The 24-Hour Knowledge Factory: Can It Replace the Graveyard Shift?

The MIT Faculty has made this article openly available. Please share how this access benefits you. Your story matters.

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>As Published</td>
<td><a href="http://dx.doi.org/10.1109/MC.2009.25">http://dx.doi.org/10.1109/MC.2009.25</a></td>
</tr>
<tr>
<td>Publisher</td>
<td>IEEE Computer Society</td>
</tr>
<tr>
<td>Version</td>
<td>Final published version</td>
</tr>
<tr>
<td>Citable Link</td>
<td><a href="http://hdl.handle.net/1721.1/59378">http://hdl.handle.net/1721.1/59378</a></td>
</tr>
<tr>
<td>Terms of Use</td>
<td>Article is made available in accordance with the publisher's policy and may be subject to US copyright law. Please refer to the publisher's site for terms of use.</td>
</tr>
<tr>
<td>Detailed Terms</td>
<td></td>
</tr>
</tbody>
</table>
With improved telecommunications and higher bandwidth, geographical distance is no longer the handicap that it once was, so it makes no sense to risk employee health with a graveyard shift. Indeed, many companies now view the time differences as a strategic edge to productivity, not an obstacle.

This new mind-set, as well as the need for a healthier alternative to night work, led to the formation of a research group at the University of Arizona with the aim of defining the 24-hour knowledge factory concept and creating a prototype system that would support it.

In this paradigm, multiple collaborating centers in time zones six to eight hours apart hand off work around the clock, ensuring that each center completes its respective part during the daytime. Of course, not all work lends itself to this arrangement, but our group found that the paradigm is adaptable for a variety of knowledge-based tasks. The key issues are how to hand over work in progress, decompose tasks into relatively autonomous units, and manage operations effectively in such a decentralized environment.

Examples from industry and academia offer insight into what is and is not currently possible or on the horizon.
Clearly, the ideas will spawn a new generation of computer professionals who will need to develop pieces of the factory infrastructure within various disciplines.

**HOW IT WORKS**

To illustrate how a 24-hour knowledge factory might operate, consider the workflow in three centers. An employee in a North or South American center ends work at 5 p.m. and transfers work to a worker in an Asian or Australian center, which is just beginning its workday. Approximately eight hours later, that worker transfers work to a third worker in a European or an African center, who at the end of the day, transfers the work back to the American center, and the cycle begins again. The paradigm requires at least three collaborating centers, but more are feasible if shifts need to be shorter.

Although this description makes the paradigm seem rather simple, it is, in fact, not that straightforward. Work transfer is really more like work transformation, since each worker adds and reshapes knowledge before passing it on. Transformed tasks, in turn, lead to redefined job descriptions, which require expertise to break down and define.

This workflow means that organizations must consider the characteristics of the work before attempting to apply the factory paradigm. Any situation in which an organization can capture and digitize underlying knowledge might be suitable, but the work must meet three criteria:

- The work can be broken down into disaggregated components.
- Several individuals can work on such components with little peer support.
- The cost of transferring work in progress is low. Cost includes the overhead involved in packaging both the work and the evolving decision rationales. As such, what constitutes “low” is an economic or business decision that is unique to each organization.

It is tempting to immediately dismiss manufacturing as a feasible application area, but in reality, it is not difficult to transmit information about individual manufacturing applications, such as design rationale, engineering, quality assurance, and sales data for individual parts, subassemblies, and finished products. With such information, geographically distant suppliers and others on the revenue stream can contribute to a manufacturing facility’s operations. The same applies to information in a range of disciplines involving accountants, architects, business managers, computer programmers, financial analysts, medical practitioners, and product designers.

**EARLY EFFORTS**

The factory concept and follow-the-sun models are not new. The notion of a collaborative 24-hour workday started with geographically dispersed call centers. Three or four centers collaborated, exploiting voice-over-IP technology to provide round-the-clock coverage—but with all employees working only during the daytime in their respective countries. Global communications networks also use multiple support centers.

The goal of leveraging the time-zone difference was to discover errors earlier in the development process, yet maintain advantageous skill and wage differences.

