

Username: Iowa State University Library **Book:** Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

2. Architectural Design

We now dive into the process of architecture design: what it is, why it is important, how it works (at an abstract level), and which major concepts and activities it involves. We first discuss architectural drivers: the various factors that “drive” design decisions, some of which are documented as requirements, but many of which are not. In addition, we provide an overview of design concepts—the major building blocks that you will select, combine, instantiate, analyze, and document as part of your design process.

Username: Iowa State University Library **Book:** Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

2.1 Design in General

Design is both a verb and a noun. Design is a process, an activity, and hence a verb. The process results in the creation of a design—a description of a desired end state. Thus the output of the design process is the thing, the noun, the artifact that you will eventually implement. Designing means making decisions to achieve goals and satisfy requirements and constraints. The outputs of the design process are a direct reflection of those goals, requirements, and constraints. Think about houses, for example. Why do traditional houses in China look different from those in Switzerland or Algeria? Why does a yurt look like a yurt, which is different from an igloo or a chalet or a longhouse?

The architectures of these styles of houses have evolved over the centuries to reflect their unique sets of goals, requirements, and constraints. Houses in China feature symmetric enclosures, sky wells to increase ventilation, south-facing courtyards to collect sunlight and provide protection from cold north winds, and so forth. A-frame houses have steep pitched roofs that extend to the ground, meaning minimal painting and protection from heavy snow loads (which just slide off to the ground). Igloos are built of ice, reflecting the availability of ice, the relative poverty of other building materials, and the constraints of time (a small one can be built in an hour).

In each case, the process of design involved the selection and adaptation of a number of solution approaches. Even igloo designs can vary. Some are small and meant for a temporary travel shelter. Others are large, often connecting several structures, meant for entire communities to meet. Some are simple unadorned snow huts. Others are lined with furs, with ice "windows", and doors made of animal skin.

The process of design, in each case, balances the various "forces" facing the designer. Some designs require considerable skill to execute (such as carving and stacking snow blocks in such a way that they produce a self-supporting dome). Others require relatively little skill—a lean-to can be constructed from branches and bark by almost anyone. But the qualities that these structures exhibit may also vary considerably. Lean-tos provide little protection from the elements and are easily destroyed, whereas an igloo can withstand Arctic storms and support the weight of a person standing on the roof.

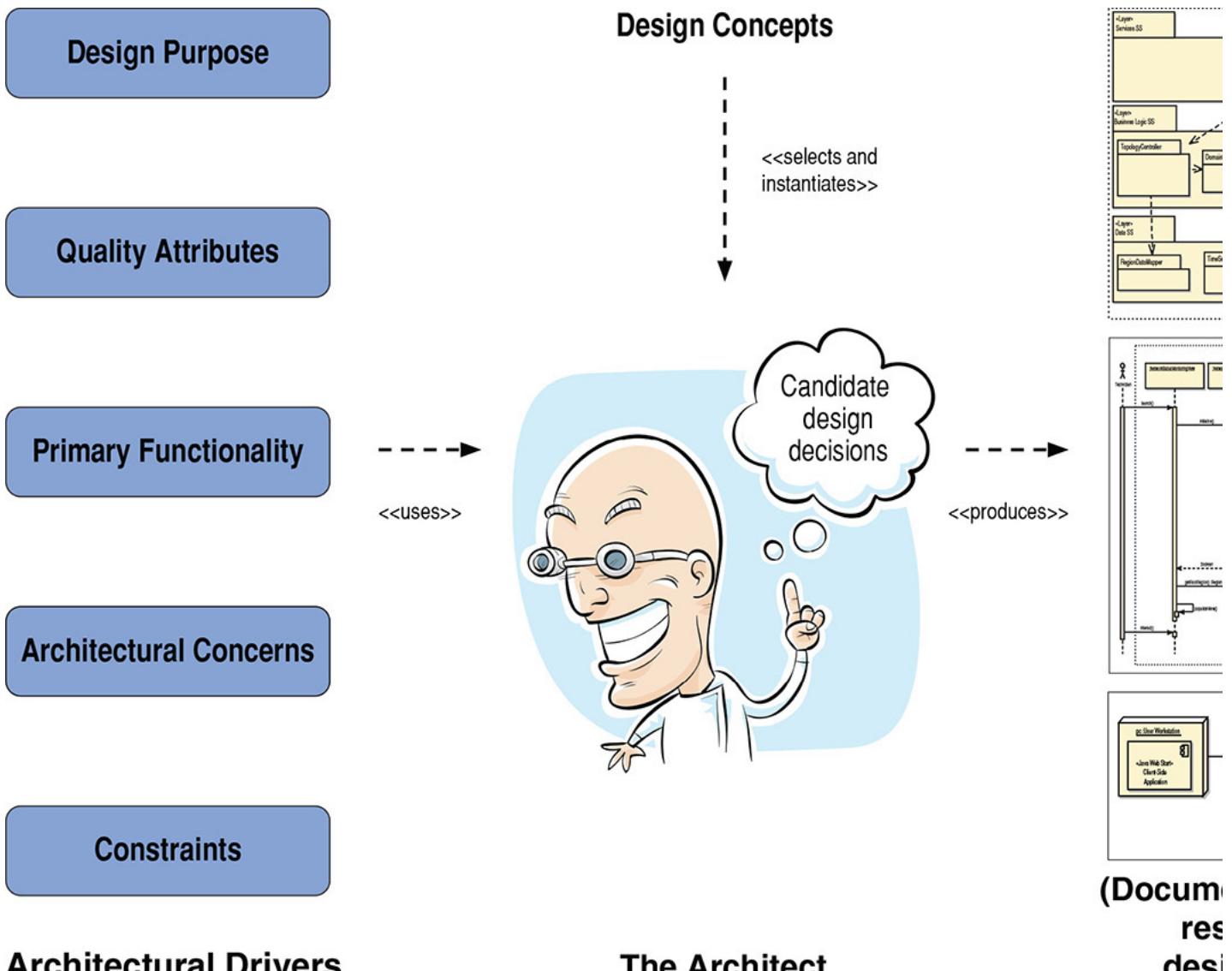
Is design "hard"? Well, yes and no. *Novel* design is hard. It is pretty clear how to design a conventional bicycle, but the design for the Segway broke new ground. Fortunately, most design is not novel, because most of the time our requirements are not novel. Most people want a bicycle that will reliably convey them from place to place. The same holds true in every domain. Consider houses, for example. Most people living in Phoenix want a house that can be easily and economically kept cool, whereas most people in Edmonton are primarily concerned with a house that can be kept warm. In contrast, people living in Japan and Los Angeles are concerned with buildings that can withstand earthquakes.

The good news for you, the architect, is that there are ample proven designs and design fragments, or building blocks that we call [design concepts](#), that can be reused and combined to reliably achieve these goals. If your design is truly novel—if you are designing the next Sydney Opera House—then the design process will likely be "hard". The Sydney Opera House, for example, cost 14 times its original budget estimate and was delivered ten years late. So, too, with the design of software architectures.

Username: Iowa State University Library Book: Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

2.2 Design in Software Architecture

Architectural design for software systems is no different than design in general: It involves making decisions, working with available skills and materials, to satisfy requirements and constraints. In architectural design, we make decisions to transform our design purpose, requirements, constraints, and architectural concerns—what we call the architectural drivers—into structures, as shown in [Figure 2.1](#). These structures are then used to guide the project. They guide analysis and construction, and serve as the foundation for educating a new project member. They also guide cost and schedule estimation, team formation, risk analysis and mitigation, and, of course, implementation.



Architectural Drivers

The Architect

FIGURE 2.1 Overview of the architecture design activity (Architect image © Brett Lamb | Dreamstime.com)

Architectural design is, therefore, a key step to achieving your product and project goals. Some of these goals are technical (e.g., achieving low and predictable latency in a video game or an e-commerce website), and some are nontechnical (e.g., keeping the workforce employed, entering a new market, meeting a deadline). The decisions that you, as an architect, make will have implications for the achievement of these goals and may, in some cases, be in conflict. The choice of a particular reference architecture (e.g., the Rich Client Application) may provide a good foundation for achieving your latency goals and will keep your workforce employed because they are already familiar with that reference architecture and its supporting technology stack. But this choice may not help you enter a new market—mobile games, for example.

In general, when designing, a change in some structure to achieve one quality attribute will have negative effects on other quality attributes. These tradeoffs are a fact of life for every practicing architect in every domain. We will see this over and over again in the examples and case studies provided in this book. Thus the architect's job is not one of finding an *optimal* solution, but rather one of *satisficing*—searching through a potentially large space of design alternatives and decisions until an acceptable solution is found.

2.2.1 Architectural Design

Grady Booch has said, "All architecture is design, but not all design is architecture". What makes a decision "architectural"? A decision is architectural if it has non-local consequences *and* those consequences matter to the achievement of an architectural driver. No decision is, therefore, inherently architectural or non-architectural. The choice of a buffering strategy within a single element may have little effect on the rest of the system, in which case it is an implementation detail that is of no concern to anyone except the implementer or maintainer of that element. In contrast, the buffering strategy may have enormous implications for performance (if the buffering affects the achievement of latency or throughput or jitter goals) or availability (if the buffers might not be large enough and information gets lost) or modifiability (if we wish to flexibly change the buffering strategy in different deployments or contexts). The choice of a buffering strategy, like most design choices, is neither inherently architectural nor inherently non-architectural. Instead, this distinction is completely dependent on the current and anticipated architectural drivers.

2.2.2 Element Interaction Design

Architectural design generally results in the identification of only a subset of the elements that are part of the system's structure. This is to be expected because, during initial architectural design, the architect will focus on the primary functionality of the system. What makes a use case primary? A combination of business importance, risk, and complexity considerations feed into this designation. Of course, to your users, everything is urgent and top priority. More realistically, a small number of use cases provide the most fundamental business value or represent the greatest risk (if they are done wrong), so these are deemed primary.

Every system has many more use cases, beyond the primary ones, that need to be satisfied. The elements that support these nonprimary use cases and their interfaces are identified as part of what we call [element interaction design](#). This level of design usually follows architectural design. The location and relationships of these elements, however, are constrained by the decisions that were made during architectural design. These elements can be units of work (i.e., modules) assigned to an individual or to a team, so this level of design is important for defining not only how nonprimary functionality is allocated, but also for planning purposes (e.g., team formation and communication, budgeting, outsourcing, release planning, unit and integration test planning).

Depending on the scale and complexity of the system, the architect should be involved in element interaction design, either directly or in an auditing role. This involvement ensures that the system's important quality attributes are not compromised—for example, if the elements are not defined, located, and connected correctly. It will also help the architect spot opportunities for generalization.

2.2.3 Element Internals Design

A third level of design follows element interaction design, which we call [element internals design](#). In this level of design, which is usually conducted as part of the element development activities, the internals of the elements identified in the previous design level are established, so as to satisfy the element's interface.

Architectural decisions can and do occur at the three levels of design. Moreover, during architectural design, the architect may need to delve as deeply as element internals design to achieve a particular architectural driver. An example of this is the selection of a buffering strategy that was previously discussed. In this sense, architectural design can involve considerable detail, which explains why we do not like to think about it in terms of "high-level design" or "detailed design" (see the sidebar "[Detailed Design?](#)").

Architectural design precedes element interaction design, which precedes element internals design. This is logically necessary: One cannot design an element's internals until the elements themselves have been defined, and one cannot reason about interaction until several elements and some patterns of interactions among them have been defined. But as projects grow and evolve, there is, in practice, considerable iteration between these activities.

Detailed Design?

The term "detailed design" is often used to refer to the design of the internals of modules. Although it is widely used, we really don't like this term, which is presented as somehow in opposition to "high-level design". We prefer the more precise terms "architectural design", "element interaction design", and "element internals design".

After all, architectural design may be quite detailed, if your system is complex. And some design "details" will turn out to be architectural. For the same reason, we also don't like the terms "high-level design" and "low-level design". Who can really know what these terms actually mean? Clearly, "high-level design" should be somehow "higher" or more abstract, and cover more of the architectural landscape than "low-level design", but beyond that we are at a loss to imbue these terms with any precise meaning.

So here is what we recommend: Just avoid using terms such as "high", "low", or "detailed" altogether. There is always a better, more precise choice, such as "architectural", "element interaction", or "element internals" design!

Think carefully about the impact of the decisions you are making, the information that you are trying to convey in your design documentation, and the likely audience for that information, and then give that process an appropriate, meaningful name.

Username: Iowa State University Library **Book:** Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

2.3 Why Is Architectural Design So Important?

There is a very high cost to a project of *not* making certain design decisions, or of not making them early enough. This manifests itself in many different ways. Early on, an initial architecture is critical for project proposals (or, as it is sometimes called in the consulting world, the *pre-sales process*). Without doing some architectural thinking and some early design work, you cannot confidently predict project cost, schedule, and quality. Even at this early stage, an architecture will determine the key approaches for achieving architectural drivers, the gross work-breakdown structure, and the choices of tools, skills, and technologies needed to realize the system.

In addition, architecture is a key enabler of agility, as we will discuss in [Chapter 9](#). Whether your organization has embraced Agile processes or not, it is difficult to imagine anyone who would willingly choose an architecture that is brittle and hard to change or extend or tune—and yet it happens all the time. This so-called [technical debt](#) occurs for a variety of reasons, but paramount among these is the combination of a focus on features—typically driven by stakeholder demands—and the inability of architects and project managers to measure the return on investment of good architectural practices. Features provide immediate benefit. Architectural improvement provides immediate costs and long-term benefits. Put this way, why would anyone ever “invest” in architecture? The answer is simple: Without architecture, the benefits that the system is supposed to bring will be far harder to realize.

Simply put, if you do not make some key architectural decisions early and if you allow your architecture to degrade, you will be unable to maintain sprint velocity, because you cannot easily respond to change requests. However, we vehemently disagree with what the original creators of the Agile Manifesto claimed: “The best architectures, requirements, and designs emerge from self-organizing teams”. Indeed, our demurral with this point is precisely why we have written this book. Good architectural design is difficult (and still rare), and it does not just “emerge”. This opinion mirrors a growing consensus within the Agile community. More and more, we see techniques such as “disciplined agility at scale”, the “walking skeleton”, and the “scaled Agile framework” embraced by Agile thought leaders and practitioners alike. Each of these techniques advocates some architectural thinking and design prior to much, if any, development. To reiterate, architecture enables agility, and not the other way around.

Furthermore, the architecture will influence, but not determine, other decisions that are not in and of themselves design decisions. These decisions do not influence the achievement of quality attributes directly, but they may still need to be made by the architect. For example, such decisions may include selection of tools; structuring the development environment; supporting releases, deployment, and operations; and making work assignments.

Finally, a well-designed, properly communicated architecture is key to achieving *agreements* that will guide the team. The most important kinds to make are agreements on interfaces and on shared resources. Agreeing on interfaces early is important for component-based development, and critically important for distributed development. These decisions *will* be made sooner or later. If you don’t make the decisions early, the system will be much more difficult to integrate. In [Section 3.6](#), we will discuss how to define interfaces as part of architectural design—both the external interfaces to other systems and the internal interfaces that mediate your element interactions.

Username: Iowa State University Library **Book:** Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

2.4 Architectural Drivers

Before commencing design with ADD (or with any other design method, for that matter), you need to think about what you are doing and why. While this statement may seem blindingly obvious, the devil is, as usual, in the details. We categorize these “what” and “why” questions as architectural drivers. As shown in [Figure 2.1](#), these drivers include a design purpose, quality attributes, primary functionality, architectural concerns, and constraints. These considerations are critical to the success of the system and, as such, they *drive* and shape the architecture.

As with any other important requirements, architectural drivers need to be baselined and managed throughout the development life cycle.

Username: Iowa State University Library **Book:** Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

2.4.1 Design Purpose

First, you need to be clear about the purpose of the design that you want to achieve. When and why are you doing this architecture design? Which business goals is the organization most concerned about at this time?

1. You may be doing architecture design as part of a project proposal (for the pre-sales process in a consulting organization, or for internal project selection and prioritization in a company, as discussed in [Section 9.1.1](#)). It is not uncommon that, as part of determining project feasibility, schedule, and budget, an initial architecture is created. Such an architecture would not be very detailed; its purpose is to understand and break down the architecture in sufficient detail that the units of work are understood and hence may be estimated.
2. You may be doing architecture design as part of the process of creating an exploratory prototype. In this case, the purpose of the architecture design process is not so much to create a releasable or reusable system, but rather to explore the domain, to explore new technology, to place something executable in front of a customer to elicit rapid feedback, or to explore some quality attribute (such as performance scalability or failover for availability).
3. You may be designing your architecture during development. This could be for an entire new system, for a substantial portion of a new system, or for a portion of an existing system that is being refactored or replaced. In this case, the purpose is to do enough design work to satisfy requirements, guide system construction and work assignments, and prepare for an eventual release.

