Hybrid Offshoring: 
Composite Personae and Evolving Collaboration Technologies

Nathan Denny, Shivram Mani, Ravi Sheshu, Manish Swaminathan, Jamie Samdal, Amar Gupta 
Nexus of Entrepreneurship and Technology 
University of Arizona

KEYWORDS: Collaborative Technologies; Distributed Work Arrangements; IS Project Teams

Abstract  Inspired by round-the-clock manufacturing, the 24-Hour Knowledge Factory endeavors to transform the production of software and other intangibles into a process of continuous development. While the concept of offshore software development is well established, few enterprises are currently able to develop the same code artifacts around the clock. We discuss the benefits of applying the 24-Hour Knowledge Factory to software development. We also present a representative scenario highlighting the problems of asynchronous communication in current offshore software development practices. Further, we introduce the notion of composite persona as a potential collaboration model within the 24-Hour Knowledge Factory and explain its ability to mitigate problems arising from communicating across cultures, languages, and time-zones. Finally, we present a suite of new collaboration tools and techniques that are being developed specifically for use by composite personae in the 24-Hour Knowledge Factory.

INTRODUCTION

Inspired by the paradigm of round-the-clock manufacturing, the concept of 24-Hour Knowledge Factory endeavors to transform the production of intellectual property and intangibles into a process of continuous development (Gupta and Seshasai 2007). More specifically, we envision a 24-Hour Knowledge Factory as an enterprise composed of multiple sites that are evenly distributed around the globe. As the sun sets on one site, it rises on another; like an ongoing relay race chasing the sun, the day's work is handed off from the closing site to the opening site.

The benefits of implementing a software development enterprise as a 24-Hour Knowledge Factory are several. We expect to realize gains from significant compression in development schedules, faster turnaround time for localization and customization of existing products, and bug fixes and critical security patches released with greater celerity. However, we also admit that the challenges of establishing a 24-Hour Knowledge Factory are significant. More specifically, we anticipate technical challenges arising from asynchronous communication, which will likely be exacerbated by cultural and linguistic differences. Non-technical challenges may grow from political and legal circumstances and from difficulties in managing and operating in such a nontraditional business environment.

In this paper, we discuss the current state of offshored and globalized software development and some of the underlying difficulties. Next, we introduce the concept of the composite persona (CP). Finally, we discuss evolving collaboration technologies that support the concept of CPs in the context of hybrid offshoring.

GLOBAL SOFTWARE DEVELOPMENT

Software development projects that involve multiple, international sites have been a reality since the 1960's (Carmel 1999, p. 17). However, those early efforts were relatively rare compared to the near ubiquity of contemporary global software development (GSD). By the year 2000, 200 of the Fortune 500 companies relied upon global software development teams or outsourced development to firms that
use them (NASSCOM 2000).

Our vision for the 24-Hour Knowledge Factory for software development is not synonymous with currently accepted GSD methods. Rather, the 24-Hour Knowledge Factory is a special case of GSD that has several unique properties while inheriting most of the problems of GSD. Subsequently, we present a brief survey of contemporary GSD methods to better define the problem of software development in the 24-Hour Knowledge Factory.

**Convention and Practice**

Engineering is the process of designing systems to solve problems. The activity of software engineering produces software systems to solve problems. Like nearly all engineering disciplines, software engineering advocates that large problems be recursively decomposed into smaller sub-problems and their corresponding sub-system solutions. Decomposition proceeds recursively until the problem-solution pair is tractable in both understanding the sub-problem and the resulting complexity of the sub-system. In modern parlance, the ultimate result of decomposition is a set of modules and classes (expressed in an object-oriented language such as Smalltalk, C++, or Java), where each class can be more-or-less completely understood by one person.

These modules and classes are then assigned to developers who then own that artifact throughout the process of coding, unit testing, and possibly even maintenance. Each module or class has one owner and only that individual may alter that artifact. Ownership confers the benefits of accountability for defects and preserves the continuity of the actual state of the artifact with the expected state of the artifact. Although a few software practices advocate otherwise, single ownership is the most generally accepted practice (Nordberg 2003).

Ideally, sub-problems can be solved in complete isolation from lateral, sibling sub-problems. This allows for sub-systems to be viewed as black-boxes: entities that have defined behavior and state and whose inner workings are not visible to those outside of its development. With respect to coding and implementing, black boxes confer autonomy on the developer and permit development to proceed concurrently. If the problem was initially well defined, with complete knowledge of the domain of interest, and there is little change in the environment in which the software is to be deployed, then there should be little lateral communication between implementers of sub-systems. It is, most unfortunately, a rare case where these perfect conditions exist.