However, call centers are not concerned with generating a product or with knowledge transformation. Software development centers were the first to experiment with a rudimentary factory concept as a way to boost efficiency over traditional offshore outsourcing. In the traditional approach, which some companies still endorse, one center works on developing code for several weeks and then sends it to the company’s complementary center for independent testing. The modules under test remain frozen until the testers are finished. If the testers discover major flaws, the code returns to the development center, and the submit-freeze-test process would begin again with the revised code.

Obviously, the develop-test cycle had to be shorter, so developers came up with the idea of leveraging the time-zone difference. The goal was to discover errors earlier in the development process, yet maintain advantageous skill and wage differences. In this model, development involves two geographically distant teams—one to write the code during the day and the other to test it during the first team’s night. The second team then returns the results at the beginning of the first team’s day. As this description implies, the teams have somewhat of a master-slave relationship, which requires careful planning.

For knowledge to flow efficiently, the distributed teams must accurately delineate tasks and understand task interdependencies. Knowledge reuse and agility are particularly relevant to follow-the-sun models, and functionality encapsulation and interface standardization are critical to leveraging and reusing knowledge.

Several software development projects have used this model. A 1999 IBM project involved several collaborating
centers in a hub-and-spoke model, with the Seattle office acting as the hub. Similarly, a 2001 project at Motorola used developers across six countries to enable project work to continue at least 21.5 hours daily.  In a 2005 project, a programming team distributed among three sites (US, UK, and Asia) used modified Extreme Programming techniques in developing mobile telephone data services.

Industries outside software development are gradually embracing this idea as well. Beridium International, an accounting company, has experts working in different time zones in the European Union and in Mauritius, making it possible to provide round-the-clock accounting support, and North America General Motors uses the 24-hour paradigm in collaborating with Australian vehicle designers (www.pbs.org/nbr/site/onair/transcripts/070209b).

**HOW EFFECTIVE IS IT?**

In 2007, I and another researcher from the Massachusetts Institute of Technology (MIT) evaluated a yearlong study of two IBM teams assigned to similar development projects and working in parallel, one based entirely in the US, and the other distributed between the US and India. Although it reported on only two collaborating centers, the study provided insights into how a distributed team compares with the traditional team structure. Because the primary deliverable had a one-year time frame, the study covered the two teams at every major point in the project life cycle. Overall, the final deliverable quality was very similar, but the distributed team produced better documentation and a higher-quality knowledge repository.

**Advantages**

One obvious advantage was *less time for task resolution*. The gradual one-to-one pairing between onshore and offshore members made it easier to transfer tasks and knowledge. In one case, the US and Indian developers working in an interlaced mode (US sending the code for testing in the evening and getting programmer feedback from India in the morning) resolved a bug in half the time they would have taken in a conventional structure.

Related to shorter task resolution is *earlier issue reporting*, which often translates to cost savings in software development, since a bug detected late in development can delay product release. With the 24-hour model, a US-based engineer could incrementally improve code during the day and pass it to a test engineer in Asia with appropriate instructions, thus making it easier to detect any bug early in the life cycle.

A by-product of constant communication is *superior documentation and knowledge dissemination*. Team members who work in the same operational environment at the same time often distribute knowledge relatively informally. In a decentralized structure, they must use more formal mechanisms to capture decision rationale and design history, thus establishing a dynamic knowledge repository. Figure 1 depicts a tool for building a chain of knowledge events that communicate the reason behind the change in a piece of source code. This trail, or event history, is invaluable in transferring knowledge about design rationales, thus avoiding the common pitfalls of reinventing the wheel and repeating unproductive design decisions.

Finally, close teamwork results in a *better incorporation of cultural issues*. Because professionals from diverse cultures are continuously involved with one another, they are consciously or subconsciously exchanging specific, culture-based insights. The resulting product or service can, therefore, be of greater value to more global markets.

**Disadvantages**

Although informal communication can work against superior documentation, it is useful in narrowing cultural and technical gaps among professionals. Thus, *loss of informal communication* is also a disadvantage in that geographic distance makes face-to-face discussion difficult. However, communication technologies, such as webcams and instant messaging, are rapidly mitigating this problem by capturing the essence of informal discussion.

