JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2002**

Vol. 1, No. 1, May-June 2002

Collaborative Software Engineering

Adele Goldberg, Neometron, Inc.

1 INTRODUCTION

Story telling is an old and revered activity, often intended to educate while entertaining. Practical work on scenario analysis, or scenario planning, is a form of story telling. In a business setting, it is a process by which multiple interests can contribute to constructing views of multiple futures in a form that can contribute to corporate strategizing. Scenario analysis is a conversational process. Participants in the conversation, both inside and outside the organization, offer diverse points of view that affect the characterization of the business climate and likely future events. The conversation takes the form of several stories leading to immediate action or long-term learning in preparation for future action. The scenario analysis approach accepts the proposition that there is not one right answer, but rather a need to prepare the organization to mobilize its resources towards innovation.

The software engineering community shares an interest with the larger business world in seeking effective ways to create such learning organizations, specifically to create new ways to capture project team experience and then to parlay that experience to improve best practice advice and harvest reusable knowledge. The requirement to produce stories can be used as a way to design new online team collaboration methods, notably methods that capture the information needed to generate experience stories in a way that helps engineers reuse knowledge from the project. Basically, these interests seek to answer the question: How might we capture actual software engineering goals and events in a non-intrusive way, so as to tell an interesting and informative story?

2 STORIES

Some people are good story tellers. I am not. I grew up with an eidetic memory. Anything I learned was stored literally in the form of a picture. I could pass math exams by turning the pages of the textbook in my mind, copying the theorems line by line. I still rely on "seeing" in order to think. The result is that I see wholes as though they materialized as-is, rather than as parts with a process that brings them together. It made it easy for me to pass exams; it makes it hard for me to share my visions. I found that telling stories

Cite this column as follows: Adele Goldberg: *Collaborative Software Engineering*, in Journal of Object Technology, vol. 1, no. 1, May-June 2002, pages 1-19, http://www.jot.fm/issues/issue_2002_05/column1

helped, leaving it to the listeners to ask questions and build their own version of the vision.

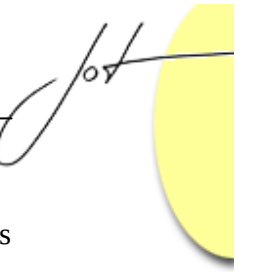
Let me start by telling you three short stories that share a common theme.

Kevin's Story. Once upon a time there was a very bright media production manager who was uncertain of his ability to understand the functioning of a very rich software system. Two more technically inclined colleagues sat with Kevin one morning to walk together through the system. Sometimes Kevin would say, "see, here is how we set up a way for the users to talk with one another." Sometimes David would say, "see, that part needs tweaking—we will find the code and make some changes." One time David said, "can't be done. We have a problem." But Gene said, "sure it can…I remember being told that we could pool sets of questions and generate tests." Back and forth, the three debated. While Gene read the manual and made suggestions, Kevin searched the system, until David yelled, "there it is." Job completed in time for Kevin to give the whole company a demo that same day.

Green and Red Cards. With 2000 participants with questions to ask the management team, a corporate user conference was about to end with a difficult open Q&A session—difficult because this company's managers often side step the hard questions. But Joe the session leader decided to reuse a process followed at a conference the previous year. Each participant entering the session room was handed one green card and one red card. Whenever anyone spoke, a green card held up meant "my question is being answered." A red card meant "keep talking, I am not satisfied that my question is being answered." Questions were collected on paper and organized by Joe before the session started. Joe read a question and then invited a management team member to the stage to give the answer—standing up, mike in hand, nowhere to hide. As the answer was given, the participants held up their green and red cards. The manager kept talking until only green cards could be seen. Sometimes a lone red card remained; its holder would be called to explain his continued discontent. And then the manager would proceed. Although not all answers were the ones the people asking wanted to hear, everyone left satisfied that at least they got an answer to all the questions they asked.

Chewing Gum Timers. Two hundred people were getting together to talk about a difficult problem. The meeting organizer was concerned that some people would dominate the meeting. So he tried a new approach. As each person came to the meeting, he or she was handed 10 pieces of chewing gum. Each piece of gum was worth ten seconds of speech. Whenever someone wanted to talk, he or she could do so as long as there were enough pieces of chewing gum. Every ten seconds, a bell rang. Sometimes, people gave the speaker gum so the comment could continue. Sometimes people gave a good speaker gum so he or she could make a new comment. Quickly, the cost of a speech converged to one piece of gum—ten seconds of interesting sound bite.

Each of these stories tells you something about a process in which several individuals come together to interact: to share their know-how, to teach one another, to learn something new. In each case, the group dynamic has properties that do not exist in any



one individual. Looking for such stories helps us uncover and then exchange techniques to build similarly effective group dynamics.

Telling good stories is, of course, often hard. A good story teller needs both a picture of the whole and the ability to build the picture for the reader, composing parts and the threads that tie them together in a manner that captures the reader's attention, or at least instills reasonable belief. A good story teller involves the reader in each part of the story, whether the part is about the development of an engaging character, the unfolding of a complex structure of events, or the intrigues born of the relationships among the characters who interact to create those events.

