CHAPTER

Practices of Software Architects in Business and Strategy—An Industry Experience Report

7

Michael Stal
University of Groningen, Groningen, The Netherlands

"If you think, good architecture is expensive, try bad architecture"
—**Joseph Yoder and Brian Foote**

7.1 Introduction

The product portfolio of Siemens, a company with over 400,000 employees worldwide, is quite diverse. It includes trains, railway control systems, health care systems, factory automation, power transmission and distribution, among many other products. These products represent embedded systems. In recent years, they moved from isolated islands to connected distributed real-time and embedded (DRE) systems that were integrated with an increasing amount of software tools and enterprise applications.

Only a few decades ago, software was considered a low value add-on to the hardware. Today, over two-thirds of all Siemens revenues depend on software. As a consequence, software development has become an important economic factor for embedded systems. Any failure or delay in software development does imply significant financial liabilities. Consequently, the success of mission-critical system development projects is essential for economic success. As software architects are responsible for designing the strategic backbone of such systems, they must be aware of economic constraints and business goals and take these factors into account in all their decisions.

Some organizations consider software architects as advanced software engineers with a high-technology bias. From their perspective, architects do not require expertise in business and strategy. Even software architects themselves tend to believe they are only responsible for technology and design decisions, but not for economic aspects. This attitude leads to solutions that are technically sound but fail to deliver the expected return on investment. If software architects do not understand economics and the business, they cannot come up with economic solutions that support the business case and business strategy. This is why mission-critical projects often do not achieve their business goals, thus incurring high additional costs. Although this holds for other engineering disciplines as well, the high flexibility expectations regarding software and its creation increases the economic risks.

Economics-Driven Software Architecture. DOI: http://dx.doi.org/10.1016/B978-0-12-410464-8.00007-6

This chapter provides war stories and practices for coping with the economic challenge software architects face. It is not based on empirical scientific studies or conceptual research, but on experiences of our company. The experiences were systematically analyzed and collected from existing and historical projects to identify common causes of failure and to extract guidelines for software architects to learn from failure and avoid it in their own projects. Being used for several years now, a framework of practices comprises the recommended process and collaboration patterns for architects to help them ensure economic success.

7.2 Identifying economic guidance for software architects

Within Siemens a team evaluated mission-critical software development projects to understand the responsibilities of software architects in development projects to achieve business and technology goals in an efficient and effective way. The analysis was the first step in setting up a companywide senior software architects certification program (Paulisch and Zimmerer, 2010). Afterward the evaluation guidelines were defined to clarify the role of software architects in business and strategy. These guidelines introduce the responsibilities, interactions, and activities of architects in the different project phases that are considered essential for project success. In additional workshops, the guidelines are also taught to managers such as heads of R&D and product (life-cycle) managers.

The project evaluation phase revealed that some of the software architects did not know the relevant business context of their projects, at least not in sufficient depth. Many software architects assumed they knew it, but actually they only understood it vaguely.

As a consequence, some large and mission-critical projects failed to achieve business and strategy goals and could not deliver on time and on budget. In a postmortem analysis, communication between software architects and product, project, and senior management often turned out to be inefficient, insufficient, and ineffective. However, mutual interaction between stakeholders is the prerequisite for creating economic and sustainable solutions that fit into the business and strategy goals of an organization. From this perspective, insufficient communication is the root of almost all technical and economic problems. Software architects need to know the economic constraints and forces in order to come up with architectural decisions that are in sync with the business case.

7.3 Structure of the chapter

In the first section, we introduce some considerations on economics-driven software before addressing the software architect's perspective. Succeeding sections in the first part of the chapter cover general facets architects should be knowledgeable about such as business context understanding and business strategy and planning. Succeeding sections address the involvement of architects in product and technology planning, cost estimations, and the treatment of legal issues and regulatory bodies.

In the middle part of the chapter, we show the responsibilities and activities of software architects during system development. This includes not only design activities, but also responsibilities in requirements engineering, testing, implementing, and integration. A coarse outline of a

systematic process for architecture design is also introduced. The next sections emphasize specific development contexts such as software product lines and embedded systems.

In the last part, the chapter explains why communication among stakeholders is a success factor for development projects and provides some final conclusions.

7.4 Considerations on economics-driven software architecture

Software architecture is the main step in mapping the problem space to the solution space as it covers strategic aspects such as Quality of Service as well as major tactical aspects such as reuse and modifiability. Tactical design, that is, fine-grained design, depends on this architectural backbone. As a consequence, each architectural decision has an impact on costs. To enable economic system development, architects must explicitly consider economic issues in all process phases and disciplines.

For driving economic system design, software architects should be actively involved in requirements engineering. In particular, they need to elaborate the economic soundness of the specification. Some requirements might stay in conflict with each other or might be economically unfeasible. For example, if many quality attributes or functional requirements are rated highly, it is difficult or even impossible to come up with economic software architecture. Quality attributes such as efficiency and exchangeability lead to trade-offs between minimizing and maximizing the number of indirection layers, which results in complex systems that are hard and expensive to maintain or evolve.

At Siemens, software architects are required to enforce a unique prioritization of requirements jointly with product managers or customers to avoid such problems.

Unit costs are another economic aspect we encountered in projects. If an embedded operating system license costs $50 per device, it is impossible to achieve a unit price less than $50. Thus, technical decisions such as the integration of third-party (Common-Off-The-Shelf (COTS)) must take care about economic implications. For this reason, architects are involved in make-or-buy decisions as well as outsourcing or offshoring considerations. Availability of a second source for most COTS components is important to minimize dependence on suppliers. Likewise, architects are in charge of deciding whether inner or outer open-source components might offer economic benefits such as cost reduction. On the other hand, open source bears the risk of hidden patent violations and licensing models that would require a company to reveal some of its business secrets. In a value-based software development, organization architects need to address these risks.

War Story: In a project for developing a medical therapy system, the integration into the medical information systems had to be accomplished using software from a specific vendor due to a mandatory customer requirement. However, the supplier was also a competitor. All adaptations of the software took months and millions of euros even for simple changes.

Software patents are of crucial value in the industry. They help to protect the business of a company, and they are important assets when negotiating cross-licensing contracts. Patent portfolios can reach values of millions or even billions of U.S. dollars. Software architects are responsible for recognizimg innovative concepts early in their projects, especially those that should be subject to intellectual property rights.

To mitigate risk early and to obtain early feedback, architects at Siemens create the architecture using a piecemeal approach, starting with the most important use cases and quality attribute scenarios. An incremental and iterative approach ensures that in case of budget cuts or time delays the system can meet important requirements but leave out less important features and qualities. Providing early feedback also helps check the economic feasibility of the project. For example, if the first increment takes much more time than estimated, then it is very likely that the succeeding increments were also subject to wrong cost and time estimations.

Strategic design, that is, functional design and design of operational quality attributes such as safety or efficiency, should always precede tactical design, which comprises developmental qualities such as modifiability or reusability. This is because in order to modify a system or reuse components, the availability of the artifact to be modified or reused is necessary. Extensibility has no meaning without precisely knowing what to extend and for what purpose. Some systems that did not follow these rules were subject to overuse of patterns such as strategy, which made the systems slow and hard to maintain. If the engineers don't know what should be flexible, they might become cautious and introduce variability in many parts of the project, even where it is not beneficial. Binding of variability and configuration in such systems tend to be complex, tedious, and error prone.

Since quality attributes are essential for a product and hence an important cost factor, architects and product management should cooperatively define the quality scenarios, derive the quality tree, and estimate the economic impact as well as the technical complexity of each quality attribute scenario (Bass et al., 2013). Qualities with high economic impact should have higher priorities than those with less economic impact. If two quality attribute scenarios have the same economic value, the one with higher technical complexity is assigned a higher priority. Thus, the order of design and implementation can be driven by economic value and technical risk. Prioritization is also useful for setting up economic risk-based test strategies and test plans. Using a risk-based strategy, testing can focus on the most critical artifacts with adequate tests. By balancing economic and risk constraints, test managers and software architects determine the right amount of testing, and an appropriate test exit strategy.

One constituent of tactical design is to come up with a systematic reuse strategy. Different projects reveal that components that are hard to use are even harder to reuse because the costs for making them usable and reusable might outweigh the reuse benefits. In the most extreme cases, usage of components with bad design for reuse turned out to be much more expensive than design from scratch. Thus, architects need to provide an economically feasible reuse infrastructure. Systematic commonality/variability analysis helps to identify potential reuse candidates as well as their cost. This is an essential activity in all software applications and is of high economic importance in product line engineering.

Reuse within Siemens projects is not constrained to organizational units, but might even be applied in a cross-organization manner. For example, control systems in energy or automation domains are enforced to use the same control systems platform that has eventually become a high economic value. Furthermore, business groups use and reuse the same engineering tools.

Reuse is not constrained to implementation artifacts. Design reuse turned out to be much more important than code reuse in several projects. Within Siemens, certified architects leverage software patterns to re-use general and domain-specific design. Patterns include architecture, design, analysis s, domain-specific s, and refactoring patterns. This pattern-based architecture approach could save up to 40% costs in some previous and current company projects.

To avoid accidental complexity, each design decision must be rooted in an economic or a business reason and abide to the "keep-it-simple" principle. Otherwise, architects might introduce unnecessary design pearls that introduce accidental complexity, that is, oversophisticated design. Oversophisticated design is an important cost factor because it causes unexpressive, complex systems that are difficult to understand, to maintain, and to use.