These purposes may be interpreted and realized differently for greenfield systems in mature domains, for greenfield systems in novel domains, and for existing systems. In a mature domain, the pre-sales process, for example, might be relatively straightforward; the architect can reuse existing systems as examples and confidently make estimates based on analogy. In novel domains, the pre-sales estimation process will be far more complex and risky, and may have highly variable results. In these circumstances, a prototype of the system, or a key part of the system, may need to be created to mitigate risk and reduce uncertainty. In many cases, this architecture may also need to be quickly adapted as new requirements are learned and embraced. In brownfield systems, while the requirements are better understood, the existing system is itself a complex object that must be well understood for planning to be accurate.

Finally, the development organization's goals during development or maintenance may affect the architecture design process. For example, the organization might be interested in designing for reuse, designing for future extension or subsetting, designing for scalability, designing for continuous delivery, designing to best utilize existing project capabilities and team member skills, and so forth. Or the organization might have a strategic relationship with a vendor. Or the CIO might have a specific like or dislike and wants to impose it on your project.

Why do we bother to list these considerations? Because they *will* affect both the process of design and the outputs of design. Architectures exist to help achieve business goals. The architect should be clear about these goals and should communicate them (and negotiate them!) and establish a clear design purpose *before* beginning the design process.

Username: Iowa State University Library **Book:** Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

2.4.2 Quality Attributes

In the book *Software Architecture in Practice*, [quality attributes](#) are defined as being measurable or testable properties of a system that are used to indicate how well the system satisfies the needs of its stakeholders. Because quality tends to be a subjective concept in itself, these properties allow quality to be expressed succinctly and objectively.

Among the drivers, quality attributes are the ones that shape the architecture the most significantly. The critical choices that you make when you are doing architectural design determine, in large part, the ways that your system will or will not meet these driving quality attribute goals.

Given their importance, you must worry about eliciting, specifying, prioritizing, and validating quality attributes. Given that so much depends on getting these drivers right, this sounds like a daunting task. Fortunately, a number of well-understood, widely disseminated techniques can help you here (see sidebar “[The Quality Attribute Workshop and the Utility Tree](#)”):

- Quality Attribute Workshop ([QAW](#)) is a facilitated brainstorming session involving a group of system stakeholders that covers the bulk of the activities of eliciting, specifying, prioritizing, and achieving consensus on quality attributes.

- Mission Thread Workshop serves the same purpose as QAW, but for a system of systems.

- The Utility Tree can be used by the architect to prioritize quality attribute requirements according to their technical difficulty and risk.

We believe that the best way to discuss, document, and prioritize quality attribute requirements is as a set of scenarios. A [scenario](#), in its most basic form, describes the system’s response to some stimulus. Why are scenarios the best approach? Because all other approaches are worse! Endless time may be wasted in defining terms such as “performance” or “modifiability” or “configurability”, as these discussions tend to shed little light on the real system. It is meaningless to say that a system will be “modifiable”, because every system is modifiable with respect to some changes and not modifiable with respect to others. One can, however, specify the modifiability response measure you would like to achieve (say, elapsed time or effort) in response to a specific change request. For example, you might want to specify that “a change to update shipping rates on the e-commerce website is completed and tested in less than 1 person-day of effort”—an unambiguous criterion.

The heart of a [quality attribute scenario](#), therefore, is the pairing of a stimulus with a response. Suppose that you are building a video game and you have a functional requirement like this: “The game shall change view modes when the user presses the <C> button”. This functional requirement, if it is important, needs to be associated with quality attribute requirements. For example:

- How fast should the function be?

- How secure should the function be?

- How modifiable should the function be?

To address this problem, we use a scenario to describe a quality attribute requirement. A quality attribute scenario is a short description of how a system is required to respond to some stimulus. For example, we might annotate the functional requirement given earlier as follows: “The game shall change view modes in < 500 ms when the user presses the <C> button”. A scenario associates a stimulus (in this case, the pressing of the <C> button) with a response (changing the view mode) that is measured using a response measure (< 500 ms). A complete quality attribute scenario adds three other parts: the source of the stimulus (in this case, the user), the artifact affected (in this case, because we are dealing with end-to-end latency, the artifact is the entire system) and the environment (are we in normal operation, startup, degraded mode, or some other mode?). In total, then, there are six parts of a completely well-specified scenario, as shown in [Figure 2.2](#).

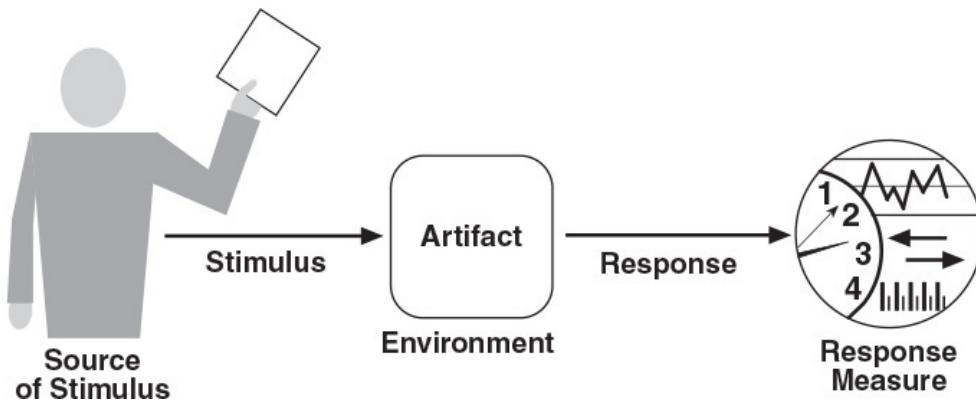


FIGURE 2.2 The six parts of a quality attribute scenario

Scenarios are testable, *falsifiable hypotheses* about the quality attribute behavior of the system under consideration. Because they have explicit stimuli and responses, we can evaluate a design in terms of how likely it is to support the scenario, and we can take measurements and test a prototype or fully fleshed-out system for whether it satisfies the scenario in practice. If the analysis (or prototyping results) indicates that the scenario’s response goal cannot be met, then the hypothesis is deemed falsified.

As with other requirements, scenarios should be prioritized. This can be achieved by considering two dimensions that are associated with each scenario and that are assigned a rank of importance:

- The first dimension corresponds to the importance of the scenario with respect to the success of the system. This is ranked by the customer.

- The second dimension corresponds to the degree of technical risk associated with the scenario. This is ranked by the architect.

A low/medium/high (L/M/H) scale is used to rank both dimensions. Once the dimensions have been ranked, scenarios are prioritized by selecting those that have a combination of (H, H), (H, M), or (M, H) rankings.

In addition, some traditional requirements elicitation techniques can be modified slightly to focus on quality attribute requirements, such as Joint Requirements Planning (JRP), Joint Application Design (JAD), discovery prototyping, and accelerated systems analysis.

But whatever technique you use, do not start design without a prioritized list of measurable quality attributes! While stakeholders might plead ignorance (“I don’t know how fast it needs to be; just make it fast!”), you can almost always elicit at least a range of possible responses. Instead of saying the system should be “fast”, ask the stakeholder if a 10-second response time is acceptable. If that is unacceptable, ask if 5 seconds is OK, or 1 second. You will find that, in most cases, users know more than they realize about their requirements, and you can at least “box them in” to a range.

The Quality Attribute Workshop and the Utility Tree

The Quality Attribute Workshop (QAW)

The [QAW](#) is a facilitated, stakeholder-focused method to generate, prioritize, and refine quality attribute scenarios. A QAW meeting is ideally enacted before the software architecture has been defined although, in practice, we have seen the QAW being used at all points in the software development life cycle. The QAW is focused on system-level concerns and specifically the role that software will play in the system. The steps of the QAW are as follows:

1. QAW Presentation and Introductions

The QAW facilitators describe the motivation for the QAW and explain each step of the method.

2. Business Goals Presentation

A stakeholder representing the project’s business concerns presents the system’s business context, broad functional requirements, constraints, and known quality attribute requirements. The quality attributes that will be refined in later QAW steps will be derived from, and should be traceable to, the business goals presented in this step. For this reason, these business goals must be prioritized.

3. Architectural Plan Presentation

The architect presents the system architectural plans as they currently exist. Although the architecture has frequently not been defined yet (particularly for greenfield systems), the architect often knows quite a lot about it even at this early stage. For example, the architect might already know about technologies that are mandated, other systems that this system must interact with, standards that must be followed, subsystems or components that could be reused, and so forth.

4. Identification of Architectural Drivers

The facilitators share their list of key architectural drivers that they assembled during steps 2 and 3 and ask the stakeholders for clarifications, additions, deletions, and corrections. The idea here is to reach a consensus on a distilled list of architectural drivers that covers major functional requirements, business drivers, constraints, and quality attributes.

5. Scenario Brainstorming

Given this context, each stakeholder now has the opportunity to express a scenario representing that stakeholder's needs and desires with respect to the system. The facilitators ensure that each scenario has an explicit stimulus and response. The facilitators also ensure traceability and completeness: At least one representative scenario should exist for each architectural driver listed in step 4 and should cover all the business goals listed in step 2.

6. Scenario Consolidation

Similar scenarios are consolidated where reasonable. In step 7, the stakeholders vote for their favorite scenarios, and consolidation helps to prevent votes from being spread across several scenarios that are expressing essentially the same concern.

7. Scenario Prioritization

Prioritization of the scenarios is accomplished by allocating to each stakeholder a number of votes equal to 30 percent of the total number of scenarios. The stakeholders can distribute these votes to any scenario or scenarios. Once all the stakeholders have voted, the results are tallied and the scenarios are sorted in order of popularity.

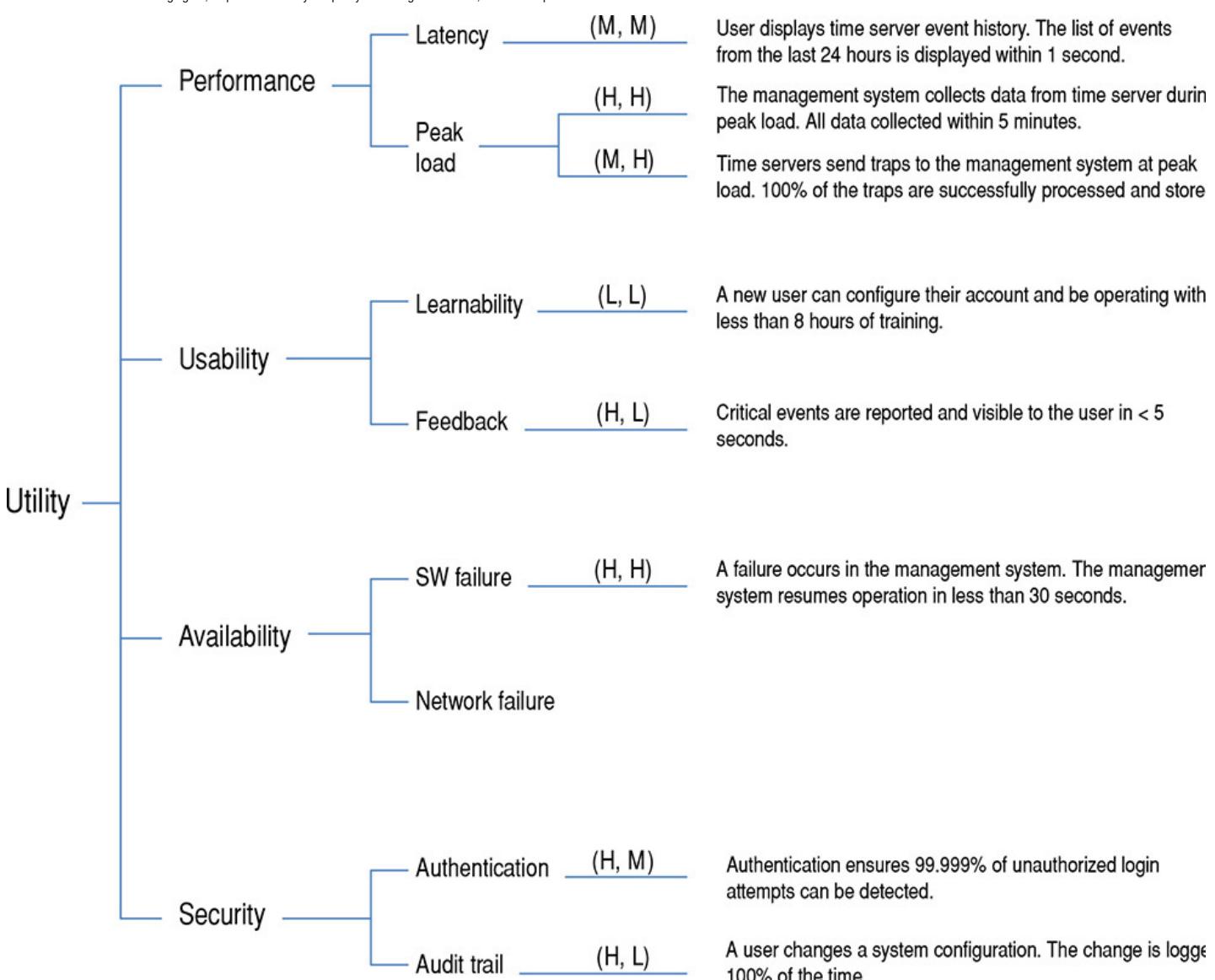
8. Scenario Refinement

The highest-priority scenarios are refined and elaborated. The facilitators help the stakeholders express these in the form of six-part scenarios: source, stimulus, artifact, environment, response, and response measure.

The output of the QAW is therefore a prioritized list of scenarios, aligned with business goals, where the highest-priority scenarios have been explored and refined. A QAW can be conducted in as little as 2–3 hours for a simple system or as part of an iteration, and as much as 2 days for a complex system where requirements completeness is a goal.

Utility Tree

If no stakeholders are readily available to consult, you still need to decide what to do and how to prioritize the many challenges facing the system. One way to organize your thoughts is to create a Utility Tree. The Utility Tree, such as the one shown in the following figure, helps to articulate your quality attribute goals in detail, and then to prioritize them.



It works as follows. First write the word "Utility" on a sheet of paper. Then write the various quality attributes that constitute utility for your system. For example, you might know, based on the business goals for the system, that the most important qualities for the system are that the system be fast, secure, and easy to modify. In turn, you would write these words underneath "Utility". Next, because we don't really know what any of those terms actually means, we describe the aspect of the quality attribute that we are most concerned with. For example, while "performance" is vague, "latency of database transactions" is a bit less vague. Likewise, while "modifiability" is vague, "ease of adding new codecs" is a bit less vague.

The leaves of the tree are expressed as scenarios, which provide concrete examples of the quality attribute considerations that you just enumerated. For example, for "latency of database transactions", you might create a scenario such as "1000 users simultaneously update their own customer records under normal conditions with an average latency of 1 second". For "ease of adding new codecs", you might create a scenario such as "Customer requests that a new custom codec be added to the system. Codec is added with no side effects in 2 person-weeks of effort".

Finally, the scenarios that you have created must be prioritized. We do this prioritization by using the technique of ranking across two dimensions, resulting in a priority matrix such as the following (where the numbers in the cells are from a set of system scenarios).

Business Importance/ Technical Risk	L	M	H
L	5, 6, 17, 20, 22	1, 14	12, 19
M	9, 12, 16	8, 20	3, 13, 15
H	10, 18, 21	4, 7	2, 11

Our job, as architects, is to focus on the lower-right-hand portion of this table (H, H): those scenarios that are of high business importance and high risk. Once we have satisfactorily addressed those scenarios, we can move to the (M, H) or (H, M) ones, and then move up and to the left until all of the system's scenarios are addressed (or perhaps until we run out of time or budget, as is often the case).

It should be noted that the QAW and the Utility Tree are two different techniques that are aimed at the same goal—eliciting and prioritizing the most important quality attribute requirements, which will be some of your most critical architectural drivers. There is no reason, however, to choose between these techniques. Both are useful and valuable and, in our experience, they have complementary strengths: The QAW tends to focus more on the requirements of external stakeholders, whereas the Utility Tree tends to excel at eliciting the requirements of internal stakeholders. Making all of these stakeholders happy will go a long way toward ensuring the success of your architecture.

Username: Iowa State University Library **Book:** Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

2.4.3 Primary Functionality

Functionality is the ability of the system to do the work for which it was intended. As opposed to quality attributes, the way the system is structured does not normally influence functionality. You can have all of the functionality of a given system coded in a single enormous module, or you can have it neatly distributed across many smaller, highly cohesive modules. Externally the system will look and work the same way if you consider only functionality. What matters, though, is what happens when you want to make changes to such system. In the former case, changes will be difficult and costly; in the latter case, they should be much easier and cheaper to perform. In terms of architectural design, allocation of functionality to elements, rather than the functionality per se, is what matters. A good architecture is one in which the most common changes are localized in a single or a few elements, and hence easy to make.