More typical is the case where, during the process of development, the problem is more complex than what was originally believed or conditions external to the project have changed. Either case causes alteration to the requirements which were used to produce the original design. Software, like a machine, requires that all of its components to operate in harmony. Changes in the requirements and subsequent behavior of one sub-system propagate to those sub-systems with which it interacts, creating a rippling wave of secondary changes in those sub-systems, and so on.

The result of this phenomenon is a need for lateral communication between sub-systems developers. In the context of GSD, lateral communication between sites often introduces delays, costs, and risks. As an illustration, consider the following fictional, yet representative, scenario.

**A Typical Scenario in GSD**

Yankee Software, Inc. is developing a new client-server application for clinical use by physicians in hospitals and other high-tech treatment centers. The new software is to be marketed internationally and must conform to local laws and standards. After thoroughly studying the problem consulting many potential customers, the architects at Yankee Software develop a design and subsequently
produce the models and documents that will be used by the programmers to implement the software.

In order to reduce coding costs and leverage significant experience and talent with coding for highly reliable networking software, Yankee Software is partnering with Muscovite Technologies in Russia. Yankee Software will develop the end-user client, workflow engine, and validating rules, while Muscovite Technologies will develop the database, cache and network server system. After the design phase and an initial kick-off meeting where both parties sign off on the design, coding begins.

The designers had anticipated several risks; key among these known risks was the vulnerability of the validating rules. These rules localize the software for operation in the various legal and standards frameworks in which each potential customer must operate. During development, this risk becomes realized in a change in record keeping standards in the United Kingdom's National Health Service. The change is significant enough to necessitate an adjustment in both validating rules and database structure. At this stage of the project, such a change will be quite costly in terms of redesign and coding. However, the market potential for the NHS is second only to the domestic US market and Yankee Software does not want to lose such an opportunity. Consequently, the requirements are altered and parts of the system undergo re-design.

Several weeks later, Kathy, a programmer for Yankee Software in San Jose (California, USA) comes across an ambiguous condition in the specification for the software. It appears to her that there is a conflict in how the server handles errors. The prior documentation, before the NHS redesign, and the posterior documentation issued after the redesign disagree in several cases. Since the affected server component has already been coded, Kathy decided to contact the owner of that code. Following is an outline of the (fictional) discussion that these two developers might have. (Note that Moscow is 11 hours ahead of San Jose.)

On day 1, Kathy discovers the ambiguity. She consults with a few co-workers at her site in San Jose and decides to e-mail the owner of the component, a developer named Feodor in Moscow, Russia. Kathy puts further development of her component on hold and continues with other work.

When Feodor logs into to read his e-mail at the beginning of day 2, he sees the message from Kathy asking for clarification. The problem is complex, English is Feodor's second, or perhaps even third language, and Kathy's writing style is not very expressive. Feodor is confused as to the exact meaning of Kathy's query and sends a reply asking for more details and further clarification.

Later in day 2, after Feodor has signed off, Kathy logs in and reads Feodor's reply. She takes a few minutes to think of a better way of explaining what she had found and what she was asking, citing the specific sections of the documents in conflict and including a fragment of the code. She sends the improved, clarifying question to Feodor.

It is day 3 and Feodor logs in to find James question easier to understand. Feodor believes he understands the problem and takes a few minutes to compose the reply. Later that same day, Kathy reads Feodor's response and believes that she understands Feodor's answer. However, Feodor accidentally replied with a double-negative and since this potential defect may be harmful to their market in the UK, Kathy composes another question for Feodor to verify his answer.

On day 4 and Feodor logs in to read his e-mail. He shakes his head for a moment, amused at his own mistake and writes a response agreeing with Kathy’s statement and verifying his own prior e-mail. After the sun has set in Moscow, Kathy reads Feodor's response and alters the client-side code that she is developing to be in harmony with the server-side code that Feodor has already written.

In the preceding story, Kathy and Feodor, separated by an 11 hour time zone difference and speaking across languages and cultures, spent 4 days solving a problem. Had the two developers been co-
located, this problem would likely have been solved within a few hours.

**COMPOSITE PERSONAE**

The pressure to collaborate has increased with the use of offshoring, in which developers need to be addressed at different locations, according to Booch and Brown (2003). About 70% of a software engineer’s time is spent on collaborative activities (Vessey and Sravanapudi 1995). As demonstrated above, this need for communication can confound software development when the distributed teams span multiple cultures, languages, and time-zones.