In order to circumvent the economic pitfalls caused by inadequate design decisions, architects are supposed to conduct an architectural analysis after each increment. The earlier the design problems are detected and resolved, the less expensive is their treatment.

Design problems are resolved by refactoring and restructuring activities before succeeding with the next increment. Sometimes, refactoring of design problems cannot be conducted because of an upcoming product release. In this case, architects should explicitly document the findings, also known as design debt, report these findings and their consequences to the management, and plan later treatment of design debt.

Design debt leads to economic implications such as higher costs when untreated or resolved in a later increment. But it might also be the case that for some minor design problems it is cheaper to keep them unresolved. For most architecture problems, early feedback by architecture analysis and immediate treatment of identified design smells often proved to be much more economic than deferring refactoring to a later increment. If bad design decisions in a top-down architecture design are not resolved, fine-grained design and evolution adds more components and dependencies to these architectural locations. After a while, any refactoring will need to consider all those additional components and dependencies that might increase development costs significantly.

In the implementation phase, architects should also implement but not on the critical path. By participating in this phase, they can encounter problems in the architecture and prevent an architectural shift caused by developers who do not understand or follow the architectural design. Otherwise, architecture shift may have a major economic impact, especially when it is discovered late in the project.

For economic reasons, developer habitability should be a major concern in architecture creation. The better the habitability of a system, the easier developers can understand and use the architecture and the less development costs will be. At Siemens, strategic architecture documents must cover different stakeholder perspectives and be written from their viewpoint and needs in a systematic, prescribed way. Additional project diaries turned out to be useful to document the rationale of decisions that could not have been documented otherwise. Whenever new ineffective and time-consuming discussions pop up in meetings that cover previous decisions, the project diary reveals why a particular path has been taken. A project diary does not need to be a formal document but may be an informal Wiki site.

7.5 The business context of software architecture

If development and product management are strictly separated, this leads to unwanted effects. Literally speaking, product managers may "throw a specification over the fence," and software engineers may "throw their architectural specification or software release back." Each role makes assumptions about their own specifications, respectively, the specifications, responsibilities, and

activities of the other roles. Some of these assumptions turn out to be wrong, incomplete, or ambiguous. When stakeholders do not practice regular and frequent communication and cooperation, there is a good chance for misunderstanding and thus for failure.

War Story: In a telecommunications project, architects thought that high flexibility was the key requirement, while in fact efficiency had the highest priority. After the first version of the base station was released, beta customers complained about the poor performance of the system. Architects and management figured out that the system architects did not regularly communicate with other roles so that nobody recognized the divergence between the specification and the actual design. It took the development team more than one year to solve the problem.

Hence, the prerequisite for architects in a development project is *to know the economic and the technical context of software architecture*, that is, the products an organization is going to build, the intended markets and customers, the business strategy of the organization, as well as the competitors (see Architects, 2008). To retrieve this knowledge, architects must communicate tightly with business roles.

Product management (see Geracie, 2010 and Mironov, 2008) should actively involve software architects in the fundamental process steps of business and strategy planning to avoid such problems. If they don't, architects should feel responsible to proactively obtain this information. Passively waiting for this information is not a viable option.

7.6 Business strategy and planning

Architects need to understand the business strategy of their organization. As a first step in software development, a SWOT analysis (Strength, Weaknesses, Opportunities, and Threats) (wikipedia.org) provides an overview of the *company's current position*.

- *Strengths* comprise various aspects, for example, the leadership position of the organization in the market, the completeness and focus of the existing product portfolio, and the availability of skilled personnel.
- *Weaknesses* may include factors such as the pricing model, lack of market awareness, lack of innovation, and inferior quality of products.
- *Opportunities* such as change of the legal environment, cost pressure of customers, standardization, compliance and conformance requirements, or outsourcing necessities lead to new or improved solutions that address these challenges.
- *Threats* put the market position of the organization at risk. For instance, when new competitors with innovative and cheap products appear, the market volume and, as a direct consequence, the profitability decreases.

From the SWOT analysis management derives improvement measures and actions. One sample measure could be to develop innovative products that attract new customer. A possible action is the search for "excitement features" that help achieve a competitive edge in the market.

Software architects should be knowledgeable about the capabilities of their company, in particular its strengths and weaknesses. They typically are not responsible for finding new opportunities or for defining measures. However, responsible management should involve them intensively in these

activities to obtain valuable feedback and ideas. Eventually, the architects are responsible for creating concrete products from business goals and requirements.

Architects support the definition of a product roadmap (see McGrath, 2000) for information on product strategies) that captures the milestones and steps required to reach a midterm or long-term vision. This comprises the evolution of current products or product releases as well as the creation of new products that are intended to complete or enhance the product portfolio. All new services and features essential for the implementation of the product roadmap are subject to R&D planning (Matheson, 1997). In this context, architects are supposed to provide their technology knowledge by analyzing relevant technology trends and determining possible implications on the architecture caused by new technologies. They should also contribute to effort and cost estimations.

An important deliverable of this activity is *a technology roadmap* (Moehrle et al., 2013) (see Figure 7.1) that synchronizes and coordinates product management with development. The purpose of a technology roadmap is to identify the technologies required for specific product releases.

War Story: In a medical product development, architects did not receive sufficient information on the product roadmap from product management. Creation of the technology roadmap was assigned to product managers as well. When development of the second version started, it turned out that some architecture and technical decisions during version one development had to be completely refactored or modified for version two. Had the architects known the product roadmap a priori, development of version two could have been completed in a much more economic way.

These technologies might be developed by the organization itself, obtained through outsourcing from other companies, based on an open-source software product, or bought from leading technology vendors. Outsourcing activities require architects to cooperate with other cultures. They need training and experience to accomplish this task. Otherwise, lack of coordination and communication may result in delayed and inadequate deliveries and in increased costs.

Which of the technology options are more beneficial depends mostly on economic and strategic aspects. Issues such as the following should be addressed in this context:

- What effort in terms of time and budget is required to develop the specific technology or service?
- Does the component contain innovations or business secrets that provide a competitive advantage, and, therefore, should not be made available to external organizations?
- Does the market provide an existing technology or service that could be easily integrated into own products?
- Is there any open-source software solution available that could be reused to implement the technology and, if yes, are there any obstacles with respect to legal issues?
- In large companies the question could even be: Is there another company that provides the required technology and could be subject to Mergers and Acquisitions? This might be a viable alternative if the technology contains business secrets or innovations.

Software architects are prepared to help answer such questions due to their technology and architecture expertise. They can estimate efforts required for the development and integration of technology as well as its impact on the software architecture. In addition, they can serve as valuable partners for due diligence activities related to technologies and services.

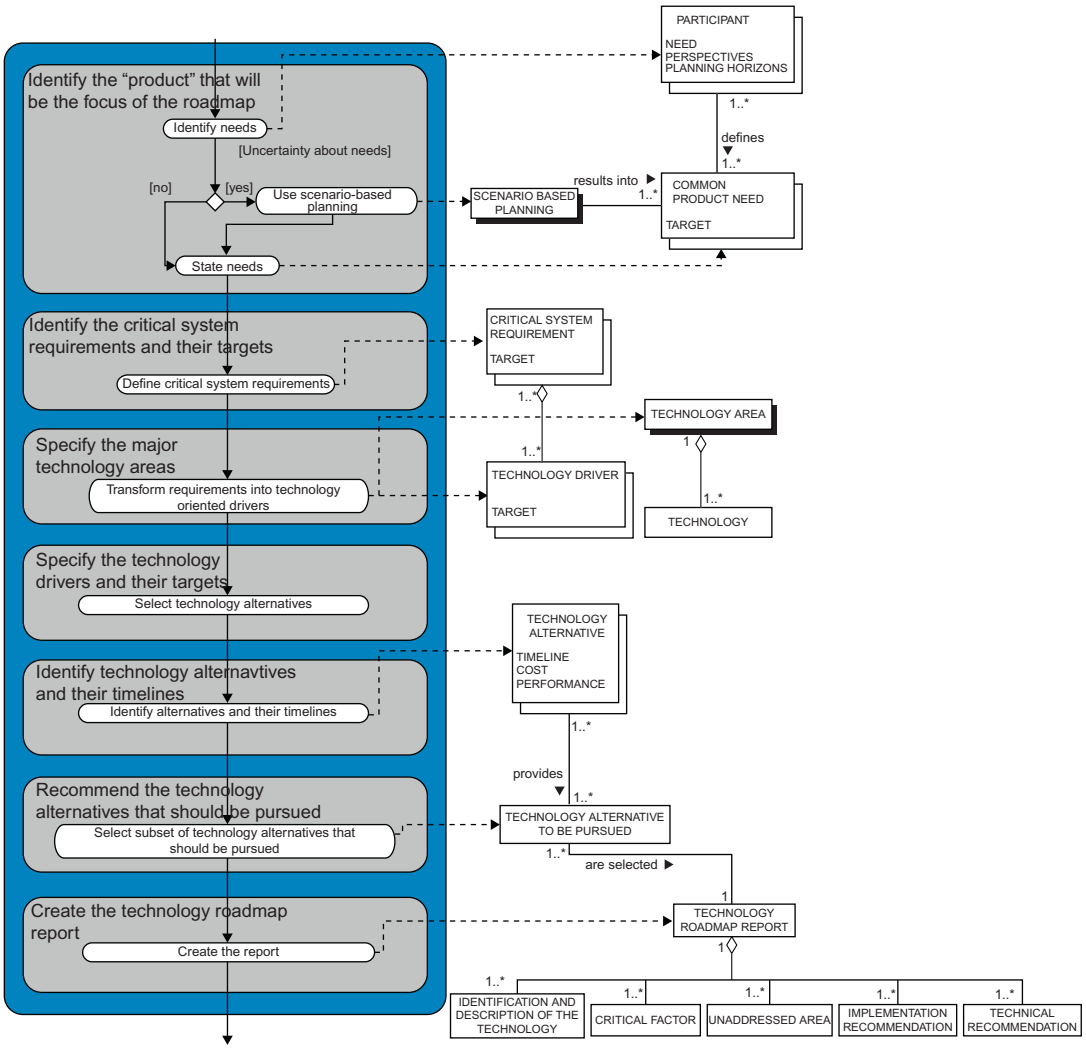

FIGURE 7.1

Extract from the process of technology roadmap creation.

