A DISCIPLINE OF SOFTWARE ARCHITECTURE

Peter J. Denning* and Pamela A. Dargan**

*CS Department, George Mason University, Fairfax, VA 22030 USA, pjd@cs.gmu.edu

**Mitre Corporation, McLean, VA 22102 USA, dargan@pacific.mitre.org

Abstract: Neither software engineering nor software design qualifies as a discipline, but software architecture might. Action-centered design is proposed as the means to build software architecture by joining the two.

Key Words: Software engineering, software design, software architecture, action-centered design, ontological mapping, workflow.

The software landscape is a mixed field of successes and failures. Most noticeable among the successes are software packages that turn personal computers into document preparers, spreadsheets, databases, animators, image displays, drawing tools, financial accountants, tax preparers, legal advisors, encyclopedias, video games, electronic mailers, network browsers, work exchangers, and much more.

The failures are disturbing by their number and spectacularity. Peter Neumann has compiled a list of the 1500 most disastrous failures of software systems since 1972; the list grew by 200 cases in 1993 (Neumann 93). Among the most widely known on that list are the aborted NASA space mission in 1975 due to a deadlock in the distributed control system, the collapse of the ARPANET in 1980 due to failure of a load control algorithm, and the shutdown of a major segment of the AT&T switching network in 1990 due to a synchronization bug in the routing software. A 1979 GAO review of nine software projects for the Department of Defense showed that about 2% of the dollars were spent on software that was delivered and in use, about a quarter on software that was never delivered, and about half on software that was delivered but never used (Charette 89). Similar problems persist today (Newport 86).

Software engineering is characterized as a set of formalisms, methods, and practices for producing "reesstmts" -- reliable, economical, efficient software systems that meet their specifications. The persistent inability to achieve this goal has led experts to say that software engineering is not a discipline, and that the word "engineering" even conveys the false impression that a discipline exists. The designers of computers recognized early on that producing "reesstmts" would be a challenge: the term "software crisis" was already being used in the 1960s and inspired the birth of software engineering in 1968 (Tichy 92). A long series of innovations including modularity, structured programming, chief programmer teams, dataflow diagrams, information hiding, data abstraction, program verifiers, syntax-directed editors, programming environments, structured languages, software-parts reusability, visual programming, object-oriented programming and design, catalogs of design paradigms, CASE tools, and, most recently, megaprogramming, have produced only modest gains but have not transformed software engineering into a

discipline for producing "reesstmts". The persistence of the software systems crisis over such a long period leads inescapably to questioning the foundations of software engineering: for the field as currently formulated is incapable of producing the results it seeks (Andriole 93).

Software design is characterized as a set of practices and implementation techniques that allow the construction of marketable software systems that provide form and function satisfying to users. Software designers tend to think of their work as a craft best learned through apprenticeship and the design process as a close participation with users. They do not claim to have a discipline. They dispute the claim of software engineers that design is subsumed in software engineering. They say that they do not use software engineering methods in their successful products, that their articles have not been welcome in the software engineering literature, and that they have founded such groups as ACM SIGCHI and the Association for Software Design to pursue investigations of design not fashionable among software engineers (Kapor 91, Winograd 96).

Software engineers and software designers share a common interest in developing a discipline whose practitioners can produce "reesstmts" systematically. They disagree on how to accomplish this. Will it require more formalism? More practice? More emphasis on systems? More emphasis on people?

What is a discipline? To be called a discipline, a field must satisfy three criteria: (a) it has a set of formalisms, methods, models, and processes that are used regularly to produce results, (b) it has a set of standard practices including generally accepted community standards for levels of competent performance and criteria for excellence, and (c) its practitioners can reliably fulfill standard promises to the satisfaction of clients. The third criterion is the most important and least discussed. The question is not whether formalisms and practices are properly balanced, but what formalisms and practices are necessary so that practitioners can keep their promises to clients. By these criteria, neither software engineering nor software design qualifies as a discipline.

Our main claim here is that the missing discipline is neither software engineering nor software design, but software architecture. It could be constructed by joining software design and software engineering, following architecture of buildings as a model. The resulting field would grow into a discipline within a few years and could resolve the software crisis.

The second claim here is that an ontology of design is needed to bring the two fields together; neither has such now. (An ontology is a conceptual framework for interpreting the world in terms of actions.) Software engineers see design as a process of gathering requirements into a formal specification and then organizing the software production process, and the software, to meet specifications efficiently and economically. Software designers see design as a collection of practices that can only be verbalized as aphorisms and rules of thumb; a design is satisfactory only when a client says it is, an assessment based in part on meeting specifications and in part on factors that defy engineering measurement, such as form, style, esthetics, effective completion of work, freedom from negative surprises, anticipation of future cases, and serving organizational and political interests.