As a mechanism to correct for asynchronous communication lag and ambiguity introduced by cultural and linguistic differences, we introduce the notion of *composite persona*. A composite persona (CP) is a highly cohesive micro-team that, like a corporation, has simultaneous properties of both individual and collective natures. That is, a composite persona to an external observer has a unique name and acts as a singular entity, even though it is the composition of several individuals.

With respect to CPs, each site is a mirror of the other, having exactly the same CPs as each other site. (Note that this does not imply that each site has the same number of staff, as one developer may belong to more than one CP.) As the world turns and sites turn on and turn off, each CP remains active, but the "driver" of each CP changes with each site.

Using CPs, development proceeds in the same manner as in a more traditional, local process. Problems are decomposed into modules and classes as they are when only single developers are assumed. However, when modules and classes are assigned ownership, the owner of each artifact is no longer an individual developer but rather a CP. Similarly, in the process of conflict resolution, discussion, and debate, each CP contributes as a single entity.

In the following subsection, we present three scenarios that exemplify what we anticipate will be common interactions within a CP and between CPs.

**A Composite Personae Scenario**

For the purposes of illustration, we assume three development sites: Tucson (USA), Wroclaw (Poland), and Sydney (Australia). We will focus on two CPs: CP Mercury and CP Minerva. CP Mercury will be composed of Tom (USA), Grzegorz (Poland), and Molly (Australia). CP Minerva is staffed by Rachel (USA), Sylwester (Poland), and Jack (Australia).

Furthermore, our scenario will take place in early November. Tucson is at GMT-7, while Wroclaw is at GMT+1. November is in Austral Summer. When the residents of Sydney set their clocks ahead at the end of October, they are at GMT+11.

**Forward Communication and Hand-off**

The most basic operation in the context of CPs is the hand-off, depicted in Figure 1. This occurs at the change of every shift, when the current driver signs off and the new driver, as the next site turns on, signs on and takes over responsibility for the CP. Our prototype hand-off procedure is inspired by the daily stand-up meeting used in the Scrum agile process (Schwaber and Beedle 2002). Scrum daily stand-up meetings, for co-located developers, are done first thing in the morning. All attending are asked to briefly summarize what they accomplished the previous workday, what problems they encountered, and what they expect to accomplish during today.
We think this to be a very succinct set of questions to be used as a basis for the hand-off from one driver to the next. Following is brief story that demonstrates a hand-off.

It is nearly 15:35 GMT, 25 minutes before Grzegorz's shift ends at his office in Wroclaw. He looks up at the clock and decides to wrap up his work for the hand-off to Tom. Grzegorz opens up the hand-off tool and begins filling in template forms and following a scripted workflow.

At 15:50 GMT, Grzegorz completes the last of the hand-off forms and officially ends his workday.

At 15:55 GMT Tom signs on and becomes the driver for CP Mercury. He takes several minutes to read what Grzegorz accomplished and finds that Grzegorz has found what he thinks is a bug in a class owned by CP Minerva. He recommends to Tom that he query CP Minerva and continue working on a couple of troublesome methods of a class owned by CP Mercey. Tom agrees and sets about his day, accordingly.

At 22:05 GMT, Tom gets an instant message from Molly. It is just after 9am DST in Sydney and she is beginning her day. She and Tom chat online to divide up the work recommend by Grzegorz. Tom is still driving CP Mercury and has checkout priority for any code artifacts owned by CP Mercury.

At 23:30 GMT, Tom walks through his handoff report and submits it at 23:50 GMT. Since there are still a few minutes of overlap, he sends an instant message to Molly. She sends an instant message back to Tom informing him that his hand-off report is understood and wishes him a good evening. At 23:57 GMT, Tom signs off and Molly becomes the driver for CP Mercury.

The hand-off process transfers knowledge from the driver signing-off to the driver signing-on. This transfer is unidirectional from past to present and forms the forward dimension in the two-dimensional communications that are possible with CPs.

**Resolving a Simple Problem with Lateral Communication**

Many simple problems can be resolved by real time communication between drivers. In co-located software development, these small problems are often resolved by a phone call or a quick chat over a cup of coffee. There is no appropriate analog in GSD, where even simple problems can result in significant delay as rounds of conversation ensue over e-mail (Herbsleb, et al. 2003). Even when developers work off hours to make real-time contact with other sites, the communication channels available carry less information than co-located face-to-face interaction.

