Reuse Practices

Peter J. Denning

George Mason University
Computer Science Department
Fairfax, VA 22030
pjd@cs.gmu.edu

Invited paper for *Communications of ACM*, February 1997

DRAFT 10/18/96

© *Copyright 1996 Peter J. Denning. You may make one exact personal copy. Further copies or distributions require express permission.*

The Reuse Paradox

The earliest electronic computers half a century ago had subroutine call and return instructions to permit programmers to modularize their codes. Subroutines, procedures, and functions were central features of the first mainstream programming languages (Fortran, Algol, and Lisp) in the late 1950s. Linking loaders made subroutines libraries available as reusable, separately compilable modules in the early 1960s. The inventors of virtual memory argued as early as 1960 that the primary objective of memory systems should be to allow address spaces to be composed of objects that may be shared and reused over and over again. Operating system structures for protecting shared objects were proposed by 1965 and were embodied in a line of object-oriented "capability machines" in the early 1970s. Capability machines were replaced by object-oriented programming and RISC machines in the 1980s. The object-oriented programming environments of the 1990s offer not only reuse of objects but of their specifications through inheritance of type information. Reusing and sharing software modules has been uppermost on the minds of users, programmers, and designers for five decades [5].

How, then, is it today that, although intended to be reusable, most software objects are not reused? If you ask users why they do not reuse, they will often say that object libraries and descriptions of their functions and inheritance

structure appear overwhelmingly complex in the new environment. This can be very perplexing to the designers of the objects, who tried very hard to make them context independent by precisely defining their simple functions. Basili et al [2] cite five additional reasons for paucity of reuse: (1) We reuse not only code but the context from which the code originates; the most relevant part of the context can be found in the experience of the code's users. (2) The reuse of experience is too informal; experience must be analyzed, described precisely, and packaged to facilitate reuse. (4) Most experiences cannot be reused "as is" but must be modified for the new context. (5) Product delivery, not reuse, is the usual mission of a software project. In light of these difficulties, how can we attain our long-sought goal of reusability?

Look at Practices, Not Complexity

Formulating the paradox as a complexity gap between designers' and users' understandings of software modules does not, unfortunately, lead us to an answer: for this formulation does not shed light on where the apparent complexity gap comes from. The complexity gap is in reality the new user's lack of knowledge of the practices of use in the original context. In real life people don't reuse software modules per se, they reuse the practices enabled by the modules. We can reframe reuse to mean the importation of practices (and their supporting tools) from another domain into your domain. You won't do this unless you can produce some benefit in your domain. Spinoza et al use the term "cross-appropriation" for reuse of business practices that produce innovations in their new settings [7]. I like that term and will use it here.

A practice is a habitual pattern of action engaged in routinely by people in a domain, usually without thought. Practices are usually supported by tools and in many cases are impossible without the tools. Typing, a central practice in using computers, is an example. Its essential tool is the keyboard: you cannot type without a keyboard. And not just any keyboard: you will have difficulty with a Dvorak keyboard if you were trained on a Qwerty keyboard. As a physical object, a keyboard has no meaning to someone without the practice of typing: it would appear as an utter mystery to a time traveler from the Middle Ages. You are unlikely to succeed at using a computer today without knowing how to type: as you learn computers you cross-appropriate typing from another domain.

Even as practices provide a context for tools, the domain provides a context for the practices. A community of visually-impaired journalists will have little use for keyboards, displays, and typing and will instead rely on spoken input and output. In most cases, there will be many layers of context and it will never be possible to completely uncover all of them.

Put differently, practices and tools live in a symbiotic relationship: each needs the other, and neither is meaningful without the other. A complex of practices lives in a series of larger contexts which give them meaning and purpose.