We propose an interpretation of design that is the beginning of an ontology. It is focused on the satisfied client and not just the specification-meeting system. It is grounded in a language-action perspective rather than a systems perspective (Erickson 93). The new interpretation observes concerns, breakdowns, standard practices, institutions, and recurring actions and produces means to connect those observations with software structures. The skill of doing this is here called *action-centered design* (Denning and Dargan 96).

Design as Deriving a System from Specifications

The dictionary lists no less than ten different senses of the verb "design". The primary meaning is "to conceive or execute a plan." Engineers use the term "design" in a specialized way that is consistent with this general definition.

Engineers say that design is the dominant paradigm of engineering. They intend "design" to describe a staged process that begins with a problem statement, constructs a statement of requirements and then specifications of a system that solves the problem, builds a prototype system, and tests it to demonstrate that it meets the specifications within given cost constraints. Like other engineers, software engineers have used the engineering design process to construct software systems; in this case they call the process the software life cycle model, the most common varieties of which are the waterfall model

and the spiral model (Boehm 76, Boehm 87). The waterfall model was proposed for systems whose requirements were fully understood in advance. The spiral model is a more recent extension that accounts for iterations and refinements of the system until it converges on one acceptable to the customer. Unlike other engineers, software engineers cannot claim success for their instance of the engineering design process. They have not developed a systematic method of producing "reesstmts".

Many explanations have been proposed to explain this anomaly. The most popular ones center on the notion that software is "complex" and violates the continuity laws that we are accustomed to in engineering -- changing of a single bit in a program can produce drastic changes of behavior. (Parnas 85) Software is seen as a "radical novelty" for which our customary engineering analogies are misleading. (Dijkstra 89) These explanations may relieve some anxiety, but none has produced a practical means of systematically producing usable, dependable software. In fact, they suggest that no practical means are possible.

The explanation of the anomaly can be found in the engineering design process, which was evolved to systematically build tangible products. Software is not tangible and becomes intricately intertwined in the smallest details of human communication and work. The three basic assumptions of engineering design are:

1. The result of the design is a product (artifact, machine, or system).

2. The product is derived from specifications; once the specifications have been agreed between customer and designer, there is little need for contact between the two parties.

3. The customer is assumed to be satisfied if the acceptance tests demonstrate that the system meets the specifications.

Many software engineers believe that, with enough knowledge and computing power, the derivation process could in principle be mechanized. The software crisis is usually seen as a breakdown in methodology, especially for obtaining "correct" specifications. The popularity of CASE (computer-aided software engineering) tools emphasizes this (Yourdon 92), as does KBSE (Knowledge-Based Software Engineering)(Lively 89).

We claim that the crisis is more usefully approached as a breakdown of customer satisfaction. The standard engineering process offers little to connect the actions of designers with the concerns of customers; the interaction between them is limited to the requirements and specifications documents (Denning 92a).

Design as Identifying and Taking Care of Concerns

Led by Europeans, a new school has emerged called "human-centered design" that seeks redress for the shortcomings of "product-centered" design. (Floyd 87, CACM 93) Don Norman says that the traditional design process focuses entirely on the machine and its efficiency, and it expects humans to adapt (Norman 93). The human-centered process, he says, leaves to humans the actions humans are good at -- empathizing, perceiving, understanding the nature of a problem, inventing solutions, listening for concerns, fulfilling commitments, satisfying others, serving interests -- and letting machines do what humans are not good at such as performing repetitive actions without error, searching large data sets, or performing movements that are beyond human capability.

Human-centered design attempts to understand the domain of work or play in which the user is acting, and to design the computers to facilitate action there. Anthropologists have played significant roles in this effort. Human-centered design operates from three assumptions:

1. The result of the design is a satisfied user.

2. The process of design is a collaboration between designers and customers; it produces a specification as a by-product.

3. The customer is not assumed to be satisfied until he or she declares satisfaction.

Human-centered design may produce good designers through apprenticeship, but not yet through systematic education and training. In their criticisms of software engineering design, some proponents of human-centered design go too far toward another extreme: they forget that customers still will not be satisfied with software systems that are too costly, do not function as performed, are too inefficient, or are too hard to maintain.

Reframing Design as Mapping

The two approaches to design just outlined do not compete. Both can be combined to yield an interpretation of design that can be the basis of a discipline of software architecture.

Terry Winograd of Stanford University and David Garlan and Mary Shaw of Carnegie-Mellon University have been among the first to propose architecture (of buildings) as a paradigm for the missing discipline (Winograd 91, Garlan et al 92). Architecture satisfies the criteria for a discipline: it has formalisms and methods for analyzing and constructing structures from given materials; it has standard practices such as blueprints, styles of buildings, building codes, and means of interacting with building contractors; it has community standards for the levels of competence that architects can have; and its practitioners can make and fulfill promises to deliver plans and buildings on time and within budget. Schools of architecture teach aspiring architects with combinations of presentations, practice with formalisms, and projects of increasing sophistication carried out under the guidance of an experienced architect.

