8 Support for Collaboration

This chapter examines collaboration with respect to design rationale. On the one hand, this is a discussion of how collaboration can support the development, codification, and use of design rationale. On the other hand, it is a discussion of how rationale supports collaboration in design and development

8.1 Introduction

8.1.1 General

People work together in software design and development because they wish to accomplish projects that are too large and complex for a single person. Although this is the fundamental basis for collaboration in all human endeavors, it is not always a simple matter of adding team members to tackle ever-greater challenges. Indeed, one of the classics of software engineering, Brooks' *Mythical Man-Month* (1975), took its title from the mistaken notion that software team productivity scales linearly with the number of team members. Brooks analyzed his own experience managing the development of the IBM Operating System 360 software, a project in which he concluded that the addition of team members eventually *reduced* productivity.

8.1.2 Objectives of This Chapter

This chapter surveys the relationship between collaboration and design rationale. First, it observes that software development is almost always collaborative, for the simple reason that most software projects are too big for solitary individuals ever to successfully tackle. This raises a set of specific challenges: collaboration aggregates individual efforts, but it also creates new sources of work for people in teams, and new risks for the products of teamwork. We then consider how collaboration supports rationale in software development—by encouraging team members to explicitly articulate their goals and plans, and therefore to create the possibility of a discussion about reasons, and by supporting a culture of software development to conventionalize and leverage social mechanisms like anthropomorphic metaphors and software patterns. Finally, we consider how rationale supports collaboration in software development—by supporting awareness of how the project is meaningful to one's collaborators, and coordination among collaborators, especially with respect to making progress in uncertainty.

8.2 Software Development as Collaborative Work

8.2.1 Collaboration Is Inescapable

The most basic driver for collaborative work is the human ambition to tackle large and complex projects: Brooks estimated that Operating System 360 took 5000 person-years. Quite simply, there is just too much work to do in many projects for one person to ever be able to carry them out. But the issue is more than one of mere additions.

Collaboration is well integrated into human psychology and sociology. For example, groups of people generate more ideas and higher-quality ideas than disaggregated individuals. People with different skills and experience often experience *synergies* in collaboration; that is, together people can develop solutions that no one of them could have conceived of or executed individually (Kelly and Littman 2001).

During its brief history, software engineering has developed as a pervasively collaborative work practice. Developing a substantial software system requires many specialized skills. The tasks of system development—requirements identification and analysis, architectural specification, software design, implementation, testing—involve a great diversity of skills. Individual software professionals cannot be expert in all or even most of these skills. Indeed, software professionals typically devote a significant fraction of their professional effort to keeping up to date with just one or two of these professional skill sets.

The tasks of software development are at least partially decomposable, as suggested—perhaps a bit optimistically—by traditional waterfall models of system development. Thus, modern software development regularly involves divisions of labor and coordination of specialized contributions. This entails fairly elaborate and articulated specializations in software project management, in addition to the primary skills of software development.

Furthermore, labor economics and the worldwide distribution of skills have produced a global distributed paradigm for software development. Today, many systems are developed by collections of technical teams scattered throughout the world, each providing some specific capabilities, and sometimes having little or no direct contact other teams. Such far-flung projects were unprecedented only a few years ago, and still constitute an area of intense innovation in collaborative work.

In this context, the example of Operating System 360 begins to appear an unrealistically *simple* case: the OS 360 software only had to run on one hardware configuration, and was developed by a colocated team; most of the designers and developers worked in direct physical proximity.

8.2.2 Collaboration Entrains Challenges

The notion of a man-month—or person-month—is mythical in the sense that adding people to a project does not enhance the total effort linearly. The basic reason for this is that *collaboration itself is work*. Two people chopping down a tree must share their plans and coordinate their efforts just to survive, let alone to experience a productivity boost. This sharing and coordination diverts and subtracts time and effort from the primary task. Thus, the tree may be cut down faster than either person alone could do it, but it is never cut down twice as fast.

The challenge of collaborative work is considerably greater than suggested by the tree-cutting example. When people work in groups, they tend to work less hard than they do when working as individuals—a phenomenon called "social loafing" (Karau and Williams 1993). Social loafing is especially prevalent when people perceive that their contribution to a collective outcome is not unique, that someone else could do the work just as well, or when they believe that their loafing will not be evident to their co-workers.

