Beta ran into this problem when multiple installations of phpBB succumbed to an internet worm, bringing down the prototype for several days. Third, patchwork prototyping requires a long-term commitment by users, and a motivated facilitator who is able to convince the users to adopt the prototype and

incorporate it into their work practices. The facilitator must collect feedback about the users' experiences. Without willing users and the collection of feedback, the prototyping process will likely fail.

FUTURE TRENDS

The use of patchwork prototyping is still in its infancy. The relative ease with which patchwork prototypes can be constructed means that the method itself affords appropriation into new contexts of use. For example, one of the biggest costs to organizations is buying software systems such as enterprise management systems. Patchwork prototyping offers a cheap and effective method for exploring a design space and evaluating features. Consequently, through prototyping managers can be more informed when shopping for software vendors, and can more effectively evaluate how effective a particular vendor's solution will be for their company (Boehm & Abts, 1999).

Because users have to integrate the prototype into their daily work practices, transitioning from the patchwork prototype to the production-scale system can be highly disruptive. One method of avoiding this is having a gradual transition from the prototype to the production-scale system by replacing prototype modules with production-scale modules. To do this, however, the prototypes must be built on a robust, extensible, modular framework because the latter component is not easily replaced. If this model is used, the system development process need never end. Prototypes of new features can constantly be introduced as new modules, and, as they mature, be transitioned into production-scale systems. As more developers and organizations support open-source development, the number and availability of OSS applications will increase. As more modules are written for particular open-source, component-based systems, the costs of doing patchwork prototyping will further decrease, as will the threshold for programming ability—perhaps to the point where users could prototype systems for themselves which embody specifications for software programmers to implement.

CONCLUSIONS

Patchwork prototyping is a rapid prototyping approach to requirements gathering which shares the advantages of speed and low cost with paper prototypes, breadth of scope with horizontal prototypes, and depth and high-functionality with vertical, high-fidelity prototypes. This makes it particularly useful for requirements gathering in highly equivocal situations such as designing cyberinfrastructure where there is no existing practice to support, because it allows future users to integrate the cyberinfrastructure into their work practices for an extended period of time and explore what they can do with it collaboratively. It has the benefit of allowing the design team to monitor the sociotechnical effects of the prototype as it is happening, and gives users the ability to provide detailed, concrete, task-relevant feedback.

OSS is particularly well suited for patchwork prototyping for several reasons. First, it is usually free, and if not, the licensing fees are often very reasonable (as compared with commercial software). Second, OSS systems tend to be fully implemented, stand-alone applications with many features and capabilities which provides a wealth of options to play with when prototyping to elicit requirements, none of which need to be hand-coded, but if particular features are missing, they usually can be added with minimal effort. Third, there is an active development community behind

the OSS software with members who are usually eager to answer the developer's questions in considerable depth, and do it for free, unlike the expensive technical support which is available for commercial products. Fourth, the code is human readable unlike that generated by CASE tools. Fifth, the code is free for reuse and repurposing as long as due credit is given. And sixth, the code of high-profile OSS is often of high quality, which means that difficult design decisions have already been made, and it is built from experience so that previous, less effective designs which might occur to someone without such experience have already been tried and discarded.

Patchwork prototyping is an excellent example of how OSS software can foster innovation. The affordances of open source code and a devoted development team create opportunities to utilize OSS in ways that go beyond the functionality of any particular application's design. The cases presented here merely scratch the surface of a new paradigm of OSS use. Further research is needed to understand the specific features of technologies which afford such innovative integration.

REFERENCES

Beynon-Davies P., Carne C., Mackay H., & Tudhope D. (1999). Rapid application development (RAD): an empirical review. *European Journal of Information Systems, 8*(3), 211-223.

Bishop, A. P., Bruce, B. C., Lunsford, K. J., Jones, M. C., Nazarova, M., Linderman, D., Won, M., Heidorn, P. B., Ramprakash, R., & Brock, A. (2004). Supporting community inquiry with digital resources. *Journal of Digital Information, 5*(3), Article No. 308.

Boehm, B. W., & Abts, C. (1999). COTS Integration: Plug and Pray? *IEEE Computer, 32*(1), 135-138.

Brooks, F. P. (1975/1995). *The Mythical Man-Mouth: Essays on Software Engineering, Anniversary Edition*. Boston, MA: Addison-Wesley.

Crabtree, A., Nichols, D. M., O'Brien, J., Rouncefield, M., & Twidale, M. B. (2000). Ethnomethodologically-informed ethnography and information systems design. *JASIS, 51*(7), 666-682.

Daft, R. L., & Lengel, R. H. (1986). Organizational information requirements, media richness and structural design. *Management Science, 32*(5), 554-571.