When designing an architecture, you need to consider at least the primary functionality. Primary functionality is usually defined as functionality that is critical to achieve the business goals that motivate the development of the system. Other criteria for primary functionality might be that it implies a high level of technical difficulty or that it requires the interaction of many architectural elements. As a rule of thumb, approximately 10 percent of your use cases or user stories are likely to be primary.

There are two important reasons why you need to consider primary functionality when designing an architecture:

1. You need to think how functionality will be allocated to elements (usually modules) to promote modifiability or reusability, and also to plan work assignments.
2. Some quality attribute scenarios are directly connected to the primary functionality in the system. For example, in a movie streaming application, one of the primary use cases is, of course, to watch a movie. This use case is associated with a performance quality attribute scenario such as "Once the user presses play, the movie should begin streaming in no more than 5 seconds". In this case, the quality attribute scenario is directly associated with the primary use case, so making decisions to support this scenario also requires making decisions about how its associated functionality will be supported. This is not the case for all quality attributes. For example, an availability scenario can involve recovery from a system failure, and this failure may occur when any of the system's use cases are being executed.

Decisions regarding the allocation of functionality that are made during architectural design establish a precedent for how the rest of the functionality should be allocated to modules as development progresses. This is usually not the work of the architect; instead, this activity is typically performed as part of the element interaction design process described in [Section 2.2.2](#).

Finally, bad decisions that are made regarding the allocation of functionality result in the accumulation of technical debt. (Of course, these decisions may reveal themselves to be bad only in hindsight.) This debt can be paid through the use of refactoring, although this impacts the project's rate of progress, or velocity (see the sidebar "[Refactoring](#)").

Refactoring

If you refactor a software architecture (or part of one), what you are doing is maintaining the same functionality but changing some quality attribute that you care about. Architects often choose to refactor because a portion of the system is difficult to understand, debug, and maintain. Alternatively, they may refactor because part of the system is slow, or prone to failure, or insecure.

The goal of the refactoring in each case is not to change the functionality, but rather to change the quality attribute response. (Of course, additions to functionality are sometimes lumped together with a refactoring exercise, but that is not the core *intent* of the refactoring.) Clearly, if we can maintain the same functionality but change the architecture to achieve different quality attribute responses, these requirement types are orthogonal to each other—that is, they can vary independently.

Username: Iowa State University Library **Book:** Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

2.4.4 Architectural Concerns

Architectural concerns encompass additional aspects that need to be considered as part of architectural design but that are not expressed as traditional requirements. There are several different types of concerns:

- *General concerns.* These are “broad” issues that one deals with in creating the architecture, such as establishing an overall system structure, the allocation of functionality to modules, the allocation of modules to teams, organization of the code base, startup and shutdown, and supporting delivery, deployment, and updates.
- *Specific concerns.* These are more detailed system-internal issues such as exception management, dependency management, configuration, logging, authentication, authorization, caching, and so forth that are common across large numbers of applications. Some specific concerns are addressed in reference architectures (see [Section 2.5.1](#)), but others will be unique to your system. Specific concerns also result from previous design decisions. For example, you may need to address session management if you previously decided to use a reference architecture for the development of web applications.
- *Internal requirements.* These requirements are usually not specified explicitly in traditional requirement documents, as customers usually seldom express them. Internal requirements may address aspects that facilitate development, deployment, operation, or maintenance of the system. They are sometimes called “derived requirements”.
- *Issues.* These result from analysis activities, such as a design review (see [Section 8.6](#)), so they may not be present initially. For instance, an architectural evaluation may uncover a risk that requires some changes to be performed in the current design.

Some of the decisions surrounding architectural concerns might be trivial or obvious. For example, your deployment structure might be a single processor for an embedded system, or a single cell phone for an app. Your reference architecture might be constrained by company policy. Your authentication and authorization policies might be dictated by your enterprise architecture and realized in a shared framework. In other cases, however, the decisions required to satisfy particular concerns may be less obvious—for example, in exception management or input validation or structuring the code base.

From their past experience, wise architects are usually aware of the concerns that are associated with a particular type of system and the need to make design decisions to address them. Inexperienced architects are usually less aware of such concerns; because these concerns tend to be tacit rather than explicit, they may not consider them as part of the design process, which often results in problems later on.

Architectural concerns frequently result in the introduction of new quality attribute scenarios. The concern of “supporting logging”, for example, is too vague and needs to be made more specific. Like the quality attribute scenarios that are provided by the customer, these scenarios need to be prioritized. For these scenarios, however, the customer is the development team, operations, or other members of the organization. During design, the architect must consider both the quality attribute scenarios that are provided by the customer and those scenarios that are derived from architectural concerns.

One of the goals of our revision of the ADD method was to elevate the importance of architectural concerns as explicit inputs to the architecture design process, as will be highlighted in our examples and case studies in [Chapters 4, 5, and 6](#).

Username: Iowa State University Library **Book:** Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

2.4.5 Constraints

You need to catalog the constraints on development as part of the architectural design process. These constraints may take the form of mandated technologies, other systems with which your system needs to interoperate or integrate, laws and standards that must be complied with, the abilities and availability of your developers, deadlines that are non-negotiable, backward compatibility with older versions of systems, and so on. An example of a technical constraint is the use of open source technologies, whereas a nontechnical constraint is that the system must obey the Sarbanes-Oxley Act or that it must be delivered by December 15.

A constraint is a decision over which you have little or no control as an architect. Your job is, as we mentioned in [Chapter 1](#), to *satisfice*: to design the best system that you can, despite the constraints you face. Sometimes you might be able to argue for loosening a constraint, but in most cases you have no choice but to design around the constraints.

Username: Iowa State University Library **Book:** Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

2.5 Design Concepts: The Building Blocks for Creating Structures

Design is not random, but rather is planned, intentional, rational, and directed. The process of design may seem daunting at first. When facing the “blank page” at the beginning of any design activity, the space of possibilities might seem impossibly huge and complex. However, there is some help here. The software architecture community has created and evolved, over the course of decades, a body of generally accepted design principles that can guide us to create high-quality designs with predictable outcomes.

For example, some well-documented design principles are oriented toward the achievement of specific quality attributes:

- To help achieve high modifiability, aim for good modularity, which means high cohesion and low coupling.
- To help achieve high availability, avoid having any single point of failure.
- To help achieve scalability, avoid having any hard-coded limits for critical resources.
- To help achieve security, limit the points of access to critical resources.
- To help achieve testability, externalize state.
- . . . and so forth.

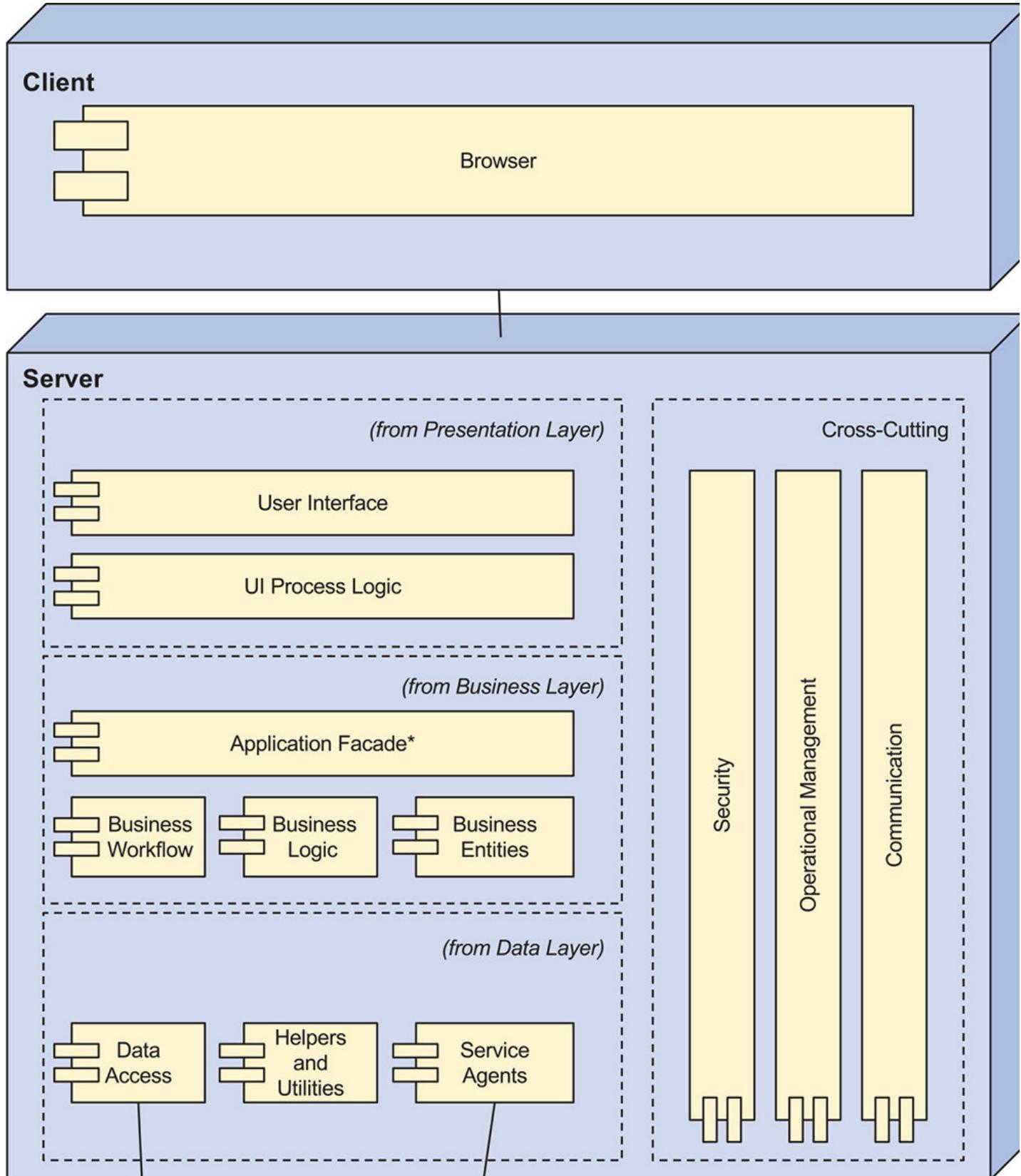
In each case, these principles have been evolved over decades of dealing with those quality attributes in practice. In addition, we have evolved reusable realizations of these abstract approaches in design and, eventually, in code. We call these reusable realizations *design concepts*, and they are the building blocks from which the structures that make up the architecture are created. Different types of design concepts exist, and here we discuss some of the most commonly used, including reference architectures, deployment patterns, [architectural patterns](#), tactics, and externally developed components (such as frameworks). While the first four are conceptual in nature, the last one is concrete.

Username: Iowa State University Library **Book:** Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

2.5.1 Reference Architectures

Reference architectures are blueprints that provide an overall logical structure for particular types of applications. A reference architecture is a reference model mapped onto one or more architectural patterns. It has been proven in business and technical contexts, and typically comes with a set of supporting artifacts that eases its use.

An example of a reference architecture for the development of web applications is shown in [Figure 2.3](#) on the next page. This reference architecture establishes the main layers for this type of application—presentation, business, and data—as well as the types of elements that occur within the layers and the responsibilities of these elements, such as UI components, business components, data access components, service agents, and so on. Also, this reference architecture introduces cross-cutting concerns, such as security and communication, that need to be addressed. As this example shows, when you select a reference architecture for your application, you also adopt a set of issues that you need to address during design. You may not have an explicit requirement related to communications or security, but the fact that these elements are part of the reference architecture require you to make design decisions about them.



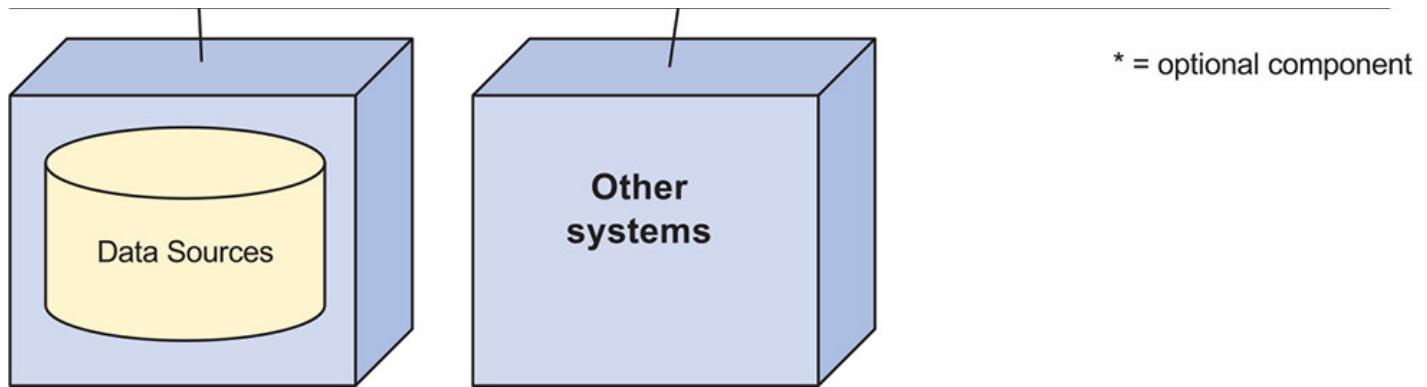


FIGURE 2.3 Example reference architecture for the development of web applications from the *Microsoft Application Architecture Guide* (Key: UML)

Reference architectures may be confused with architectural styles, but these two concepts are different. Architectural styles (such as "Pipe and Filter" and "Client Server") define types of components and connectors in a specified topology that are useful for structuring an application either logically or physically. Such styles are technology and domain agnostic. Reference architectures, in contrast, provide a structure for applications in specific domains, and they may embody different styles. Also, while architectural styles tend to be popular in academia, reference architectures seem to be preferred by practitioners—which is also why we favor them in our list of design concepts.

While there are many reference architectures, we are not aware of any catalog that contains an extensive list of them.

Username: Iowa State University Library **Book:** Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

2.5.2 Architectural Design Patterns

[Design patterns](#) are conceptual solutions to recurring design problems that exist in a defined context. While design patterns originally focused on decisions at the object scale, including instantiation, structuring, and behavior, today there are catalogs with patterns that address decisions at varying levels of granularity. In addition, there are specific patterns to address quality attributes such as security or integration.

While some people argue for the differentiation between what they consider to be architectural patterns and the more fine-grained design patterns, we believe there is no principled difference that can be solely attributed to scale. We consider a pattern to be architectural when its use directly and substantially influences the satisfaction of some of the architectural drivers (see [Section 2.2](#)).

[Figure 2.4](#) shows an example architectural pattern that is useful for structuring the system, the Layers pattern. When you choose a pattern such as this one, you must decide how many layers you will need for your system. [Figure 2.5](#) shows a pattern to support concurrency, which is useful to increase performance. This pattern, too, needs to be instantiated—that is, it needs to be adapted to the specific problem and design context. Instantiation is discussed in [Chapter 3](#).