One of the architect's principal products is a map of the building -- the blueprint. Working from sketches and interviews, the architect must come to understand the interests of the client including form, function, social relations, styles of those who will use the building, efficiency, and affordability. This understanding is manifested in the blueprint. The same blueprint conveys sufficient information that the building contractor can calculate the time, materials, and costs to assemble the structure. It is used later to assess the building after people move in, and to reconfigure portions of it if they demand changes.

At present, the fields of software design and engineering have no method of mapping analogous to the blueprint. Software designers, who correspond roughly with architects, have no common language to coordinate with software engineers, who correspond with building contractors. The sketches and drawings used by designers are not easily connected to specifications of software structures; the formal specifications are generally unintelligible to clients.

The key to transforming software design and engineering into a single discipline is the development of a method of mapping that would be simultaneously intelligible to clients, designers, and engineers. The function of a map is to enable the parties to navigate the territory depicted by the map and to coordinate their actions. What would such a map include?

Since a customer makes assessment of satisfaction relative to the domain of action in which the software system has been deployed, it is reasonable to investigate maps of domains of action. The phrase “domain of action” is general and includes not only work, but entertainment, law, career, education, finance, family, sociability, membership, health, world, and even dignity and spirituality. Software designers who frame design as “a process of building software whose form and function satisfies users,” and engineers who frame design as “a process of deriving a software system from specifications,” must reframe design as “a process of assembling computing technologies to enable members of a community to more effectively carry out the standard practices of their domain and to invent new practices.”

Such a reframing would align software with the practices of a community, enable radical forms of extensibility, permit reconfiguration and evolution with the needs and concerns of a community, and permit productivity of knowledge work to increase.

What Led Us to this Claim

In the spring of 1992 we undertook to investigate why so many engineers were interested in object oriented design (OOD) as a way to approach the requirements analysis stage of the engineering design process, and object oriented programming (OOP) as an approach to the implementation. The former approach (OOD) focuses on the objects and functions that will be needed, and the latter (OOP) on their implementation as data structures and procedures. We asked: Are these approaches leading us to a discipline of design?

To get a preliminary answer, we posed a more concrete question to a number of prominent software engineers: “What is the concern that drives people to find object-oriented programming so attractive?” Most see object-oriented programming as another fad, not a “silver bullet” that will end the software crisis. Most think that object-oriented programming appeals to intuitive senses about how the software will be used and thereby reduces the apparent complexity of the software. The sidebar quotes them.

THE APPEAL OF OBJECT ORIENTED PROGRAMMING

We asked several prominent computer scientists, "What is the fundamental concern that drives so many people to find object oriented programming appealing?"

Les Belady: "The desire to construct programs more out of components than out of statements."

Jim Horning: "OOP is a technology that has been gaining acceptance over the last 25 years (since Simula-67). Properly used, it is another tool in our arsenal for attacking complexity. So are modularity, abstraction, specification, etc. Just because none of them solves the entire problem doesn't mean that they aren't useful."

Jean Sammet: "I have been in this field a long time. I have seen numerous 'silver bullets' which were going to be the savior of the software field. Examples include structured programming, chief programmer teams, and numerous specific languages (e.g., COBOL, PL/I, Ada). There are lots of other examples of alleged 'silver bullets' ... Obviously none of these purported 'silver bullets' served in that role, or you wouldn't need to ask the question."

Bill Wulf: "In retrospect it's hard to believe, but there was comparable hype over the use of higher level languages, elimination of the 'goto', top-down design, etc. None of them was THE silver bullet, but the best of each has become woven into the fabric of the discipline. Taking the long view, a central problem of CS/SE [Computer Science / Software Engineering] is managing complexity, and certainly abstraction is a key tool for doing that. That aspect of OOP will certainly persist, but whether in the particular form seen in C++, for example, is problematical. Based on history, one would expect not -- simply because once the issues that OOP addresses are under control another set of problems will appear to be the most important and another type of mechanism will be needed."

Peter Wegner: "Object-oriented models determine abstractions that specify the domain-dependent behavior of real-world objects by their applicable operations. They determine the behavior of objects in a domain of discourse without any commitment to the execution of a particular task. Data is structured into encapsulated chunks whose behavior is determined by the operations that can act on the data. The focus on nouns rather than verbs as the basis for describing a domain of discourse makes it fundamentally declarative in the sense that it describes properties of a modeled world rather than particular computational processes."