Daft, R. L., & Macintosh, N. B. (1981). A Tentative Exploration into the Amount and Equivocality of Information Processing in Organizational Work Units. *Administrative Sciences Quarterly, 26*(2), 207-224.

Ehn, P., & Kyng, M. (1991). Cardboard Computers: Mocking-it-up or Hands-on the Future. In J. Greenbaum, & M. Kyng (Eds.), *Design at Work* (pp. 169-196). Hillsdale, NJ: Laurence Erlbaum Associates.

Finholt, T. A. (2002). Collaboratories. *Annual Review of Information Science and Technology*. *36*(1), 73-107.

Floyd, C. (1984). A Systematic Look at Prototyping. In R. Budde, K. Kuhlenkamp, L. Mathiassen, & H. Zullighoven (Eds.), *Approaches to Prototyping* (pp. 1-18). Berlin: Springer-Verlag.

Grudin, J. (1988). Why CSCW Applications Fail: Problems in the Design and Evaluation of Organizational Interfaces. *CSCW 88: Proceedings of the Conference on Computer-Supported Cooperative Work* (pp. 85-93). Portland, OR: ACM.

Kling, R. (2000). Learning About Information Technologies and Social Change: The Contribution of Social Informatics. *The Information Society, 16*, 217-232.

Lakhani, K. R., & von Hippel, E. (2002). How open source software works: "free" user-to-user assistance. *Research Policy, 1451*, 1–21.

Lim, K. H., & Benbasat, I. (2000). The Effect of Multimedia on Perceived Equivocality and Perceived Usefulness of Information Systems. *MIS Quarterly, 24*(3), 449-471.

Martin, J. (1991). *Rapid Application Development*. New York, NY: Macmillan Publishing Co.

Nielsen, J. (1993). *Usability Engineering*. San Diego, CA: Morgan Kaufman.

Pressman, R. S., Lewis, T., Adida, B., Ullman, E., DeMarco, T., Gilb, T., Gorda, B., Humphrey, W., & Johnson, R. (1998). Can Internet-Based Applications Be Engineered? *IEEE Software, 15*(5), 104-110.

Rettig, M. (1994). Prototyping for tiny fingers. *Communications of the ACM, 37*(4), 21-27.

Rudd, J., Stern, K., & Isensee, S. (1996). Low vs. high-fidelity prototyping debate. *Interactions 3*(1), 76-85.

Star, S. L., & Ruhleder, K. (1996). Steps Toward an Ecology of Infrastructure: Design and Access for Large Information Spaces. *Information Systems Research, 7*(1), 111-134.

Thomke, S., & von Hippel, E. (2002) Customers as Innovators: New Ways to Create Value. *Harvard Business Review, 80*(4), 74-81.

Trist, E.L. (1981). The sociotechnical Perspective: the Evolution of Sociotechnical Systems as a Conceptual Framework and as an Action Research Program. In A. H. van de Ven, & W. F. Joyce (Eds.) *Perspectives on Organization Design and Behavior* (pp. 19-75). New York, NY: John Wiley & Sons.

GLOSSARY OF TERMS

Sociotechnical system – refers to the concept that one cannot understand how a technology will be used in a particular environment without understanding the social aspects of the environment, and that one cannot understand the social aspects of the environment without understanding how the technology being used shapes and constrains social interaction. Thus, one can only understand what is going on in an environment by looking at it through a holistic lens of analysis.

Rapid Prototyping – Rapid prototyping is a method which involves creating a series of prototypes in rapid, iterative cycles. Normally, a prototype is created quickly, presented to users in order to obtain feedback on the design, and then a new prototype is created which incorporates that feedback. This cycle is continued until a fairly stable, satisfactory design emerges, which informs the design of a production-scale system.

Paper Prototyping – a rapid prototyping method for creating low-fidelity prototypes using pencils, paper, sticky notes, and other "low-tech" materials which can be quickly iterated in order to explore a design space. Often used in interface design.

Patchwork prototyping – a rapid prototyping method for creating high-fidelity prototypes out of open-source software which can be integrated by users into their every-day activities. This gives users something concrete to play with and facilitates a collaborative process of sociotechnical systems development. It is ideal for highly equivocal design situations.

COTS Integration – the process by which most businesses integrate commercial off the shelf (COTS) software systems in order to create a computing environment to support their business activities.

Uncertainty – the name for a lack of knowledge which can be addressed by obtaining more information, such as by researching an answer, looking it up in reference materials, or by collecting data.

Equivocality – the name for a lack of knowledge which cannot be mitigated simply by doing research or gathering more information. In an equivocal situation, decisions often need to be made, definitions created, and procedures negotiated by various (often competing) stake-holders.