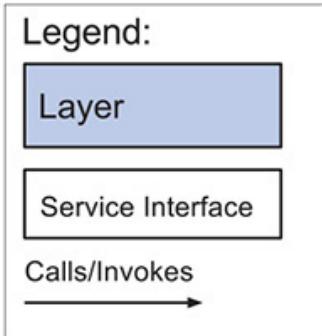
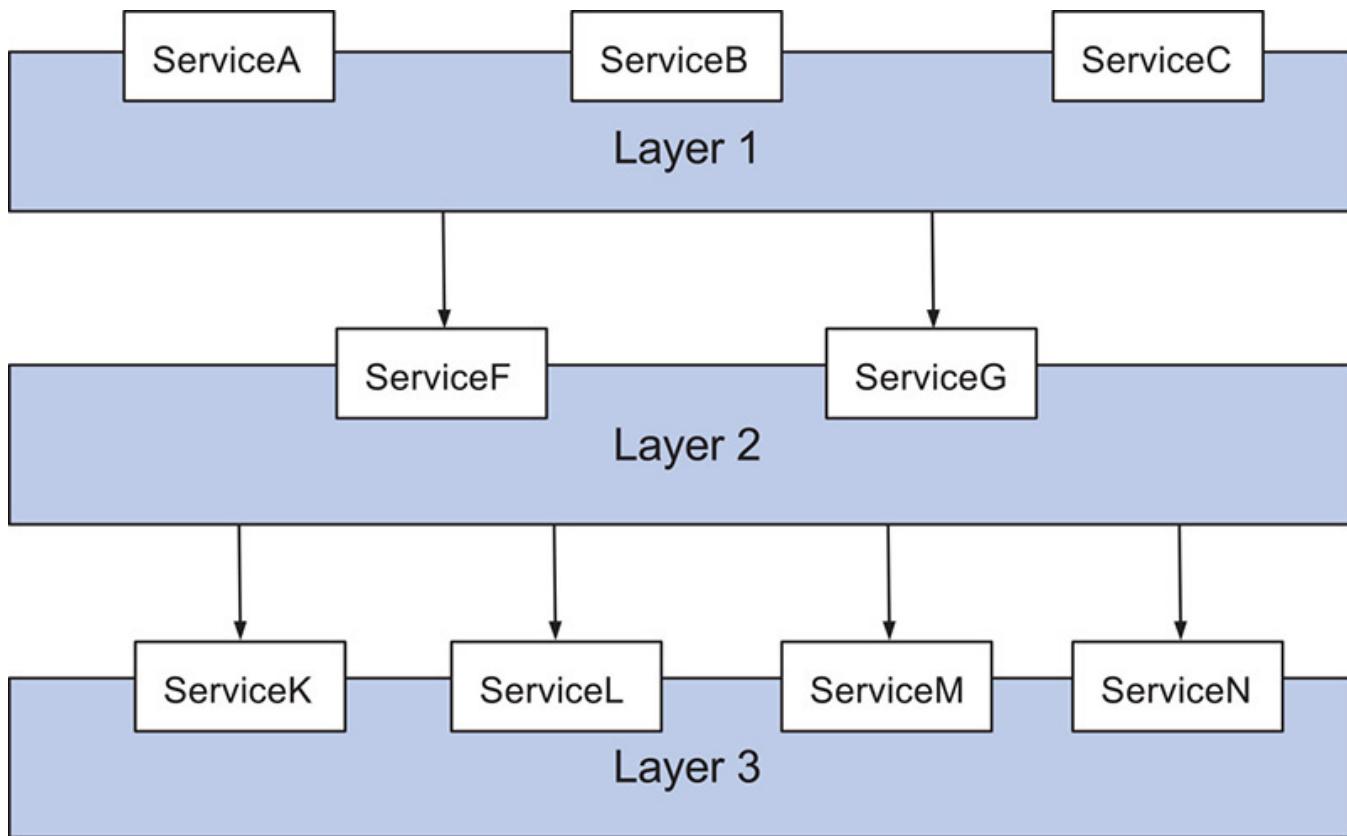


FIGURE 2.4 The Layers pattern for structuring an application from *Pattern-Oriented Software Architecture*

Synchronous Service Layer

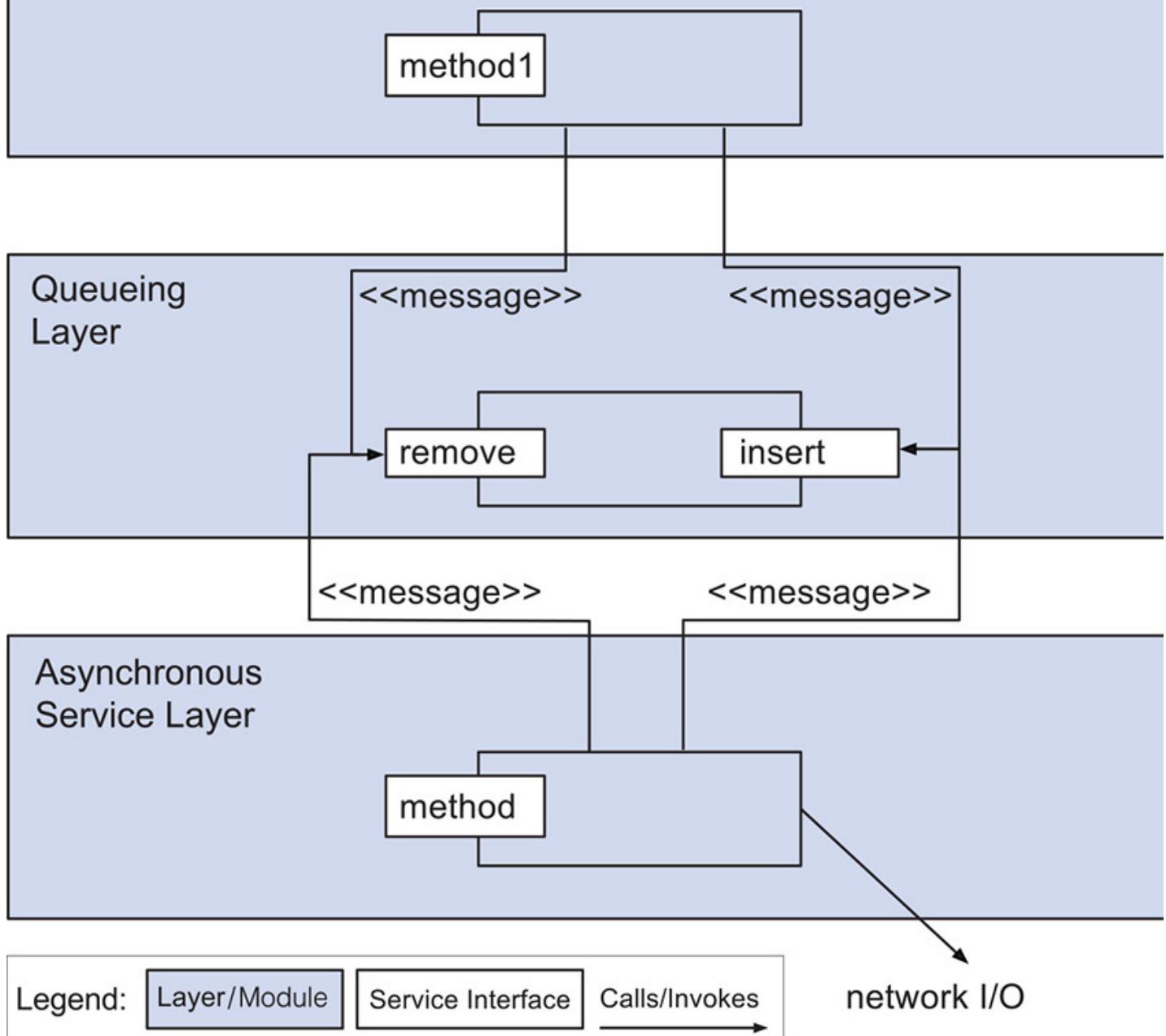


FIGURE 2.5 The Half-Sync/Half-Async pattern to support concurrency from *Pattern-Oriented Software Architecture* (Source: Softserve)

Although reference architectures may be considered as a type of pattern, we prefer to consider them separately because of the important role they play in structuring an application and because they are more directly connected to technology stacks. Also, a reference architecture typically incorporates other patterns and often constrains these patterns. For example, the reference architecture for web applications shown in [Figure 2.3](#) incorporates the Layers pattern but also establishes how many layers need to be used. This reference architecture also incorporates other patterns such as an Application Facade and Data Access Components.

Username: Iowa State University Library Book: Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

2.5.3 Deployment Patterns

Another type of pattern that we prefer to consider separately is [deployment patterns](#). These patterns provide models on how to physically structure the system to deploy it. Some deployment patterns, such as the one shown in [Figure 2.6](#), are useful to establish an initial physical structure of the system in terms of tiers (physical nodes). More specialized deployment patterns, such as the Load-Balanced Cluster in [Figure 2.7](#), are used to satisfy quality attributes such as availability, performance, and security.

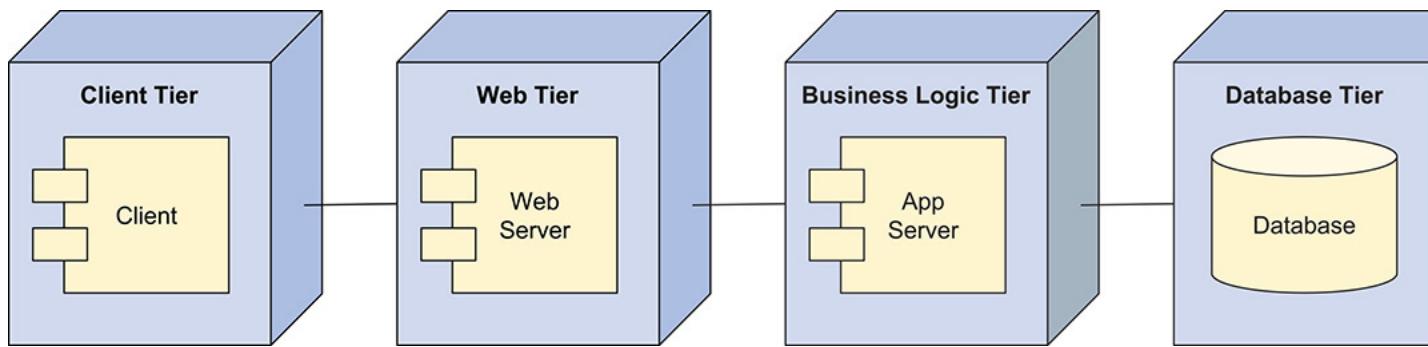


FIGURE 2.6 Four-tier deployment pattern from the *Microsoft Application Architecture Guide* (Key: UML)

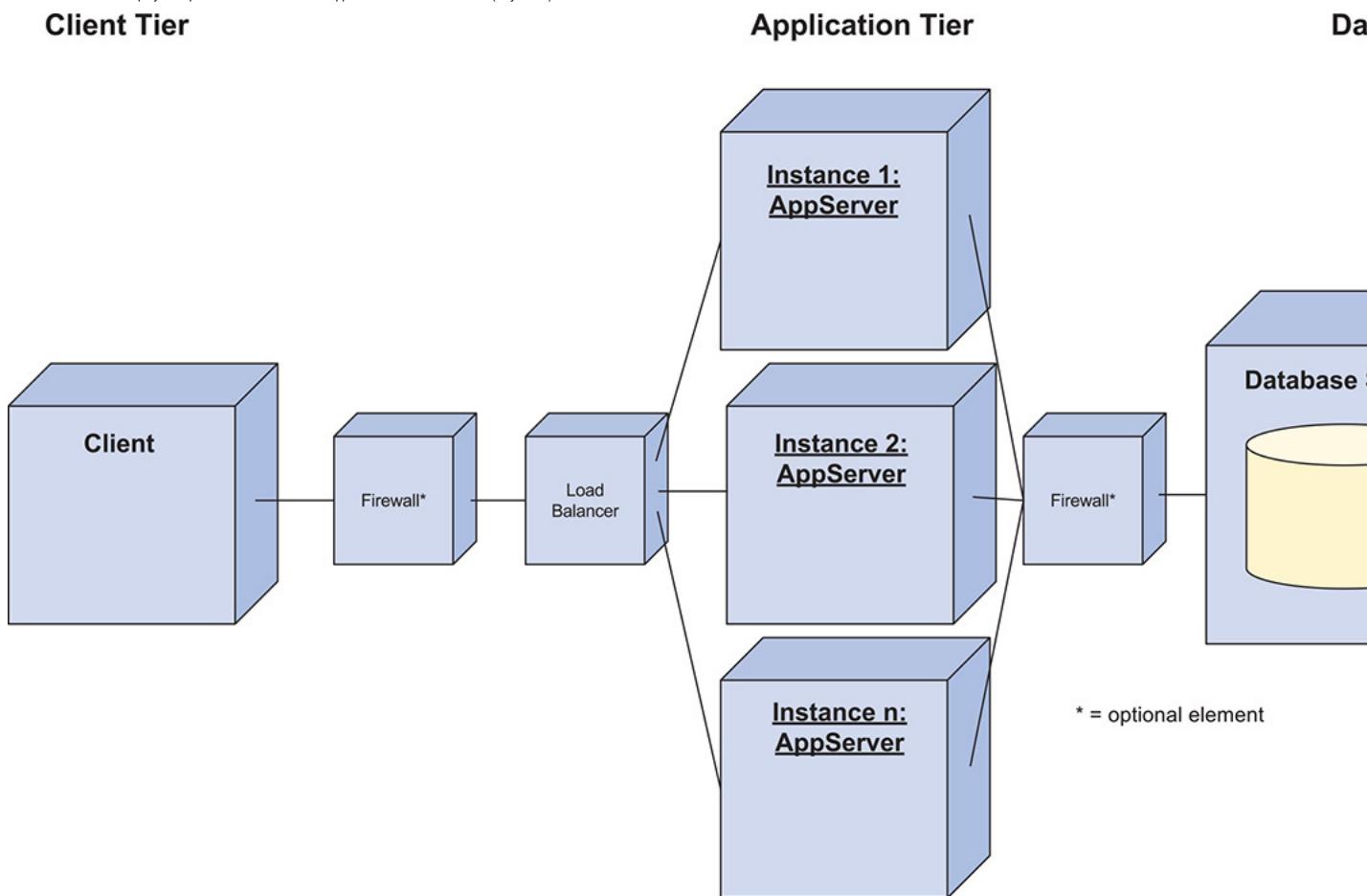


FIGURE 2.7 Load-Balanced Cluster deployment pattern for performance from the *Microsoft Application Architecture Guide* (Key: UML)

In general, an initial structure for the system is obtained by mapping the logical elements that are obtained from reference architectures (and other patterns) into the physical elements defined by deployment patterns.

Username: Iowa State University Library **Book:** Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

2.5.4 Tactics

Architects can use collections of fundamental design techniques to achieve a response for particular quality attributes. We call these architectural design primitives [tactics](#). Tactics, like design patterns, are techniques that architects have been using for years. We do not invent tactics, but simply capture what architects actually have done in practice, over the decades, to manage quality attribute response goals.

Tactics are design decisions that influence the control of a quality attribute response. For example, if you want to design a system to have low latency or high throughput, you could make a set of design decisions that would mediate the arrival of events (requests for service), resulting in responses that are produced within some time constraints, as shown in [Figure 2.8](#).

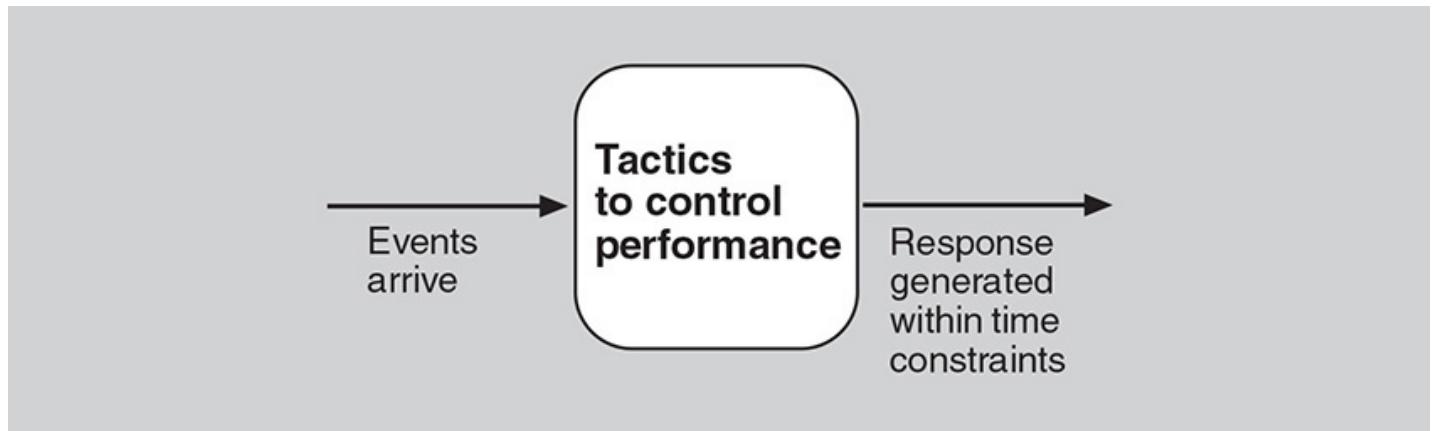
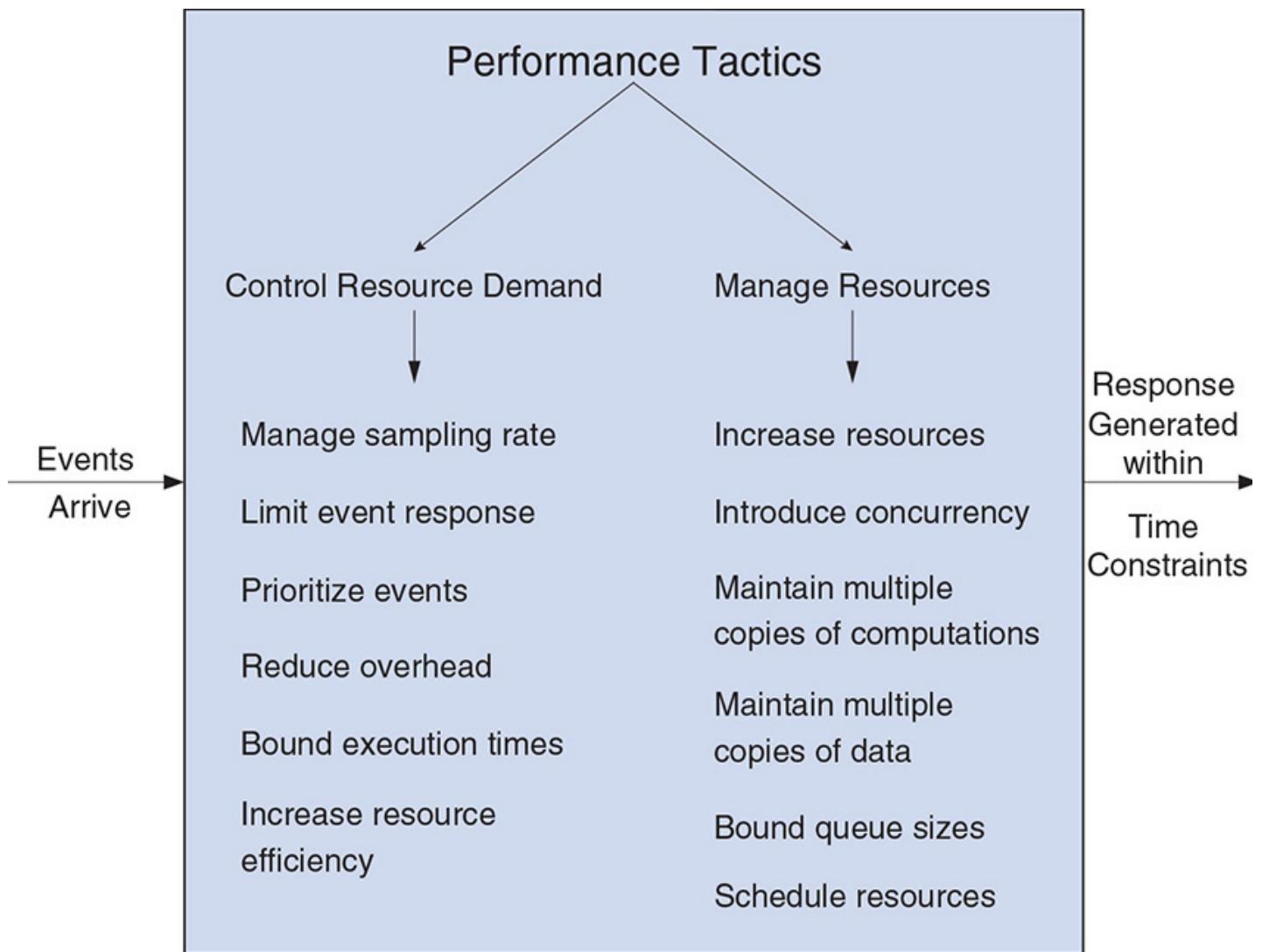


