Learning Design from Emergent Co-Design: Observed Practices and Future Directions

Ingbert R. Floyd
Graduate School of Library and Information Science
University of Illinois at Urbana Champaign
Champaign, IL 61820 USA
ifloyd2@illinois.edu

Michael B. Twidale
Graduate School of Library and Information Science
University of Illinois at Urbana Champaign
Champaign, IL 61820 USA
twidale@illinois.edu

ABSTRACT
In everyday work environments, systems for work evolve constantly in response to changing environments, the need to overcome technical and social obstacles, or out of the desire by individuals to try something new or satisfy their curiosity. This proposal briefly reviews some emergent design activities, namely patchwork prototyping and a recent trend in academic computing for adopting and modifying open-source software. On the basis of this past work, the authors provide suggestions for future research on how design activity by both professional and amateur designers can be studied to inform both the design of systems to support co-design, and to learn better about how to do intentional design.

Keywords
Participatory design, patchwork prototyping, open-source software, emergent design, intentional design

INTRODUCTION
Design research, whether exploring co-design or other forms of design, typically focuses on intentional design: cases where a company, a manager, or a developer design a system, product, or service to solve a particular or imagined need [1]. Yet in everyday work environments, systems for work evolve constantly in response to changing environments, the need to overcome technical and social obstacles, or out of the desire by individuals to try something new or satisfy their curiosity. Typically, this evolution occurs when individuals or groups develop work-arounds to deal with limitations in existing work-flow systems [6], when they adopt new technologies or practices which they find satisfy a particular need (better), when they appropriate existing technologies to satisfy a need [4], or when they innovate with their existing work-flows to make the process more effective or efficient. Thus, the evolution consists of changing work practice, which is motivated by and has consequences for technology use, workplace culture, the effectiveness of policies, etc., a classic example of the task-artifact cycle [3].

This evolution often is reactive and ad-hoc in nature, and while it does involve conscious decisions, these decisions are variable in their reflectiveness, and are typically made by people who are not professional designers, in contexts where they might not consider their decisions to be design-oriented, despite the fact that professional designers would.

This reactive, in-situ designing takes many forms, but it is almost always sociotechnical in nature. The actions involving technologies can take various forms, e.g.: copying-and-pasting data from one application to the next [12], tailoring or tweaking existing applications, cobbling together different pieces of computing and analog technology via bricolage [2,9]; and if the person's skill level is high enough it can involve programmatic customization of the technology. Yet these activities inevitably also involve other actions, e.g.: negotiating a new work-flow pattern with co-workers, appealing to a superior to change organizational policy, recruiting a co-worker to teach them how to use a promising technology, etc.

Emergent, ad-hoc design often occurs by trial-and-error experimentation informed by previous experience and conversations with colleagues—frequently conversations which happen serendipitously. The consequence is that work systems are remarkably robust because workflows are constantly evolving to adjust to changing circumstances, individuals are constantly innovating to maintain them, and they are not fixed by policy, a particular technology structure, etc. The flexibility and informality of this emergent design is its strength, which is important to keep in mind when trying to support it via intentional design for co-design. Yet, that does not mean that the professional designer has no place for aiding and abetting this process.

The innovation that can occur via this reactive, ad-hoc design is limited by the imagination of the amateur designer(s), by their technical expertise, and by the constraints on the mutability of the technology. Thus the professional designer can contribute design expertise, a knowledge of the possibilities of what technology can support, and the skill to construct technology which the amateur designer can envision but not create. For these

contributions to be effective and to preserve the creative robustness of the natural ad-hoc design, an environment of true on-going co-design must be created.

To explore how to create such an environment, we consider two kinds of emergent, on-going co-design that we have observed. By emergent, we mean they were not planned; they evolved in response to the circumstances of the organizations in which they occurred. These projects exhibit a co-design via bricolage that is different from what [2] describes due to new technological capabilities. An analysis of these co-design processes leads to implications for intentional design for co-design via bricolage.

EMERGENT CASES OF DESIGN FOR CO-DESIGN

We discuss two cases that involve co-design activity: patchwork prototyping [5,8] and a recent trend in academic computing that harnesses the power of free/libre open-source software (FLOSS). Our focus in this discussion is not on the details of the processes, but rather on how the structures of the various design environments enable serious co-design activities to occur, in the hopes that this will inform more intentional design for co-design.