Many current discussions of software reusability is often little more than conversations about moving tools and objects without regard to their associated practices and contexts. We would never tolerate this in other domains! If someone proposed to "reuse keyboards", we would immediately ask the proposer to tell us about whether typing makes sense in the intended target domain. We do not ask this very often with software. Perhaps this is a result of our practice of thinking of software as a context-independent function mapping certain input symbols into output symbols: a way of thinking that disinclines designers to examine the context of use.

Besides our inability to see the original context of use, there is another major impediment to cross-appropriation. To the people in the new domain, the other practices may appear as not very relevant to their concerns; those practices may appear marginal and of little value. It may require an act of leadership by the proposer of reuse to accomplish the cross-appropriation. I will return to this point shortly, after discussing how we might observe practices of use in a universal way that would facilitate their reuse.

Coordinative Practices

Let me distinguish between individual and group practices. Typing on a keyboard is an individual practice; we do not normally share our keyboards with others. Cross-appropriating an individual practice can be difficult. Much greater challenges to cross-appropriation arise when the practices involve cooperation and coordination among a group of people.

Coordinative practices are everywhere. We can see them, for example, in the everyday routines of our offices and labs, or in mail-order houses, banking, market transactions, transportation, air traffic control, space flight, and much more. They are often hidden behind what look like tools, especially when the tools are an infrastructure -- have you ever estimated the number of people who must be on their jobs in order that a single telephone call can go through?

Our greatest difficulties in reusing software occur in the settings where the software supports coordinative practices. The more people involved, the harder it is to effect reuse. The first step toward reuse must be a standard way of observing the recurrent actions that constitute coordinative practices. We could draw maps of the recurrences. We could and annotate the maps with

the tools that are needed, explicitly establishing links between practices and objects. The maps would reveal opportunities for cross-appropriation.

The action workflow diagram, invented by Action Technologies in 1989, is a powerful method for mapping coordinative processes. A workflow map exhibits the network of loops enacting the commitments engaged in by a group of people. It is easy to observe when the participants cause state changes in their loops because the state changes are marked by characteristic speech acts. These maps have made it is easy to take direct measurements of cycle times, congestion, and satisfactions among people who are coordinating actions together in a business. Although originally developed as a way to picture business processes, workflow maps are also being used in software design, where they help with the requirements phase of object-oriented designs [4].

It is also easy to annotate workflow maps with information objects that are used as tools during the performance sections of loops. This establishes a direct connection between software objects and the practices they support. Figure 1 shows a standard object-oriented dataflow diagram for a university personnel process and a workflow diagram, and Figure 2 shows a workflow map for the same process. The recurrent practices of mutual commitments made by the participants are clear in the workflow map.

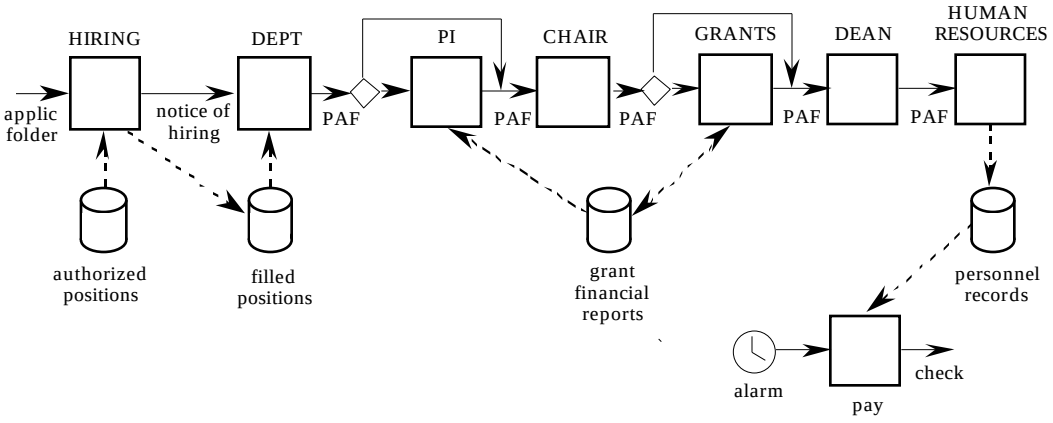

Figure 1. A standard object-oriented dataflow representation of a university personnel process depicts the steps followed from when an individual applies for a job until that individual is on the payroll. It shows a series of eight functions (boxes) that transform input forms and data into output forms and data. The main form here is the personnel action form (PAF). The data are held in databases (cylinders).