FIGURE 2.8 Tactics mediate events and responses.

Tactics are both simpler and more primitive than patterns. They focus on the control of a single quality attribute response (although they may, of course, trade off this response with other quality attribute goals). Patterns, in contrast, typically focus on resolving and balancing multiple forces—that is, multiple quality attribute goals. By way of analogy, we can say that a tactic is an atom, whereas a pattern is a molecule.

Tactics provide a top-down way of thinking about design. A tactics categorization begins with a set of design objectives related to the achievement of a quality attribute, and presents the architect with a set of options from which to choose. These options then need to be further instantiated through some combination of patterns, frameworks, and code.

For example, in [Figure 2.9](#), the design objectives for performance are "[Control Resource Demand](#)" and "[Manage Resources](#)". An architect who wants to create a system with "good" performance needs to choose one or more of these options. That is, the architect needs to decide if controlling resource demand is feasible, and if managing resources is feasible. In some systems, the events arriving at the system can be managed, prioritized, or limited in some way. If this is not possible, then the architect can manage resources only as part of an attempt to generate responses within acceptable time constraints. Within the "[Manage Resources](#)" category, an architect might choose to increase resources, introduce concurrency, maintain multiple copies of computations, maintain multiple copies of data, and so forth. These tactics then need to be instantiated. As an example, an architect might choose the Half-Sync/Half-Async pattern (see [Figure 2.5](#)) as a way of introducing (and managing) concurrency, or the Load-Balanced Cluster deployment pattern (see [Figure 2.7](#)) to maintain multiple copies of computations. As we will see in [Chapter 3](#), the choice, combination, and tailoring of tactics and patterns are some of the key steps of the ADD process. There are existing tactics categorizations for the quality attributes of availability, interoperability, modifiability, performance, security, testability, and usability.

FIGURE 2.9 Performance tactics from *Software Architecture in Practice*

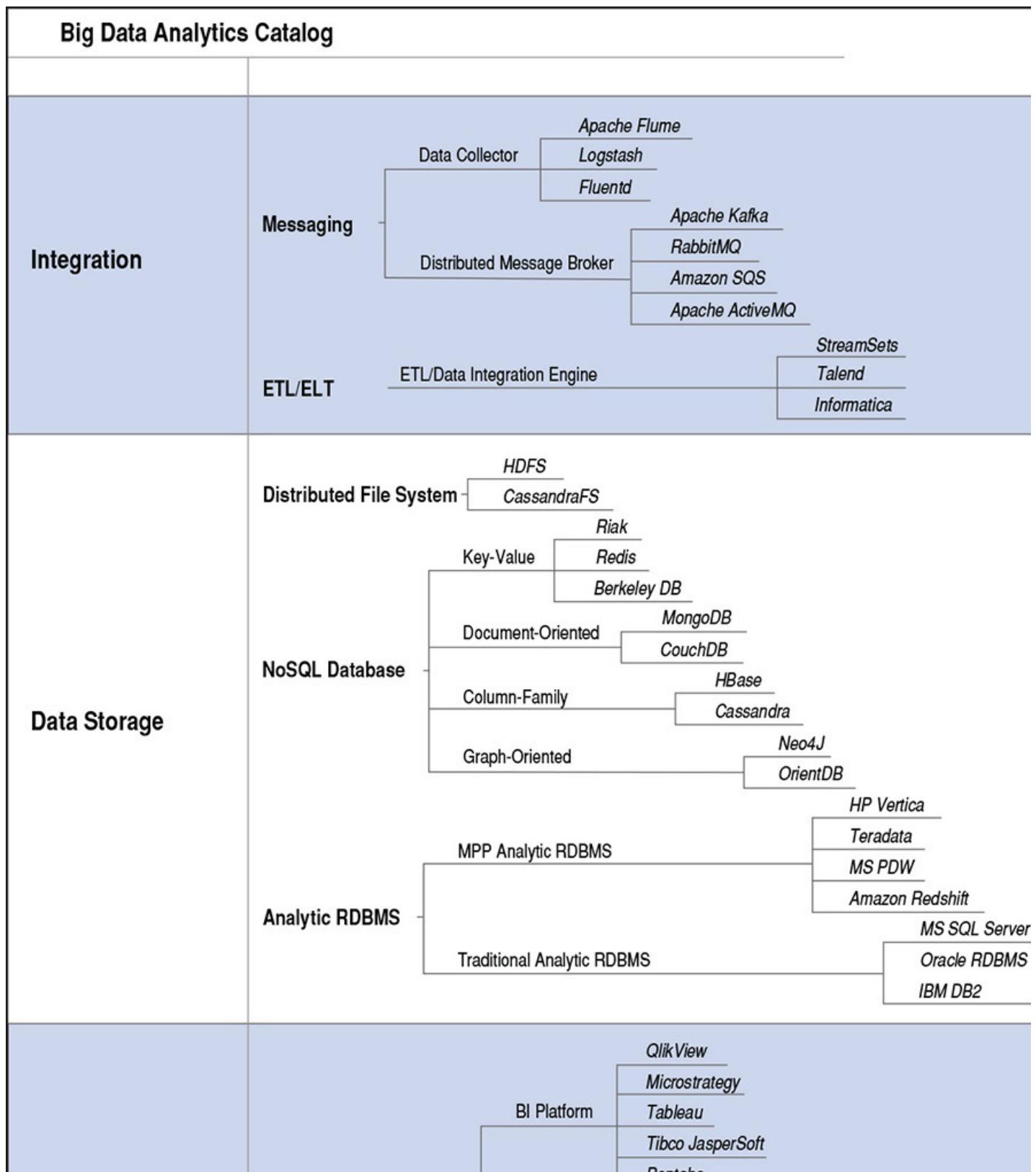
Username: Iowa State University Library **Book:** Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

2.5.5 Externally Developed Components

Patterns and tactics are abstract in nature. However, when you are designing a software architecture, you need to make these design concepts concrete and closer to the actual implementation. There are two ways to achieve this: You can code the elements obtained from tactics and patterns or you can associate technologies with one or more of these elements in the architecture. This "buy versus build" choice is one of the most important decisions you will make as an architect.

We consider technologies to be *externally developed components*, because they are not created as part of the development project. Several types of externally developed components exist:

■ **Technology families.** A *technology family* represents a group of specific technologies with common functional purposes. It can serve as a placeholder until a specific product or framework is selected. An example is a relational database management system (RDBMS) or an object-oriented to relational mapper (ORM). [Figure 2.10](#) shows different technology families in the Big Data domain (in regular text).



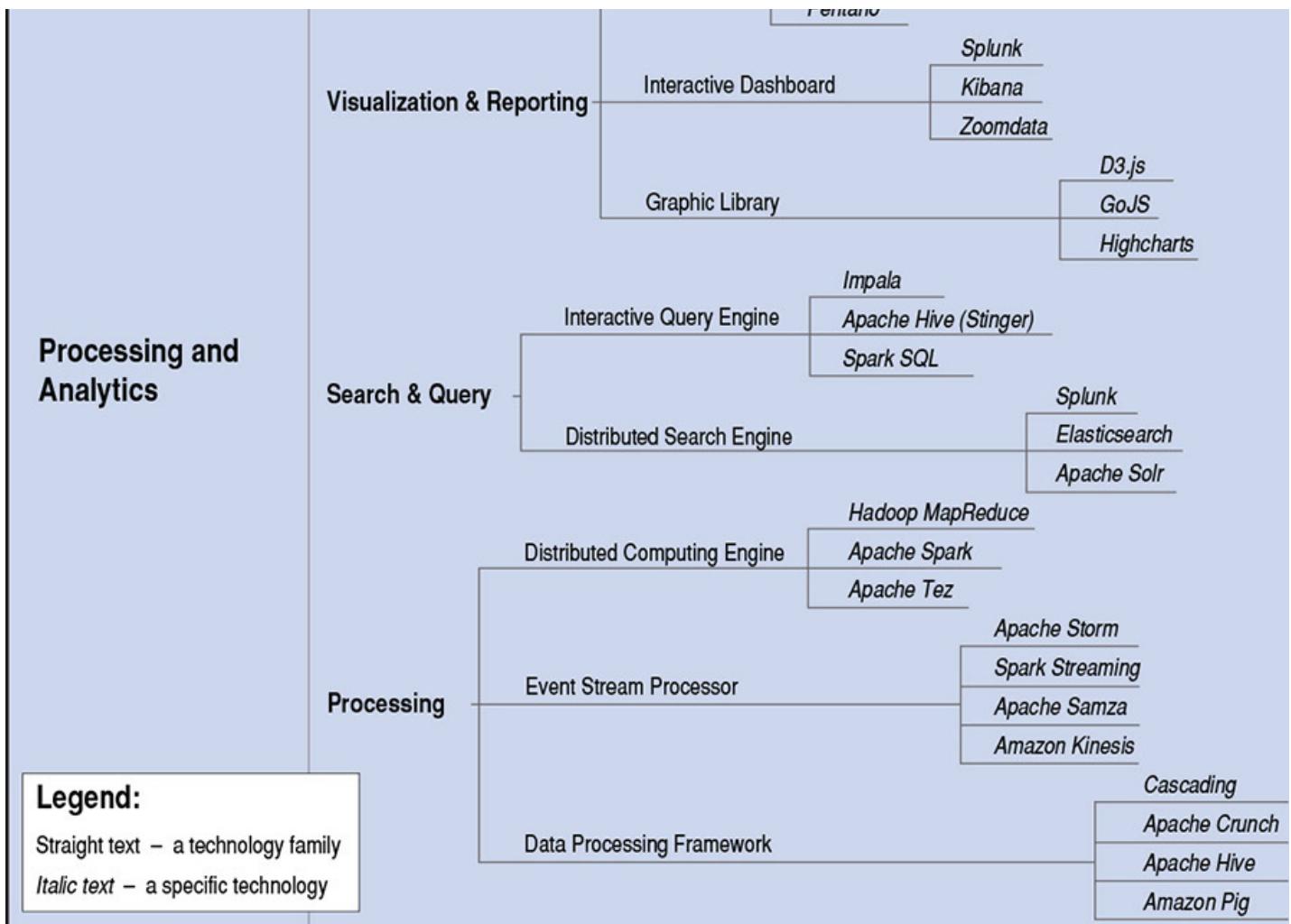


FIGURE 2.10 A technology family tree for the Big Data application domain

■ **Products.** A product (or software package) refers to a self-contained functional piece of software that can be integrated into the system that is being designed and that requires only minor configuration or coding. An example is a relational database management system, such as Oracle or Microsoft SQL Server. [Figure 2.10](#) shows different products in the Big Data domain (in *italics*).

■ **Application frameworks.** An application framework (or just framework) is a reusable software element, constructed out of patterns and tactics, that provides generic functionality addressing recurring domain and quality attribute concerns across a broad range of applications. Frameworks, when carefully chosen and properly implemented, increase the productivity of programmers. They do so by enabling programmers to focus on business logic and end-user value, rather than underlying technologies and their implementations. As opposed to products, framework functions are generally invoked from the application code or are “injected” using some type of aspect-oriented approach. Frameworks usually require extensive configuration, typically through XML files or other approaches such as annotations in Java. A framework example is Hibernate, which is used to perform object-oriented to relational mapping in Java. Several types of frameworks are available: Full-stack frameworks, such as Spring, are usually associated with reference architectures and address general concerns across the different elements of the reference architecture, while non-full-stack frameworks, such as JSF, address specific functional or quality attribute concerns.

■ **Platforms.** A platform provides a complete infrastructure upon which to build and execute applications. Examples of platforms include Java, .Net, or and Google Cloud.

The selection of externally developed components, which is a key aspect of the design process, can be a challenging task because of their extensive number. Here are a few criteria you should consider when selecting externally developed components:

■ **Problem that it addresses.** Is it something specific, such as a framework for object-oriented to relational mapping or something more generic, such as a platform?

■ **Cost.** What is the cost of the license and, if it is free, what is the cost of support and education?

■ **Type of license.** Does it have a license that is compatible with the project goals?

■ **Support.** Is it well supported? Is there extensive documentation about the technology? Is there an extensive user or developer community that you can turn to for advice?

■ **Learning curve.** How hard is it to learn this technology? Have others in your organization already mastered it? Are there courses available?

■ **Maturity.** Is it a technology that has just appeared on the market, which may be exciting but still relatively unstable or unsupported?

■ **Popularity.** Is it a relatively widespread technology? Are there positive testimonials or adoption by mature organizations? Will it be easy to hire people who have deep knowledge of it? Is there an active developer community or user group?

■ **Compatibility and ease of integration.** Is it compatible with other technologies used in the project? Can it be integrated easily in the project?

■ **Support for critical quality attributes.** Does it limit attributes such as performance? Is it secure and robust?

■ **Size.** Will the use of the technology have a negative impact on the size of the application under development?

Unfortunately, the answers to these questions are not always easy to find and the selection of a particular technology may require you do some research or, eventually, to create prototypes that will help you in the selection process. These criteria will have a significant effect on your total cost of ownership.

Username: Iowa State University Library **Book:** Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

2.6 Architecture Design Decisions

As we said at the beginning of this chapter, design is the process of making decisions. But the act of making a decision is a *process*, not a moment in time. Experienced architects, when faced with a design challenge, typically entertain a set of “candidate” decisions (as shown in [Figure 2.1](#)); from this set, they choose a best candidate and instantiate that. They might select this “best” candidate based on experience, constraints, or some form of analysis such as prototyping or simulation. The reality is that an architect will often make a choice and “ride the horse until it drops”—that is, commit to a decision and revisit it only if it appears to be compromising the success of the project. These decisions have serious consequences!