I know the conventions of story telling, but as a reader, not a writer. Each character's depiction has to make sense; it has to be able to stand alone—a consistent representation of a person and their role in the story. The thread of events must contribute to drawing the landscape or to targeting outcomes, with each event plugged in at the right place and involving a logical subset of the characters. A relationship is formed so that information or artifacts can be exchanged—a teacher of a whiz kid student, a long lost half sister with the right blood type, a spy. An event might cause a consequence, perhaps an artifact that sits around during the remainder of the story and impacts some later event—maybe a conversation left on a tape, a diary or daily calendar, or a scientific discovery that saves the life of the heroine 3000 miles away. The character's roles hint at likely participation in events, and the relations suggest how the story might unfold. The choice of story elements has to make sense; the elements have to fit together for the story to be understood. Looked at independently, the elements might themselves be mini-stories. When added together, they might create an epic, having properties that differ from the properties of any one element.

Good stories form pictures, and pictures are useful aids in understanding and remembering complex materials. Whether a book is a history, mathematics text, or computer science text—the better the story is painted, the easier I can create a picture in my mind, and remember it. It may be unusual to think of a textbook as a story, but I regularly test the understandability of a textbook (or a technical manual or a paper) by seeing if the table of contents outlines a sequence of actions or information exchanges that build upon one another and lead to a conclusion. I see if this story plan represents something from which I could learn.

A good software system is like a good story. There is much to recommend the analogy. Both software system and story have roles, with attributes and responsibilities—a system's users, a story's characters. They both have a framework—a system's architecture, a story's threads of events. They both have parts or elements contributing to the whole—a system's modules, a story's events. The parts can be appreciated when isolated from the whole, but they also need to fit together and, in doing so, subordinate themselves to the whole. The meaning of any event in a story derives from the story's whole pattern; each occurrence of an event depends on those that precede and are anticipated by those that follow.

This is, in fact, how we characterize good software systems.

Both system and story are written. The analogy, however, breaks down when we consider the respective writers. The writer of a story is often an individual, whereas the writer of a system is typically a group of individuals, a team. Perhaps it is the medium and not the story itself that should be the basis for comparison. A story for a movie is written by a group, and so movies may offer a better analogy. Textbooks are also often written by a group, with contributions at a variety of different levels.

Let's now suppose that the story is about the discovery of the algorithm—a result of decades of contributors in mathematics. Or the story is a country's history—clearly created by many, usually working in parallel. Of course, I am taking some license here, muddying *what* a story is about with *who* documents a history, who tells the story of that history. But imagine that we could write the history in an automatic way. Then who really wrote the story? I would say that either the historical participants themselves wrote the story, or the engineers who wrote the software that wrote the story by capturing and recasting the historical events—they wrote the story. Such software organizes the historical content according to some programmed rules for retrieval and replay. The rules can then generate stories when the required content is collected and represented in an accessible way.

The ability to write a story derived from real life actions and ideas—mathematics, history, biography, invention, system building—relies on two skills:

- The first skill is the ability to retell the sequence of events in a manner whose pattern is recognizable, so that elements of the story telling can be anticipated or remembered in the context of the pattern. For example, the retelling might present the process followed, the milestones met, the surprises, and even the reversals. The recognizable pattern is part of the picture that helps readers like me understand and remember.
- The second skill is the ability to lead the reader to the desired conclusions, such as the lessons learned or the reusable knowledge conveyed. By connecting a lesson learned to a story, the lesson is recognized when a context similar to that of the story is encountered.

As system writers, we try to share experience, but we do not always find it easy to do so in a way that is sufficiently supportive of knowledge reuse. We extract an artifact for reuse (code, documentation, or process), often losing the situation that had simultaneously created the artifact and made it effective. A good story writer might give birth to the two independently—an artifact and a framework or context for its use. Combining the two creates new properties. The way the two interact, placing the artifact in context and allowing the context to embrace the artifact, is an important aspect of the story writing.

Just as the theory and practice of algorithms is the result of a process of discovery and analysis, every software system is the result of a process of design and implementation. The story of that process is clearly written by the team members who adopted it, evolved it as the needs of the system building unfolded, and interacted with one another in the context of that process. Our ability to generate such stories depends

either on our skills as journalistic harvesters—a labor intensive task—or it depends on our foresight which motivates us to capture the software engineers' output as the work proceeds (process, knowledge creation and usage, decision making, and so on), in a manner that can be electronically retrieved and replayed.

The opportunity to automate story retelling is now finally becoming available to us. As more and more people work together online, in the context of a software engineering community, special community tools become natural extensions of other processes. We need to focus the design of these online communities to include features that garner the context useful to retelling project stories, because it is often easier to understand new ideas when they are in story form. [1]

[1] The notion that stories are a good format for making ideas and facts easier to remember has taken various forms over the years. Peter Schwartz's *The Art of the Long View* [Schwartz, 1991] offers story telling as a way to create an objective view of the future based on potentially conflicting assumptions. This is the scenario planning referred to in the opening of this paper. See also [van der Heijden, 1996]. An article in *Computerworld* [Anthes, 2000] notes the use of stories as case studies as a part of the U.S. Navy's knowledge management effort.

3 COLLABORATION AND DISTRIBUTED COGNITION

I want to practice a kind of software engineering that builds a story about each experience, one that, in the sharing, creates reusable knowledge that promotes collaboration. I want to do this both for the team *in situ*—that is, while the project is ongoing—as well as for use later, in the team's debriefing or by teams on future projects. Further, these stories can be the documentation of reusable artifacts, so that a candidate re-user can learn the role of an artifact in a previous project by reading an experience story.

