Chapter 1
Introduction

Software – Yet Another Knowledge Industry

Knowledge industries such as electronics, space, pharmaceuticals, or software are special. On the surface, they're the hotly-argued-upon backbone of the new economy, a concept that's no longer new. In our opinion, it's the *approach* to business that makes the difference, rather than a company's niche or age. Some old-economy veterans, such as global-automation vendor ABB, have rapidly expanded their R&D initiatives and resources, employing many more IT specialists than many so-called new high-profile IT firms. IT provides a foundation to a variety of current business ideas, including customer-driven manufacturing where a web customer configures the product or even the software guiding an industrial robot in manufacturing the chosen customized product.

Obviously, knowledge industries are more special under the shell than at this slightly superficial mass-media/thematic level. On one hand, they have business processes similar to other industries but, on the other hand, production/operations is a small part of any business dominated by R&D and by marketing the know-how of that organization.

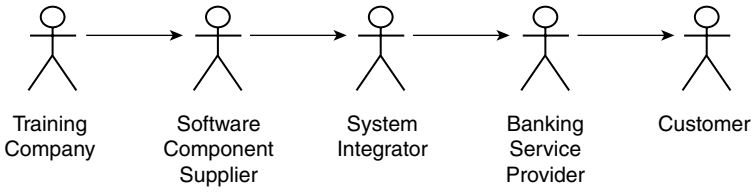


Figure 1-1 **A possible knowledge industry value chain.**

Awareness of knowledge-industry specifics is a project-time saver, both within the software industry itself and with the rapidly increasing number of its customers in the other knowledge industries. Knowledge industries are often interleaved with traditional industry sectors – today, you find computer chips and software in all the flagships of industrialism, from heavy trucks to railways. But, in a high-tech region, the complete knowledge-business value chain can sometimes grow remarkably long without any *tangible* ("hard") products whatsoever (Figure 1-1). For example, your customer might be a training company, whose customer is someone selling tools and methodology to a software house, some of whose customers provide Internet banking to e-traders, others providing sales configurators for customized insurance packages, and on it goes; sometimes, all the tangible hardware might seem to be produced on some other planet. Nevertheless, whichever the surrounding corporate culture or age of the enterprise, its IT parts must be considered a knowledge industry.

Classifying the Knowledge Industry

Figure 1-2 shows a kind of classification, pioneered 15 years ago by Karl-Erik Sveiby's team,[1] which makes us aware of the climate in our firm or project by starting from the extremes:

- A traditional *office:* a lack of real organization, of explicit common objectives, of know-how. Professor Parkinson's Laws apply. For example, an office of more than 150 employees doesn't need any external input because of generating its workload itself!
- A traditional *factory:* traditionally a hierarchy. Even in a modern factory, there's more focus on processes, work instructions/procedure

[1] Visit this Swedish-Australian writer and pioneer of knowledge management at *www.sveiby.com.au*. Books include *Managing Knowhow,* by Sveiby and Lloyd (Bloomsbury, London, 1997) and *The New Organizational Wealth,* by Sveiby (Berrett-Koehler, San Francisco, 1997).

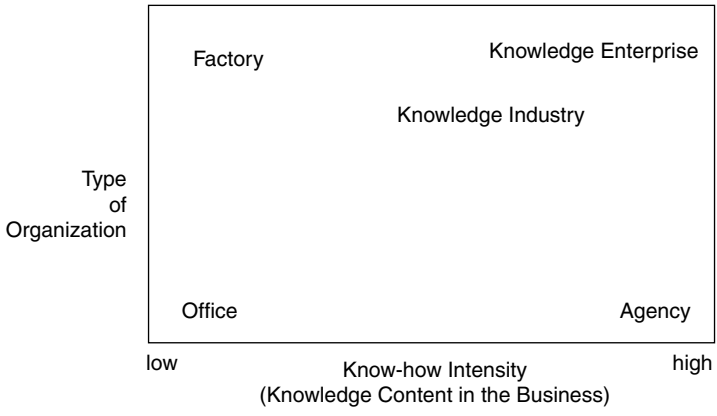


Figure 1-2 **Where is your corporate culture?**

steps than on creativity. In the past, the personnel were roughly supposed to take their hands with them in the morning – leaving their heads at home.