Consider the following example from Figure 2: At 21:10 GMT, Tom, driving for CP Mercury, gets an instant message from Rachel. Rachel is currently driving CP Minerva. One of the components owned by CP Minerva is using a component owned by CP Mercury. Rachel asks Tom for an example of how to use a certain feature of CP Mercury's component. Tom thinks about it for a moment and then sends a reply with a code fragment for her to study. She thanks Tom for the example.

This form of communication is lateral between CPs, and is orthogonal to the forward communication dimension. The third communication pattern is established when information is flowing along both...
dimensions.

**Resolving a Complex Problem Using Both Communication Dimensions**

For the purpose of clarification, we will define a complex problem as a problem that involves two or more CPs and cannot be solved within a single working shift. This problem must therefore be handed off from one driver to the next. A complex problem that can be handled completely internal to the CP can be resolved in much the same way as development. (Writing code is somewhat equivalent to solving a long and complicated problem.)

At 13:10 GMT, Grzegorz reads an e-mail sent to CP Mercury about a test that had failed and indicating a fault in a component owned by CP Mercury. Fortunately, the message contained a trace of the code that created the fault as well as the parameters of the test. Grzegorz finds that the problem seems to arise only in test cases where the component owned by CP Mercury is used by a component owned by CP Minerva. Grzegorz contacts Sylwester via instant messenger and they begin the work of tracking down the bug. At the end of their shift, each of them complete their respective signoff procedures, and end the day hoping that Tom and Rachel will be able to find the solution.

Tom and Rachel sign in within a few minutes of each other. Tom, seeing that Rachel is also at work, sends her an instant message and they agree to continue working on the problem. Each takes a few minutes to catch up on the notes from their counterparts. Grzegorz and Sylwester have done most of the work and it does not take much longer before Tom and Rachel find the source of the fault and have it corrected.

**Benefits of CPs in the 24-Hour Knowledge Factory**

We anticipate several benefits of applying the composite personae strategy in the context of the 24-Hour Knowledge Factory.

**Smooth Transition:** The process of decomposition, design, and assigning ownership is essentially the same whether development is done by individuals or by CPs. The introduction of CPs augments the body of engineering knowledge and does not deprecate anything. Furthermore, during migration from current practices to one that incorporates CPs, an enterprise is able to mix and match as needed. Some artifacts may be owned by individuals, others by CPs.

**Increased Trust:** Trust is notoriously difficult to establish in GSD projects (Handy 1995, McDonough et al. 2001) but essential for success (Sarker et. al., 2001). Humans in collective efforts work best in teams where each member of the team shares the common goal of the team and members trust the intentions and efforts of each other member. Here we are looking for the jelled team described by DeMarco and Lister (1987). As group size increases and the frequency and expressiveness of communication decreases, trust within the team suffers greatly (Carmel and Bird 1997).

CPs are very small, typically comprising of 3 or 4 individuals, and trust is easier to establish between these few members than it would be within a larger group. Although most communication will be done asynchronously, communication between members of CPs will be frequent and informative. While asynchronous communication is much less expressive than synchronous communication (Sproull and Kiesler, 1991), we expect that such persistent, frequent communication will lead to greater trust than is common in contemporary GSD projects (Jarvenpaa et al., 1998; Javenpaa and Leidner 1999; Maznevski and Chudoba, 2001).
**Convergence:** At this conceptual, stage of our development of the CP, we believe that communicating the changes made by each driver to the other members will require less costly communication than associated with current global software development practices. We expect that after working on the same artifact for some time, the knowledge held individually by each member of a CP will converge. That is, we expect that developers within the same CP will, after some warm-up period, have essentially the same concept of the problem domain, solution options, and the utility space of those options. For a given question about some code artifact, each member of the CP that owns the artifact is likely to give very similar answers. Consequently, between members of a CP, communicating the purpose of incremental changes and the overall state of the artifact will become increasingly efficient as their mutually shared experience grows with time.

Here we can draw upon the experience of Extreme Programming (XP), from which our CPs are inspired (Beck 1998; Beck 1999). XP advocates collective ownership of code, allowing anyone in the project to alter any artifact. This is made possible by creating mutual experience and shared knowledge through *pair programming*: the process of having two people work on the same artifact, simultaneously, using the same workspace. While XP is not without controversy, especially for larger projects and distributed development, we believe our CP method to be an especially useful balance for the 24-Hour Knowledge Factory. CPs prevent surprises and preserve accountability as per the single ownership model, yet enable many other collective ownership benefits that are discussed here.