Patchwork Prototyping

Patchwork prototyping (discussed extensively elsewhere [5,8]) is an emergent design method that we have observed develop independently in several different projects. The essence of the method is that many different FLOSS applications are patched together with minimal glue code (at times nothing more than hyperlinks) by professional developers to create high-fidelity prototypes which can be rapidly iterated to explore a design space, and which are tested by having actual users incorporate the prototypes into their daily work activity. These prototypes are sometimes augmented using web APIs, but are much more complex than typical mash-ups. Radical iterations can happen in time-spans of less than a week because entire FLOSS applications can be added or removed from the prototype, or the features which are exposed can be altered simply by changing the configuration files. The key to the method is that it involves extensive collection of feedback on the design by project leaders who come from the user community, and thus understand the needs of different user groups much better than the professional designers and developers. It is a method for requirements gathering, not for production-scale systems development, but the transition from prototype to production-scale system can be relatively seamless as the modularity of the prototype affords incremental substitution of production-scale code.

What is interesting about patchwork prototyping is that we have seen it develop as an emergent method in several different projects. While the end result is very participatory in nature, none of the designers or developers in the individual projects came from the co-design tradition, although a few individuals had superficial knowledge about the tradition. Supplementary to our observations, we have anecdotal evidence that the method seems to emerge repeatedly. This is likely due to its pragmatic resolution of the requirements gathering problem, and ease of supporting real user participation in actual use of the evolving prototype.

The traditional software development literature considers the choice between high-fidelity prototyping and rapid prototyping to be one of the fundamental trade-offs a design team has to negotiate at different times in the design process [10]. Patchwork prototyping seems to bypass this tradeoff. Therefore, what is it about the current software development environment that facilitates this kind of activity?

One of the primary reasons we believe patchwork prototyping has only emerged recently as a design method is the emergence of more and more high-quality FLOSS. These applications are feature-rich with many customizable options, and have benefited from cycles of use and feedback over time.

The free as in beer nature of the software is important, in that it keeps prototyping costs down, however the openness of the code is what is more important, because the software can be customized for the purpose of prototype development, and thus developers can help users explore the design space by presenting different versions of the functionality in quick succession. This prevents situated use lock-in; users becoming too comfortable with any particular instance of the interface; and so reluctant to change. These rapid changes also give users the opportunity to get a more visceral feel for the design space of computational possibilities. For example, in hopes of satisfying a particular need, introducing a group to a single wiki software implementation which the users find distasteful can lead users to dismiss wikis as a technology when in fact it is simply a poor implementation of wiki functionality or interface which they are responding to. However, iterating with a second or even a third wiki implementation not only can lead immediately to a desired solution, but can also give the non-technical user a better sense of, in this case, the rather abstract concept of "wikiness". Developing this understanding of the design space is important for users who do not have the same computational sense [13] as experienced computer users or computer programmers.

Patchwork prototyping is not just a result of the technological environment, however; it is also a product of various values and attitudes present in the organization. The approach aligns well with the software engineering value of code reuse over building from scratch wherever possible. However, developer attitudes alone are insufficient to support patchwork prototyping as a method. As noted earlier, the project leadership (design team leadership) in these patchwork prototyping projects always includes (if is not wholly composed of) leaders of the future users. These leaders were advocates for *using* the prototype iterations, and led by example. This had many components including establishing a clear understanding of the need, and a vision for the possibility offered by innovative

software, even if they did not necessarily know in advance what would emerge out of the process. They were invested in using the results of the project, and in addition had the attitude that their job was to get the software or service working.

Finally, the intended users of the product must have an ethic of participation [7]. When patchwork prototyping occurs, there exists a general recognition that these projects are for the mutual good (nobody's going to be put out of work by them, you're job's just going to be easier if it works). Thus users are willing and happy to provide feedback to the design team, as long as it does not take too much of their time (and as long as they are reminded to do so). Typically, this is either because the users have joined the project in order to benefit from the software, or because trusted technology support personnel are developing the project in-house.