- An *agency:* creativity in an organizational chaos. Everyone is working hard and loves it – forgetting about surroundings, lunches, and colleagues. Anyone who becomes a burnout is considered an admirable role model.
- A *knowledge enterprise:* expertise combined with a common vision, structure, and cooperation. A knowledge enterprise solves complex problems of customers, while a service enterprise solves simple problems with appropriate repeatable procedures.

As you can see, no organization fits into any of the previous cartoons. The engine of the global economy is a gray zone that we prefer calling the knowledge industry: firms or projects that package their know-how into well-defined products and procedures, yet stay knowledge-intensive. Here, component-based approaches boosted by standards are the engine in most improvement efforts.

Consequences of the Knowledge Industry

Know-how intensity has some important practical consequences.

The production process becomes a *packaging machine* for the realization of the know-how, for example, the pharmaceutical factory for the know-how of R&D specialists. It must not fail, so all bottlenecks are banned, but these

production costs are pennies compared to the acquisition and development of this know-how.

The silicon chip, the medical pill, the software CD, or the download-site is a *wrapper* for the know-how. We don't buy pills by weight. We pay for the expected improvement instead, no matter how it's packaged. Similarly, buying software by kilo-lines, (kLocs) of code doesn't make much sense. We pay for the expected business improvement, no matter the amount of new code or reused components. As the Object Management Group (OMG) points out, modern software projects avoid writing all the code for the programs. In other words, they reuse more infrastructure parts, off-the-shelf software components/business-oriented components than traditional projects do. In knowledge industries *know-how is the real thing,* whereas the wrapper is hardly relevant.

Unlike traditional mass production, the competitive edge isn't in the workflows of production/operations/administration, but in the mechanisms of *sharing and processing know-how* across the firm. Therefore, a traditional mechanical Business Process Reengineering (BPR) approach tends to solve the wrong problem when applied in a knowledge industry because it's focusing on the basic activities, without considering the complexity of the business logic in that activity.

The Asset Paradox

When the main asset of the firm is knowledge, then the trick is to stay fairly independent of individuals by turning a knowledge enterprise into a knowledge *industry.* This requires storing more knowledge in a format accessible to as many co-workers as possible, most often using computers. The labor market is simply a market. Therefore, even in a rather holistic bookkeeping approach, the (fancy) knowledge-asset figures must be adjusted by a factor reflecting their infrastructure, structuring, standardization, methodology, component-sharing, and so forth.

Acquiring and keeping unique knowledge is key in a clear-cut knowledge enterprise, whereas in a knowledge industry, the *structure* of the know-how – and the *infrastructure* used in keeping it current and in feeding it through – is as important as the know-how itself, a fact deserving attention from both knowledge managers and financial analysts.[2] Typically, a knowledge enter-

[2] Estimating/forecasting high-tech shares has been hotly argued since the 1980s. Focusing on knowledge structure and infrastructure is less fuzzy than trying to quantify pure knowledge. We hope to see less roller-coaster rides on NASDAQ in the future, as tech shares become less volatile when all such factors are thoroughly worked through and taken into account by analysts, ahead of IPOs or mergers. As we show in Chapter 6, configurable components can boost sales activities as well, by enabling a closer and cheaper match between bids and specific customer needs in a variety of niches.

prise *sells* knowledge, whereas a knowledge industry *sells its capability to apply and deploy its knowledge* packaged as, for example, software.

Sharing the Knowledge

Given all these specifics, the efficiency of specification and development activities is extremely important in any knowledge industry. The toolkit of improvement is all about knowledge sharing by:

- Standardizing the terms and the notation
- Practicing a common approach
- Sharing pretested components

UML has standardized the terms and the notation by providing a set of diagrams with a defined syntax. Unlike other knowledge industries, software can't be expressed by drawings or photographs of some spatial/physical, musical, biological, or chemical properties. Even under a fancy microscope, software stays *invisible and intangible.* A software-blueprint isn't as intuitive as a land map showing ice in white and water in blue. Rather, it presupposes a general industry-wide agreement in the first place, on the agreed meaning of every single symbol or relationship.