For this reason, software architects should be actively involved in technology vendor assessment and management. Architects help to decide which vendors should become strategic suppliers. They make sure that technology deliveries abide by the expectations and necessities of product development. Their assessment includes the conformance of external deliveries to legal issues and relevant standards. If possible, architects should help identify a second source to decrease dependence on certain suppliers.

7.7 Products—definition and development

Architects are in charge of strategic design for products and solutions. But they also ought to participate in the product or solution definition phase.

For understanding customer expectations, architects and management leverage methods and tools such as KANO analysis (Kano et al., 1984). A KANO diagram (see Figure 7.2) illustrates the basic requirements the overall system must meet. It also identifies the additional excitement features that are not expected by customers, but provide a competitive advantage.[1]

Although KANO analysis focuses mainly on functional aspects such as features, one of the most critical and important facets relates to the definition, implementation, and validation of quality attributes. A product that meets all functional requirements but fails to deliver sufficient performance or robustness may result in customer dissatisfaction. For example, long interruptions of a mail system or website can incur millions of losses. And even worse, a medical device that causes lethal injuries becomes a major hazard for health care providers, patients, and the vendor.

Defining quality requirements and assigning unique priorities to these qualities requires close interaction between architects and business stakeholders. Within projects, stakeholders leverage utility trees and scenario diagrams (Bass et al., 2013) to achieve this goal.

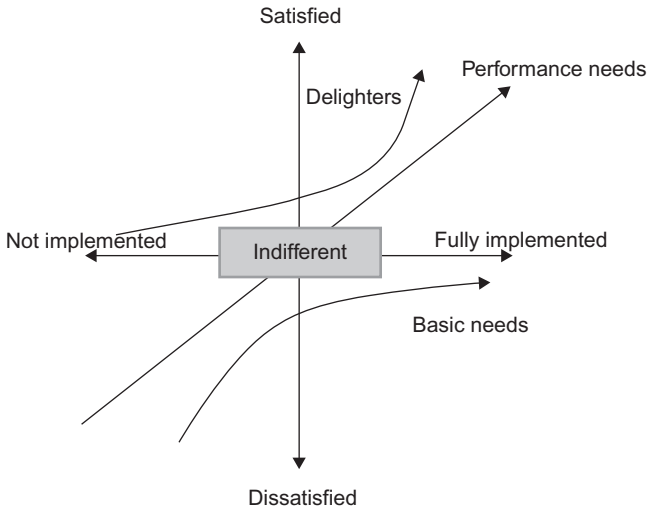

FIGURE 7.2

The Kano model is a theory of product development and customer satisfaction developed in the 1980s by Professor Noriaki Kano. It defines how a product addresses basic needs of users and usage efficiency, as well as the delighters it provides, i.e. properties user do not expect but that delight them.

[1]A well-known example for an excitement feature is the touch-based UI of the first iPhone.

War Story: Product managers of a business application defined flexibility to be one of the key requirements. In addition, they rated almost all other quality attributes as high-priority requirements as well. Two problems emerged: First, the managers had completely different definitions in mind when discussing flexibility, but no stakeholder was aware of the problem. Second, all quality attributes were defined very vaguely, and some of them even conflicted with each other. In the project, the architects hold full responsibility for technical aspects of quality attributes, and product managers were responsible for business aspects. Thus, the product managers did not know the technical aspects, while the architects did not know the business aspects. The resulting system was overly complex and could not meet some important customer requirements. It had to be completely reengineered.

Operational and developmental quality attributes are specified with this kind of diagram by product managers and architects—which additionally fosters common understanding of the business case and the development challenges. The interaction also reduces the risk of incompleteness or inconsistence of the requirements specification. In a further step, architects estimate the complexity and efforts for implementing each scenario, while business stakeholders define the economic relevance of the scenarios. Scenarios with high business relevance and high implementation complexity are then developed first. From an economic viewpoint, design and implementation can focus on most business-relevant and risky scenarios, which helps mitigate risks and reduce costs.

7.8 Cost and effort estimations

Architects are important partners for project and product management with respect to cost and effort estimations. In the Siemens software architect certification program, estimation methods such as COCOMO (Steece et al., 2000) are introduced, including their pros and cons. However, the main focus is on the planning poker (Steece et al., 2000) (see Figure 7.3), a method provided by Scrum. Multiple business units are using Scrum as their preferred process model, or at least they mix-in some Scrum ingredients in their existing development process. Appropriate cost and effort estimations turned out to be of crucial importance in many projects. Because of work package dependencies, a delay in one milestone might cause delays in multiple dependent milestones and work packages.

Architects are supposed to leverage methods like the planning poker to come up with more realistic effort estimations. This method does not calculate costs, but rather combines different cost estimations to a joint and more realistic estimation by different stakeholders and persons.

War Story: In an automotive project for developing an entertainment system, management decided to use Windows CE as the underlying operating system. To increase the development team, they hired developers experienced in the development environment Visual Studio, most of whom had no knowledge of embedded systems development. Management's basic assumption upon cost and time estimation was that Visual Studio developers would be as productive in developing embedded apps as they were in building desktop applications. Eventually, the developers needed unplanned competence ramp-up for embedded development, which caused significant delays in the project.

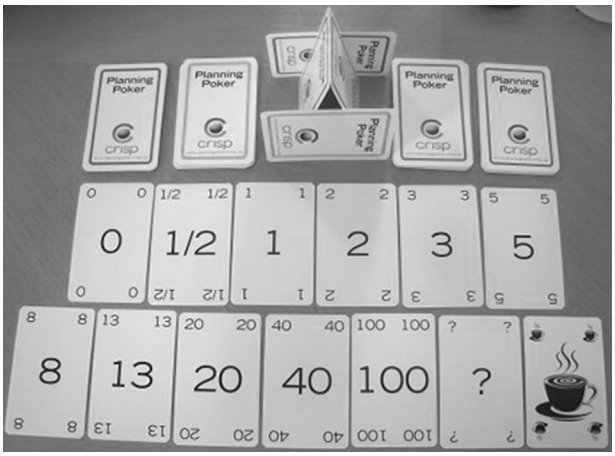

FIGURE 7.3

In planning poker, cards with Fibonacci numbers are used to estimate efforts.

Specific cost analysis methods are not prescribed, but methods such as CBAM (cost-benefit analysis method) (Nord et al., 2013) provide guidance to software architects and software engineers. Architects typically ask product managers or customers about their requirements and the value they assign to the quality attributes and features. This information helps select appropriate design tactics and software patterns from various alternatives. Each selection must also take economic aspects into account. Technology and solution selection then serve as the base for detailed cost and effort estimations. This approach connects features and quality attributes with business value and costs so that software architects can consider the economic consequences of their decisions.

7.9 Legal environment

Whenever software architects are in charge of software design and implementation, they generally need to cope with legal issues as well. A legal services unit can offer sophisticated support, but is seldom knowledgeable about technical aspects. Various opportunities in software development projects require specific care. Without thorough considerations, some pitfalls may lead to economic risks.

War Story: During the development of a communication middleware, architects designed new means for efficient message-based communication. After project completion, it was detected that some of the open-source components used in the system would have required the organization to provide its source code publicly according to the open-source license models, thus revealing business secrets. Another problem was the use of a smart component loading/unloading mechanism that turned out to be already patented by another company. The legal services organization had to prove that this prior patent was invalid due to prior use, causing a delay of the project.

Licensing models for open-source software (Lindberg, 2008) may force an organization to publish its own source code. If the internal code contains sensitive information such as innovations or business secrets, this can cause significant reengineering or rewriting efforts to get rid of the open-source components. The economic impact of licensing models might be huge, especially if inappropriate licensing models are used. Thus, architects are required to closely interact with legal experts before using open-source software components and to make sure their organization can live with the specific constraints of the license models.

Patents (Blind et al., 2005; Cleland-Huang et al., 2013; Hahn, 2005; Hall et al., 2006; Lundberg et al., 2011; Rosenberg and Apley, 2012; Stobbs and Kluwer Law International (Firm), 2008) may become another trap. If an open-source component, a commercial COTS product, or an internal code artifact violates existing patents of competitors or suppliers, then costs for dealing with the patent violations will emerge. This subsumes costs for substituting the respective code with new implementations, costs for identifying and proving the invalidity of patents due to prior art, or costs for paying license fees to the patent owners. If competitors are among the patent owners, licensing may become quite challenging, if possible at all. Utilizing software such as Black Duck is helpful to identify possible patent violations in company-internal code and lead to large cost savings in some projects.