**24-Hour Access to Owner:** The practice of single ownership of code artifacts works well when all development is done in the same time zone. However, for sites that have very little or no overlap in work schedules, the single ownership model becomes a hindrance. For instance, in the coding phase of the linear development model, bug fixes are typically routed through the owner of the artifact. In this way, the owning programmer is knowledgeable of any changes made to the artifact and understands how those changes alter the behavior of the artifact.

If we assume only one owner and three non-overlapping development sites, then 16 of the 24 work hours in a day are done by centers which do not have direct, synchronous communication with the owner. Necessary changes must wait until the owner's site turns on, creating a bottleneck for related work in the other two centers. Accordingly, by our method of assigning ownership to CPs, each piece of code artifact can be modified at almost any time.

**Higher Truck Number:** So-called from the imagined worst-case scenario where a project member gets struck by a runaway truck, the *truck number* of a project is an amusing but useful metric for expressing vulnerability to the loss of critical talent. The collective code ownership model and consequent degree of redundancy of knowledge about any given artifact increases the truck number of the project and decreases development risks introduced by the possibility of losing critical talent and specialized knowledge (Nordberg 2003).

With three-member CPs, a CP could lose a single member and the enterprise will still retain two people that have intimate knowledge over the artifacts owned by that CP. While productivity may decrease for the period that the CP remains understaffed, the project survives. When a new member is brought in to the CP, the new member has the project artifacts and notes from the previous developer; in addition, (s)he has direct and frequent access to the two remaining members in the CP.

This feature may be particularly important when including labor from nations where worker turnover is high. Indeed, some recent observations imply that high employee turnover in offshore companies often is a significant risk in contemporary offshoring strategies (Carmel and Agarwal 2002; Lewin and
Peters 2006; Offshoring Times 2006). This makes the higher truck number inherent to CPs to be a major advantage over current offshore development practices.

**Localizing Lateral Communication:** In a case study done by Herbsleb, et. al (2003), little difference was found in the number of lateral communication delays incurred by a local development project and a GSD project. However, the GSD project experienced longer delays. In this particular study, the mean delay time for the locally developed project was 0.9 days while the mean delay time for the GSD project was 2.4 days. Our own work on studying communication in technical USENET conversations (Denny 2007) suggests an average delay of between 2 and 5 days. With each CP represented at each site, lateral communication is now mostly local. Consider a question by CP Minerva about a code artifact owned by CP Mercury. If the Tucson site is on, then Tom will be answering a question by someone else co-located at his site. If Tucson is off and Sydney is on, the Australian driver for CP Minerva will be asking Molly, instead. Further, since each site is relatively mono-cultural (with respect to the entire enterprise), lateral communication will be nearly free of ambiguity introduced by differing linguistic and cultural norms. That is not to say that communication will be completely unambiguous, but the prevailing ambiguity will be resolved more easily by synchronous conversation between the drivers of the CPs engaging in active, spontaneous dialog.

We have not completely done away with asynchronous communication, but have instead moved into a different realm. Each member of a CP must have nearly equivalent knowledge of the artifacts owned by the CP. If the current driver modifies a code artifact owned by his CP, then the next driver must be familiar with that alteration. So within the CP there will be significant communication, almost all of which will be asynchronous in nature.

**Evolving Collaboration Technologies**

DeSanctis and Gallupe (DeSanctis 1987) described a space and time classification frame-work in a 2x2 matrix (Figure 3) that classifies tools based on the temporal characteristics of activities and location of the teams.

The 24-Hour Knowledge Factory paradigm fits into the 4th quadrant. More sophisticated tools than available asynchronous communication tools are needed that to tap and effectively transfer tacit knowledge.

In distributed collaboration software development, tools that seamlessly enable communication without loss of tacit knowledge are critical. The communication methods that were successful in multi-site transfer of knowledge, particularly when dealing with remote or global teams, included teleconferencing, video-conferencing, chat, email, and document exchange. While some of these methods are not necessarily effective for most of each shift in the 24-Hour Knowledge Factory system (those involving real-time communication), they might be particularly important during the hand-off between each shift in order to convey a large amount of information in a short amount of time.