Fred Brooks: "My answer, set forth in detail in 'No Silver Bullet', is that the next radical step forward in programming has to involve shifting the conceptual level at which one thinks and writes programs. The fundamental hope for OOP is that it permits one to move up a level by building a set of objects for the domain of discourse of the particular application area, hence allowing one to program within the intellectual context and using the elementary concepts of the particular application area. Most of today's programming languages are general-purpose, and work a level below this."

We then asked, What does it mean for a design to be intuitive and to be judged as complexity-reducing? To obtain preliminary answers, we interviewed the designers of several award-winning software packages:

- Quicken (by Intuit), a personal financial management system;
- MeetingMaker (by On Technologies) and Synchronize (by Crosswind Technologies), systems for scheduling meetings and managing calendars in groups;
- Topic (by Verity), a system for retrieval from texts; and
- Macintosh user interface (by Apple).

We asked them what they had done to achieve a good design. There was a surprising level of unanimity in their answers:

- Pick a domain in which many people are involved and which is a constant source of breakdowns for them. (The designer of Quicken chose personal finance.)
- Study the actions of people in that domain, especially repetitive actions. What do people complain about most? What new actions would they like to perform next? (In personal finance, the repetitive actions

include writing checks and balancing the checkbook; the most common complaints include errors in arithmetic and discrepancies between bank statements and personal balance sheets; the most-sought new action is to automatically generate reports for income tax purposes.)

- Choose software functions that are obvious to anyone already practicing in the domain -- there is little to learn to get started because the software merely helps perform familiar actions. Include functions that permit actions most have wished they could do but could not do manually. (In personal finance, this includes paying by filling out screen images of checks, editing a facsimile of the checkbook register, labeling register entries by income-tax type, and reconciling bank statement with register.)

- Deploy prototypes in selected customer domains; see how people react and what kinds of breakdowns they experience.

- Because the customers frequently shift concerns, especially after they are seasoned users, it is necessary to continue observing and to take the observations into account in the next version of the design. Thus there are many beta-test sites, individual follow-up, hot lines, highly attentive technical and customer support, suggestion boxes, bug advisories, and the like. It is of central importance to stay in communication with customers.

- All the designers said they did not pay much attention to software engineering methodology. Several said that the internal structure of their code is ugly and is not well modularized. When fixing bugs they make patches; when the system gets too patchy, they declare the next version and reengineer it completely.

One way to summarize these findings is this: Software systems have customers. Quality means customer satisfaction. Customers are more likely to be satisfied by software that is transparent in their domain of work because it allows them to perform familiar actions without distraction and allows them to perform new actions that previously they could only speculate about.

Customer satisfaction is not static: customers change expectations and the software they use must be rapidly reconfigurable to track their shifting expectations.

Larger Systems

The software packages that we examined are "small systems"; much of the software crisis concerns "large systems". So we asked, What large systems exist that are widely acclaimed to be of high quality? We turned to the list of winners of ACM software systems awards:

1983	Unix
1984	Xerox Alto
1985	VisiCalc
1986	TeX
1987	Smalltalk
1988	System R and Ingres
1989	PostScript
1990	NLS
1991	TCP/IP
1992	Interlisp
1993	Sketchpad
1994	Remote Procedure Call
1995	World Wide Web and Mosaic

By examining the published papers of the authors of these systems, we came to similar conclusions. Each was immersed in the concerns of their customers, especially the major breakdowns those customers faced. Each provided a system that facilitated familiar actions and resolved the breakdowns. Each allowed new actions that could only previously be speculated about. And each was designed to be rapidly changeable as the concerns and interests of customers shifted.

Ontological Mapping as a Basis of a Discipline of Architecture

A discipline of software architecture would be capable of training its practitioners to systematically fulfill promises to build and install software systems that are judged useful and dependable by their customers. To accomplish this we need ultimately to

1. Understand the linguistic structure common to all domains of action, so that one can observe a particular domain for its basic distinctions, repetitive processes, standard practices, standards of assessment, strategies, and driving concerns.

2. Know how to connect the linguistic structure of the domain to software structures that will assist in carrying out standard actions and know how to coordinate the implementation of those structures.

3. Know how to assess the satisfaction of people in the domain so that their effectiveness in the presence of the software can be measured.

Mapping is a powerful metaphor that connects these three elements. It is not difficult to see what is needed from maps of the domains in which software will be used. The same map should depict the domain structure (step 1), guide the software implementation (step 2), and help assess the outcome (step 3). The domain actors use it to navigate as they work (step 1); the software producers use it to navigate as they configure software systems (step 2); and managers or others it to guide their assessments (step 3).