The other side of the coin is *securing intellectual property rights*. New solutions or products typically contain innovative ideas that qualify for becoming valuable patents. This might be even more important for products in markets with many competitors. Architects are responsible for identifying, leveraging and harvesting these patent opportunities in their projects (Knight, 2013). As large companies mostly have organizational units that support patenting, architects should communicate with patent lawyers very early about securing new inventions. In the industry, patent portfolios can have an economic value of several millions or even billions of U.S. dollars. Patent portfolios are also helpful in cross-license negotiations with other companies. The economic value and impact of patents should not be underestimated. Creativity and innovation workshops educate project participants as to how to find and create patents in a systematic way.

7.10 Standards and regulations

Safety- and security-critical systems, as well as enterprise and other systems, are often subject to standardization *conformance and regulations*. A medical device, a weapon, or a railway control system needs to be certified by national organizations such as the the United States' FDA (Food and Drug Administration) or Germany's TÜV (Technischer Überwachungs Verein). Otherwise, these products must not be sold in some target markets. The time period needed for certification, however, can span several months or even years. An organization has to explicitly consider such constraints and necessities in product planning. Software architects are responsible for the conformant design and implementation of the architecture. For example, they define concepts of how to integrate specific safety measurements where required. They, together with certification experts, also need to make sure that there are no violations of standards or legal regulations such as DiCOM or SIL in the product.

Not only the parts produced by an organization, but also deliveries from third-party suppliers need to undergo certification. This could be one of the criteria for vendor selection and for the

creation of a technology roadmap. It is a core responsibility of architects to check third-party COTS components, implementation artifacts, and tools for possible conformance issues.

War Story: In developing medical products, safety is the most critical concern. In a project based on Microsoft Windows 7, architects only checked their own components for safety, but were asked later in the project to ensure the safety of the underlying operating system. Among many other efforts, developers had to provide additional layers on top of the Windows communication stack to provide checksums and other safety mechanisms. These activities had not been anticipated in the project plan.

Organizations have to keep in mind that regulation and standardization efforts might be huge, especially when different markets and thus different regulation bodies are addressed.

But it is not only mandatory regulation and standardization that are important. Sometimes customers expect products to implement specific standards (see Figure 7.4). One driver for this requirement is the interest of customers in not depending on a single vendor. Other facets may include the extensibility, customizability, compatibility, or configurability of products. An example is the availability of a REST-based API that allows easy product integration into customer environments. Software architects are in charge of evaluating and meeting these requirements.

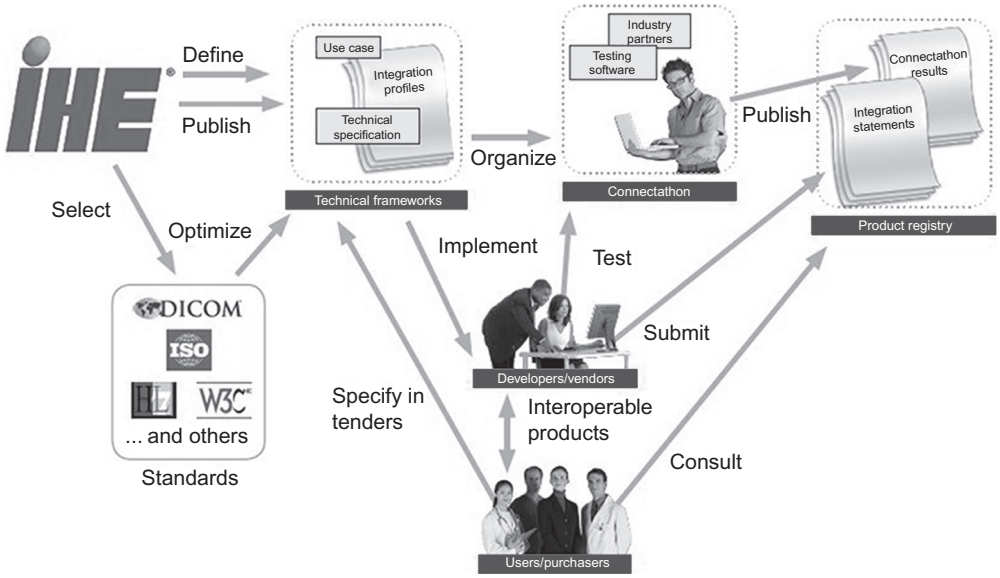

FIGURE 7.4

Health care modalities such as the MRI (magnetic resonance imaging) must undergo certification by the United States' FDA. In addition, they must adhere to standards such as IHE (Integrating the Healthcare Enterprise), DiCOM (Digital Imaging and Communication in Medicine) and HL/7 (Health Level 7).

7.11 Project management

The roles of software architects and project managers differ but complement each other. One key learning from projects is that these roles should be strictly separated. Only in very small projects nay an engineer wear both hats. If a software architect is also responsible for project management (Jones, 2010; Pries and Quigley, 2011; Whitaker, 2010), she cannot cover all software architecture-related responsibilities as well, at least not with the necessary depth. In such situations, either project management or software architecture will suffer, both of which may cause economic liabilities, for example, delay of development.

War Story: The project manager of a communication middleware system had to estimate the development efforts. She just reused the experiences from earlier projects for this purpose. However, all former products had been developed with C++, while the new product was supposed to be the first Java-based middleware. After the project started. it became clear that developers required competence ramp-up and that the work packages would take longer development time than expected.

The tight interaction between those roles respectively persons turned out to be one of the main success factors within Siemens. Architects need project management so that they can focus on the technical and economic aspects of software, while project managers are in charge of the logistics and planning aspects. Project managers need a close interaction with architects for making effort and cost estimates, for defining the project plan, for increasing the skill level of the staff, and for coping with problems in the project such as the delay of hardware development in an embedded systems environment. It is beneficial if software architects have some experience with project management, and, of course, if project managers have some experience in software engineering.

7.12 Economic considerations in the development process

7.12.1 Requirements engineering

In requirements engineering, architects interact with requirements engineers, customers (in solution organizations), and product managers (in product organizations) to support the definition of a system architecture (Berenbach and ebrary Inc., 2009; Broy and Rumpe, 1998; Chemuturi, 2013; Dorfman, 1994). Many of the activities are related to business aspects. KANO analysis and scenario diagrams are essential tools for this activity (see also Products—Definition and Development).

A very common challenge is the lack of a domain language for stakeholder communication. The same terminology may mean different things to different persons or roles, which often leads to wrong design decisions due to false interpretation of requirements. This problem typically becomes a critical economic factor when it remains untreated. Consequently, the establishment of a common language must be the first step in the requirements engineering phase. Domain languages range from informal glossaries to full-blown formal languages. They introduce a common understanding between stakeholders regarding the problem domain and the requirements. Domain analysis and domain-driven design (Evans, 2003) provide the basis for creating a domain language.

Another challenge arises from the *validation of the software requirements*. Is the quality of the requirements specification sufficient? For instance, are requirements consistent, unambiguous and

complete? In other words, could they be an adequate basis for designing and implementing the system. Architects will also need to prove software feasibility. Is it possible to design and implement the system with reasonable time and budget, so that the result can meet the business goals? Requirements such as "product should use expensive DBMS system from vendor A" and "unit costs should not exceed X US-$" may be conflicting requirements that cannot be resolved. Architects must be aware of all such sensitivity and trade-off points in the architecture and inform management about problems.

War Story: In a large-scale product line development several business units were supposed to base their products on the same product line platform. The domain development team asked the two largest business units to provide their requirements and derived the reference architecture from these requirements. The first version of the platform failed because the other business units not involved in requirements analysis had different requirements with different priorities. Thus, they could not use the product line. Product line development had to be restarted.

To come up with reasonable decisions, a unique requirements prioritization is necessary. This prioritization captures the business value of requirements as well as technical complexity. If architects encounter conflicts in the specification such as trade-off points, they can take the prioritization as a base for their decisions. Of course, the prioritization must be sound. Sometimes there are requirements specifications in which high priority is assigned to almost all requirements. If this is the case, architects must address requirements engineering and ask for a finer-grained prioritization. A viable strategy for architects is to make assumptions about the prioritization, document them, and ask requirements engineering to provide feedback.

For quality attributes such as performance or modifiability, the usage of scenario diagrams and utility trees (see Figure 7.5) (Bass et al., 2013) helps in efficiently setting up a joint prioritization.

As Frederick P. Brooks has pointed out (Brooks, 2010), *in many cases it is not useful to perform architecture design and requirements engineering independently or by a fixed time sequence*. The reason for this is uncertainty. Initially, architects have insufficient understanding and knowledge of the requirements, while the stakeholders in charge of requirements have insufficient information about the feasibility and technical implications of their specification. An additional issue is that new technologies, services, or products may not be fully understood by project participants, which further increases uncertainty. From an economic viewpoint, this phase of uncertainty should be addressed by parallelizing and coordinating requirements specification and architecture design, at least at project start. The Twin Peaks model (Cleland-Huang et al., 2013) recommends and promotes such "parallelism." Prototypes and technology evaluations should be conducted in this phase, for example, the creation of prototypes that support assessment of technical and economic feasibility. If the organization enforces a strict waterfall model, requirements and architecture will be separately and sequentially conducted. All problems in the requirements specifications as well as insufficient technology knowledge will cause significant economic implications. The later a design problem is detected, the more expensive it is to solve it, if it is solvable at all.