When people pool their ideas, when they collectively brainstorm and develop new ideas, they tend to adjust their contributions toward positions taken by others they perceive to be competent or powerful, or toward existing majority opinions—the status quo. This tendency to conform undermines the extent to which collaborative intellectual activity can generate more and better ideas, and over time causes groups to become more homogenous and less effective (Latane and Bourgeois 2001).

However, diversity in groups also entails collaborative challenges. People with different technical backgrounds commonly have different fundamental values and beliefs; they can find it difficult to appreciate one another's contributions, or even understand what is being contributed (Pelled et al. 1999). Thus, diversity in collaborative groups frequently leads to conflicts, often very deep, value-based conflicts.

Phenomena like social loafing and conformity/conflict have significant derivative effects on group dynamics. Derogatory terms like "slacker" and "overachiever" reveal the tensions that can be created in a group over social loafing. Effective group performance requires a foundation of common ground, that is, shared knowledge about local context, conventions, and co-reference to enable efficient and reliable interactions. Sustained group performance requires the development of trust and generalized reciprocity, sometimes called *social capital* (Coleman 1990).

Many of the challenges of collaboration are inherent tradeoffs; they can be addressed, and perhaps balanced, but not solved *tout court*. For example, designating a "coordinator" to receive and direct all group communications can improve group problem solving efficiency, but decreases satisfaction with the group activity (Leavitt 1951). Similarly, including a "skeptic" in brainstorming allows groups to produce more and better ideas, but also decreases members' satisfaction with the group activity (Connolly et al. 1990).

These collaborative challenges are as old as human organizations, but they are exacerbated by the very nature of knowledge work like software design and development. In knowledge work the interim work products, sometimes even the final work products, can be quite insubstantial; they are plans and strategies, architectures, algorithms, and heuristics. The products of knowledge work are also typically arcane; indeed, software systems are possibly the best example there is of this.

8.3 Collaboration Supports Rationale

Collaboration is an important social resource for design rationale. The collaborative interactions of various software professionals ineluctably and naturally externalize rationales, though often incompletely. Collaborative interactions in software development also shape the software development process in ways that favor rationale.

8.3.1 Collaboration Externalizes Rationales

The creation of design rationale is often conceived of as a documentation activity within the software development lifecycle, and it is certainly true that design rationale can be a kind of documentation. Incorporating rationale into formal documentation activities is useful and efficient, since rationale provides causal foundation for other categories of documentation such as final specifications, reference, and maintenance manuals, and user documentation like online help and tutorials.

However, rationale is more broadly the reasoning that occurs throughout design and development, *whenever and however it is codified and used*. One of the most important consequences of collaborative work is that co-workers must articulate and externalize knowledge, assumptions, and reasoning that otherwise might remain tacit. If you watch one programmer at work, you would most likely get little insight into programming. The work activity is mostly mental, and the occasional external inscriptions that are produced are quite arcane, but if you watch *two* programmers collaborating, you see quite a lot about programming. More specifically, you see quite a lot of rationale.

Software development is a complex, intellectual task in which there are never singularly correct solutions. More typically, there are many satisfactory solutions, each entailing a variety of partially understood tradeoffs and side-effects. Elsewhere in this book we have characterized these problems as wicked (Rittel and Weber 1973) or ill-structured (Reitman 1965). When people work on this kind of task collaboratively there is lots to talk about, indeed, lots to analyze, justify, and debate.

As Kraut (2003) put it, this kind of collaborative work follows a “trust-supported” heuristic in which group performance can be only as good as the second-best member. Groups pool and weigh different perspectives; they identify and repair errors in candidate solutions and the rationale for candidate solutions. Producing a solution requires both the technical enterprise of identifying and developing a proposal, but also the social enterprise of convincing one’s colleagues.

An old chestnut of software engineering is that no-one wants either to produce or to use documentation. But in collaborative contexts, in which one must obtain the support of colleagues in order to make a technical decision, there is no shortage of design rationale. Indeed, the culture of software development work has evolved a variety of mechanisms to capture, preserve, and discuss these materials, such as commenting and literate programming (Knuth 1992), bug reports and frequently asked questions (FAQ) forums, and indeed the entire spectrum of Usenet communities. Collaboration in software development unavoidably and voluminously generates rationale.