<table>
<thead>
<tr>
<th></th>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Same Time (Synchronous)</td>
</tr>
<tr>
<td></td>
<td>Different Time (Asynchronous)</td>
</tr>
<tr>
<td>SPACE</td>
<td></td>
</tr>
<tr>
<td>Distributed</td>
<td>1st Quadrant</td>
</tr>
<tr>
<td>Same Space</td>
<td>Spontaneous collaborations,</td>
</tr>
<tr>
<td></td>
<td>formal meetings, classrooms</td>
</tr>
<tr>
<td></td>
<td>2nd Quadrant</td>
</tr>
<tr>
<td></td>
<td>Design rooms, Project</td>
</tr>
<tr>
<td></td>
<td>scheduling</td>
</tr>
<tr>
<td></td>
<td>3rd Quadrant</td>
</tr>
<tr>
<td></td>
<td>Video conferencing, net</td>
</tr>
<tr>
<td></td>
<td>meetings, phone calls</td>
</tr>
<tr>
<td></td>
<td>4th Quadrant</td>
</tr>
<tr>
<td></td>
<td>Emails, writing, voice mails,</td>
</tr>
<tr>
<td></td>
<td>fax</td>
</tr>
</tbody>
</table>

*Figure 3: Tool Matrix*
Synchronous hand-off is shown to be successful in the following tools:

Microsoft’s NetMeeting (Netmeeting 2006) facilitates the social interaction required for sharing tacit knowledge. IBM’s Workplace Collaboration Services provides a full range of integrated communication and collaboration tools (IBMWCS 2005). Integrated development environments (IDEs) such as Eclipse are useful collaborative software development tools. Collaber is a collaboration framework that is built on Eclipse Software and includes features such as task management and group discussion. Eclipse goes a step further by its integration of JAZZ (Cheng et al. 2003), an awareness tool that allows developers to initiate and archive synchronous communication from the IDE in the code context. In all these distributed collaboration environments including the tools mentioned above, the developers rarely have a time overlap and hence synchronous communication tools are of no avail.

The tools required for a 24-Hour Knowledge factory using the CP model necessitates not only distributed remote collaboration but is also limited to an asynchronous mode. Wiki (Cunningham 2001) is a web-based collaborative tool to use the strength of collective intelligence. MASE (Chau and Maurer 2004) extends the wiki concept into the realm of agile software processes and can be used for knowledge sharing by both co-located and distributed teams. The limitations with such tools are that users have to explicitly store documents and there is little to no context from which decisions can be analyzed by succeeding shifts. Furthermore, there is a strict dependency on the exact terminology in order to relate different content in the repository, which is a weak data mapping model. Distributed document management is also critical in a collaborative development environment. There are many configuration management systems such as CVS (Berliner 1990), IBM’s ClearCase (Allen et al. 1995), and Subversion (Collins-Sussman 2002). These do a good job in defining mechanisms for managing different versions of their work products. In such tools, the developers tend to get inundated with notification/updates of the various control events. Allowing the developers to configure their artifacts of interest is of good value as it reduces the information overload on the developer. Software tools such as CVS Watch (Berliner 1990) allow the developers to define their artifacts of interest for which they get automatically notified on any access or modification.

WorkSmart.net (2007) is a hosted asynchronous collaboration service which provides the ability to configure a set of events whenever any action is taken. For example, an event may be triggered when a new folder is created, document deleted or a new version of a document has been checked in.

Most of the commercial collaboration tools that have been built are either add-ons to existing tools or those that integrate several existing tools tuned to a synchronous mode. These tools rely significantly on user initiated annotation and documentation and do not address the project hand-off from one user to another in an efficient manner. The free flow of information and assistance is vital to the 24-Hour Knowledge Factory environment. All of these needs have created an environment that requires innovative and more useful processes and tools.

**CPro – Composite Personae Software Process**

Software processes typically aim to improve quality and productivity. Quality and productivity are measured in terms of cost, effort, delivery schedules, and defects. However, many of these processes like PSP (Humphrey 1995) assume single ownership of code. Extreme Programming (XP) (Nagappan and Williams 2003) supports multiple ownership of code, assumes horizontal decomposition of work with more than one developer sitting together working on the same task. This is no longer the case where CPs are collaborating in the 24-Hour Knowledge Factory. This work model poses many more challenges in terms of estimation, work allocation, knowledge transfer and defect control. To overcome these problems and to facilitate offshoring in the 24-Hour Knowledge Factory model we have come up with a software process which we call CPro.
Planning: When an overall schedule of the CP to complete a task needs to be obtained, one developer cannot estimate for the whole CP. This is because, more than one developer present in the CP share the same task and productivity can vary greatly from one programmer to the other. This has long been observed to be quite wide (Sackman et al. 1968; Humphrey 1995). Due to a variety of factors, we can know a priori neither the subtask allocation within the CP nor the exact productivity of each member on any given task. Therefore, in CPro each developer gives his or her own estimates for all of the subtasks within a task. A scheduler then executes a Monte Carlo simulation on the possibilities for productivity and project evolution. The result is a probabilistic schedule that project managers can use to estimate the delivery date of project artifacts within a specified degree of confidence.