A further deliverable of requirements engineering is the implementation and control of an IPR (intellectual property rights) strategy. Architects may even already identify and create patents in this phase.

Technical feasibility might be checked by implementing feasibility prototypes during the requirements engineering phase and architecture design. This helps rule out technologies that cannot meet technical and economic goals.

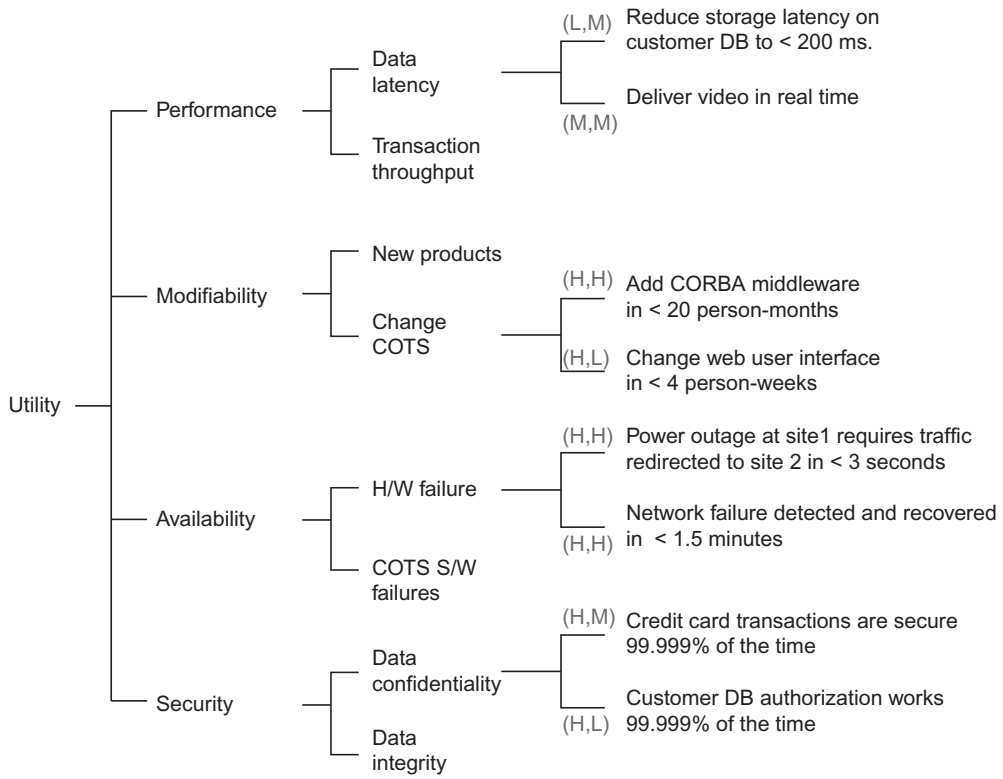

FIGURE 7.5

Utility trees are commonly used to prioritize quality attribute scenarios. Business decides on the business relevance of each scenario (left in tuple: High, Medium, Low), while development prioritizes according to development complexity (right in tuple: High, Medium, Low).

In some projects, delays emerged from the "uncertainty principle." Architects thought they needed a requirements specification where 100% of all requirements were fully specified in order to start with strategic design. Of course, all other information should be available as well. Although this completeness would never happen, architects were caught in an idle loop waiting for all information to flow in. Such behavior causes economic penalties, as it results in budget problems and time delays. Software architects should accept the idea that they live in a space of uncertainty—they won't be capable of creating a perfect design because requirements keep changing or growing in all projects. However, it is sufficient if at least the most important of the requirements are available at project start. Highly prioritized strategic requirements affect the architecture more than less important strategic or tactical requirements. Thus, it is possible to start with a sound but incomplete conceptual design that will be continuously refactored and refined during the project.

Together with test managers, architects will also participate in defining a (risk-based) test strategy during or after the requirements engineering phase. Without an adequate test strategy (Patton,

2005), testing activities may cover the wrong and less important things, while neglecting risky and important issues. The responsibility of testing (Veenendaal et al., 2006) is to provide sufficient information to architects and developers (and management) for efficient and successful quality control. With an appropriate test strategy in place, testing efforts can be minimized and focused on the risks. Testing activities such as integration testing also serve as a safety net for architects when they design the architecture. In addition, the overall test strategy includes a test exit strategy, which determines when the quality is considered high enough for product release.

7.12.2 Design and implementation

In the design phase (Bass et al., 2013) the main focus of architects is, of course, the architecture design itself. But additional activities have an economic impact, such as establishing the integration sequence and integration concepts (Summers, 2013), or setting up supplier agreements and reviewing development partnerships. These activities are fundamental for solution or product development in that errors in this phase mostly have a significant financial impact.

This is especially true for system development projects with various disciplines such as mechatronics, electronics, electrics, mechanics, and sometimes even including building construction (Blanchard, 2008)

War Story I: In an engineering project, the different disciplines were not synchronized because project management failed to take care of this responsibility, and lead system or software architects had not been established. The engineers at the construction site were forced to wait for deliveries from the development sites, and the development sites could not check for integration issues in an appropriate time frame. All this resulted in large delays that caused the customer to stop the project and to demand high penalty payments from the product organization. Eventually, even the product organization itself was closed.

War Story II: Architects responsible for the development of a credit card production system focused primarily on flexibility. The main objective of the project had been to quickly set up a configuration for banks ordering new credit cards. Eventually, the system was highly configurable, but it took experts several weeks to prepare the production. The system could not be refactored. Instead, a new development project had to be started from scratch.

Architects mainly deal with strategic design such as defining the technical architecture and defining the main architecture principles that should govern the implementation. They are also in charge of preventing architecture shift. Without establishing and enforcing such principles, the implementation will suffer from structural and operational quality issues as well as unnecessary efforts for reinventing the wheel multiple times. An example is the usage of different patterns or solutions to solve identical problems in the same problem context. Allowing multiple solution strategies for recurring problems produces systems that are hard to understand, assess, change, extend and maintain because reviewers, architects, and developers will need to deal with several approaches instead of one. For economic and technical reasons, the enforcement of software pattern usage and of quality indicators and architecture metrics should be mandatory in order to increase the overall quality.

It is mandatory that software architects relate all of their design or technology decisions to the business case as well as to business goals and business strategy. Each decision must have a concrete economic rationale. Otherwise, software engineers may create complex design pearls without

economic value. As a consequence, the probability of accidental complexity will increase, which has a direct and negative impact on costs for maintenance, testing, bug tracking, or refactoring. In addition, oversophisticated designs may reduce the overall quality by making the architecture more complex than is necessary instead of following the KiSS principle (Keep it stupid simple). Hence, quality control should be a permanent activity of architects to detect and avoid design flaws early.

Suitable instruments for ensuring quality are the following:

- Architecture governance for continuous architecture enforcement
- Regular code, design, and architecture assessments
- Testing

Architecture and design are not constrained to construction-oriented activities, building feature after feature but also for early refactoring. At each development cycle, architects are supposed to analyze the system for possible architecture or implementation problems such as wrong design decisions, design errors, or problems with structural and operational quality properties. Especially in architecture and design, problems that are detected and eliminated early reduce costs, since architecture is the backbone of software systems and hence provides a high economic value.

Software architects not only design the architecture, but they are in charge of conducting regular qualitative and quantitative architecture assessments that might be based on Architecture Tradeoff Analysis Method (ATAM) or a company-internal method, for example, an experience-based approach. Quantitative methods comprise prototyping or benchmarking.

If risks are detected such as design flaws in strategic aspects, measures are derived for eliminating these problems. Architecture refactoring helps to systematically get rid of problems in the architecture design. If architecture deficiencies are neglected, the architecture may grow and grow, continuously introducing new artifacts and dependencies. In a later stage, solving architecture problems might become very expensive or even impossible because the sum of deficiencies often leads to design erosion. If an architecture component with flawed design gets further refined and extended, dependencies between the component and its environment increase. To remove such flaws would consequently lead to huge efforts because many parts of the architecture and implementation have to be changed. Tight coupling between architecture components negatively affects structural and operational qualities of the system, thus increasing maintenance costs and service costs. Even a sound architecture design may turn into a serious bottleneck, when it is changed or developed in an unsystematic and wrong way. This is a main issue for product lines and platforms on which not one but many products depend. Such problems may cause huge maintenance costs, especially when the system is supposed to run for many years or even decades, which is a common challenge in industries like railway control, trains, or power plants.

These are the reasons why architects are responsible for taking the right actions. They design the architecture, conduct regular architecture assessments, obtain testing information, and improve the architecture before continuing their design activities.

Design and technical debt denote critical issues in this context. Shortly before product release, architects should not refactor the architecture or implementation if possible. They need to defer modifications and improvements to the next development cycle. Very critical and hence unacceptable problems are the only exception to this rule. However, a development organization increases design debt and technical debt whenever it postpones refactoring activities. It is essential that an organization keeps track of design and technical debt, that is, by managing a database with

unresolved architecture problems and their criticality and priority. When the implementation of the next release starts, architects must resolve critical issues first before further extending the architecture. Design debt must be paid soon, not ignored.