Erickson characterizes this way of observing a community as the "language-action perspective" and contrasts it with the familiar systems perspective (Erickson 93). The systems perspective interprets a community as a network of functions that receive input and produce output. The language-action perspective regards a community as people who plan and initiate action in their language together; action consists in making and fulfilling commitments. Action cannot be completed if there is a breakdown in communication, and thus the central concern of a designer is to provide computers that avoid or resolve breakdowns. The language/action perspective does not discard formalisms, but does look beyond their limitations (Denning 91).

The software architect must know the "cartography" of software blueprints in general and must be able to develop specific blueprints for each new system. Winograd and Flores discussed the framework in which design can take place (Winograd 87). The common elements of every domain of action are:

- A set of *linguistic distinctions* (verbs, nouns, jargon, etc) around which people in the domain organize their actions.

- A set of *speech acts* -- statements whose utterance initiate actions and signify completions.

- A set of *standard practices* -- recurrent actions, organizational processes, roles, and standards of assessment -- in which members of the domain engage.

- A set of *tools and equipment* people use to perform actions. A tool is "ready-to-hand" if the person using it does so with skill and without distraction.

- Possible *breakdowns* -- interruptions of progress caused by tools breaking, people failing to complete agreements, external circumstances, and the like.

- A set of *ongoing concerns* of the people in the domain, for example their common mission, interests, or fears.

- The history and institutions of the domain.

They call the framework outlined above the "ontology" of the domain. Notice the concern for action in this ontology. Building on this terminology, we propose "action-centered design" as the name of the skill that a software architect must master. The architect's skill of creating a blueprint and using it to coordinate builders and later assessments is an example of this skill. Action-centered design can be learned and is the basis of a discipline of software architecture.

The designers of the prize-winning software packages and systems had all become observers of these phenomena in the particular domains in which they were designing.

The ontological map of a domain consists of two parts. One is the "dictionary" that indicates specifically the constitution of each domain element (Winograd 87). Examples of these ontological dictionaries for personal finance (served by the program Quicken by Intuit) and drawing (served by the program KidPix by Brøderbund) are shown in the sidebars. The other part of the map is a diagram of the recurring "workflow loops" of the domain -- these are the processes by which various people, acting as customers and performers, make requests, reach agreements, produce results, and declare satisfaction. (Denning 92, Medina-Mora et al 92. Denning and Medina-Mora 95) Workflow maps convey information about incomplete tasks that cannot be seen from pure information-structure diagrams.

Some of the tasks that make up workflows store, retrieve, consume, or manipulate information. By identifying the exact points in each loop where information-processing actions occur, the map-maker establishes a connection between workflows and information processes. In other

ONTOLOGICAL DICTIONARY FOR QUICKEN

Quicken is a popular program that assists in the management of one's personal finances. The "ontology" of this domain is as follows:

- The linguistic distinctions include checks, checkbooks, ledgers, paying agents, bank accounts, bank statements, budgets, and tax forms.
- The speech acts include writing a check, depositing a payment, balancing the ledger, preparing a budget, and preparing a tax return.
- The standard practices include check writing, depositing and withdrawing from bank accounts, paying bills, balancing the books, filling out tax forms, preparing tax reports, entrusting funds to a bank.
- The institutions include banks, payment agents, merchants, and the IRS.
- The tools and equipment include checks, checkbooks, ledgers, and data management systems.
- Possible breakdowns include improperly filled-out checks, errors in the checkbook, late payments, missed deposits, negative cash flows, and inability to prepare an IRS report.
- The set of ongoing concerns includes financial solvency, timeliness of payments, creditworthiness, payment of taxes.

words, workflow maps connect the domain's ontology to software structures. In effect, the points of interaction can be formalized as functions applied to information objects -- exactly the point of object-oriented design.

Workflows that are performed by software processes may encounter points at which the customer must make choices -- for example, among possible requests in the first phase of a loop, among possible conditions of satisfaction in the second, among possible performance strategies in the third, and among possible degrees of satisfaction in the fourth. Menus and dialog boxes are the means to present these choices act on the one selected.

To engineers it might seem strange that scientific principles can be deployed to develop systematic competence at producing customer satisfaction. In his recent book, Peter Drucker discusses how Frederick Taylor in the early 1900s averted a class war by using scientific principles to make manufacturing work more productive, thereby improving the lot of each worker (Drucker 93). Drucker says something similar must happen with knowledge work.

Although he suggests that Taylor's principles might constitute an obsolete common sense, Drucker says the principles were still operative as recently as World War II. At the onset of World War II, Hitler assessed (correctly) that Germany but not the US, had great expertise in optics, needed for accurate bombsights, and (incorrectly) that it would take the US five years to build the same expertise. This assessment led Hitler to declare war on the US, a deadly miscalculation: Six months later, having applied Taylor's principles to train workers how to assemble optical systems, the US was producing bombsights the equal of Germany's. Such a discipline is possible for software production.