Recall that, in the early stages of design, decisions focus on the biggest, most critical choices that will have substantial downstream consequences: reference architectures, major technologies (such as frameworks), and patterns. Reference architectures, deployment patterns, and other kinds of patterns have been widely discussed—there are many books, websites, and conferences devoted to the creation and validation of patterns and pattern languages. Nevertheless, the output of these activities is always a set of documented patterns. Interpreting the patterns from a pattern catalog is a critical part of the selection activity for an architect. Each candidate pattern must be chosen and its instantiation must be analyzed. For example, if you chose the Layers pattern from [Figure 2.4](#), you would still have many decisions to make: how many layers there will be, how strict the layering will be, which specific services will be placed into each layer, what the interfaces between these functions will be, and so forth. If you chose the Load-Balanced Cluster deployment pattern from [Figure 2.7](#), you would have to decide how many servers will be balanced, how many load balancers you will use, where these servers and load balancers will physically reside, which kinds of networks will connect these servers, which form of encryption you will use on those network connections, which form of health monitoring the load balancers will employ, and so forth. These decisions are important and will affect the success of the instantiated pattern, so they need to be analyzed. In addition, the quality of the *implementation* of these decisions will affect the success of the pattern. As we like to quip, the architecture giveth and the implementation taketh away.

Furthermore, the many catalogs and web pages that present design concepts use different conventions and notations. The focus of our book is on the design method and how it can be used with these external sources. For this reason we just take examples from outside sources and show them here as they were originally presented. This book is not intended to be another design patterns catalog—we want to alert you to the presence of these catalogs and show how they can be an incredibly useful resource for an architect, but they must be interpreted and used with care! In fact, one of your many jobs as an architect is to understand and interpret these catalogs, with their different notations and conventions. This is the reality that you will have to deal with.

Finally, once a [design decision](#) has been made, you should think about how you will *document* it. You could, of course, do no documentation. This is, in fact, what is most common in practice. Architectural concepts are often vague and conveyed informally, in “tribal knowledge”: personal communications, emails, naming conventions, and so forth. Alternatively, you could create and maintain full, formal documentation, as is done for some projects with demanding quality attribute requirements, such as safety-critical or high-security systems. If you are designing flight-control software, you will probably end up at this end of the spectrum. In between these endpoints is a broad set of possibilities, and in this space we see less formal (and less costly) forms of architecture documentation, such as sketches (as we will discuss in [Section 3.7](#)).

The decision of what, when, and how to document should be risk based. You should ask yourself: What is the risk of *not* documenting this decision? Could it be misinterpreted and undermined by future developers? Could it contribute to near-term or long-term problems in the system? For example, if the rationale for layering is not carefully documented, the layering will inevitably break down, losing coherence and tending toward increased coupling. Over time, this trend will increase the system’s technical debt, making it harder to find and fix bugs or add new features. To take another example, if the rationale for allocation of a critical resource is not documented, that resource might become an unintended contention area, resulting in bottlenecks and failures.

Username: Iowa State University Library **Book:** Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

2.7 Summary

In this chapter, we introduced the idea of design as a set of decisions to satisfy requirements and constraints. We also introduced the notion of “architectural” design and showed that it does not differ from design in general, other than that it addresses the satisfaction of *architectural drivers*: the purpose, primary functionality, quality attribute requirements, architectural concerns, and constraints. What makes a decision “architectural”? A decision is architectural if it has nonlocal consequences *and* those consequences matter to the achievement of an architectural driver.

We also discussed why architectural design is so important: because it is the embodiment of early, far-reaching, hard-to-change decisions. These decisions will help you meet your architectural drivers, will determine much of your project’s work-breakdown structure, and will affect the tools, skills, and technologies needed to realize the system. Thus architectural design decisions should be scrutinized well, as their consequences are profound. In addition, architecture is a key enabler of agility.

Architectural design is guided by certain principles. For example, to achieve good modularity, high coupling, and low cohesion, the wise architect will probably include some form of layering in the architecture being designed. Similarly, to achieve high availability, an architect will likely choose a pattern involving some form of redundancy and failover, such as active–passive redundancy, where an active server sends real-time updates to a passive server, so that the passive server can replace the active server in case it fails, with no loss of state.

Design concepts, such as reference architectures, deployment patterns, architectural patterns, tactics, and externally developed components, are the building blocks of design, and they form the foundation for architectural design as it is performed using ADD. As you will see in our step-by-step explanation of ADD in [Chapter 3](#), some of the most important design decisions that an architect makes are how design concepts are selected, how they are instantiated, and how they are combined. Also, in [Appendix A](#), we present a design concepts catalog that includes several instances of the design concepts presented here.

From these foundations, an architecture can be confidently and predictably constructed.

Username: Iowa State University Library **Book:** Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

2.8 Further Reading

A more in-depth treatment of scenarios and architectural drivers can be found in L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed., Addison-Wesley, 2012. Also found in this book is an extensive discussion of architectural tactics, which are useful in guiding an architecture to achieve quality attribute goals. Likewise, this book contains an extensive discussion of QAW and Utility Trees.

The Mission Thread Workshop is discussed in R. Kazman, M. Gagliardi, and W. Wood, "Scaling Up Software Architecture Analysis", *Journal of Systems and Software*, 85, 1511–1519, 2012; and in M. Gagliardi, W. Wood, and T. Morrow, *Introduction to the Mission Thread Workshop*, Software Engineering Institute Technical Report CMU/SEI-2013-TR-003, 2013.

An overview of discovery prototyping, JRP, JAD, and accelerated systems analysis can be found in any competent book on systems analysis and design, such as J. Whitten and L. Bentley, *Systems Analysis and Design Methods*, 7th ed., McGraw-Hill, 2007. The combination of architectural approaches with Agile methods will be discussed in [Chapter 9](#).

A catalog of reference architectures and deployment patterns appears in the book by the Microsoft Patterns and Practices Team: *Microsoft® Application Architecture Guide*, 2nd ed., Microsoft Press, 2009. This book also provides an extensive list of architectural concerns associated with the reference architectures that are documented.

An extensive collection of architectural design patterns for the construction of distributed systems can be found in F. Buschmann, K. Henney, and D. Schmidt, *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*, Wiley, 2007. Other books in the POSA (Patterns Of Software Architecture) series provide additional pattern catalogs. Many other pattern catalogs specializing in particular application domains and technologies exist. A few examples are listed here:

- E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.
- E. Fernandez-Buglioni. *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*. Wiley, 2013.
- G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2004.

The evaluation and selection of software packages is discussed in A. Jadhav and R. Sonar, "Evaluating and Selecting Software Packages: A Review", *Journal of Information and Software Technology*, 51, 555–563, 2009.

The "bible" for software architecture documentation is P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond*, 2nd ed., Addison-Wesley, 2011.

The technology family tree for the Big Data application domain is based on the Smart Decisions Game by H. Cervantes, S. Haziye, O. Hrytsay, and R. Kazman, which can be found at <http://smartdecisionsgame.com>.

Chapter 1

Understanding Software Architecture

1.1 What is Software Architecture?

The last 15 years have seen a tremendous rise in the prominence of a software engineering subdiscipline known as software architecture. *Technical Architect* and *Chief Architect* are job titles that now abound in the software industry. There's an International Association of Software Architects,¹ and even a certain well-known wealthiest geek on earth used to have "architect" in his job title in his prime. It can't be a bad gig, then?

I have a sneaking suspicion that "architecture" is one of the most overused and least understood terms in professional software development circles. I hear it regularly misused in such diverse forums as project reviews and discussions, academic paper presentations at conferences and product pitches. You know a term is gradually becoming vacuous when it becomes part of the vernacular of the software industry sales force.

This book is about software architecture. In particular it's about the key design and technology issues to consider when building server-side systems that process multiple, simultaneous requests from users and/or other software systems. Its aim is to concisely describe the essential elements of knowledge and key skills that are required to be a software architect in the software and information technology (IT) industry. Conciseness is a key objective. For this reason, by no means everything an architect needs to know will be covered. If you want or need to know more, each chapter will point you to additional worthy and useful resources that can lead to far greater illumination.

So, without further ado, let's try and figure out what, at least I think, software architecture really is, and importantly, isn't. The remainder of this chapter will address this question, as well as briefly introducing the major tasks of an architect, and the relationship between architecture and technology in IT applications.

¹<http://www.iasahome.org/web/home/home>

1.2 Definitions of Software Architecture

Trying to define a term such as software architecture is always a potentially dangerous activity. There really is no widely accepted definition by the industry. To understand the diversity in views, have a browse through the list maintained by the Software Engineering Institute.² There's a lot. Reading these reminds me of an anonymous quote I heard on a satirical radio program recently, which went something along the lines of "the reason academic debate is so vigorous is that there is so little at stake".

I've no intention of adding to this debate. Instead, let's examine three definitions. As an IEEE member, I of course naturally start with the definition adopted by my professional body:

Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.

[ANSI/IEEE Std 1471-2000, *Recommended Practice for Architectural Description of Software-Intensive Systems*]

This lays the foundations for an understanding of the discipline. Architecture captures system structure in terms of components and how they interact. It also defines system-wide design rules and considers how a system may change.

Next, it's always worth getting the latest perspective from some of the leading thinkers in the field.

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

[L.Bass, P.Clements, R.Kazman, *Software Architecture in Practice (2nd edition)*, Addison-Wesley 2003]

This builds somewhat on the above ANSI/IEEE definition, especially as it makes the role of abstraction (i.e., externally visible properties) in an architecture and multiple architecture views (structures of the system) explicit. Compare this with another, from Garlan and Shaw's early influential work:

[Software architecture goes] beyond the algorithms and data structures of the computation; designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.

[D. Garlan, M. Shaw, *An Introduction to Software Architecture*, Advances in Software Engineering and Knowledge Engineering, Volume I, World Scientific, 1993]

It's interesting to look at these, as there is much commonality. I include the third mainly as it's again explicit about certain issues, such as scalability and

²<http://www.sei.cmu.edu/architecture/definitions.html>

distribution, which are implicit in the first two. Regardless, analyzing these a little makes it possible to draw out some of the fundamental characteristics of software architectures. These, along with some key approaches, are described below.

1.2.1 *Architecture Defines Structure*

Much of an architect’s time is concerned with how to sensibly partition an application into a set of interrelated components, modules, objects or whatever unit of software partitioning works for you.³ Different application requirements and constraints will define the precise meaning of “sensibly” in the previous sentence – an architecture must be designed to meet the specific requirements and constraints of the application it is intended for.

For example, a requirement for an information management system may be that the application is distributed across multiple sites, and a constraint is that certain functionality and data must reside at each site. Or, an application’s functionality must be accessible from a web browser. All these impose some structural constraints (site-specific, web server hosted), and simultaneously open up avenues for considerable design creativity in partitioning functionality across a collection of related components.

In partitioning an application, the architect assigns responsibilities to each constituent component. These responsibilities define the tasks a component can be relied upon to perform within the application. In this manner, each component plays a specific role in the application, and the overall component ensemble that comprises the architecture collaborates to provide the required functionality.

Responsibility-driven design (see *Wirfs-Brock* in Further Reading) is a technique from object-orientation that can be used effectively to help define the key components in an architecture. It provides a method based on informal tools and techniques that emphasize behavioral modeling using objects, responsibilities and collaborations. I’ve found this extremely helpful in past projects for structuring components at an architectural level.

A key structural issue for nearly all applications is minimizing dependencies between components, creating a loosely coupled architecture from a set of highly cohesive components. A dependency exists between components when a change in one potentially forces a change in others. By eliminating unnecessary dependencies, changes are localized and do not propagate throughout an architecture (see Fig. 1.1).

³Component here and in the remainder of this book is used very loosely to mean a recognizable “chunk” of software, and not in the sense of the more strict definition in Szyperski C. (1998) *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley

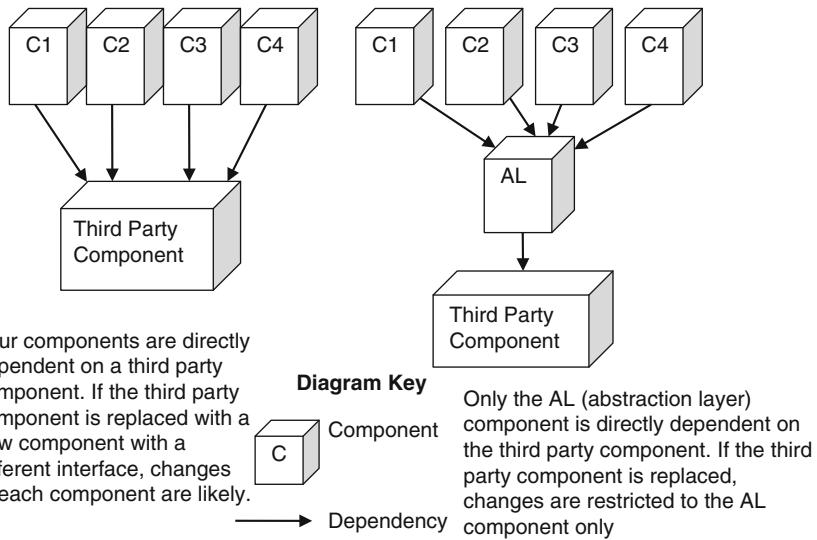


Fig. 1.1 Two examples of component dependencies

Excessive dependencies are simply a bad thing. They make it difficult to make changes to systems, more expensive to test changes, they increase build times, and they make concurrent, team-based development harder.

1.2.2 *Architecture Specifies Component Communication*

When an application is divided into a set of components, it becomes necessary to think about how these components communicate data and control information. The components in an application may exist in the same address space, and communicate via straightforward method calls. They may execute in different threads or processes, and communicate through synchronization mechanisms. Or multiple components may need to be simultaneously informed when an event occurs in the application's environment. There are many possibilities.

A body of work known collectively as architectural patterns or styles⁴ has catalogued a number of successfully used structures that facilitate certain kinds of component communication [see *Patterns* in Further Reading]. These patterns are essentially reusable architectural blueprints that describe the structure and interaction between collections of participating components.

Each pattern has well-known characteristics that make it appropriate to use in satisfying particular types of requirements. For example, the client–server pattern

⁴Patterns and styles are essentially the same thing, but as a leading software architecture author told me recently, “the patterns people won”. This book will therefore use patterns instead of styles!

has several useful characteristics, such as synchronous request–reply communications from client to server, and servers supporting one or more clients through a published interface. Optionally, clients may establish sessions with servers, which may maintain state about their connected clients. Client–server architectures must also provide a mechanism for clients to locate servers, handle errors, and optionally provide security on server access. All these issues are addressed in the client–server architecture pattern.

The power of architecture patterns stems from their utility, and ability to convey design information. Patterns are proven to work. If used appropriately in an architecture, you leverage existing design knowledge by using patterns.

Large systems tend to use multiple patterns, combined in ways that satisfy the architecture requirements. When an architecture is based around patterns, it also becomes easy for team members to understand a design, as the pattern infers component structure, communications and abstract mechanisms that must be provided. When someone tells me their system is based on a three-tier client–server architecture, I know immediately a considerable amount about their design. This is a very powerful communication mechanism indeed.

1.3 Architecture Addresses Nonfunctional Requirements

Nonfunctional requirements are the ones that don't appear in use cases. Rather than define *what* the application does, they are concerned with *how* the application provides the required functionality.

There are three distinct areas of nonfunctional requirements:

- *Technical constraints*: These will be familiar to everyone. They constrain design options by specifying certain technologies that the application must use. “We only have Java developers, so we must develop in Java”. “The existing database runs on Windows XP only”. These are usually nonnegotiable.
- *Business constraints*: These too constraint design options, but for business, not technical reasons. For example, “In order to widen our potential customer base, we must interface with XYZ tools”. Another example is “The supplier of our middleware has raised prices prohibitively, so we’re moving to an open source version”. Most of the time, these too are nonnegotiable.
- *Quality attributes*: These define an application’s requirements in terms of scalability, availability, ease of change, portability, usability, performance, and so on. Quality attributes address issues of concern to application users, as well as other stakeholders like the project team itself or the project sponsor. Chapter 3 discusses quality attributes in some detail.

An application architecture must therefore explicitly address these aspects of the design. Architects need to understand the functional requirements, and create a platform that supports these and simultaneously satisfies the nonfunctional requirements.

1.3.1 Architecture Is an Abstraction

One of the most useful, but often nonexistent, descriptions from an architectural perspective is something that is colloquially known as a *marketecture*. This is one page, typically informal depiction of the system's structure and interactions. It shows the major components and their relationships and has a few well-chosen labels and text boxes that portray the design philosophies embodied in the architecture. A *marketecture* is an excellent vehicle for facilitating discussion by stakeholders during design, build, review, and of course the sales process. It's easy to understand and explain and serves as a starting point for deeper analysis.

A thoughtfully crafted *marketecture* is particularly useful because it is an abstract description of the system. In reality, any architectural description must employ abstraction in order to be understandable by the team members and project stakeholders. This means that unnecessary details are suppressed or ignored in order to focus attention and analysis on the salient architectural issues. This is typically done by describing the components in the architecture as black boxes, specifying only their *externally visible properties*. Of course, describing system structure and behavior as collections of communicating black box abstractions is normal for practitioners who use object-oriented design techniques.