OSS in Academic Computing

In academic computing environments, resources to build custom software is often unavailable, and many commercial off-the-shelf (COTS) solutions are insufficient for the unit's needs. In the past, academic units typically had to settle for COTS software that they could bricolage [2,9] together, or would deal with poor quality custom software that required many man-hours to work around, since labor in these organizations is often less expensive than professional software customization. However, these solutions created frustrating working environments, very inefficient workflows, resentment by users to management and IT staff for providing such poor solutions, and resentment by managers to vendors due to false promises.

Recently we have noted academic units, including our own school and the main library on campus, deciding to spend resources on people and skills, and not on software and software licenses. They look for FLOSS solutions for a computing need, such as digital library software, e-learning software, and content management systems. As long as FLOSS solutions are not significantly worse than COTS solutions, the FLOSS is adopted to solve the computing need, and professional software developers are hired to modify the FLOSS to better fit the needs of the academic unit. Furthermore, these developers are encouraged to join the FLOSS development teams so as to contribute back to the FLOSS community the code modifications, customizations, or improvements that the developers needed to make anyway. This kind of development not only happens on a local level, but is also starting to be embraced by inter-university consortia. In order to refer to this phenomenon more efficiently, we will refer to it henceforth as academic development of FLOSS (ADF).

The reasons we believe ADF to have started occurring are in part the same reasons for patchwork prototyping's occurrence. First, the availability of good quality FLOSS is important. For academic institutions, the FLOSS does not need to be production-scale, but it does need to be designed to evolve into production-scale software. These institutions even embrace imperfect software because they are willing to hire developers to improve the software, and customize it to their needs. However, the decision to adopt FLOSS vs. COTS often hinges on the availability of one or more critical features: the modularity of the software, the ability to create multiple instances of the software from a single management back-end (reducing administrative overhead), the openness of the FLOSS developers to accepting code modifications and new features, and source code which is clean enough and structured in such a manner that modifications are not very difficult to make (e.g.: functionality is generally not hard-coded). The latter feature is particularly important because the developers these institutions hire are often working on 3-5 different projects, and sometimes even more, so the time they can spend on ADF projects is very limited.

The success of ADF also depends on the fact that the developers are bona-fide members of the user communities. The developers may or may not be actual users of a particular software application, but they are embedded in the community that uses it, and thus they are in a position to know what their community needs better than any outside developer can. This also means that the developers are most interested in supporting the needs of their user communities. As a result, not only can the developers represent their academic community's needs to the FLOSS development community, but also their development efforts are geared at serving real user needs.

This process is strengthened further by the way that collecting feedback is built into all of these development efforts. In all of the academic units in which we have observed ADF, the developers are not leading the efforts. Rather, the administration of the units lead the efforts, and collecting feedback on user experiences, soliciting requests for features and functionality, and observing user behavior are explicitly treated as core activities in the development process. Thus there is a feedback loop in place where people try out the new systems, experience using them, and provide feedback based on authentic use.

INTENTIONAL DESIGN FOR CO-DESIGN

The two examples of environments enabling on-going co-design seem to have certain features in common. First, it is important for intended users to have the opportunity to try out technology in their everyday work activities. It is only by using technologies in authentic settings that most users can provide meaningful feedback, and can develop a sense of what is technologically possible.

Second, the collection of feedback facilitated by leaders of the user community is important, (a) because they are in a position of authority where their requests are more likely to be heeded, and (b) because they know the needs and language of their community better than a developer who is primarily in a support role. Thus, they can obtain and translate feedback into a form that is useful for developers.

Third, there needs to be a culture of trust where both designers and managers are seen as making good-faith

efforts to improve the work environment, and enable users in new ways.

Fourth, the technologies used to support co-design must be structured to allow modification and customization. Open source, modularity in design, clean source-code, and other technological features mentioned above are unlikely to be the only factors. Examining how various features of technology and software enable and disable co-design over time is another promising area for future research.

However, these examples also make clear that successful co-design happens only because of a potent sociotechnical cocktail of technological affordances, workplace values, institutional policy, and user trust, and the nature of the ingredients can vary a great deal from environment to environment. Because there is no one "right" mix, searching for "best practices" is as quixotic an endeavor as seeking to create ideal technologies that will always support co-design. Thus, one of the major lessons of the above examples is that we need to study the mechanisms by which the different ingredients of various successful cocktails interact to produce on-going co-design.