7.13 Implementation and integration

In the implementation and integration phase, architects need to frequently join the development teams to prevent design problems and to obtain valuable feedback. If possible, projects must follow the "architect always implements" mantra to ensure that architects are actively involved in implementation and architecture enforcement activities. They review the fine-grained design and its conformance to design guidelines, check the deliveries of external suppliers, establish an open-source strategy, and control software integration. All these responsibilities include economic considerations such as cost estimations, or validating the fulfillment of business goals. By preventing drift between architecture and design, architects can control implementation and integration costs.

War Story: In a project for managing prepaid mobile phone cards, the architects created a "beautiful" architecture design document that they provided to the development teams. One of the key design principles the architects used was strict layering. When implementation was completed, it turned out that each change of the database schemas led to a crash of the presentation clients. In an architecture review, the UI client architect confessed that he had intentionally ignored the strict layering and accessed the lower layers from the client; his management had told him to do so for performance reasons. This resulted in a direct dependency between persistence layer and UI layer and caused huge restructuring costs. Other developers told us that they just ignored the details of the architecture document.

After roll-out, the responsibilities of architects do not end. They must continuously manage the architecture, monitor quality indicators and metrics, and identify needs for refactoring, restructuring, or re-implementing.

7.14 Development process for architecture design

On one hand, using a strict waterfall model in design is neither effective nor efficient, as the previous sections illustrate, because they often turn later modifications into high economic risks.. On the other hand, projects might require models such as the V-Model, because of customer requirements, which is very common in government projects, safety-critical system development, and infrastructure projects. In the last-named case, an iterative waterfall model is recommendable which allows iterating through phases and returning to earlier development phases whenever required.

War Story: For the development of an innovative control system, the organization strictly followed the V-Model. However, during development, the requirements specification used to change frequently. In addition, unexpected issues were encountered upon implementation of some innovative new patterns and algorithms. Late changes caused substantial modification efforts. Most of the errors and change requests emerged only after release of the first product version. Those modifications led to an overly complex architecture and other indicators for design erosion.

As illustrated in figure 7.6, architects should use an agile model for their design activities. In each increment they focus on one or a few use cases or scenarios, starting with the ones of highest priority. The existing base-line architecture keeps growing with each architecture extension or refinement. After the design activity, architects invite stakeholders and other architects to conduct an assessment of the current state of the base-line architecture (Clements et al., 2001). These reviews help identify design flaws, bugs, inefficient design decisions, quality defects, and other issues. Before continuing with fine design or the next increment, the detected issues are prioritized and, if possible, eliminated by architecture refactoring. Some refactoring necessities can be postponed for economic reasons to the next increments if the issues are less critical and require substantial rework shortly before important events such as product release. However, all postponed issues should be documented as design debt (Sterling, 2010) that must be paid back later in the project.

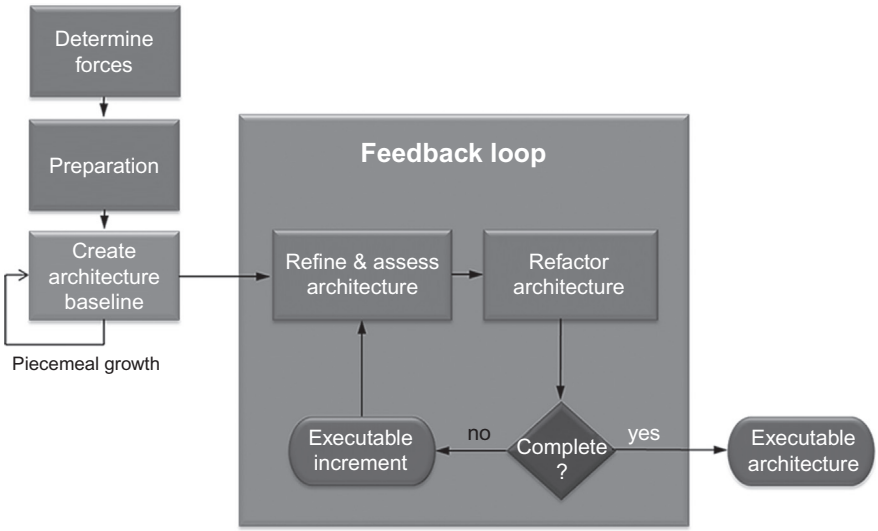

FIGURE 7.6

Architecture creation should follow a feedback loop that includes not only architecture refinement, but also architecture assessments and refactoring activities.

7.15 Product line organizations and re-use

Economic considerations are particularly essential in the context of developing and evolving product families and platforms. While evolution of a single product is typically constrained to the product itself, the evolution of a product line platform (see Figure 7.7) directly affects all members of the product family. One wrong decision harms all family members. In a product line organization, additional efforts are required to synchronize platform development with product development in a seamless and effective way (Bosch, 2000; Linden, et al., 2007; Pohl et al., 2005; Weiss and Lai,

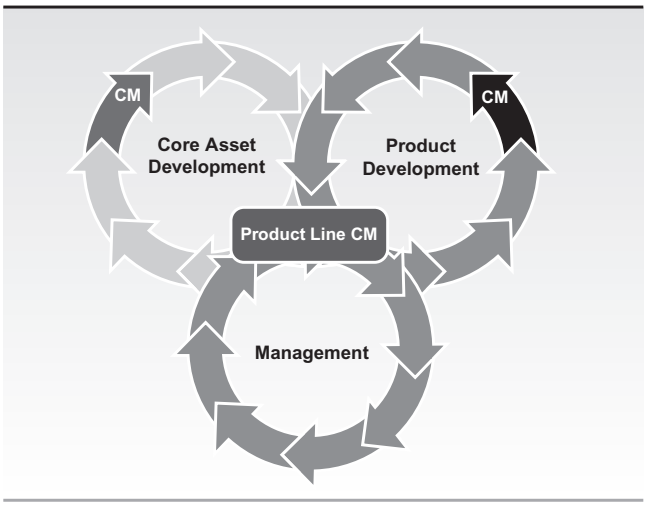

FIGURE 7.7

Product line engineering requires the management of multiple software development activities, in particular those for creating the product line assets common for all products as well as the individual products.

1999). Architects of products and the product line platform, together with business-oriented stakeholders, should jointly define the product line roadmap. One of the many challenges is the decision of whether a new asset should be integrated in the product line itself or provided by the minority of product organizations that require the asset. It is in the responsibility of architects to analyze all alternatives and rate their impact on complexity, costs, and sustainability.

Even more critical, architects are involved in the initial decision of whether a product line approach is feasible from a technical and an economic viewpoint. For this purpose, the organization should conduct a commonality/variability analysis (Authors, 2013) and, if required, further activities of domain engineering and application engineering such as constructing a conceptual architecture. If the overlap of various one-off products is significant in terms of commonalities and if, at the same time, variability is not too complex to implement, then a product line may turn out to be the best economic choice. In a joint activity, software architects and management then derive an effort and cost estimation from the preceding C/V analysis to calculate the return on investment of the software line approach compared to the current approach. For instance, one fundamental question is: How many products or product releases are needed until the expected revenues cover the additional estimated costs of a product line?

Success Story: At Siemens a product line platform called syngo was developed in the 1990s as a base for all modalities such as computed tomography (CT), magnetic resonance tomography (MRT), ultrasound, angiography, and positron emission tomography (PET). Previously, each modality used its own implementation. Thus, the various development teams had developed their own version of the same functionality. Some of the issues with this approach were that modalities could not connect with each other, and that customers complained about the different UIs offered

for the different products. In addition, the redevelopment of the same functionality proved to be a great time and cost issue. By establishing a domain architecture organization, they came up with their common syngo platform, which made Siemens a leading vendor for health care products.

But also in organizations without a product line approach a more basic, albeit *systematic reuse strategy* (Dusink et al., 1991), helps in saving costs. Re-using functionality that would otherwise be implemented many times can become a strategic advantage. Not only software might be reused, but also design solutions for recurring problems that are documented as software patterns (Buschmann et al., 1996) or design tactics (Bass et al., 2013). In addition to pattern catalogs and systems for common problems, organizations may document and apply domain-specific patterns (Fowler, 1996) or use a DSL (domain-specific language) (Voelter, 2013) for generating at least parts of the implementation (Parr, 2010). Architects are responsible for fostering reuse whenever it makes sense, but also for figuring out whether the benefits of reusing a particular component outweigh the liabilities (e.g., by conducting a cost-benefit analysis). It requires some efforts to make components usable and also reusable. These efforts might be substantial—for example, when extracting a reusable functionality from an existing system that was not built with reusability in mind. Another question is who should be responsible for maintaining reusable functionality. If this question remains unanswered, reuse will not work in an economic way. In product line organizations, a clear separation between the domain engineering organization and the various product organizations is introduced to foster systematic reuse.

7.16 Embedded systems development

In business units that develop embedded systems, some additional challenges must be addressed by systems and software architects (Balarin, 1997; Domeika & Books24 × 7 Inc., 2008; Gupta, 1995). One general issue we observed is that in the area of embedded systems development the BoM (bill of material) (Haik and Shahin, 2011) and the unit costs can become economic obstacles for software engineering. If the product represents a mass product, then each economic decision regarding hardware and unit costs can also affect software development.