Clearly, story telling contributes to reuse. Why also emphasize collaboration? Because software engineering is inherently a collaborative social practice. Building a complex system—one with lots of parts and lots of relations—requires actions and has properties we associate with collaboration. These include: distribution of labor, shared vocabulary by which such distribution is organized and managed, and shared awareness about tasks and tools and skills. Collaboration in software engineering is itself complex. It deals with how work is pushed down to individuals, how results are gathered and combined, how tools are used individually and collectively, how groups use tools to coordinate and cooperate, and how technology resources and communications channels are organized. It is also about how each team member maintains active awareness of his or her own work and work context. And how this active awareness contributes to shared learning.

The design of an online *project community* for software engineering—which is what we have come to call these Web-based interactions—relies on understanding the rules of participation and the rules of interaction that support the collaborative nature of the engineering team. At any point in team work, three factors affect the structure of the work and what actually transpires: conduct of the activity, development of the practitioner, and development of the practice.

- **Conduct of the activity** is how the activity of software engineering is actually practiced. To tell this part of the story, there has to be some model or framework that holds the parts together and allows us to look at isolated events in the context of how those events came about and then how they contributed to the ongoing story. For example, we might define a model for software engineering projects, with ingredients such as:
  - Process frameworks: ones that succeed and ones that fail
  - Roles, and how they determine levels of participation in team activities
  - Tools used by individuals in isolation that create artifacts for the team
  - Tools used by the team that coordinate contributions of individuals

We expect this model to describe:

- How information is shared, whether the information is about project process or subject matter expertise, and

- How ongoing work on one project can be connected to and influenced by work on another project.

The model should also include: rules that define who has access to what aspects of the work, and the structure within which participants can interact. These are the rules of participation and interaction mentioned earlier.

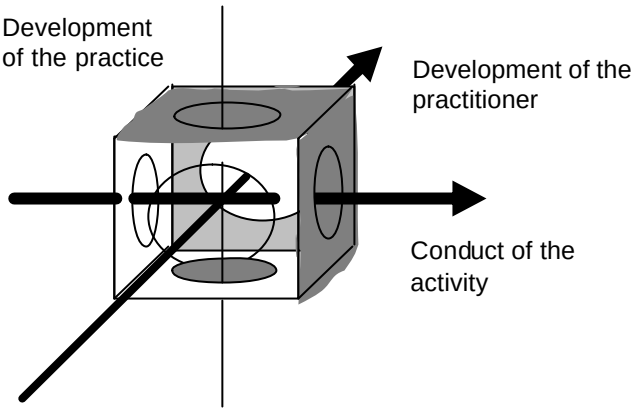
- **Development of the practitioner** is how the individual software engineer develops skills and know-how, answering questions such as:
  - What form does collaboration need to take to create learning opportunities?
  - How do we enable an individual software engineer to reflect on his or her actual practice?
- **Development of the practice** is how the rules for participation and interaction in a single software engineering project evolve, and then further evolve when used in multiple projects. The elements of this aspect of the story answer questions such as:
  - What influences change, specifically, change in a project model in terms of objectives, processes, knowledge flow, or roles?
  - How do teams self organize to set up workable communications channels?
  - What style(s) of communication best serves the team given location, frequency to exchange issues or ideas, and nature of negotiation or decision-making?

To tell this story, we start with an explicit model that can be customized by team members. We then track the changes to that model over time, eventually using the customizations to produce alternative models to share.

Edwin Hutchins identifies these same three factors in his book C*ognition in the Wild* [Hutchins, 1995]. Hutchins used a graphical representation to show that these factors combine at any moment in human practice (Figure 1[2]). He uses his observations of the practice of ship navigation to argue the case for a cognitive science that studies *cultural cognition*. Cultural cognition is the affect that the context of work has on the ability to do the work, for example, how the spatial layout of people and tools and accessible communications channels affect how people are able to contribute to the work.

[2][2] Hutchins uses the length and boldness of the arrows to depict rate of change and density of interaction of the elements, respectively.

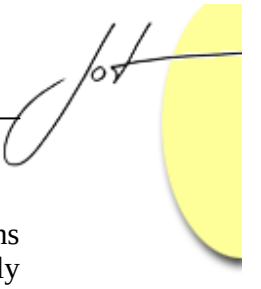
Figure 1.
A moment of human practice.
From Hutchins,
C*ognition in the Wild,* p. 372.