Geographic distance also means considerable *handoff overhead*. Work is transitioned every eight hours, and time and effort are involved in each handoff activity. The factory paradigm also involves considerable upfront effort in *task decomposition*. Someone must decide how to break down tasks in the various disciplines.

*Management and quality control* become more burdensome. With possibly three programmers across three continents, someone must provide overall guidance and management. One option is to employ three managers, but at some point the teams need a central coordinator, for example, to respond to urgent calls at any hour. Determining code quality is another control issue. If each programmer has a different idea of quality, who prevails? Should the team put the issues to a majority vote, with two against one, which might lead to relationship problems? Should an independent party decide? Decisions by consensus frequently involve significant time and correspondence, and the quality issue will come up during every code review. The notion of consensus is also tied to team size. When the size varies, the largest team tends to dominate, so it is best to establish teams of similar size.

Another overhead source is *infrastructure duplication*, since each country’s center requires its own set of hardware and software. Obviously, one set would cost less, but overhead is not the only concern. If all programmers used a single set of hardware and software in a 24-hour shift, someone would have to work at night. The ultimate goal is to have no one work at night, and the findings of the World Health Organization and American Cancer Society could motivate companies (and countries) to adopt...
measures that enforce day work wherever possible, even with the overhead.

**SUPPORTING KNOWLEDGE ASSIMILATION**

As this list of pros and cons implies, implementing the 24-hour knowledge factory will require new management tools. Office Tiger (now part of R.R. Donnelly Inc., with operations in the US, India, Sri Lanka, Singapore, and the Philippines) has developed a T-Tracks system to enable the fine-grained tracking of work in progress at distributed sites. Wipro, a major offshoring company, uses the Veloci-Q tracking system.

Although these tools are useful, the factory paradigm requires more comprehensive support than merely tracking work. Between 2000 and 2004, an MIT research team explored the feasibility of such support. This work resulted in KnowFact, a concept demonstration prototype that analyzes the capabilities for automated capture of evolving knowledge. The prototype successfully assimilated the evolving understanding of design requirements, as well as the history of experience and behavior. With rapid access to such knowledge, test subjects (system users) increased their productivity—in some cases, by almost a factor of 10.

With minimal user intervention, KnowFact captured information on evolving code, as well as associated decisions, and it could handle frequent handoffs. The next challenge was to broaden the collaborative environment so that multiple programmers could work in a sequential mode with automated capture and use information about prior key decisions that various programmers made during code evolution.

Between 2004 and 2008, the research group at the University of Arizona worked on developing MultiMind, a collaboration environment for agile software development in a 24-hour knowledge factory. MultiMind incorporates a chronologically ordered persistent database to collect objects—encapsulated knowledge about the development
Traditional software development techniques strongly emphasize recursive decomposition until every unit breaks down to a module or class. Ideally, each module then has one assigned owner, who strives to improve the module until its actual state matches its expected state. In most cases, one person per module is the norm, although some practitioners have advocated otherwise. Consequently, module owners must constantly interact, and collaboration time can be formidable. Two decades ago, Tom DeMarco and Tim Lister estimated that time to be as much as 70 percent of the total work time for software engineers, and although the medium has changed, communication overhead can still come close to this figure in an array of programming situations.

Communication support

To address this asynchronous communication lag, MultiMind uses a composite personae (CP) structure, which defines a highly cohesive micro-team that possesses both individual and collective attributes simultaneously—a kind of minicorporation. The CP notion is based on the realization that it is not practical to decompose a problem horizontally (assume single ownership of a module) past a certain threshold.

Thus, in the CP structure, the developer role is virtual: The outward appearance is of a single entity with a unique identity that owns and develops work artifacts, accumulates project performance metrics, and so on. But in reality, a CP has multiple members—at least one at each development site—and an individual can be a member of multiple CPs. The CP driver is the member at the currently active development site and the one with complete control over the CP’s identity and allocated resources during that time.

The CP Mercury in Figure 3a, for example, spans three development sites: Tucson, Arizona; Sydney, Australia; and Wroclaw, Poland. While the sun is shining on Tucson, Tom is driving Mercury. As the day goes on and Tucson ends its day, Tom transfers control of Mercury to Molly in Sydney. Likewise, Molly drives Mercury until her shift ends, and she passes control to Grzegorz in Wroclaw.