8.3.2 Software Development Communities of Practice

Software development is diverse and somewhat fragmented as a profession, but it is a profession that is all about the skills and practices of constructing software. Software professionals have developed a *culture* of software development—communication and work practices to coordinate work and to teach and coach one another (Curtis et al. 1988; DeMarco and Lister 1999; Lammers 1988). For example, software developers frequently talk about software components and their interactions in explicitly anthropomorphic terms; thus, a component is said to *know* things—such as how to put a file on the print queue—or to *expect* things from other components (e.g., Herbsleb 1999; Madsen 1994). In this sense, software development is a *community of practice* (Lave and Wenger 1991).

One could regard the cultural practices of software developers as curiosities, but in fact social practices emerge, evolve, and persist because they add something to human activity. Thus, it seems prudent to consider how the ways software professionals talk about and construct software—particularly those work practices that are *not* taught in formal education or encouraged by industry standards, corporate policies, or managerial directives—may reveal important characteristics about how experts think about software, and how they coordinate software development work.

In this light, consider the issue of anthropomorphic and other metaphorical language. Formal education and normative practices in software engineering have traditionally placed high value on explicit and correct representations such as specification languages, programming languages, and a variety of diagrams. Notably these formal representations are pretty much strictly declarative; they describe the structures and interactions in a software design and implementation. Classic articles on computer science education by Dijsktra (1989), among others, have specifically argued against metaphorical language.

Why then would software developers employ anthropomorphic and other metaphorical language? Carroll and Mack (1985) argued that metaphorical representations clarify new domains by leveraging concepts that are already known, while at the same time highlighting mismatches in the mapping of old-to-new, and thereby flagging conceptual problems that need attention. Rosson and Alpert (1990) suggested that the anthropomorphic metaphors of object-oriented design facilitate upstream communication among developers by reducing the need for explicit point-by-point clarification and refinement entailed by more explicit representations. For example, saying that software component A knows about software component B is both succinct and rich. It conveys that the behavior of A depends in some way on the behavior of B, and that the specific nature of the contingency is either not yet known or not needed for present purposes.

Herbsleb (1999) elaborated this conjecture by noting that the strategy of anthropomorphic representation allows software developers to leverage "naïve psychology" (Clark 1987)—the near-universal understandings that humans share about animate entities. Naïve psychology allows people to reason what an animate entity must have known to have acted as it did, or what it is trying to do given its behavior and knowledge. In other words, it bundles declarative understanding of what is happening with direct perception of its rationale—that is, *how* an entity is able to do what it does and *why*. It is believed that naïve psychology capabilities were selected in evolution because individuals who could draw these inferences were better able to succeed in the early social world (Clark 1987).

Herbsleb (1999) analyzed a corpus of 1800 system behavior descriptions identified in a series of software engineering domain analyses. The domain analyses involved teams of 3–5 experts analyzing message-passing protocols in telephony or switch maintenance software. Herbsleb found that 70% of the behavior descriptions were metaphorical. Each domain analysis involved a series of meetings; for each series, Herbsleb analyzed one early meeting, one meeting from the middle of the series, and one of the final meetings. He found that, through the course of the three domain analysis meetings, teams of software engineers came to rely *increasingly* on certain of these metaphorical descriptions—the ones derived from naïve psychology. That is, metaphors were not used as ephemeral ice-breakers, to replaced with more proper and explicit descriptions. Instead, they became established in the domain analysis as a sort of local technical language for the teams.

Communities of practice are social mechanisms for the codification and social transmission of practices and their rationales. Collaborative software development work requires sharing extraordinarily complex information fluently. Software development has evolved as a community of practice to leverage naïve psychology via anthropomorphic metaphors, selectively hiding and emphasizing information, while bundling description and rationale. Another example of this in contemporary software practice is pattern languages (Gamma et al. 1995). Both simplify and speed communication in software collaborations by leveraging rationale.

8.4 Rationale Supports Collaboration

The relationship between rationale and collaboration is reciprocal. The role of rationale in software development is motivated and facilitated by collaborations among professionals, but rationale also supports collaboration. It provides a compelling management tool for keeping projects on track spanning time, distance, and organizational change. It facilitates awareness of one's team members, contributing to the development of common ground and trust, and it facilitates coordination, particularly in project contexts of high uncertainty.

8.4.1 Awareness

In order to collaborate effectively in a large and complex project one needs to know many things about one's collaborators (Carroll et al. 2006): Who are they? What do they want to do? What are they doing now? What tools are they using? To what other resources do they have access? Who do they work with? What are they thinking about? What do they know? What do they expect? What are they planning to do in the near future? What sorts of significant relevant experiences have they had in the past? What disciplinary biases and assumptions do they bring to this interaction? What do they value? What criteria will they use to evaluate joint outcomes? How is their view of the shared plan and the work accomplished evolving over time?