This makes us extremely dependent on a standard notation for any software-related communication and specification, all the way from a project developing a system from scratch to one selecting an off-the-shelf package. As development projects become increasingly global, UML also helps those of us communicating in our second or third language. For example, IBM development labs are located in dozens of countries, each with its own native language or languages, or the new Airbus Superjumbo involves industries from most of Europe. Atop of that, all natural languages include some natural ambiguity.[3] All things considered, word processors aren't enough as a tool of specifying requirements.

Practicing a common approach or method framework across projects, supported by a regularly upgraded knowledge aid, such as online mentors, built-in hyperbooks, or intelligent checks in a PC-based tool (a UML case tool), is knowledge sharing in a narrow sense. With cheap tools, we simply *access* the expert knowledge of others (typically, using standard search engines and hyperlinks) whereas, with automation tools, we can even *run* it

[3] A fact easily "rediscovered" while we're writing this book and asking others to read our first-draft text.

on a computer, and then simply access the results of the run (or let the computer use them), be it calculations or a more qualitative business logic.

We share pretested components across the firm and across the software industry**.** This kind of "canned know-how" from colleagues is a superior stage of knowledge sharing – we can activate the result right away, without ever acquiring the know-how that created it. This component that encapsulates the expert's knowledge and experience is kept up-to-date by the expert, leaving all the other developers free to concentrate on the business solution. This is an effective technique and a rather down-to-earth one when contrasted to preaching knowledge management at a thematic level. As we point out in the Chapter 6, this degree of automation can be increased further by smart configurator tools in the near future.

By and large, we encourage IT teams to exchange and adopt best practices from other sectors of industry. That said, we recommend *knowledge industries as sources* of ideas: many Business Process Reengineering cases and books described processes with a low-to-medium knowledge content, hardly applicable in the context of software specification and development. Although the bottom line might look deceptively similar, the devices and the activities generating that bottom line do differ, and those differences may be significant.

Sharing the Responsibility for Getting It Right

Even the buyer, the reengineer, or the process owner is involved in specifying and improving requirements throughout the project. In any knowledge industry, the customer and the vendor *share* this responsibility. Here, "the customer is always right" translates into "the customer always has the right to get the *right solution to the right problem*." If you go to your car dealer and order a thirsty six-wheel-drive monster for driving from home to a job just around the corner, your dealer might laugh, as Figure 1-3 shows, but he offers and sells the monster to you anyway. On the other hand, if you try

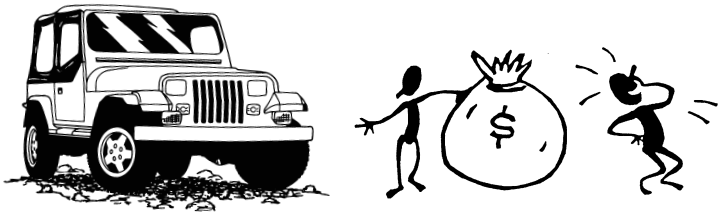


Figure 1-3. **Some simple old approaches** to customer requirements don't count in a knowledge industry because a shared responsibility exists for the specification and its fit-for-purpose.

something similar in a knowledge industry, a serious vendor will raise strong objections on the mismatch between the business and your requirements because of this shared responsibility.

Sharing responsibility across the negotiation table involves *communication* at a pretechnical high level, as does sharing know-how within a team. Having combined rigor with easy-to-learn diagrams, UML has proven to be an excellent common IT language. UML is an unrivaled smorgasbord[4] of diagram ingredients matching a variety of needs. In business modeling, the stakeholder or the buyer works closely with the project team, gradually transferring work to the IT staff members as we move on (iterate) through the full system-development cycle.

A standard notation (or modeling language) greatly reduces ambiguity throughout the project.[5] This is important because ambiguity is a major source of confusion. You say the same thing, which is understood/reacted to in different ways by the listeners. For example, the clear statement "secure the building" will cause the Marines to form a taskforce and storm the building, a legal department to negotiate a long lease on the property, and the security experts to install and manage an access control system.

A good analogy exists on being multilingual. Milan speaks Swedish in Stockholm or Czech in Prague, just as you're fluent in your business language, be it in reinsurance, meteorology, switching, billing, or train control. Methodology experts or developers of a UML-tool understand UML at this level of detail, that is, all the diagrams' types, syntax, and rules. Milan can also speak a "standard language" – English – in frequent areas such as software, but not in areas like bug species (except software bugs of course).