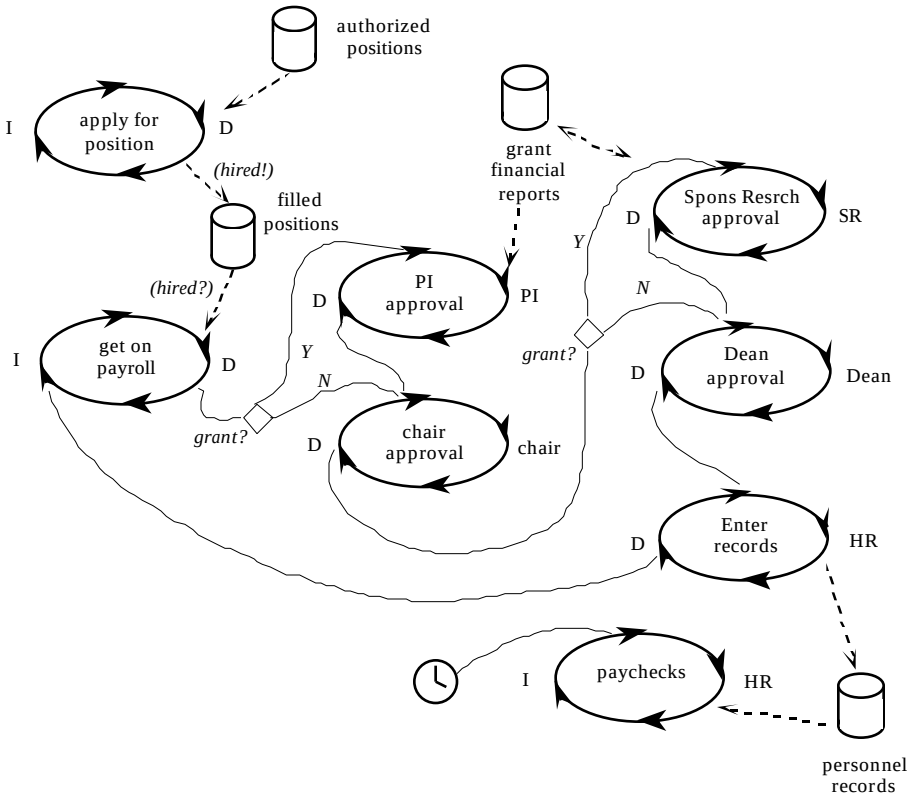

Figure 2. Workflow map of the personnel process consists of eight loops connecting seven roles and four databases. Each loop, which connects a requester and a performer, consists of phases for making a request, negotiating the terms of mutual satisfaction, performing the work, and declaring the task done satisfactorily. The eight functions of the dataflow representation are embedded in the performance phases of these eight loops. The workflow diagram depicts the set of recurrent actions (practices) of the seven people who make this process work.

Examples of Cross-Appropriation

Electronic mail is a splendid example of cross-appropriation. For nearly two decades, e-mail was long a proprietary practice among Internet techies. About ten years ago, business managers began to see that the practice of moving files through e-mail was similar to the business practice of moving documents by courier and fax -- and a lot faster and cheaper. Importing e-mail into the workplace made current business practice more effective.

The World Wide Web is another example of cross-appropriation. The WWW was originally developed to allow physicists to look up each other's papers simply by "clicking on" the text-string citations to them in a document currently displayed on a computer screen. The designers of the Mosaic browser believed that many other people would see such a practice as useful in electronic commerce -- and indeed they were right. The WWW is being absorbed rapidly into business practice.