What We Have Done Here

We have proposed here that a new skill, which we call "action-centered design", is the basis of a discipline of software architecture. This skill consists of observing the ontology of a domain, constructing its ontological dictionary, and constructing a workflow map of its recurring standard processes. The map can be used by the software architect to review with the client how

the system will satisfy each concern, and to coordinate the implementation of the information and software structure with the software engineers. It can be used later to assess and reconfigure the system. The one map can be used by the designer, the implementor, the assessor, and the maintainer. Evidence of this skill can be seen in the examples of award-winning software packages and systems. The field of architecture illustrates that the skill can be learned.

We propose this interpretation not as a final answer, but as a preliminary step. We offer it as an opening for a new direction in the investigation of software design.

ONTOLOGICAL DICTIONARY FOR KIDPIX

KidPix is a popular program used by children to draw pictures and to paint on the computer screen. Some of the functions behave in fanciful ways, such as those that erase the screen. The "ontology" of this domain is:

- The *linguistic distinctions* include paper, pens, brushes, erasers, paint cans, sprayers; lines, curves, boxes, ellipses, arrows, shading, colors, fillings, and borders.
- The *speech acts* include drawing lines, boxes, ellipses, etc.; painting or shading a region, erasing, pointing and dragging.
- The *standard practices and institutions* include doodling, preparing presentations, amateur artwork, entertainment.
- The *tools and equipment* include brushes, paint, canvas, paper, the personal computer, mouse, and printer.
- Possible *breakdowns* include hard-to-use software, deterministic and predictable behavior of brushes, pens, inks, etc.
- The set of *ongoing concerns* includes entertainment, esthetics, art others admire, explorations of artistic methods.