Hutchins' introduced the term and practice of *distributed cognition*. Distributed cognition is a simple idea: Work is accomplished through an awareness of and actions taken because of the fact that the knowledge (or skills) needed to complete an activity—whether it be a computation, a document, or a complete software system—resides in multiple places (people, sources) –*and* (here the "and" is crucial to the point)—and the recognition that the knowledge resides in multiple places is known by all of these places. That is, know-how is distributed and so is the awareness that know-how is distributed. According to Hutchins:

> *"All divisions of labor, whether the labor is physical or cognitive in nature, require distributed cognition in order to coordinate the activities of the participants… When the labor that is distributed is cognitive labor, the system (of work) involves the distribution of two kinds of cognitive labor: the cognition that is the task and the cognition that governs the coordination of the elements of the task." [Hutchins, 1995, p. 176]*

Software engineering is a cognitive labor that produces several tangible artifacts: the system itself and its documentation. The work requires practitioners to know how to program, how to use programming tools, and how to write about programs, as well as how to coordinate the production of the programs and the use of tools by multiple engineers. The conduct of each activity proceeds, as Hutchins summarizes in his explanation about distributed cognition, "…by the operation of functional systems that bring representational media into coordination with one another." [Hutchins, 1995, p. 372] In actual software engineering projects, there is a continual process of gathering or creating information in one representational media that is then propagated across media, for example, text of requirements specifications into design diagrams into expressions of a programming language, and further into completed forms for requirements change requests. This propagation across media is one way to observe the requirement for software engineers to collaborate in order to carry out their shared work.



According to the rubric of distributed cognition, to enhance or extend the collaborations needed for successful software engineering, an online community should simultaneously support:

- The tasks for a single individual;
- The single individual's coordinated use of a set of tools;
- The interactions of a group of individuals with one another, that accomplish the work, and evolve the way the work is practiced, or evolve the skills of the practitioners; and
- The interactions of a group of individuals with a set of tools.

I use the word "tools" here to refer to both manipulators (such as text editors, compilers, debuggers, version controllers, or change managers) and visualizers /presenters (of plans, work flow and work status, instruction manuals, code libraries, and informational content). These tools, and the form in which they are presented to the users of the online community, represent the conduct of the activity of software engineering. In an online community, tools take a specialized form and are accessed through a user interface that directs the user's tool choice—often controlling when tools are available and who can share the results of tool use.

Collaborative software engineering requires that each individual community member be a software engineer, be a part of a software engineering team, be a collaborator, and be in a collaborative environment. All four usages are needed.

4 COLLABORATION AND SOFTWARE ENGINEERING

To understand collaboration, we need to consider it in all of its senses: conceptual, practical, and educational.

> *Conceptual collaboration* is sharing information and work, and sharing responsibility and leadership.
>
> *Practical collaboration* is the decomposition of the work, the integration of work results, and the management coordination given differences in expertise with the practice.
>
> *Educational collaboration* is helping one another learn on the job and learn on demand, with activities that support reflecting on the lessons learned from practical experiences.

When the context for conceptual and practical collaboration is structured to encourage educational collaboration, then everyone has an opportunity to learn through observing others (especially others who are more skilled in some aspects of the work). Everyone has an opportunity to adapt the work context itself in order to facilitate both the conduct of the activities and the points at which learning can take place. It typically takes longer to learn how to do a task than it does to do it. Learning while doing both introduces the less

skilled practitioner as an earlier contributor to the work, and also allows learning to take place in the context of ongoing changes to how the work is done. No software engineer wants to spend four years of college studying, for example, how to program in a language or style that is obsolete before he or she enters real world practice. Rather, this engineer needs to study the concepts of software construction, and, simultaneously, to explore practical construction in a variety of forms, developing an ability to apply concepts to new forms as they appear.

The challenge to an online project community then is to combine, not separate, the three kinds of collaboration. A successful online project community is a place in which the practical gives immediate access to the conceptual. And it is a place where on-the-job mentoring, discussions about decisions and tradeoffs, and knowledge reuse, all together, form learning opportunities. They become educational collaborations among the project's team members, and may encourage individual reflection in the form of skills assessments, seminars, and courses.

Now, where does story telling enter into this picture of collaborative software engineering? Story telling is a way to support collaboration by providing a memorable format for transforming and transferring experience. Stories capture the content of actual software engineering practices. Story frameworks can be based on conceptual collaboration—how sharing takes place, how knowledge flows—since these typically form the scaffolding for project structures and processes. Then the actual decisions for how to apply the concepts and actual experience with such application—the practical collaboration—can fill out the story structure and create an experience report that can be associated with reusable artifacts (such as a particular process, development method, team structure, or product/code result).

To tell a software engineering story involves developing a storyline in terms of what is known, how it is known, who knows what, how all players recognize the distributed nature of knowledge and skills—what is the nature of the distributed cognition and what is the culture in which that cognition is practiced. Distributed cognition incorporates both doing things with others, and coordinating doing things in a highly decentralized manner. Distributed cognition focuses on creating a community of peers working together towards a shared purpose. It encourages more explicit representation and visualization of project elements, so that all team members can understand and contribute.

The following examples of story-telling illustrate how devising a plan to tell a story might help to determine the structure for collaboration in a software engineering project community. Each example contributes to our understanding the three factors introduced earlier: conduct of work activity, development of the practice, and development of the practitioner. The people whose work impacted these ideas are: Donald Schon [Schon, 1984] whose reflection-in-action ideas encourage us to "let the situation talk back"; Doug Engelbart [Engelbart, 1984; Victor, 1977] who emphasized the importance of a vocabulary for talking about work, and who was one of the first to practice collaborative software engineering; and Carol Strohecker [Strohecker *et al*, 1999; Strohecker, 1997],

whose early work on interactive digital stories motivated the investigations reported on here.