This may seem like a long list, but in fact it is quite incomplete. Consider the issue of coordinating nuances in vocabulary. A user interface designer and a software architect may both support prioritizing design elegance; they may even be able to talk at length about how and why this objective is important. But in practice, they may have entirely different notions of what elegance is. If goals are not adequately analyzed and codified, this kind of failure of common ground can quickly lead to conflict, putting the collaboration and the project outcome at risk.

Of course the mere fact that different professional perspectives differ with respect to technical concepts and skills, values and priorities, and so forth is *not* the problem. Indeed, such differences are required for a successful large-scale software project, or any other large-scale human endeavor. Professional diversity can be, should be, and often is a resource to a software development team. The challenge is to efficiently recognize and effectively manage these differences.

This is where rationale can help. To the extent that people share concepts, skills, values and priorities, they can more easily create and develop common ground and trust. This is the essence of a *community of practice* (Lave and Wenger 1991), as discussed earlier in this chapter. When team members *do not* share disciplinary concepts, skills, values and priorities - as in the example of the user interface designer and the software architect discussing *elegance*, they need to construct common ground socially by exchanging perspectives and attaining mutual understanding. (Analogous points could be made for social structures other than discipline and community of practice, such as culture and ethnicity.) Members of a software development team can construct common ground by sharing their goals and visions for a project, their ideas about how to turn these into plans and actions, what they most value, and what they think they can contribute to the project.

A great variety of groupware tools are being developed, deployed, and investigated to provide awareness support in collaborative work, for example, tools for online discussion about, or direct annotation of project objects, various activity visualizations, personal profiles and social networks, and activity integrators (Carroll et al. 2006). All of these tools help to codify bits of rationale; many have the effect of making personal rationales more permanently accessible to other team members, or more closely integrated with project data objects. They help people share more of their reasoning and their reasons with one another, and that helps them collaborate more effectively.

8.4.2 Coordination

In their empirical study of collaborative coordination in large software organizations, Kraut and Streeter (1995) found that *informal discussion among team members* was both the most valued and the most used coordination technique among the 18 coordination techniques they studied. Curiously, they also found that members of the software teams they studied valued informal interaction more than they actually engaged in it. More generally, Kraut and Streeter found that less formal coordination mechanisms—such as group meetings, discussions with one's manager, requirement reviews, design reviews, and customer testing—mechanisms that bring to light diverse viewpoints, were judged as valuable given the extent to which they were used, whereas more formal coordination mechanisms—like status reviews, code inspections, CASE tools, data dictionaries, milestone schedules, and source code—were judged as not valuable relative to the extent to which they were used.

In this investigation, the rated importance of informal and social coordination mechanisms in large software projects was strongest during periods of high uncertainty, such as in requirements and early design. In other words, the easy exchange of rationale, facilitated by less formal coordination mechanisms such as meetings and discussions, was critical to collaboration among software developers in high uncertainty, upstream stages.

Kraut and Streeter also noted a somewhat alarming tendency for projects to increasingly *de-emphasize* informal interaction through the course of development. Managers tended to prefer formal coordination mechanisms, and to shift towards these when possible.

Kraut and Streeter concluded that an important potential advantage in software management would be to devise better tools and techniques to enhance informal and interpersonal communication among team members throughout the development process. They noted that many of the most prominent and celebrated techniques in software engineering, such as formal specification languages, are designed to minimize interpersonal communication. These may satisfy a manager's desire for formal coordination mechanisms, but they do not facilitate the easy exchange of rationale and were rated by developers as valueless relative to their use.

8.5 Summary and Conclusions

This chapter surveys the relationship between collaboration and design rationale. First, it observed that software development is almost always collaborative work, for the simple reason that most software projects are too big for solitary individuals ever to successfully tackle. This raises a set of specific challenges: collaboration aggregates individual efforts, but it also creates new sources of work for people in teams, and new risks for the products of team work. We then considered how collaboration supports rationale in software development: by encouraging team members to explicitly articulate their goals and plans, and therefore to create the possibility of a discussion about reasons, and by supporting a culture of software development to conventionalize and leverage social mechanisms such as anthropomorphic metaphors and software patterns. Finally, we consider how rationale supports collaboration in software development—by supporting awareness of how the project is meaningful to one's collaborators, and coordination among collaborators, especially with respect to making progress in uncertainty.