War Story: The decision to reduce the amount of memory in a car entertainment system forced software developers to introduce compression algorithms. However, the usage of these algorithms implied unacceptable performance penalties. A possible solution was to use a faster CPU, but then the unit costs would have been much higher than in the original design, that is, the design without memory reduction. Key learning:saving hardware costs may affect the efforts of software development as well as software architecture.

Higher customer expectations denote a further challenge in embedded systems development. When using embedded systems, customers will not accept large response or start-up times or instabilities of the system. Thus, an extended focus on quality control is inevitable, not only for technical but also for economic reasons. This includes effective Q&A and testing, in addition to frequent code and design reviews as well as regular quantitative and qualitative architecture assessments.

For mobile systems that are distributed with thousands of units across the planet, tracking down software defects, patches, and their reasons is difficult. On one hand, over the air or online updates

can be an appropriate solution for quick and easy problem fixes, but on the other hand, these systems might contain different hardware and software versions or have a different patch state. (Software and system-) architects are responsible for coming up with an economic update concept that is reasonable and acceptable for the service organization and for customers.

In terms of development processes, there are additional phases relevant for systems engineering, for example, the production phase, the commissioning phase, and the maintenance and service phase. Extra requirements for software emerge in these phases from hardware and system infrastructure properties. For example, during assembly of a stepper motor, lubricant is applied as a lump on the winding. In order to distribute it evenly, the motor must be driven full cycles back and forth several times, slowly so as to not jerk the lubricant away. This "slow mode" is only needed when first used and never again. In production, it is necessary to move gauges to defined positions, turn on all light-emitting diodes, and display test patterns for automated "calibration" of gauges, especially gas and speed.

Systems and software architects have to know and consider such efforts. Ignoring these requirements leads to unplanned efforts and time delays (Maier, 2009). Regular communication and cooperation with systems engineers are essential to be prepared for utility software requirements.

Another common property of embedded systems development projects stems from the necessity to integrate different disciplines (see Figure 7.8) such as hardware, firmware, and software. Software development typically comprises significantly more development cycles than hardware development. Thus, integration of the individual artifacts to a complete product is one of the most critical steps. These integration steps need to be planned in advance with the help of system and software architects. Otherwise, integration tests cannot be executed at predefined milestones. Delay of one discipline causes other disciplines to remain idle due to their mutual (complex)

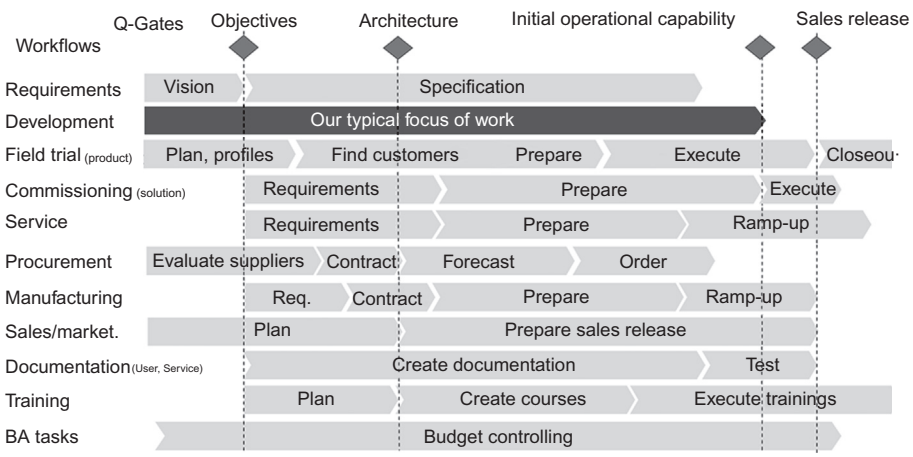

FIGURE 7.8

In systems engineering, multiple disciplines cooperate, for example, electronics and software development. Software development for embedded systems must be integrated into the systems engineering context such as synchronizing and integrating developments at quality gates.

dependencies, which in turn increases costs. Thus, project management and architects must thoroughly plan these milestones and continuously control progress. As the development cycles are different and integration tests occur less often, hardware and software integration requires all disciplines to deliver the best possible quality (Duvall et al., 2007).

A number of challenges present themselves when two low-quality artifacts are combined in a big bang approach. To reduce quality problems, software development adds more tooling to compensate for the slower hardware development. One common tool is model-based simulation of the target hardware, which enables software testing in the absence of hardware. In some industries automotive tools such as MATLAB/SimuLink help create simulations (Klee and Allen, 2011). With this approach the problem of different development cycles can be addressed. There is a caveat, however. Sometimes simulation can be more expensive than the delays between disciplines. And, of course, a simulator can only model the hardware but is not a full surrogate. In certain scenarios, simulation might behave differently than the real-world system, especially when timing is subtle and critical. Hence, simulators are very helpful but cannot substitute testing with the real target hardware. The rule here is to test on the target hardware as soon as possible.

7.17 Communication

As the chapter illustrated, communication is one of the most essential prerequisites for the economic success of software development. Alistair Cockburn's observation, "software engineering is a collaborative game" (Cockburn, 2006), is exactly what project experiences show. Architects need to communicate frequently and effectively with other stakeholders. They do not present the only communication hub, but they have an important role (see Figure 7.9).

In detail, architects communicate with

- systems architects to create a sustainable system architecture as well as to coordinate software development and system integration with other disciplines.
- software developers to enforce the architectural vision, obtain feedback, and prevent a drift between architecture and implementation.
- test managers to help create an appropriate test strategy and test plans.
- requirements engineers (i.e., the persons in charge of requirements engineering) to understand the requirements and ensure the quality of the requirements specification as a basis for architecture development. These stakeholders include requirements engineers, product (lifecycle) managers, and customers.
- project managers to achieve proper cost and effort estimations as well as to support project planning.
- heads of R&D to understand the business strategy and business goals of the organization.

Architects spend up to 40% of their time for communication, at least in large projects. It is obvious that communication must be effective. Tom DeMarco (DeMarco and Lister, 1999) once consulted two printer companies in the United States about how to improve their productivity. As it turned out, the company with fewer meetings and better meeting culture was much more productive than its meeting-addicted competitor. This is what industry projects show as well. Constant creation

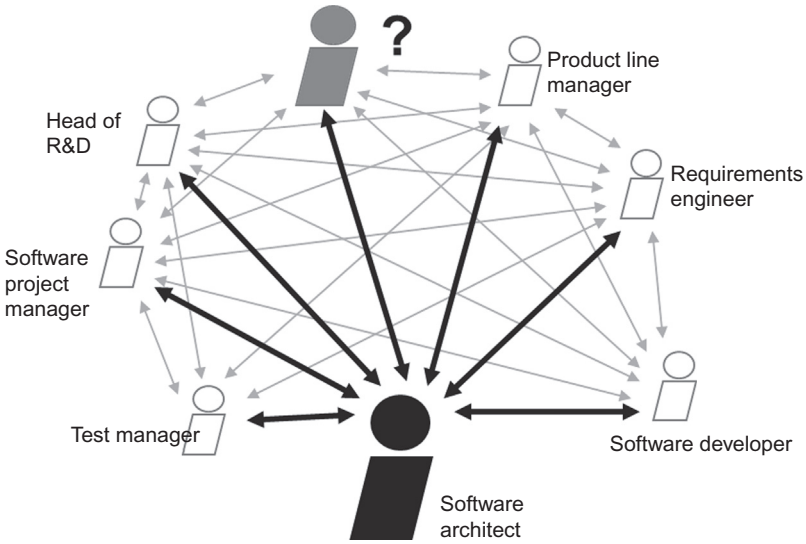

FIGURE 7.9

Architects need to interact with many other roles in a project to develop an economic solution. The figure shows product organizations as an example. In solution organizations, other roles such as the customer are essential.

of new task forces, frequent mandatory and long-lasting meetings without agenda and goals, combined with other unplanned ad-hoc activities, significantly reduce productivity. E-mail-centric communication is mostly very ineffective, especially if the amount of mail exceeds the limits persons can handle. Tom DeMarco calls this kind of mail overload "Corporate Spam." In turn, face-to-face communication is often the only effective option.

It is the responsibility of architects to demand but also to apply effective communication patterns. If they spend more than 40 to 50% of their time on communication, this might be an indication either of ineffective personal communication habits or of organizational communication overload. In the latter case, architects should escalate to management, while in the former case they should obtain feedback from colleagues, reflect, and then change and improve their habits.

7.18 Conclusions

In large industry companies such as Siemens, more than 50% of revenues depend on software. Since software architecture builds the backbone of each software system, the business value of software architecture represents a substantial economic and strategic factor. Conversely, problems in software architecture design lead to increased development costs. Software architects must understand and consider the economic impact of their design decisions, instead of focusing merely on technical aspects. They should be well educated, experienced and trained in software architecture

as well as other related aspects such as business and strategy, quality assurance, requirements engineering, or soft skills.