5 LEARNING ABOUT TEAM CONDUCT: IDENTIFYING NOTICEABLE EVENTS

Carol Strohecker and her colleagues have been studying the structure of interactive stories. In her designs, readers can interact with core story elements to secure individual presentation, and to add observations to be shared with other readers. Readers are treated as though they and the team members belong to a Greek chorus, members of which can query characters and comment on events. This story form can be used to create interactive narratives about the conduct of team activity—one of Hutchin's three factors about work introduced earlier.

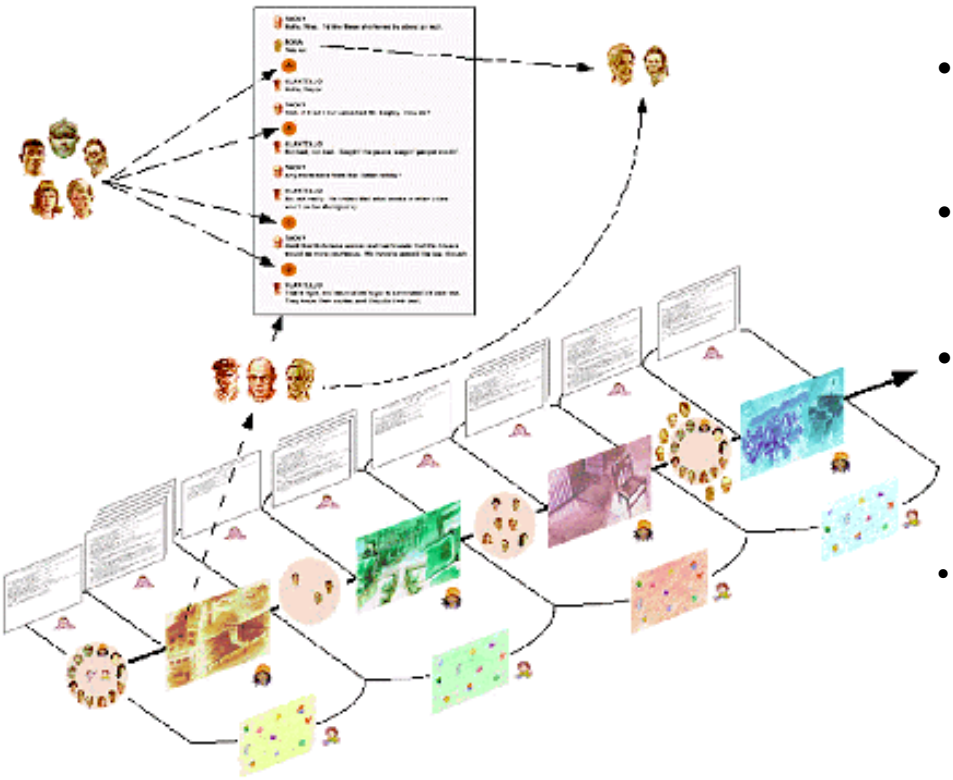
The story of interest is a set of personal narratives told in the context of historical events. Strohecker notes that history is, "…not a matter of objective fact, but of perspective. People experience, remember, and relate events and circumstances differently." Such narratives are best browsed rather than read in sequence, some loss of logical sequencing and organizational detail can be tolerated, despite the existence of a consistent story-line and characters. Reader comments can be shared with other readers, enhancing the information shared with the captured narrative. This research employed the story of the 1955 arrest of Rosa Parks that sparked the Montgomery, Alabama bus boycott as the first significant event in the U.S. civil rights movement.

The chorus can consist of just those characters involved in the chronicled events—the chorus present, or include characters in history that play some role—the chorus past, and characters in the future affected by the events—the chorus future. For the Strohecker example, these last two are Africans taken as slaves, and urban dwellers today—the former looking at the present events wondering how they could occur, and the latter wondering if anything really changed. In a software engineering story, the chorus past comes from experiences with similar problems or solutions, while the chorus future might be harvesters commenting on opportunities for knowledge reuse.

The reader has control of how the story unfolds, at the depth of the retelling. There is a default retelling of the basic story-line, and a peek at the conversations of characters surrounding each event. Each chorus member can relate additional observation, and the reader can add his or her own (posted to what Strohecker calls the graffiti wall). And, finally, there is a chorus member who provides access to the references used to construct the event (for the software engineering version, these would likely be a summarized view of the project community settings that affected or were affected by the event).

The interactive structure of this form of narrative is a set of scenes and images of characters that can be clicked to give information or engage in dialog. Figure 2 is borrowed from Strohecker's online web page describing the project.

Figure 2. Components of interactive narrative. Taken from Strohecker *et al, 1999*.[3]



This style of narrative and interaction can be reused for our software engineering stories only if we can capture or harvest in some way what Strohecker *et al* refer to as "noticeable events"—the pivotal occurrences that highlight change or rationale in a software engineering project. Specifically, how do we construct a project community such that we can automatically extract noticeable events and the points of view of the many roles (engineers, managers, customers) that surround these events? How do we then determine the personal narratives to be queried (presumably of team members and other project stakeholders)? And through this ability to query, how can we have readers collaborate with team members to examine what transpired, moving the practical collaboration of carrying out the project into an educational collaboration?

One solution is to provide a community structured by gathering places or *settings* in which interactions occur towards accomplishing some task. All work takes place in a setting. Access is restricted to those team members who collaborate on the task. There are inputs, outcomes, and task-specific tools, and links to relevant information. Each team

[3] Included with permission from Carol Strohecker, personal communications.

member, as a community member, is profiled according to skills and responsibilities; it is possible to determine if the same individual engineer is fulfilling multiple roles. The definition of a setting can extend the responsibilities of a team member to assume setting-specific roles, such as meeting leader or code-review recorder.