Most software developers understand UML at this standard level.[6] UML resembles a grammatical language, such as Spanish or German, because of its predefined syntax and semantics. Nevertheless, we approach it in quite an idiomatic, example-based manner as common with today's English. With

[4] Usually translated as "Swedish table," a large table of ready-made dishes located in the middle of a restaurant, where the guests choose and pick their preferred combinations and quantities themselves, and then eat at their restaurant-tables.

[5] Language and reasoning are closely interrelated. As UML pioneer Dr. Ivar Jacobson points out, IT people used to think as humans until attending computer science classes at the university level, where they learn to think as computers (i.e., sequential Von Neumann machines splitting the world into data values and procedural instructions, which are poorly, or hardly, interrelated). UML provides the language necessary for reinventing the natural, human way of reasoning in the context of software systems. You can view it as a set of well-defined, preshrunk, standard mind maps that are useful to both the project team members and the software development tools to be used in the project.

[6] Typically, they also provide UML guidance to others throughout a project. The IIIE's list of software requirement qualities implies a cooperation here, stating that requirements shall be *unambiguous,* complete, correct, *consistent,* traceable, modifiable, *understandable,* verifiable, and *ranked* for importance and stability.

this language metaphor in mind, we found several good *Webster's* dictionaries are around for UML (addressing the "native"), as well as an extensive English course book or three (addressing the ambitious "guest scientist from abroad").

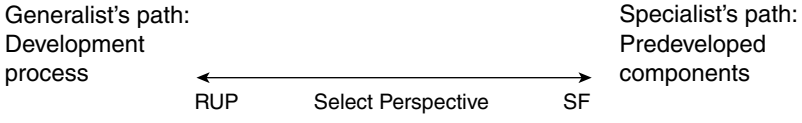
The missing link so far was a tour book on the language, accessible to many "frequent visitors" in the landscape of software projects. This tour book needs to fit in a lightweight cabin bag and be reasonably comprehensible, even during jet lags. From our customers, the pressure was on as well – so we wrote one.

Overconsumption of languages is excellent for brains, overconsumption of standard notations is far from excellent for a project approaching delivery deadline. With the smorgasbord principle in mind, let's pick up what we want and skip the cookies. If you're a software specialist, you'll soon read deeper books anyway.

Methods and Processes

UML standardizes the system documentation independent of how you produce it. Methodologies, on the other hand, are paths to take you from the problem to the solution and, during that journey, deliver the relevant UML diagrams.

UML provides diagram notations for most kinds of applications, so it works with all up-to-date methodologies, that is, with a component-based approach. Nevertheless, various practical methodologies are based on various ambitions and priorities. Some organize the overall problem-solving activities within a project – the cookbook approach – whereas others provide more how-to and the ingredients for the problem solver – the toolkit approach. Likely, this scale looks familiar to most readers who are specialists in non-IT areas. Of course, you can combine both ends of the scale in the same project: the UML notation works fine. Let's briefly compare three approaches in the following:



The Rational Unified Process™ (RUP)[7] makes the development process in a software project visible, from inception to deployment. Stressing, step by step, roles (30 kinds of "workers") and responsibilities for 60+ predefined

[7] from The Rational Corporation; visit *www.rational.com*.

BASIC STANDARDIZATION AND CREATIVITY BOOST EACH OTHER!

The recent standardization effort put into UML resembles trends from knowledge industries of the past. For centuries, classical music has been pushing its ubiquitous mix of science and creativity on a global market. We also find standard constructs in the American tradition, from a 12-bar blues to a jazz standard tune. Interestingly, when scaling-up sheer creativity into a knowledge industry, people always try to standardize the basics, to enable a shift of focus from low-level work to the big picture, that is, to *what we do* with the basics.