E-mail and WWW were cross-appropriations from a technological domain into a business domain. I am proposing here that workflow mapping be cross-appropriated in the opposite direction: software designers can adopt a practice from business to help them become more effective at reusing software.

Achieving Cross-Appropriation

It might seem that workflow maps might lead to a simple methodology of software cross-appropriation. You could use a map to ascertain when a practice from another domain is compatible with your own domain and capable of bringing a benefit. The map would then help you guide the reconfiguration of your group's practices and the importation of the software tools you will need.

Alas, matters are not this simple. I mentioned earlier the need for leadership. This is where it comes in. We are talking about getting people to change the way they do business. Accomplishing this requires leadership, persuasion, and salesmanship. If the recipients see no value for them, or if the new practice would conflict with their interests, they will resist it. There is a famous example of this. In the mid 1980s, Action Technologies sold an e-mail program called Coordinator that assisted people in carrying out a single loop in the form of a one-on-one transaction. The Coordinator became moderately popular, with nearly half a million copies installed by 1990. At the conferences on cooperative work, the Coordinator came under criticism from some groups as being "fascist software" [7]. In some workplaces Coordinator was loved and in others hated. The reason for this paradox can be seen in the practices of the workplace. The ones in which timely, rigorous completion of promises was prized were the ones that welcomed Coordinator. The ones in which everyone valued ambiguity about promises and deadlines were the ones that hated Coordinator -- to some of them, Coordinator looked like another surveillance mechanism that management could use against employees.

Thus it is not a foregone conclusion that a new practice can be appropriated into every group. There is no fixed "method" for cross-appropriation.

Although we can develop some rules of thumb and guidelines, we can no more automate cross-appropriation than salesmanship. It is a skill that we will have to learn and practice.

Conclusion

My suggestion to think of software reuse as cross-appropriation of practices accomplished by acts of leadership accords with some current lines of investigation in software design that have not yet made the mainstream. One is Vic Basili's "experience factory" for setting up processes to observe and transport experience along with the software tools when moving them to a new domain [2]. The other is James Coplien's "software use pattern", which is a set of relationships among entities that solves a problem in a context [3]; this work derives from the work of the architect and urban planner, Christopher Alexander [1]. Both these lines buck the conventional inclination of software engineers to think of their programs as context independent mathematical functions. Their approaches describe patterns and say what is important about them in context.

The title of this paper can be interpreted in two ways: an observer's report "on the practices of reusing software" or the activist's exhortation to "reuse practices" rather than software objects. I hope to have planted the suggestion that you can be the activist. You can use workflow maps to understand the complex of people, practices, and objects that must be understood before you can cross-appropriate the practice and its tools into a new domain.

Readings

1. C. Alexander. 1979. *The Timeless Way of Building*. Oxford Press.

2. V. Basili, G. Caldiera, and D. Rombach. 1994. "The experience factory." In *Encyclopedia of Software Engineering*. Wiley. See also the publications list of the Experimental Software Engineering Group at U. Maryland, http://www.cs.umd.edu/projects/SoftEng/ESEG.

3. J. Coplien and D. Schmidt. 1995. *Pattern Languages of Program Design*, Addison-Wesley.

4. P. Denning and P. Dargan. 1996. "Action Centered Design." In *Bringing Design to Software* (T. Winograd, ed.), Addison-Wesley.

5. C. V. Ramamoorthy and W-T Tsai. 1996. "Advances in Software Engineering." IEEE Computer (October), 47-58.

6. J. Seymour. 1998. “The Coordinator”. In *PC/Computing* (October), 82-83.

7. C. Spinoza, F. Flores, and H. Dreyfus. 1995. “Disclosing new worlds: entrepreneurship, democratic action, and the cultivation of solidarity.” Inquiry 38 (June), 3-64.