The setting acts as a context in which practical collaboration happens. Information, the subjects of the work, is constructed and referenced. This information is explorable by team members who have access to the setting. Any team member can maintain awareness of events by declaring interest in changes to any information, or by monitoring combinations of data. In addition, each setting may include at least one conversation or forum in which team members discuss their work and any issues that might arise. Additional discussions can be added by the team members, possibly with side channels for highlighting actions, decisions, or votes. Contributions to completing tasks and to discussions are all time stamped so that it is possible to correlate otherwise independent actions—who knows what, acted in what manner, added new information, commented, and so on—in a common time frame.

The setting structure for an online project community is based on the model for the project structure; it defines the distribution of access to information that is created in the many settings that make up a community (conceptual collaboration). These many settings are related according to a flow of knowledge so that outcomes from one setting provide inputs to another. The combination of shared access and shared knowledge enables the coordination of expectations, which is the basis for coordinated actions.

The online project community explicitly presents to all team members the structure of the community's settings, including the assignments of team members, and the flow of knowledge in the form of inputs and outcomes. Each team member directly shares the knowledge created in settings to which he or she has access. Each team member is aware that knowledge is created in other settings.

Participants in a setting can observe one another's actions, so that it is possible to learn from one another. In capturing a history of these actions, participants can review their work. For example, conversations can be initiated to explore how a tool is used, as well as how a task can best be accomplished. A project community, then, forms an ideal way in which the researcher interested in distributed cognition can, as Rogers & Ellis suggest [Rogers & Ellis, 1994], map out the ontology of shared and individual knowledge, and the means by which it is communicated, adapted, and used in distributed cognitive activities.

So, the basic idea of community—with settings holding explicitly defined tasks, tools, and shared knowledge, and individually defined notifications and data monitors—gives us the context for collaboration. But what about the attitudes of the team members? What point of view did each have on the history, that affected their commentary and decision making? What story would these members tell if we could construct an interactive narrative with readers and team members in Strohecker's Greek chorus?

Suppose we add a feature to each setting, call it the *team narrative*. The narrative is constructed collaboratively by team members with access to the setting. Simply, each

team member comments, whenever he or she pleases, on what is going on in the setting. At any given time, there is only one comment per team member. The comment is time stamped and labeled by role name or member identity. The team narrative is the sum of all the comments, and presents a combined view of the conduct of activity in a setting at any time.

It is likely that lots of change to the team narrative will occur around events that produce significant changes in the status of work in a setting. When the project is over, we can look for times when there is a lot of change in each setting narrative, and use that as a hint that some noticeable event has taken place. We can then look across all settings to determine times when lots of churning was taking place in the community, and then designate these times as the elements of the main story-line. The initial Greek chorus consists of each of the team members with access to knowledge about the event; a reader can then browse the narrative contributions to learn about each team member's perspective and how it changes over time.

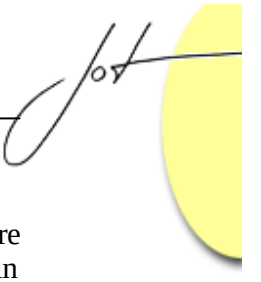
When a project model underlies the team interactions, the model can be used as a basic structure for the story. The model gives us an explicit representation for project vocabulary, team structure, knowledge flow, knowledge access, and communications. These can in turn be presented to team members to help them coordinate their activities. Where the model itself can be modified while the project is ongoing, team members can adjust their working context according to personal and group preferences.

Given the addition of team narratives in the project community, it is possible to construct a view of the project from each team member's point of view. Much of the churning we might expect to see in the narratives likely occurs around changes in project structure or roles—this is certainly our experience in the real world! We would then mix these observations with the actual history of events carried out in the context defined by the project model in order to create interactive stories like those proposed by Strohecker.

6 LEARNING ABOUT A PRACTITIONER: MODELING ROLES IN A PROJECT

A story begins with a plot, embedded with character development. The plot in the story of a project is the project process. It starts with the definition of objectives, it concludes when those objectives are met. And in the middle are the many characters who participate and interact to produce the desired outcomes, and sometimes perhaps undesired outcomes! How might we tell the story of the evolving practitioner—the second of Hutchin's factors in the practice of work?

One form of project story is the simple retelling of how any one role contributes to the project. The story follows the role through time, highlighting those noticeable events in the project in which the role participated, giving access to the role's actions, with whom the role interacted such as in conversations, what knowledge was solicited to carry