Unsophisticated music is as old as humanity itself. However, the "Big Art" music of the Western world emerged from extensive *standardization* only a couple of centuries ago. Before J. S. Bach, most churches used their own proprietary scales, some of which were impossible to play on instruments. Also, a tone could be pitched differently in different scales; thus, the same tone was played on different keys of the same keyboard. In cooperation with keyboard vendors, Bach pioneered *standard tempered scales* (major and minor, with standard tone intervals), enabling a leap in composer work and in interplay of instruments. A century later (W. A. Mozart and the classical period in music), common *architectural templates* already existed, such as a concerto in three movements (the slow one in the middle) or a symphony in four movements (the two slow ones in the middle, the latter of them a minuet.*) Similar architectural rules also governed the structure within each movement. A de-facto standard guided staging appropriate numbers of appropriate instruments in an orchestra, which gave the composer the necessary hints upfront in "design time," while composing the music – regarding the hardware to deploy the music later, onstage. As musicians were always borrowing-extending-reusing jerks and themes invented by someone else, even what we now call a *component approach* became frequent in the beginning of the classical period. For example, in large divertimentos, an evening or event was configured from a small "library" of ready-made components (movements). This greatly simplified and streamlined the requirement specification, yet matched the preferences of that particular evening's sponsor.

The long-term focus on Mozart in most creative professions** creates a major obstacle for a minority of programmers still trying to claim "no standards and no components, please – this is creativity." Long-term experience from other knowledge industries indicates exactly the opposite: extremely creative individuals benefit from architectural standards and components.

---

* To be exact, Mozart's Prague Symphony is the widely known exception to this rule because it omits the minuet movement (according to the BBC's "Best on Record," some 80+ recordings of the symphony exist worldwide).

** Many readers might remember Milos Forman's film *Amadeus* or Ingmar Bergman's *Magic Flute,* or several BBC documentary films on Mozart's music (among others). The creativity dimension was recently explored by Don Campbell in his book *The Mozart Effect* (Avon Books, 1997) and his CD-production, *Music for Creativity and Imagination* (Spring Hill Music®, 1997). In arguing that history repeats itself, we've also checked facts with Jiří Kratochvíl (Milan's father), a woodwind history expert at the Prague Academy of Music (see Pamela Weston: *Clarinet Virtuosi of Today,* Egon Publishers Ltd, 1989).

kinds of artifacts, RUP is a process framework suited for large projects, roughly of 70 members or more, with a large number of components to be constructed. RUP also outlines splitting the project into use-case-based (see Chapter 3) miniprojects, some running in sequence and some in parallel, in several iterations. Because RUP is distinctly use-case driven, some strengths and limitations of use cases affect the process itself. For example, a data warehouse/data mining or knowledge-based system implies hard work inside the system, despite rather simple external interaction, whereas use cases are easy to apply to telecom switching or to order handling, where a much larger proportion of external interaction (often with end users) takes place.

To a potential user of the process, we strongly recommend acquiring a thorough knowledge of UML to ensure the right aspects are dealt with in the right documents (artifacts). Providing guidelines from the requirement specification all the way to test, the process has become rather heavyweight, which implies some extensive process customization to start with to make the process fit the purpose. This customization needs to be done in two steps: first, for the enterprise, and second, for the project. In some 4,000+ web pages, this process framework defines roles, artifacts, work flows/activities, and project management.

IBM's WebSphere® Business Components,[8] an application framework previously known as the SF (for San Francisco or Shared Frameworks) is, on the other hand, a wholly *component-driven* approach. IBM supplies off-the-shelf, pretested components, books, best practices, and instruction to solution suppliers who target customers requiring e-business, CRM, and ERP packages. Thus, SF is a component *framework* for application projects – large or small ones – typically employing more reused pretested components than new ones. SF motivates the doers rather directly: here we have a box of software Lego bricks and the directions for use, so let's go ahead.

SF's strengths and limitations are typical of a specialist's method. Such methods are precustomized for certain systems – in SF's case, the closer to ERP/CRM/e-business, the more useful it is. We hope similar complete frameworks will also emerge in some other niches. By shrinking development timescales, SF guides projects into smooth construction work: more assembly, less programming. As senior developers at Swedish ERP-vendor IBS[9] as well as their R&D Manager and Vice President Tomas Bräne points out, having found a couple of appropriate SF components, a day might sometimes be enough to develop a sophisticated "new" one.

[8] from the IBM Corporation; visit *www.ibm.com/software*.

[9] At the end of 2001, IBS is ranked third in the world by AMR Research, and Frost & Sullivan in the field of supply chain management (visit *www.ibs.se*).