One of the most powerful mechanisms for describing an architecture is hierarchical decomposition. Components that appear in one level of description are decomposed in more detail in accompanying design documentation. As an example, Fig. 1.2 depicts a very simple two-level hierarchy using an informal notation, with two of the components in the top-level diagram decomposed further.

Different levels of description in the hierarchy tend to be of interest to different developers in a project. In Fig. 1.2, it's likely that the three components in the top-level description will be designed and built by different teams working on the

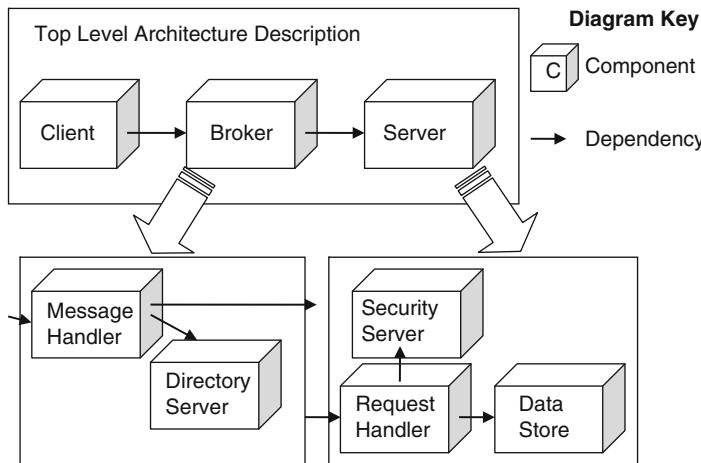


Fig. 1.2 Describing an architecture hierarchically

application. The architecture clearly partitions the responsibilities of each team, defining the dependencies between them.

In this hypothetical example, the architect has refined the design of two of the components, presumably because some nonfunctional requirements dictate that further definition is necessary. Perhaps an existing security service must be used, or the *Broker* must provide a specific message routing function requiring a directory service that has a known level of request throughput. Regardless, this further refinement creates a structure that defines and constrains the detailed design of these components.

The simple architecture in Fig. 1.2 doesn't decompose the *Client* component. This is, again presumably, because the internal structure and behavior of the client is not significant in achieving the application's overall nonfunctional requirements. How the *Client* gets the information that is sent to the *Broker* is not an issue that concerns the architect, and consequently the detailed design is left open to the component's development team. Of course, the *Client* component could possibly be the most complex in the application. It might have an internal architecture defined by its design team, which meets specific quality goals for the *Client* component. These are, however, localized concerns. It's not necessary for the architect to complicate the application architecture with such issues, as they can be safely left to the *Client* design team to resolve. This is an example of suppressing unnecessary details in the architecture.

1.3.2 Architecture Views

A software architecture represents a complex design artifact. Not surprisingly then, like most complex artifacts, there are a number of ways of looking at and understanding an architecture. The term “architecture views” rose to prominence in Philippe Krutchen’s 1995⁵ paper on the *4+1 View Model*. This presented a way of describing and understanding an architecture based on the following four views:

- *Logical view*: This describes the architecturally significant elements of the architecture and the relationships between them. The logical view essentially captures the structure of the application using class diagrams or equivalents.
- *Process view*: This focuses on describing the concurrency and communications elements of an architecture. In IT applications, the main concerns are describing multithreaded or replicated components, and the synchronous or asynchronous communication mechanisms used.
- *Physical view*: This depicts how the major processes and components are mapped on to the applications hardware. It might show, for example, how the database and web servers for an application are distributed across a number of server machines.

⁵P.Krutchen, *Architectural Blueprints—The “4+1” View Model of Software Architecture*, IEEE Software, 12(6) Nov. 1995.

- *Development view:* This captures the internal organization of the software components, typically as they are held in a development environment or configuration management tool. For example, the depiction of a nested package and class hierarchy for a Java application would represent the development view of an architecture.

These views are tied together by the architecturally significant use cases (often called scenarios). These basically capture the requirements for the architecture and hence are related to more than one particular view. By working through the steps in a particular use case, the architecture can be “tested”, by explaining how the design elements in the architecture respond to the behavior required in the use case. We’ll explore how to do this “architecture testing” in Chap. 5.

Since Krutchen’s paper, there’s been much thinking, experience, and development in the area of architecture views. Mostly notably is the work from the SEI, colloquially known as the “Views and Beyond” approach (see Further Reading). This recommends capturing an architecture model using three different views:

- *Module:* This is a structural view of the architecture, comprising the code modules such as classes, packages, and subsystems in the design. It also captures module decomposition, inheritance, associations, and aggregations.
- *Component and connector:* This view describes the behavioral aspects of the architecture. Components are typically objects, threads, or processes, and the connectors describe how the components interact. Common connectors are sockets, middleware like CORBA or shared memory.
- *Allocation:* This view shows how the processes in the architecture are mapped to hardware, and how they communicate using networks and/or databases. It also captures a view of the source code in the configuration management systems, and who in the development group has responsibility for each modules.

The terminology used in “Views and Beyond” is strongly influenced by the architecture description language (ADL) research community. This community has been influential in the world of software architecture but has had limited impact on mainstream information technology. So while this book will concentrate on two of these views, we’ll refer to them as the structural view and the behavioral view. Discerning readers should be able to work out the mapping between terminologies!

1.4 What Does a Software Architect Do?

The environment that a software architect works in tends to define their exact roles and responsibilities. A good general description of the architect’s role is maintained by the SEI on their web site.⁶ Instead of summarizing this, I’ll briefly describe, in no

⁶http://www.sei.cmu.edu/ata/arch_duties.html

particular order, four essential skills for a software architect, regardless of their professional environment.

- *Liaison:* Architects play many liaison roles. They liaise between the customers or clients of the application and the technical team, often in conjunction with the business and requirements analysts. They liaise between the various engineering teams on a project, as the architecture is central to each of these. They liaise with management, justifying designs, decisions and costs. They liaise with the sales force, to help promote a system to potential purchasers or investors. Much of the time, this liaison takes the form of simply translating and explaining different terminology between different stakeholders.
- *Software Engineering:* Excellent design skills are what get a software engineer to the position of architect. They are an essential prerequisite for the role. More broadly though, architects must promote good software engineering practices. Their designs must be adequately documented and communicated and their plans must be explicit and justified. They must understand the downstream impact of their decisions, working appropriately with the application testing, documentation and release teams.
- *Technology Knowledge:* Architects have a deep understanding of the technology domains that are relevant to the types of applications they work on. They are influential in evaluating and choosing third party components and technologies. They track technology developments, and understand how new standards, features and products might be usefully exploited in their projects. Just as importantly, good architects know what they don't know, and ask others with greater expertise when they need information.
- *Risk Management:* Good architects tend to be cautious. They are constantly enumerating and evaluating the risks associated with the design and technology choices they make. They document and manage these risks in conjunction with project sponsors and management. They develop and instigate risk mitigation strategies, and communicate these to the relevant engineering teams. They try to make sure no unexpected disasters occur.

Look for these skills in the architects you work with or hire. Architects play a central role in software development, and must be multiskilled in software engineering, technology, management and communications.

1.5 Architectures and Technologies

Architects must make design decisions early in a project lifecycle. Many of these are difficult, if not impossible, to validate and test until parts of the system are actually built. Judicious prototyping of key architectural components can help increase confidence in a design approach, but sometimes it's still hard to be certain of the success of a particular design choice in a given application context.

Due to the difficulty of validating early design decisions, architects sensibly rely on tried and tested approaches for solving certain classes of problems. This is one of the great values of architectural patterns. They enable architects to reduce risk by leveraging successful designs with known engineering attributes.

Patterns are an abstract representation of an architecture, in the sense that they can be realized in multiple concrete forms. For example, the publish–subscribe architecture pattern describes an abstract mechanism for loosely coupled, many-to-many communications between publishers of messages and subscribers who wish to receive messages. It doesn't however specify how publications and subscriptions are managed, what communication protocols are used, what types of messages can be sent, and so on. These are all considered implementation details.

Unfortunately, despite the misguided views of a number of computer science academics, abstract descriptions of architectures don't yet execute on computers, either directly or through rigorous transformation. Until they do, abstract architectures must be reified by software engineers as concrete software implementations.

Fortunately, the software industry has come to the rescue. Widely utilized architectural patterns are supported in a variety of prebuilt frameworks available as commercial and open source technologies. For a matter of convenience, I'll refer to these collectively as commercial-off-the-shelf (COTS) technologies, even though it's strictly not appropriate as many open source products of very high quality can be freely used (often with a *pay-for-support* model for serious application deployments).

Anyway, if a design calls for publish–subscribe messaging, or a message broker, or a three-tier architecture, then the choices of available technology are many and varied indeed. This is an example of software technologies providing reusable, application-independent software infrastructures that implement proven architectural approaches.

As Fig. 1.3 depicts, several classes of COTS technologies are used in practice to provide packaged implementations of architectural patterns for use in IT systems. Within each class, competing commercial and open source products exist. Although these products are superficially similar, they will have differing feature sets, be implemented differently and have varying constraints on their use.

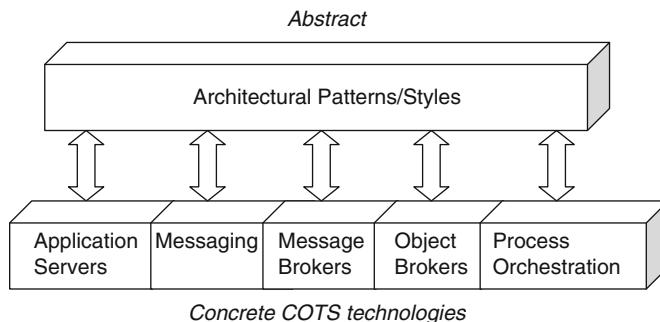


Fig. 1.3 Mapping between logical architectural patterns and concrete technologies

Architects are somewhat simultaneously blessed and cursed with this diversity of product choice. Competition between product vendors drives innovation, better feature sets and implementations, and lower prices, but it also places a burden on the architect to select a product that has quality attributes that satisfy the application requirements. All applications are different in some ways, and there is rarely, if ever, a *one-size-fits-all* product match. Different COTS technology implementations have different sets of strengths and weaknesses and costs, and consequently will be better suited to some types of applications than others.

The difficulty for architects is in understanding these strengths and weaknesses early in the development cycle for a project, and choosing an appropriate reification of the architectural patterns they need. Unfortunately, this is not an easy task, and the risks and costs associated with selecting an inappropriate technology are high. The history of the software industry is littered with poor choices and subsequent failed projects. To quote Eoin Woods,⁷ and provide another extremely pragmatic definition of software architecture:

Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled.

Chapters 4–6 provide a detailed description and analysis of these infrastructural technologies.

1.6 Architect Title Soup

Scan the jobs advertisements. You'll see chief architects, product architects, technical architects, solution architects (I want to place a spoof advert for a problem architect), enterprise architects, and no doubt several others. Here's an attempt to give some general insights into what these mean:

- *Chief Architect*: Typically a senior position who manages a team of architects within an organization. Operates at a broad, often organizational level, and coordinates efforts across system, applications, and product lines. Very experienced, with a rare combination of deep technical and business knowledge.
- *Product/Technical/Solution Architect*: Typically someone who has progressed through the technical ranks and oversees the architectural design for a specific system or application. They have a deep knowledge of how some important piece of software really works.
- *Enterprise Architect*: Typically a much less technical, more business-focus role. Enterprise architects use various business methods and tools to understand, document, and plan the structure of the major systems in an enterprise.

The content of this book is relevant to the first two bullets above, which require a strong computer science background. However, enterprise architects are somewhat

⁷<http://www.einwoods.info/>

different beasts. This all gets very confusing, especially when you're a software architect working on enterprise systems.

Essentially, enterprise architects create documents, roadmaps, and models that describe the logical organization of business strategies, metrics, business capabilities, business processes, information resources, business systems, and networking infrastructure within the enterprise.⁸ They use frameworks to organize all these documents and models, with the most popular ones being TOGAF⁹ and the Zachman Framework.¹⁰

Now if I'm honest, the above pretty much captures all I know about enterprise architecture, despite having been involved for a short time on an enterprise architecture effort! I'm a geek at heart, and I have never seen any need for computer science and software engineering knowledge in enterprise architecture. Most enterprise architects I know have business or information systems degrees. They are concerned with how to "align IT strategy and planning with company's business goals", "develop policies, standards, and guidelines for IT selection", and "determine governance". All very lofty and important concerns, and I don't mean to be disparaging, but these are not my core interests. The tasks of an enterprise architect certainly don't rely on a few decades of accumulated computer science and software engineering theory and practice.

If you're curious about enterprise architecture, there are some good references at the end of this chapter. Enjoy.

1.7 Summary

Software architecture is a fairly well defined and understood design discipline. However, just because we know what it is and more or less what needs doing, this doesn't mean it's mechanical or easy. Designing and evaluating an architecture for a complex system is a creative exercise, requiring considerable knowledge, experience and discipline. The difficulties are exacerbated by the early lifecycle nature of much of the work of an architect. To my mind, the following quote from Philippe Krutchen sums up an architect's role perfectly:

The life of a software architect is a long (and sometimes painful) succession of sub-optimal decisions made partly in the dark

The remainder of this book will describe methods and techniques that can help you to shed at least some light on architectural design decisions. Much of this light comes from understanding and leveraging design principles and technologies that have proven to work in the past. Armed with this knowledge, you'll be able to

⁸http://en.wikipedia.org/wiki/Enterprise_Architecture

⁹<http://www.opengroup.org/togaf/>

¹⁰<http://www.zachmaninternational.com/index.php/the-zachman-framework>

tackle complex architecture problems with more confidence, and after a while, perhaps even a little panache.

1.8 Further Reading

There are lots of good books, reports, and papers available in the software architecture world. Below are some I'd especially recommend. These expand on the information and messages covered in this chapter.

1.8.1 General Architecture

In terms of defining the landscape of software architecture and describing their project experiences, mostly with defense projects, it's difficult to go past the following books from members of the Software Engineering Institute.

- L. Bass, P. Clements, R Kazman. *Software Architecture in Practice*, Second Edition. Addison-Wesley, 2003.
- P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford. *Documenting Software Architectures: Views and Beyond*. 2nd Edition, Addison-Wesley, 2010.
- P. Clements, R. Kazman, M. Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, 2002.

For a description of the “Decomposition Style”, see *Documenting Software Architecture*, page 53. And for an excellent discussion of the *uses* relationship and its implications, see the same book, page 68.

The following are also well worth a read:

Nick Rozanski, Eion Woods, *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*, Addison-Wesley 2005

Richard N. Taylor, Nenad Medvidovic, Eric Dashofy, *Software Architecture: Foundations, Theory, and Practice*, John Wiley and Sons, 2009

Martin Fowler's article on the role of an architect is an interesting read.

Martin Fowler, *Who needs an Architect?* IEEE Software, July-August 2003.

1.8.2 Architecture Requirements

The original book describing use cases is:

- I. Jacobson, M. Christerson, P. Jonsson, G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

Responsibility-driven design is an incredibly useful technique for allocating functionality to components and subsystems in an architecture. The following should be compulsory reading for architects.

- R. Wirfs-Brock, A. McKean. Object Design: Roles, Responsibilities, and Collaborations. Addison-Wesley, 2002.

1.8.3 *Architecture Patterns*

There's a number of fine books on architecture patterns. Buschmann's work is an excellent introduction.

- F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal., *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, 1996.
- D. Schmidt, M. Stal, H. Rohnert, F. Buschmann. *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2000.

Two recent books that focus more on patterns for enterprise systems, especially enterprise application integrations, are well worth a read.

- M. Fowler. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002.
- G. Hohpe, B. Woolf. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley, 2003.

1.8.4 *Technology Comparisons*

A number of papers that emerged from the Middleware Technology Evaluation (MTE) project give a good introduction into the issues and complexities of technology comparisons.

- P. Tran, J. Gosper, I. Gorton. *Evaluating the Sustained Performance of COTS-based Messaging Systems*. in Software Testing, Verification and Reliability, vol 13, pp 229–240, Wiley and Sons, 2003.
- I. Gorton, A. Liu. *Performance Evaluation of Alternative Component Architectures for Enterprise JavaBean Applications*, in IEEE Internet Computing, vol.7, no. 3, pages 18–23, 2003.
- A. Liu, I. Gorton. *Accelerating COTS Middleware Technology Acquisition: the i-MATE Process*. in IEEE Software, pages 72–79, volume 20, no. 2, March/April 2003.

1.8.5 *Enterprise Architecture*

In my humble opinion, there's some seriously shallow books written about enterprise architecture. I survived through major parts of this book, so would recommend it as a starting point.

James McGovern, Scott Ambler, Michael Stevens, James Linn, Elias Jo and Vikas Sharan, *The Practical Guide to Enterprise Architecture*, Addison-Wesley, 2003.