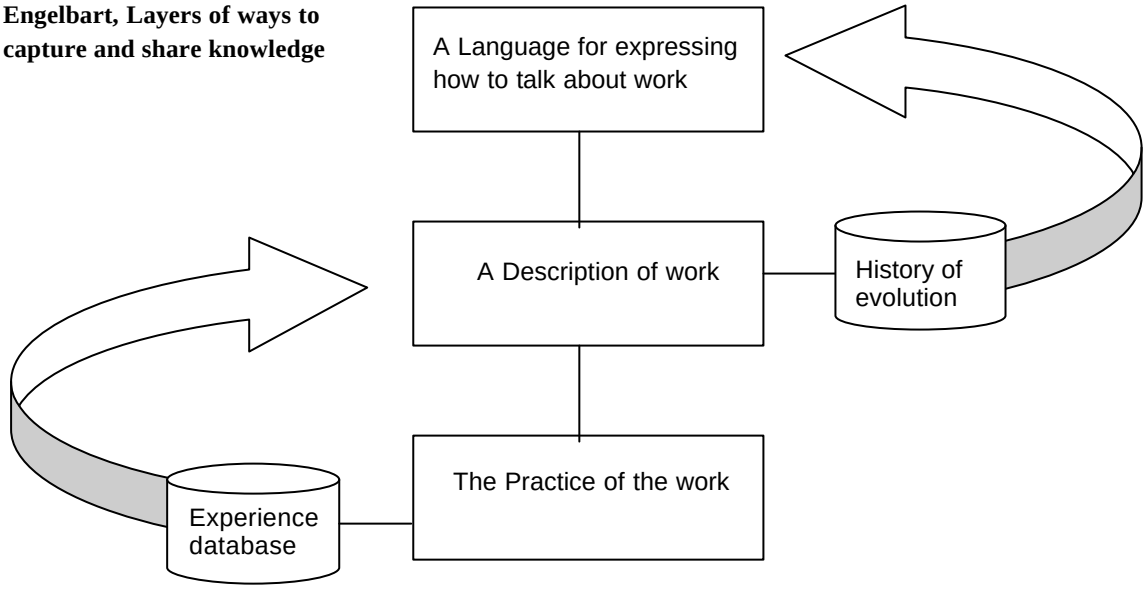
out a task, what artifacts were produced, and so on. Such a story is useful to help software engineers learn how to take on a new role by observing experienced people perform in that role in prior projects. Such a story is also useful to understand why certain skills are required. Teams become dysfunctional when members do not understand one another's roles. We might then follow the examples of a Le Carre spy story, a play like Akutagawa's *Rashomon*, or a TV soap opera—examining the same sequence of events, but from the perspective of different roles.

To be able to generate such stories, we need an explicit model of the project, including the responsibilities, assumed skills, and assignments of each role.

A project community is an online, virtual community whose structure and operation is defined by such a model. This model is expressed in a language of templates that specifies elements such as: the project purpose; objectives; measures to determine progress towards objectives; community roles; community vocabulary; settings with tasks, team assignments, setting-specific roles, and tools; and knowledge flow among settings. The model defines these elements generally so that several projects can be formed based on the same model, but created with different details. The model then evolves as the team work proceeds, and settings—notably for handling conversations, negotiations and decision making—are constructed as needed.

The idea to devise a language for describing work patterns is due to Doug Engelbart. Engelbart was interested both in having a way to talk about the work that is taking place or should take place, and in having a way to talk about how to talk about that work. This double layer serves an important purpose, as it allows us to capture information about actual work (for example, in an experience database as noted in Figure 3) and to recognize explicitly when we change how we discuss our work (the history of evolution database noted in Figure 3). An experience database contains the history of events, time stamped so that parallel actions can be coordinated in a story telling. The evolution of the model should bring us to a best practice model or set of models, differentiated at decision points whose different criteria for choice are tied to work circumstances. Exploring decision points might be a way to tell the story of the evolving practice. Both forms of information gathering provide insight into possible forms of educational collaboration.

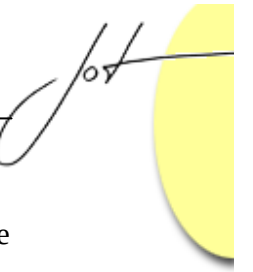
Figure 3.
Engelbart, Layers of ways to capture and share knowledge



7 LEARNING ABOUT A PRACTICE: MONITORING PROCESS AND KNOWLEDGE FLOW

The information derivable from the model used in a project is pivotal to telling many different forms of project stories that contribute to understanding the evolving practice—the third of Hutchins factors. The entire context of what transpired is defined by the schema given by the model and the content of the project's use of that model: who did what work, what information was available when conversations took place, who shared mutual knowledge when decisions were made or issues raised and resolved, and so on. The status of the project community can be seen at any time: milestones reached, results produced, information brought in as background or evidence, tools used. Where the project interactions are conducted online, the intentional communications are captured and can be reviewed in the context of actions taken. Where the project community includes spontaneous chats and less formal conversations, these exchanges can also be reviewed for their impact on actions.

In *Hamlet on the Holodeck* [Murray, 1997], Janet Murray exposes a myriad of story forms possible when stories are presented as computer-based interactions. One of my favorites takes a single moment in time and explores a house full of characters, one room at a time. In one view, a character leaves a room and is not seen again; where did he go?



In a second view, at just the right time, the character enters to continue his part of the story—carrying with him some artifact or knowledge critical to the plot.

These different rooms in the Murray example-story format map readily to the settings of a project community in which knowledge flows in and out, people playing the various roles converse, artifacts are constructed, and decisions are made. Rather than monitor a particular role, we might instead monitor the project process and track events according to single points in time that interest us because changes to artifacts took place. We could make use of the story form idea offered by Murray, by creating an interactive story in which we post a time line of project milestones. We then look at different settings at overlapping time slots that bracket the timing of the milestones in order to examine conversations and task status.