Figure 3 shows the three possible communication patterns within a CP and between CPs. In the handoff, information flows within the CP and around the clock from one site to the next—from Tom to Molly to Grzegorz. Lateral communication occurs between the drivers of two or more CPs that are colocated and is common in resolving a conflict in the module interface. Lateral communication is rapid and informal, since both developers use the same language and are of the same culture. The third communication pattern, lateral + handoff, recognizes that not all problems are solvable in a single shift. In this pattern, new CP drivers come online to continue working on the
problem. At present, MultiMind supports a combination of handoff and lateral communication and encourages conducting lateral + handoff transactions as a sequence of the two simpler patterns.

The CP structure fosters a smooth transition within a CP and across CPs, maintains trust among development peers, enables faster convergence on the overall goal, and provides greater resilience against sudden changes in module ownership. The problem of communication overhead will become less severe as technology improves. Our research group has already developed EchoEdit, a supplementary tool that integrates code documentation and voice recognition. Developers can record a comment, play it back, and convert audio to text for indexing and rapid access. Programmers can provide annotations in text, audio, or a combined format. After a handoff, the new CP driver can play back the recording by selecting an icon displayed in the margin.

**Quality and productivity support**

Most software processes aim to improve quality and productivity, measured in terms of cost, effort, delivery schedule, and defect incidents. Core to both quality and productivity are task-time estimation, work allocation, knowledge transfer, and defect control. However, existing estimating tools assume single ownership of code (horizontal decomposition). To support developers working in the factory paradigm, our research group created CPro.

To support task-time estimation, CPro encourages developers to provide their respective time estimates for all subtasks. A schedule caster then executes a Monte Carlo simulation to determine how the project should evolve. As the subtasks proceed, the simulation compares actual times with their respective estimates so that developers can realign and reassign subtasks as needed.

CPro can assign work according to either random- or selected-work assignment. In the selected-work assignment algorithm—the more sophisticated of the two—CPro first arranges all open tasks in increasing order of complexity. When a developer selects a task, CPro generates a simulation of the effort required to complete it. If the developer cannot complete the task by the end of that shift and is unwilling to work overtime, CPro makes the task available to another developer. If the originally selected developer intends to complete that task, CPro locks the task to prevent its assignment to another developer.

To support defect reduction, CPro incorporates a three-pronged strategy:

- **Periodic artifact review.** Within a CP, peer reviews aim to improve software quality and promote communication and the understanding of the module being reviewed.
- **Test-driven development.** CPro validates module development in various phases using test cases that the team creates to evaluate and debug modules. Because peer reviews take place inside the CP, the test cases are a form of communication that helps keep the CP coherent.
- **Heuristic assignment.** CPro incorporates heuristics that optimize knowledge sharing within the CP as a by-product of task assignments.
TOWARD A GENERAL INFRASTRUCTURE

So far, applications of the knowledge factory paradigm have focused on software development, but the paradigm can benefit other professions, such as transportation. Much work remains, however, to develop a generic computer-based infrastructure that will be applicable across diverse industries.

Knowledge-based approaches

Collaboration across time and space requires novel knowledge-based approaches that will accommodate continuous communication and knowledge sharing. One approach encompasses knowledge acquisition through traditional information sources, in conjunction with knowledge discovery, management, and dissemination. Knowledge acquisition provides participants with accurate and comprehensive material; knowledge discovery requires continuous and automated mining of multimedia in the capture of underlying knowledge. Knowledge management tools can address the heterogeneities that invariably exist when input must cross geographic and temporal borders. Finally, knowledge dissemination techniques can aid in extracting, customizing, and directing knowledge to appropriate destinations and users.

Task breakdown

A generic infrastructure will also need ways to split tasks into well-defined components that a professional can execute without having to know all the idiosyncrasies of other tasks and how to assemble tasks into a work product. Individual components will not necessarily be following a standard, unlike physical parts and assembly-line subsystems. The combination of software components can take many forms, and development teams often do not need to fully delineate how a component will fit with other components.