Extrapolated Implications for Designing for Co-Design

Thinking critically on our experience and the design literature has also led us to consider ad-hoc design and its implications in general. Most ad-hoc design does not involve much reflection. It is a side project, a distraction from the "real" work that the individual wants to do. Because they see it as a distraction, there are strong pressures for satisficing [11] behavior to occur. Thus, individuals are typically happy with *any* solution that solves their problem, and they often take the first solution they think of or encounter, implement it, and move on. What is missing is a consideration of the consequences of their decision, and the benefits that arise from considering *multiple* solutions—i.e., the benefits that accrue from exploring a design space.

Thus, when designing for co-design, it seems it would be productive to consider ways in which individuals engaging in reactive design can encounter multiple solutions to their problems without intentional effort on their part. This might take the form of a social solution, where recreational discussions about how people have overcome different obstacles in their work are encouraged (i.e., people are encouraged to complain about their problems and how they solved them). Or it might take the form of a sociotechnical solution where many different technologies and policies are developed, and individuals can consult experts (perhaps IT support professionals) about their different options. If individuals routinely encounter creative emergent design solutions in their everyday activities, then it becomes less "work" to consider alternatives before implementing the first thing that occurs to them. Similarly, if individuals are encouraged to complain about how poor decisions made by other individuals in the organization affect their work, a greater awareness of the consequences of their own actions can be fostered.

REFERENCES

1. Battarbee, K., Cabrera, A. B., Mattelmäki, T., Rizzo, F. Designed for Co-designers. *Proceedings of the Participatory Design Conference, PDC 2008*, (Bloomington, IN, Oct. 2008), ACM Press.
2. Büscher, M., Gill, S., Mogensen, P., Shapiro, D. Landscapes of Practice: Bricolage as a Method for Situated Design. Computer Supported Cooperative Work (CSCW), 10, 1 (2001), 1-28.
3. Carroll, J. M., Kellogg, W. A., Rosson, M. B. The Task-Artifact Cycle. In: J. M. Carroll (Ed.) *Designing Interaction: Psychology at the Human-Computer Interface*. Cambridge University Press, 1991.
4. Eglash, R. Appropriating Technology: An Introduction. In R. Eglash, J. Crossiant, G. Di Chiro, and R. Fouché (Eds.) *Appropriating Technology: Vernacular Science and Social Power*. University of Minnesota Press, Minneapolis, MN, 2004.
5. Floyd, I. R., Jones, M. C., Rathi, D., Twidale, M. B. Web Mash-ups and Patchwork Prototyping: User-driven technological innovation with Web 2.0 and Open Source Software. *Proceedings of HICSS 2007* (2007).
6. Gasser, L. The integration of computing and routine work. *ACM Transactions on Information Systems*, 4, 3 (1986), 205-225.
7. Greenbaum, J., Kyng, M. Introduction: Situated Design. In J. Greenbaum and M. Kyng (Eds.) *Design at Work: Cooperative Design of Computer Systems*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1991.
8. Jones, M. C., Floyd, I. R., Twidale, M. B. Patchwork Prototyping with Open-Source Software. In St. Amant, K. and Still, B. (Eds.) *The Handbook of Research on Open Source Software*. Idea Group, Inc., PA, 2007.
9. Levi-Strauss, C. *The Savage Mind*. Translated by George Weidenfeld and Nicolson Ltd. Oxford University Press, Oxford, UK, 1996.
10. Rudd, J., Stern, K., Isensee, S. Low vs. high-fidelity prototyping debate. *Interactions*, 3, 1 (1996), 76-85.
11. Simon, H. A. Invariants of Human Behavior. *Annual Review of Psychology*, 41 (1990), 1-19.
12. Twidale, M.B. Over the shoulder learning: supporting brief informal learning. *Computer Supported Cooperative Work*, 14, 6 (2005), 505-547.
13. Twidale, M.B., Nichols, D.M. Computational Sense: The Role of Technology in the Education of Digital Librarians. To appear in *The Encyclopaedia of Digital Libraries*. Idea Group Inc., 2008.