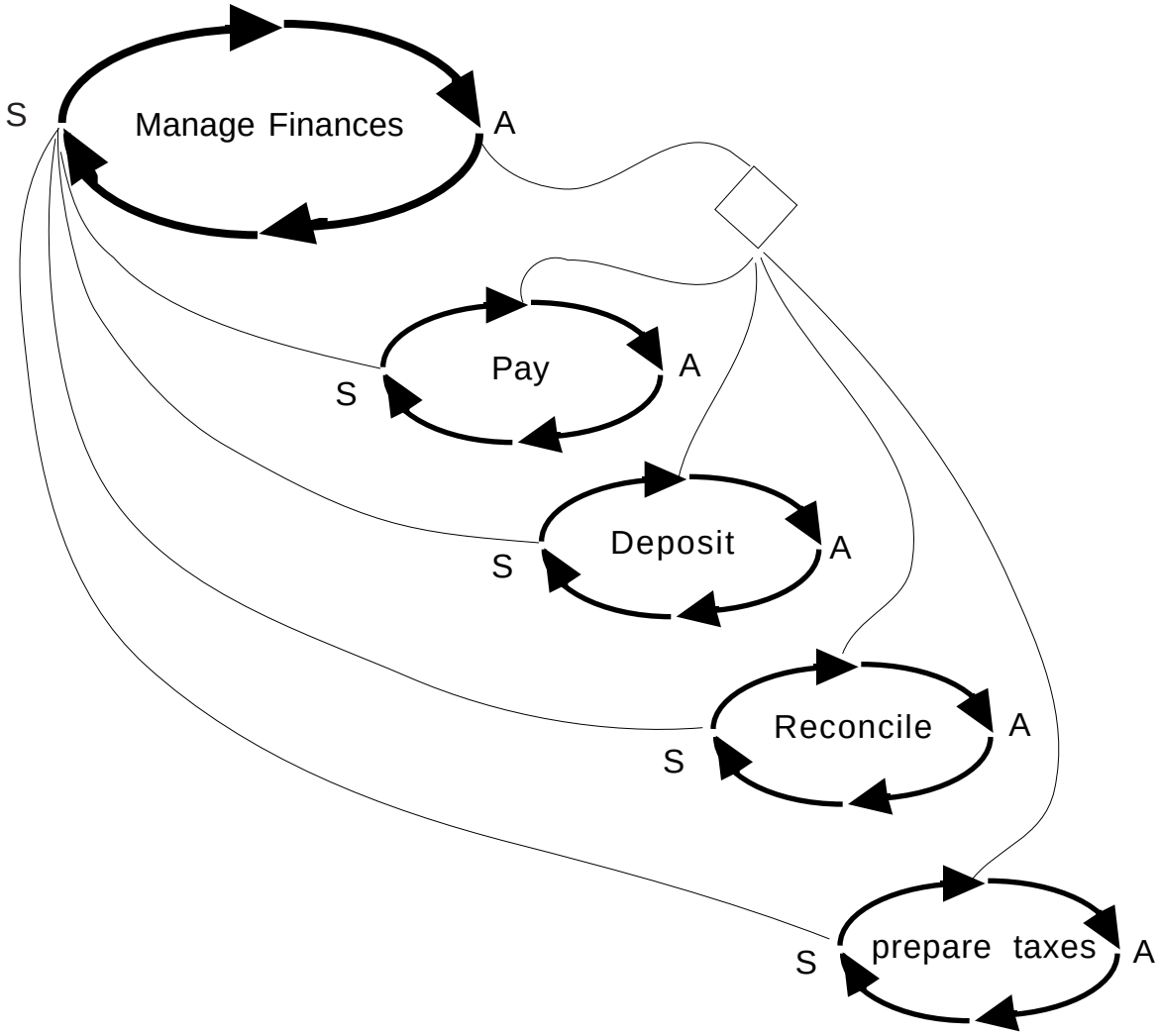

Figure 1. The action workflows of the domain of personal finance depict the repetitive processes in which people engage. The main loop (top) shows oneself (S) interacting with a financial advisor (A). The loop connecting them consists of four stages denoted by arrows: request, agreement, performance, and satisfaction. For each interaction with the financial manager, S can make only one of four requests, shown as secondary workflows initiated from the performance stage of the main loop. Through these workflows, S and A act on several datasets including the checkbook ledger, the bank statement, and the budget.