Knowledge Transfer and Defect Control: Defect reduction is one of the primary goals of any software process that aims at improving quality and productivity. In the CP model, since tasks are vertically decomposed among drivers of the CP, each driver changes the state of the task. In the absence of a structured handoff procedure, defects may also occur due to lack of understanding of the current state of the task. Therefore, many of the current defect reduction techniques cannot be utilized as is.

CPro makes use of the existing project artifacts as implicit hand-off documents. This by itself not only conveys the work done in the previous shift, but also provides inputs to the current driver in the form of a project artifact. When the current driver reads the artifact to continue the work, he would be equipped with the knowledge of the job done by the driver in the previous shift. While reading the artifact, the developer in the current shift would in effect review the artifact and could give suggestions and other feedback.

To achieve this, we suggest the use of Test Driven Development (Williams et al. 2003). Test cases for some method are written by one driver, which would act as a unit test document for that method. This also would serve as a hand-off document. In this way, the current driver who codes on the method receives input from the previous driver in the form of unit test cases. The effect of this being that any discrepancies in understanding the design are clarified at this stage. Test-cases must be machine interpretable and consequently lack the ambiguity of natural language. The developer in the second shift would then code the method defending the unit test cases and the CP peer in the next shift would review the code. In this way, all three members of the CP would be aware of the code artifact in good detail and extra communication in the form of status email or documents are avoided.

MultiMind

Suchan and Hayzak (2001) found that a semantically rich database was useful in creating a shared language and mental models. MultiMind is a novel collaboration tool under development and experimentation which aims to provide a semantically rich environment for developers collaborating using the CP method. MultiMind aims to improve upon DICE (Sriram 2002) and other collaborative engineering tools.

Our initial implementation efforts have focused on providing a proof of concept for the 24-Hour Knowledge Factory model. Based on feedback from the initial users we plan to judge the efficacy of the model and tune our development efforts accordingly. MultiMind is founded on the following technologies:

Software Processes: MultiMind is process aware and guides the CP members along the CPro process. In addition to CPro, MultiMind also incorporates concepts from XP and Scrum. The tool automates some of the Scrum process, keeping developer intervention to a minimum. More specifically, the
responsibility of creating work summaries has been offloaded from the developers to the embedded project management system. A templated scrum-style system is incorporated which, on sign-out, gathers quick statements about the anticipated future state of open tasks. On sign-in, during the subsequent shift, a scrum report is automatically generated by synthesizing information from the previous scrum entry and the state of various archived project artifacts. Figure 6 depicts the usage of Scrum and CPro processes during hand-off.

**Lifestream:** Objects and events relevant to the project are posted and logged into a monotonically increasing persistent database along with a time stamp. This database, sometimes called a Lifestream (Freeman and Gelernter 1996), allows human users and intelligent agents to create a local or comprehensive view of the state of the project at any given time. This capability is central to justify decisions made by one user to a different user that is trying to understand why a particular decision was made and the original acting user cannot be contacted in real-time.

Towards facilitating the driver’s understanding of the current state of the task, we have designed a study tool which draws inspiration from Activity theory. This is, of sorts, a decision support system in reverse: a decision justification system that justifies the decisions made by the previous workers. The integrated project knowledge base (Lifestream) is mined for relevant knowledge objects (project artifacts, speech acts, events) that were consumed by the previous worker during the course of his action. The aim is thus, to supply the driver with only relevant and condensed knowledge objects.

**Speech Acts Theory:** Speech Acts (Austin 1963, Searle 1975), used in linguistics, are acts of communication. The Speech Act theory systematically classifies communication messages into various acts thereby establishing a common ontological base using which agents can communicate and understand each other. Both KQML (Knowledge Query Manipulation Language), developed by DARPA (Finin et al. 1993) and ACL (Agent Communication Language), developed by FIPA (O’Brien and Nicol 1998), rely on the Speech Act Theory for their protocol for communication between software agents in Knowledge Based systems.

In the CP model, Speech Acts assist in synthesizing relevant information based on the high-level semantic nature of communication events. Existing asynchronous communication mechanisms that use free text lack significant semantic structure making it impractical for a tool to classify the communication type or do meaningful analysis.