Grady Booch [Booch, 1995] uses a charting technique to lend visibility to what he calls the "regular rhythm of a project." He labels the x-axis by the phases of a project, and the y-axis by the artifacts to be produced. In actuality, the x-axis represents the march of time. The y-axis lists artifacts such as requirements, architecture, code, plan, tests. Coloring in the chart, from light to dark according to the amount of activity spent on an artifact in any phase, produces a kind of picture of the work effort—where the team places its emphasis. The picture can be compared to expectation, different patterns hinting at different cultures or sources of trouble.

A project model with settings to denote phases, subjects of the settings to denote the desired artifacts, and outcomes of settings to denote deliverables, can easily generate these project rhythm pictures. Select a point in the picture, and get the detail about the status of any settings denoted at that point, and access any setting conversations that provide some understanding about what the team members in the setting were doing.

Charts are often constructed of the number of outstanding software bugs in a software system. These charts show timing and frequency of discovering and fixing errors, and give the team manager a sense of whether the software system is converging to its expected quality level. Suppose, instead of bugs, we examine issues raised and resolved during the practice of software engineering. We might generate a chart reminiscent of the bug tracking chart, but one that is interactive. As in the Strohecker example for noticeable events and personal perspectives, we could use the chart of outstanding issues and issue resolutions, to access the setting in which an issue was raised, or the meeting in which the issue was discussed and resolved. If we wish to generate such an interactive pictorial form of story, we need to provide an explicit representation for "issues" in the project model, and we need to design a specific kind of setting in which meetings or conversations about resolving issues are held, so that they can be recognized as such, and retrieved for our story telling purposes. In doing so, we formalize a way to capture project conflict.

By specifying a particular structure for issue resolution, we create a recognizable style by which team members talk to one another about issues. We can translate this style into a story format that allows the reader to quickly scan for unusual or interesting situations.

8 CONCLUSION

I have described project communities designed around the idea of making explicit what is often implicit knowledge. We choose what should be explicit by considering what context and content is needed to produce informative stories. These are the stories that we can use to understand the work we do as software engineers, and to be able to use our work experience to help develop better practices and practitioners. The knowledge that we capture about a particular team is selected because its availability is useful to the team as it practices, and to this and other teams as part of reflecting on lessons learned and learning from one another.

ACKNOWLEDGEMENT

Any paper is a collaboration, conceptual, practical, and educational. I owe a debt of thanks at all three levels to colleagues Angela Coppola, Dennis Allison, and David Leibs, for their willingness to comment, discuss, and edit.

REFERENCES

Anthes, Gary, "Charting Knowledge Management Course", *Computerworld*, August 21, 2000.

Booch, Grady, *Object Solutions: Managing the Object-Oriented Project*, Addison-Wesley Longman, October 1995. Also online in *Conducting a software architecture assessment*, a white paper prepared for Rational and found at http://www.rational.com/products/whitepapers/391.jsp.

Engelbart, Douglas C., Collaboration Support Provisions in AUGMENT, *OAC '84 Digest: Proceedings of the AFIPS Office Automation Conference*, Los Angeles, CA, February 20-22 1984, pp. 51-58. [http://www.bootstrap.org/oad-2221.htm]

Hutchins, Edwin, *Cognition in the Wild*, MIT Press, 1995.

Murray, Janet. *Hamlet on the Holodeck*, Free Press, 1997 and also MIT Press, 1998.

Rogers, Yvonne, and Ellis, J., Distributed Cognition: an alternative framework for analysing and explaining collaborative working. *Journal of Information Technology*, 9(2), 1994, pp. 119-128. (Also http://www.cogs.susx.ac.uk/users/yvonner/dcog94.html)

Schon, Donald A., *The Reflective Practitioner: How Professionals Think in Practice*, Basic Books, 1984.

Schwartz, Peter, *The Art of the Long View*, Doubleday Currency, 1991.

Strohecker, Carol, Brooks, Kevin M., and Friedlander, Larry, *Tired of Giving In: An Experiment in Narrative Unfolding*, Mitsubishi Electric Corporation Research Laboratory, TR99-16, April, 1999. [http://www.merl.com/reports/TR99-16/index.html]

Strohecker, Carol, A Case Study in Interactive Narrative Design, *Symposium on Designing Interactive Systems*, 1997, pp. 377-380.

Van der Heijden, Kees, *Scenarios: The Art of Strategic Conversation*, John Wiley & Sons, 1996.

Victor, Kenneth E., A Software Engineering Environment, *Proceedings of AIAA/NASA/IEEE/ACM Computers In Aerospace Conference*, Los Angeles, CA, October 31-November 2, 1977, pp. 399-403. [http://bootstrap.org/augment-29292.htm]

About the author

Dr. Adele Goldberg is a director of Neometron, Inc. Neometron is a California-based consultancy working towards the use of virtual communities to support more effective teamwork. (http://www.neometron.com). Previously, she was the Chairman of the Board and a Founder of ParcPlace Systems, Inc. She served as CEO and President from inception until 1991, and Chairman until 1995. Prior to the creation of ParcPlace, Dr. Goldberg received a Ph.D. in Information Science from the University of Chicago and spent 14 years as researcher and laboratory manager at the Xerox Palo Alto Research Center. From 1984-1986, she served as president of the ACM, the U.S. computer professional society.