Siemens has established an internal (Senior) Software Architect Certification curriculum to address this problem. The curriculum intends to increase the economic awareness and responsibility of software architects and to foster networking between architects and business units. It was built on experiences from success stories and war stories in former projects. In the last five years, the CEOs of various Siemens businesses have returned positive feedback regarding the actual value of the curriculum. We were told that senior architects began to intensively participate in business, product, and strategy planning. They closely follow the guidelines that were illustrated in this chapter to relate business goals and business plans with design decisions. They feel responsible for requirements engineering and testing as well because both activities establish the fundamentals for economic and sustainable design. It has become obvious that software architects have a huge impact on economic aspects, especially when so many embedded products depend on software. Perhaps Grady Booch perfectly explains the economic value of software architecture when he states: "Software architecture is about everything that is costly to change".

References

Architects, A.I.O., 2008. The Architect's Handbook of Professional Practice. fourtenth ed. Wiley, Hoboken, NJ.

Authors, V., 2013. Systems and Software Variability Management: Concepts, Tools and Experiences. Springer, New York.

Balarin, F., 1997. Hardware-Software Co-Design of Embedded Systems: The POLIS Approach. Kluwer Academic Publishers, Boston.

Bass, L., Clements, P., Kazman, R., 2013. Software Architecture in Practice. third ed. Addison-Wesley, Upper Saddle River, NJ.

Berenbach, B., Ebrary Inc., 2009. Software and systems requirements engineering in practice. Retrieved from <http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10286222>.

Blanchard, B.S., 2008. *System Engineering Management* (Wiley Series in Systems Engineering and Management). Wiley, Hoboken, NJ.

Blind, K., Edler, J., Friedewald, M., 2005. Software Patents: Economic Impacts and Policy Implications. Edward Elgar, Cheltenham, UK.

Boehm, B., Abts, C., Brown, A.W., Chulani, S., Clark, B.K., Horowitz, E., et al., 2000. Software Cost Estimation with Cocomo II. Prentice Hall, New Jersey, USA.

Bosch, J., 2000. Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach. Addison-Wesley, Reading, MA.

Brooks, F.P., 2010. The Design of Design: Essays from a Computer Scientist. Addison-Wesley, Upper Saddle River, NJ.

Broy, M., Rumpe, B., 1998. Requirements Targeting Software and Systems Engineering International Workshop RTSE '97, Bernried, Germany, October 12–14, 1997 Lecture Notes in Computer Science 1526. Retrieved from <http://dx.doi.org/10.1007/b71630>.

Buschmann, F., Meunier, R., Rohner, H., Sommerlad, P., Stal, M., 1996. Pattern-Oriented Software Architecture. Volume 1: A System of Patterns. Wiley, New York.

Chemuturi, M., 2013. Requirements Engineering and Management for Software Development Projects. Retrieved from <http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10607533>.
Cleland-Huang, J., Mirakhorli, M., Supakkul, S., Hanmer, R.S., 2013. The Twin Peaks of Requirements and Architecture. IEEE (IEEE Software Magazine), Available from http://dx.doi.org/10.1109/MS.2013.39.
Clements, P., Kazman, R., Klein, M., 2001. Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley Professional, Reading, MA.
Cockburn, A., 2006. Agile Software Development: The Cooperative Game. second ed. Addison-Wesley Professional, Reading, MA.
DeMarco, T., Lister, T., 1999. Peopleware: Productive Projects and Teams, second ed. Dorset House, New York, USA.
Domeika, M., Books24x7 Inc., 2008. Software development for embedded multi-core systems a practical guide using embedded Intel architecture. Newnes; 1st edition Boston, USA.
Dorfman, M., 1994. Standards, Guidelines, and Examples on System and Software Requirements Engineering. IEEE Computer Society Press, Los Alamitos, CA.
Dusink, L., Hall, P.A.V., British Computer Society, 1991. Software re-use, Utrecht 1989: Proceedings of the Software Re-use Workshop, November 23–24, 1989. Springer-Verlag, Utrecht, The Netherlands. London.
Duvall, P.M., Matyas, S., Glover, A., 2007. Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley Professional, Reading, MA.
Evans, E., 2003. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, Reading, MA.
Fowler, M., 1996. Analysis Patterns: Reusable Object Models. Addison-Wesley Professional, Reading, MA.
Geracie, G., 2010. Take Charge Product Management: Take Charge of Your Product Management Development; Tips, Tactics, and Tools to Increase Your Effectiveness as a Product Manager. Actuation Press, Santa Clara, CA, USA.
Gupta, R.K., 1995. Co-synthesis of Hardware and Software for Digital Embedded Systems. Kluwer Academic Publishers, Boston.
Hahn, R.W., 2005. Intellectual Property Rights in Frontier Industries: Software and Biotechnology. AEI Press, Washington, DC.
Haik, Y., Shahin, T.M.M., 2011. Engineering Design Process. second ed. Cengage Learning, Stamford, CT.
Hall, B.H., MacGarvie, M., National Bureau of Economic Research, 2006. The private value of software patents NBER working paper series working paper 12195. Retrieved from <http://papers.nber.org/papers/w12195>.
Jones, C., 2010. Software Engineering Best Practices: Lessons from Successful Projects in the Top Companies. McGraw-Hill, New York.
Klee, H., Allen, R., 2011. Simulation of Dynamic Systems with MATLAB and Simulink. second ed. CRC Press, Boca Raton, FL.
Knight, H.J., 2013. Patent Strategy for Researchers and Research Managers. Wiley, Hoboken, NJ.
Lindberg, V., 2008. Intellectual Property and Open Source. O'Reilly, Sebastopol, CA.
Linden, F.V.D., Schmid, K., Rommes, E., 2007. Software Product Lines in Action : The Best Industrial Practice in Product Line Engineering. Springer, New York.
Lundberg, S.W., Durant, S.C., McCrackin, A.M., American Intellectual Property Law Association, 2011. Electronic and Software Patents: Law and Practice. third ed. Bureau of National Affairs, Arlington, VA.
Maier, M.W., 2009. The Art of Systems Architecting. third ed. CRC Press, Boca Raton, FL (Systems Engineering).
Matheson, D.M.J.E., 1997. The Smart Organization: Creating Value Through Strategic R&D. Harvard Business Review Press, Boston, MA, USA.
McGrath, M., 2000. Product Strategy for High Technology Companies. McGraw-Hill, New York.

Mironov, R., 2008. The Art of Product Management: Lessons from a Silicon Valley Innovator. BookSurge Publishing, Charleston, SC, USA.

Moehrle, M., Isenmann, R., Phaal, R., 2013. Technology Roadmapping for Strategy and Innovation: Charting the Route to Success. Springer, New York.

Nord, R., Barbacci, M.R., Clements, P.C., Kazman, R., Klein, M.H., Tomayko, J.E., 2013. Integrating the Architecture Tradeoff Analysis Method (ATAM) with the Cost Benefit Analysis Method (CBAM). Software Engineering Institute Pittsburgh, PA, USA.

Noriaki, S., Tsuji, S., Seraku, N., Takerhashi, F., 1984. Attractive Quality and Must-Be Quality, in: Hinshitsu Quality. The Journal of the Japanese Society for Quality Control Vol. 14 (No. 2), S39−S48.

Parr, T., 2010. Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages (Pragmatic Programmers). Pragmatic Bookshelf, Frisco, TX, Raleigh, NC.

Patton, R., 2005. Software Testing. second ed. Sams Publishing, Indianapolis, IN, USA.

Paulisch, F., Zimmerer, P., 2010, May 2−8. A role-based qualification and certification program for software architects: An experience report from Siemens Paper presented at the Software Engineering, 2010 ACM/IEEE 32nd International Conference on Software Engineering.

Pohl, K., Böckle, G.N., Linden, F.V.D., 2005. Software Product Line Engineering : Foundations, Principles, and Techniques. first ed. Springer, New York.

Pries, K.H., Quigley, J.M., 2011. Scrum Project Management. CRC Press, Boca Raton, FL.

Rosenberg, M.D., Apley, R.J., 2012. Business Method and Software Patents: A Practical Guide. Oxford University Press, Oxford, UK.

Sterling, C., 2010. Managing Software Debt: Building for Inevitable Change (Agile Software Development Series). Addison-Wesley Professional, Reading, MA.

Stobbs, G.A., Kluwer Law International (Firm), 2008. Software Patents Worldwide. Wolters KluwerFirm), 2008, Alphen aan den Rijn, The Netherlands, Frederick, MD, Sold and distributed in North, Central, and South America by Aspen Publishers.

Summers, B.L., 2013. Effective methods for software and systems integration (p. 1 online resource (xix, 163 p.)). Retrieved from <http://www.crcnetbase.com/isbn/978-1-4398-7662-6>.

Veenendaal, I.P., Bob van de, B., Dennis, J., Erik, V., 2006. Successful Test Management: An Integral Approach. Springer, New York.

Voelter, M., 2013. DSL Engineering: Designing, Implementing and Using Domain-Specific Languages. CreateSpace Independent Publishing Platform, Hamburg, Germany.

Weiss, D.M., Lai, C.T.R., 1999. Software Product-Line Engineering : A Family-based Software Development Process. Addison-Wesley, Reading, MA.

Whitaker, K., 2010. Principles of Software Development Leadership: Applying Project Management Principles to Agile Software Development Leadership. Charles River Media/Course Technology, Cengage Learning, Boston, MA.

wikipedia.org. SWOT Analysis. from <http://en.wikipedia.org/wiki/SWOT_analysis>.