In the MultiMind tool, the communication entity is embedded as a conversation panel and it appears in the context of every project artifact. The conversation panel provides the facility to post messages based on a communication act and these messages are logged to the Lifestream.

**Vocal Annotations in Program Source Code**

The best scenario for transferring code from one programmer to the next is to have the first programmer sit with the next programmer to go over the code line by line and have him or her explain
what the program does and why it was designed and implemented the way is was (Chiueh 2000). This is not likely in the 24-Hour Knowledge Factory.

In the absence of appropriate documentation, maintenance of the code becomes a nightmare and grows worse over time (Miller et al. 1992). Dekleva cites in these surveys that lack of documentation is one of the biggest problems people who maintain the code have to face (Dekleva 1992). She takes the survey one step further and uses the Delphi technique to ask system maintainers what their largest obstacles are. The Delphi technique consisted of three rounds of surveys where the consensus increases with each survey round. In all three rounds that were conducted, system documentation was in the top four issues seen by system maintainers and it ended up tied for third when the survey was over.

Raskin (2005) explains the need for the thorough use of internal code documentation, not only for improving code understandability but also improving reusability and productivity. Literate programming introduced by Donald Knuth (Knuth 1984), is a methodology that combines the programming language with documentation language. The most well known languages of literate programming are TeX, Cweb, and METAFONT by Knuth himself. The most apparent drawback is that it necessitates a competent writing ability to be an effective literate programmer.

Fletcher describes how dictation becomes a more relaxed process since the user can now concentrate on the content of what is being dictated instead of the process of getting the text into the computer (Fletcher 1997).

Combining code documentation and voice recognition may be a good way to use an established technology to solve a problem that is continuing to grow in the software industry. The prototype developed takes audio comments, translates them to text and displays the text comments in the code. It also has the feature that the audio comments can be played back so if the translation is incorrect or not comprehended by the next programmer, he can play back the audio comment and hear exactly what was said.

**Eclipse Extensions**

While email is a commonly used asynchronous communication method, the knowledge sharing requirements of the 24-Hour Knowledge Factory make any one-to-one communication methods a poor choice for project communication. However, in order to convince software developers to move away from this method of communication, a convenient alternative that makes information available to the entire team must be provided. Project communication is not always technical in nature. Often, target dates, requirements decisions, priorities, and even team member schedules must be shared such that an entire team, regardless of where they are located, must be able to quickly assimilate.

The solution under development proposes integrating tools into a common development environment. The Jazz project at IBM (Cheng et al. 2003) is a recent initiative that begins to address these concerns by providing a framework for distributed collaboration that is also process aware and combines technical and managerial tasks into the development environment. This framework is built as an extension to the Eclipse Development Environment, and provides many tools that are ideal for supporting the 24-Hour Knowledge Factory.

What Jazz does not provide is an alternative to email that is functions as a team-based project communication method. One solution may be discussion forums. Forums allow informal communication to be stored in a structured format that is accessible by all team members. Therefore,
work is being done to study the existing functionality of the Jazz platform that would work well in the 24-Hour Knowledge Factory and to create appropriate plug-ins that integrate discussion forums into the Jazz platform.

CONCLUSION

As long-term strategic offshore partnerships evolve into the 24-Hour Knowledge Factory, new methods of work must be employed. Here, we have introduced the Composite Persona (CP), a long-lived micro-team that has simultaneous individual and collective characteristics. CPs have the potential to resolve much of the difficulty in contemporary offshore development by mitigating the cultural, linguistic, and time-zone differences between partners.

Modern groupware technologies cannot fully realize the possibilities made available by the 24-Hour Knowledge Factory and the CP method. Here, we have presented a new software process (CPro) customized for use by developers organized into CPs. MultiMind, our new groupware tool and framework builds upon a variety of novel technologies to aid software developers that employ CPro and automates many management and process functions in order to reduce the need for explicit cooperation between the collaborating developers. Furthermore, we are experimenting with speech recognition technology as it can be applied to vocal annotation of project artifacts in distributed and international development. On the near horizon, we are also preparing extensions to the popular Eclipse integrated-development environment that will provide a smooth path for moving from contemporary offshore development methods into the 24-Hour Knowledge Factory. Beyond technology, we have created an international partnership. With aid from the University of Technology - Sydney, Australia and the University of Wroclaw, Poland we can test theory and tools and move closer to the 24-Hour Knowledge Factory.

BIBLIOGRAPHY


Raskin, J. (2005). Comments are more important than code. *ACM Queue, 3*(2), pp. 64-. ff.