Business rule recording

Given the diversity of professions, business rules naturally have dissimilar formats and intersect in dissimilar ways within an information system or business process. Designers must capture any rule change in multiple systems, and this is particularly difficult with a globally distributed team.

Untangling business rules means unlocking the ability to represent specific elements of business knowledge only once in a knowledge repository. The repository then becomes the basis for designing specific knowledge elements that will naturally manifest themselves in appropriate forms. Structured programming, reusable code libraries, relational databases, expert systems, object technology, computer-aided software engineering tools, code generators, and computer-aided process engineering tools are frameworks that have attempted to create such a repository. A more recent approach focuses on knowledge reuse by recording comprehensive information of the particular business, untangling the underlying business rules, and then enabling the re-creation of many key business processes without the need to define them from scratch.

Reuse frameworks

Generalizing the knowledge factory paradigm will require better frameworks for leveraging and reusing knowledge. Members of geographically distributed teams must reconcile differences in concepts and terminologies; problem domain understanding; and training, knowledge, and skills.

The solution must be an architectural approach that normalizes knowledge and provides reusable artifacts in an accessible knowledge repository. Identifying knowledge components requires distinguishing two kinds of assertions: ones that when divided lose no information and ones that when divided will sacrifice meaning. A division sacrifices meaning when it causes the loss of information that is not recoverable after pieces are reassembled. Rules that are impossible to break down further without irreparable information loss are indivisible or atomic rules or irreducible facts.

The Object Management Group is developing the Ontology Definition Metamodel, which aims to link Common Logic and the Resource Description Framework (RDF)/Web Ontology Language (OWL) with a model-driven architecture. Common Logic is an International Standards Organization standard designed to facilitate knowledge and information exchange in computer-based systems. The evolution of these metamodels for specifying knowledge will encourage knowledge reuse, which will make it easier to deploy the knowledge-factory paradigm.

MultiMind has served as a conceptual architecture for identifying next steps. Our research group plans to incorporate more new tools to create a common development environment for commercial implementation. Augmentations to the current-generation development environments, such as IBM’s Jazz, are likely to provide additional capabilities. Jazz, developed as an extension to the Eclipse development environment, offers several capabilities that can support the knowledge factory environment. Our team has already developed a plug-in that integrates discussion forums into Jazz, and EchoEdit is suitable for the Eclipse environment. With these continual improvements, most companies will soon be able to implement the knowledge factory paradigm with minimal effort.

Organizations that adopt the 24-hour knowledge factory model stand to gain a considerable productivity and
market edge. The model integrates contributions of key players from around the world, enables all personnel to work during the daytime in their respective countries, avoids the health hazards that characterize working the graveyard shift, and reduces overall development time. The use of personnel from different continents makes it more likely that the final products or services will have greater global appeal. Having each collaborating group act as a peer auditor can lead to overall quality gains. Finally, the factory paradigm is robust: If one center becomes suddenly inoperational, the other centers can continue work.

Acknowledgments

Thanks to Nathan Denny, Shivram Mani, Satwik Seshasai, Igor Crk, R. Sheshu, M. Swaminathan, Jamie Samdal, Kate O’Toole, Tapasya Patki, Ben Chen, and Sarah Trainor for their valuable assistance. Thanks also for the valuable input and assistance provided by Computer’s reviewers.

References


Amar Gupta is the Thomas R. Brown Endowed Chair of Management and Technology and professor of entrepreneurship, management information systems, management of organizations, and computer science—all at the University of Arizona. He is also a visiting professor in the College of Engineering at the Massachusetts Institute of Technology for part of the year. Gupta received a PhD in computer science from the Indian Institute of Technology, Delhi. He is a senior member of the IEEE and a member of the ACM. Contact him at gupta@arizona.edu.

Call for Articles

IEEE Pervasive Computing seeks accessible, useful papers on the latest peer-reviewed developments in pervasive, mobile, and ubiquitous computing. Topics include hardware technology, software infrastructure, real-world sensing and interaction, human-computer interaction, and systems considerations, including deployment, scalability, security, and privacy.

Author guidelines:
www.computer.org/mc/pervasive/author.htm

Further details:
pervasive@computer.org

www.computer.org/pervasive