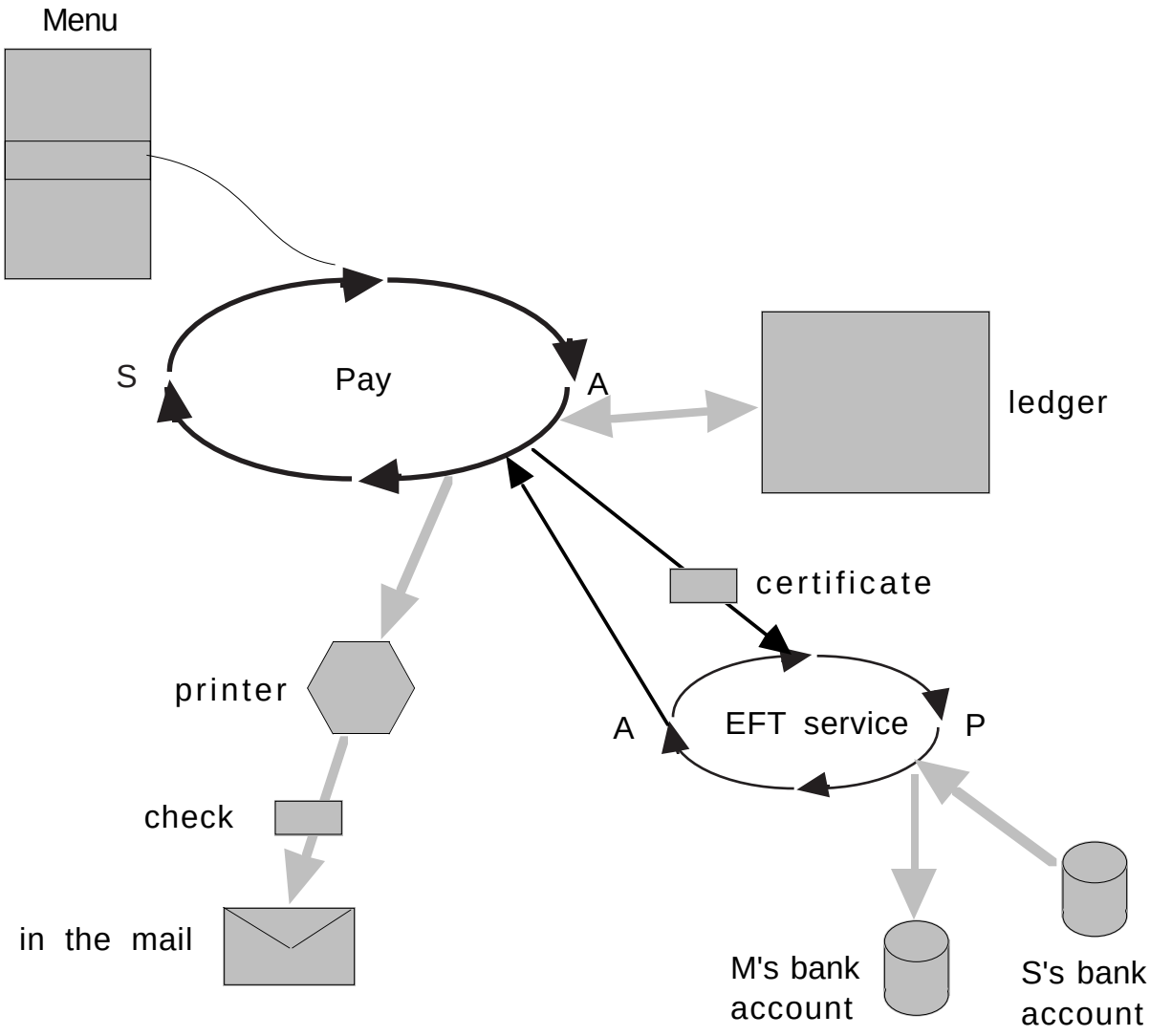

Figure 2. A more detailed view of one of the payments workflow shows the various data objects acted on by S and A. The selection of this loop occurred in the choice box of Figure 1 (diamond), represented here as a menu item. Electronic payments and regular checks are two options available in the performance phase. Electronic payments involve a request on a tertiary workflow, the electronic funds transfer service.

Readings

Stephen J. Andriole and Peter A. Freeman. 1993. "Software systems engineering: the case for a new discipline." IEEE *Software Engineering Journal* (May). 165-179.

Barry Boehm. 1976. "Software engineering." IEEE *Transactions on Computers C-25*, 12 (December). 1226-1241.

Barry Boehm. 1987. "A spiral model of software development and enchancement." IEEE *Computer* (September). 43-57.

Fred Brooks. 1987. "No silver bullet." IEEE *Computer* (April).

CACM. 1993. Special issue on human-centered design. *Communications of ACM 36*, 6 (June).

R. H. Charette. 1989. *Software Engineering and Risk Analysis Management*. McGraw-Hill Intertex.

Peter Denning. 1991. "Beyond Formalism". *American Scientist 79*, 1 (January-February). 8-10.

Peter Denning. 1992. "What is software quality?" *Communications of ACM 35*, 1 (January).

Peter Denning. 1992. "Work is a closed loop process." *American Scientist*. July-August. 314-317.

Peter Denning and Raúl Medina-Mora. 1995. "Completing the loops." ORSA/TIMS *Interfaces 25*, 3 (May-June), 42-47.

Peter Denning and Pamela Dargan. 1996. "Action-centered design." In *Bringing Design to Software* (T. Winograd, ed.), Addison-Wesley.

Edsger Dijkstra. 1989. "On the cruelty of really teaching computer science." *Communications of ACM 32*, 12 (December). 1398-1404.

Peter Drucker. 1993. *Post Capitalist Society*. Harper Business.

J. David Erickson. 1993. "Beyond systems: better understanding the user's world." *Computer Language 10*, 3 (March). 51-56.

Christiane Floyd. 1987. "Outline of a paradigm change in Software Engineering." In *Computers and Democracy: A Scandinavian Challenge* (Bjerknes, Ehn, Kyng, Eds.) Gower Press.

David Garlan, Mary Shaw, Chris Okaskaki, Curtis Scott, and Roy Songer. 1992. "Experience with a course on architectures for software systems." *Software Engineering Education: Lecture Notes in Computer Science* (C. Sledge, Ed.). Springer-Verlag.

Mitchell Kapor. 1991. "A software design manifesto: Time for a change." *Dr. Dobb's Journal*. January.

William M. Lively. 1989. "Where AI/KB techniques fit into software development/engineering." *Proc. 13th Annual Int'l Computer Software and Applications Conf*. IEEE Press. 777ff.

Raúl Medina-Mora, Terry Winograd, Rodrigo Flores, and Fernando Flores. "The Action Workflow approach to workflow management technology." *Proc. ACM CSCW Conference*, November 1992.

J. P. Newport. 1986. "A growing gap in software." *Fortune 113* (April 28). 132-142.

Peter Neumann. 1995. *Computer Related Risks.* Addison-Wesley & ACM Press.

Donald Norman. 1993. "Human centered design". *Technology Review 96*, 5 (July). 47-53.

Donald Norman. 1993. *Things That Make Us Smart*. Addison-Wesley.

David Parnas. 1985. "Software aspects of strategic defense systems." *Communications of ACM 28*, 12 (December). 1326-1335.

Walter Tichy, Nico Habermann, Lutz Prechett. 1992. "Future directions in Software Engineering". ACM SIGSOFT *Software Engineering News 18*, 1 (January). 35-43.

Terry Winograd and Fernando Flores. 1987. *Understanding Computers and Cognition*. Addison-Wesley.

Terry Winograd. 1991. "Introduction to the project on people, computers, and design." Center for the Study of Language and Information, Stanford University, Report CSLI-91-150PCD-1 (April).

Terry Winograd (Editor). 1996. *Bringing Design to Software*. Addison-Wesley.

Edward Yourdon. 1992. "Whither or wither?" *American Programmer* (November). 16-27.