Another good general, practical book is:

Marc Lankhorst, *Enterprise Architecture at Work*, Springer-Verlag, 2009

I'm sure there's joy to be had in the 700+ pages of the latest *TOGAF version 9.0* book (Van Haren publishing, ISBN: 9789087532307), but like Joyce's *Ulysses*, I suspect it's a joy I will never have the patience to savor. If the Zachman Framework is more your thing, there's a couple of *ebooks*, which look informative at a glance:

<http://www.zachmaninternational.com/index.php/ea-articles/25-editions>

Chapter 2

Introducing the Case Study

2.1 Overview

This chapter introduces the case study that will be used in subsequent chapters to illustrate some of the design principles in this book.¹ Very basically, the application is a multiuser software system with a database that is used to share information between users and intelligent tools that aim to help the user complete their work tasks more effectively. An informal context diagram is depicted in Fig. 2.1.

The system has software components that run on each user's workstation, and a shared distributed software "back-end" that makes it possible for intelligent third party tools to gather data from, and communicate with, multiple users in order to offer assistance with their task. It's this shared distributed software back-end that this case study will concentrate on, as it's the area where architectural complexity arises. It also illustrates many of the common quality issues that must be addressed by distributed, multiuser applications.

2.2 The ICDE System

The Information Capture and Dissemination Environment (ICDE) is part of a suite of software systems for providing intelligent assistance to professionals such as financial analysts, scientific researchers and intelligence analysts. To this end, ICDE automatically captures and stores data that records a range of actions performed by a user when operating a workstation. For example, when

¹The case study project is based on an actual system that I worked on. Some creative license has been exploited to simplify the functional requirements, so that these don't overwhelm the reader with unnecessary detail. Also, the events, technical details and context described do not always conform to reality, as reality can be far too messy for illustration purposes.

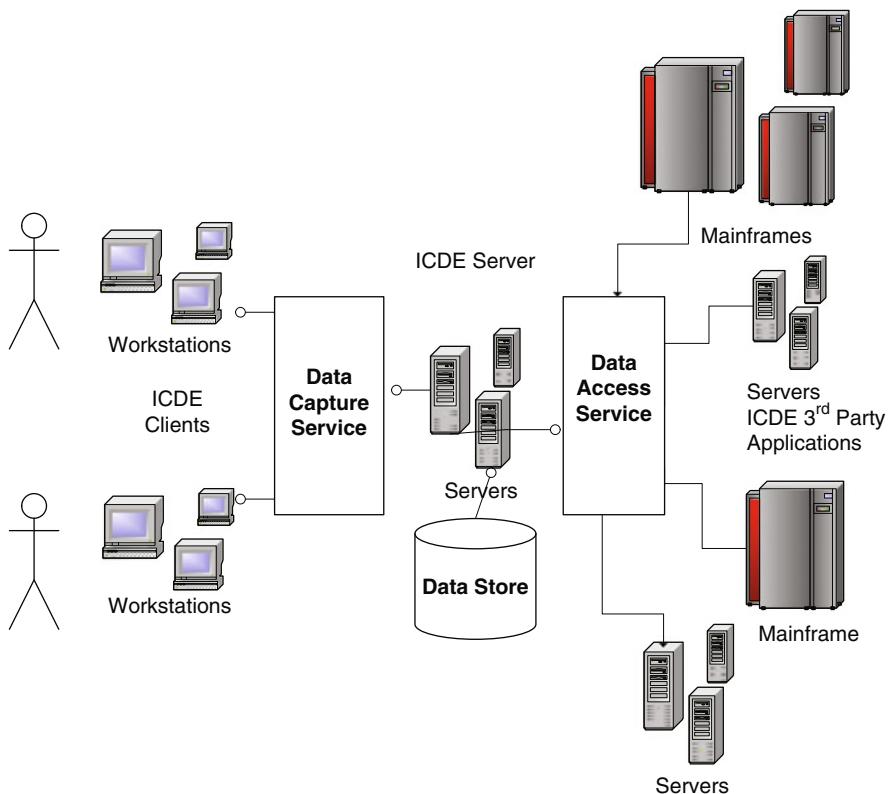


Fig. 2.1 ICDE context diagram

a user performs a Google search, the ICDE system will transparently store in a database:

- The search query string
- Copies of the web pages returned by Google that the user displays in their browser

This data can be subsequently retrieved from the ICDE database and used by third-party software tools that attempt to offer intelligent help to the user. These tools might interpret a sequence of user inputs, and try to find additional information to help the user with their current task. Other tools may crawl the links in the returned search results that the user does not click on, attempting to find potentially useful details that the user overlooks.

A use case diagram for the ICDE system is shown in Fig. 2.2. The three major use cases incorporate the capture of user actions, the querying of data from the data store, and the interaction of the third party tools with the user.

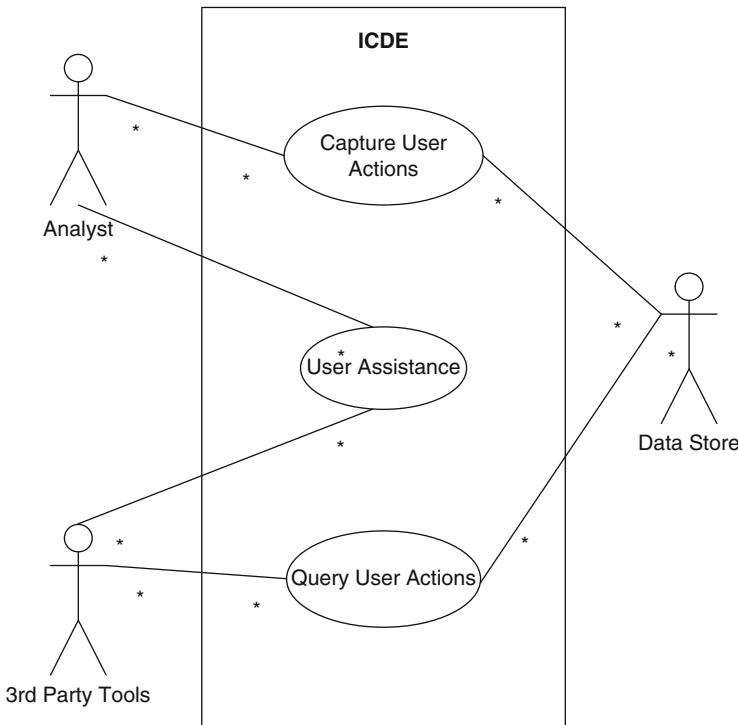


Fig. 2.2 ICDE system use cases

2.3 Project Context

Few real projects are green-field efforts, allowing the design team to start with a clean and mostly unconstrained piece of paper. The ICDE system certainly isn't one of these.

An initial production version (v1.0) of ICDE was implemented by a small development team. Their main aim was to implement the *Capture User Actions* use case. This created the client component that runs on each user workstation, and drove the design and implementation of the data store. This was important as the data store was an integral part of the rest of the system's functionality, and its design had to be suitable to support the high transaction rate that a large number of users could potentially generate.

ICDE v1.0 was only deployed in a small user trial involving a few users. This deployment successfully tested the client software functionality and demonstrated the concepts of data capture and storage. The design of v1.0 was based upon a simple two-tier architecture, with all components executing on the user's workstation. This design is shown as a UML component diagram in Fig. 2.3. The collection and analysis client components were written in Java and access the data store

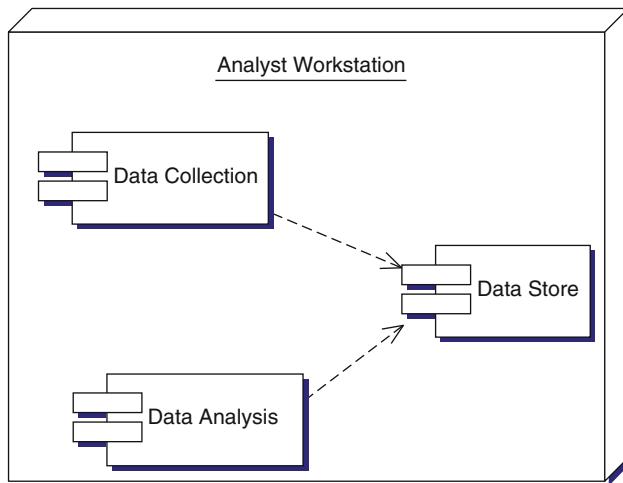


Fig. 2.3 ICDE Version 1.0 application architecture

(server) directly using the JDBC² API. The complete ICDE application executed on Microsoft Windows XP.

The role of each component is as follows:

- *Data Collection*: The collection component comprises a number of loosely coupled processes running on a client workstation that transparently track the user's relevant activities and store them in the *Data Store*. The captured events relate to Internet accesses, documents that are opened and browsed, edits made to documents, and some basic windowing information about when the user opens and closes applications on the desktop. Each event has numerous attributes associated with it, depending on event type. For example, a mouse double click has (x, y) coordinate attributes, and a window activation event has the associated application name as an attribute.
- *Data Store*: This component comprises a commercial-off-the-shelf (COTS) relational database. The relational database stores event information in various tables to capture the user activities, with timestamps added so that the order of events can be reconstructed. Large objects such as images on web pages and binary documents are stored as Binary Large Object Fields (BLOBS) using the native database facilities.
- *Data Analysis*: A graphical user interface (GUI) based tool supports a set of queries on the data store. This was useful for testing purposes, and to give the third party tool creators an initial look at the data that was being captured, and was hence available to them for analysis.

²Java Database Connectivity.

2.4 Business Goals

ICDE v2.0 had much more ambitious aims. Having proven that the system worked well in trial deployments, the project sponsors had two major business objectives for the next version. These were:

- Encourage third party tool developers to write applications for the ICDE system. For example, in finance, a third party developer might build a “stock advisor” that watches the stocks that an analyst is looking at in their browser and informs them of any events in the news that might affect the stock value.
- Promote the ICDE concept and tools to potential customers, in order to enhance their analytical working environment.

Clearly, both these objectives are focused on fostering a growing business around the ICDE technology, by creating an attractive market for third party tools and an advanced advisory environment for users in a range of application domains. Achieving these goals requires detailed technical and business plans to be drawn up and followed through. From a purely technical perspective, leaving out such activities as sales and marketing, the following major objectives were identified – see Table 2.1:

In order to attract third party tool developers, it is essential that the environment has a powerful and easy-to-use application programming interface (API) that could be accessed from any operating system platforms that a developer chooses to use. This would give tool developers flexibility in choosing their deployment platform, and make porting existing tools simpler. Surveys of existing tools also raised the issue that powerful analytical tools might require high-end cluster machines to run on. Hence they’d need the capability to communicate with ICDE deployments over local (and eventually wide) area networks.

Another survey of likely ICDE clients showed that potential user organizations had groups of 10–150 analysts. It was consequently important that the software could be easily scaled to support such numbers. There should also be no inherent design features that inhibit the technology from supporting larger deployments which may appear in the future.

Table 2.1 ICDE v2.0 business goals

Business goal	Supporting technical objective
Encourage third party tool developers	Simple and reliable programmatic access to data store for third party tools
	Heterogeneous (i.e., non-Windows) platform support for running third party tools
	Allow third party tools to communicate with ICDE users from a remote machine
Promote the ICDE concept to users	Scale the data collection and data store components to support up to 150 users at a single site
	Low-cost deployment for each ICDE user workstation

Equally important, to keep the base cost of a deployment as low as possible, expensive COTS technologies should be avoided wherever possible. This in turn will make the product more attractive in terms of price for clients.

2.5 Constraints

The technical objectives were ambitious, and would require a different architecture to support distributed data access and communications. For this reason, it was decided to concentrate efforts on this new architecture, and leave the client, including the GUI and data capture tools, stable. Changes would only be made to the client to enable it to communicate with the new data management and notification architecture that this project would design. For this reason, the client-side design is not dealt with in this case study.

A time horizon of 12 months was set for ICDE v2.0. An interim release after 6 months was planned to expose tool developers to the API, and allow them to develop their tools at the same time that ICDE v2.0 was being productized and enhanced.

As well as having a fixed schedule, the development budget was also fixed. This meant the development resources available would constrain the features that could be included in the v2.0 release. These budget constraints also influenced the possible implementation choices, given that the number of developers, their skills and time available was essentially fixed.

2.6 Summary

The ICDE application makes an interesting case study for a software architecture. It requires the architecture of an existing application to be extended and enhanced to create a platform for new features and capabilities. Time and budget constraints restrict the possible options. Certainly a redevelopment of the existing ICDE v1.0 client and data store is completely out of the question.

In Chap. 9, the design for the ICDE back-end will be elaborated and explained. The next few chapters aim to provide the necessary background knowledge in designing architectures to meet quality attributes, and exploiting technologies to make the creation of such systems tractable.

Chapter 3

Software Quality Attributes

3.1 Quality Attributes

Much of a software architect's life is spent designing software systems to meet a set of quality attribute requirements. General software quality attributes include scalability, security, performance and reliability. These are often informally called an application's “-ilities” (though of course some, like performance, don't quite fit this lexical specification).

Quality attribute requirements are part of an application's nonfunctional requirements, which capture the many facets of *how* the functional requirements of an application are achieved. All but the most trivial application will have nonfunctional requirements that can be expressed in terms of quality attribute requirements.

To be meaningful, quality attribute requirements must be specific about how an application should achieve a given need. A common problem I regularly encounter in architectural documents is a general statement such as “The application must be scalable”.

This is far too imprecise and really not much use to anyone. As is discussed later in this chapter, scalability requirements are many and varied, and each relates to different application characteristics. So, must this hypothetical application scale to handle increased simultaneous user connections? Or increased data volumes? Or deployment to a larger user base? Or all of these?

Defining which of these scalability measures must be supported by the system is crucial from an architectural perspective, as solutions for each differ. It's vital therefore to define concrete quality attribute requirements, such as:

It must be possible to scale the deployment from an initial 100 geographically dispersed user desktops to 10,000 without an increase in effort/cost for installation and configuration.

This is precise and meaningful. As an architect, this points me down a path to a set of solutions and concrete technologies that facilitate zero-effort installation and deployment.

Note however, that many quality attributes are actually somewhat difficult to validate and test. In this example, it'd be unlikely that in testing for the initial

release, a test case would install and configure the application on 10,000 desktops. I just can't see a project manager signing off on that test somehow.

This is where common sense and experience come in. The adopted solution must obviously function for the initial 100-user deployment. Based on the exact mechanisms used in the solution (perhaps Internet download, corporate desktop management software, etc), we can then only analyze it to the best of our ability to assess whether the concrete scalability requirement can be met. If there are no obvious flaws or issues, it's probably safe to assume the solution will scale. But will it scale to 10,000? As always with software, there's only one way to be absolutely, 100% sure, as "it is all talk until the code runs".¹

There are many general quality attributes, and describing them all in detail could alone fill a book or two. What follows is a description of some of the most relevant quality attributes for general IT applications, and some discussion on architectural mechanisms that are widely used to provide solutions for the required quality attributes. These will give you a good place to start when thinking about the qualities an application that you're working on must possess.

3.2 Performance

Although for many IT applications, performance is not a really big problem, it gets most of the spotlight in the crowded quality attribute community. I suspect this is because it is one of the qualities of an application that can often be readily quantified and validated. Whatever the reason, when performance matters, it *really* does matter. Applications that perform poorly in some critical aspect of their behavior are likely candidates to become road kill on the software engineering highway.

A performance quality requirement defines a metric that states the amount of work an application must perform in a given time, and/or deadlines that must be met for correct operation. Few IT applications have *hard real-time* constraints like those found in avionics or robotics systems, where if some output is produced a millisecond or three too late, really nasty and undesirable things can happen (I'll let the reader use their imagination here). But applications needing to process hundreds, sometimes thousands and tens of thousands of transactions every second are found in many large organizations, especially in the worlds of finance, telecommunications and government.

Performance usually manifests itself in the following measures.

3.2.1 Throughput

Throughput is a measure of the amount of work an application must perform in unit time. Work is typically measured in transactions per second (tps), or messages

¹Ward Cunningham at his finest!

processed per second (mps). For example, an on-line banking application might have to guarantee it can execute 1,000 tps from Internet banking customers. An inventory management system for a large warehouse might need to process 50 messages per second from trading partners requesting orders.

It's important to understand precisely what is meant by a throughput requirement. Is it average throughput over a given time period (e.g., a business day), or peak throughput? This is a crucial distinction.

A stark illustration of this is an application for placing bets on events such as horse racing. For most of the time, an application of this ilk does very little work (people mostly place bets just before a race), and hence has a low and easily achievable average throughput requirement. However, every time there is a racing event, perhaps every evening, the 5 or so minute period before each race sees thousands of bets being placed every second. If the application is not able to process these bets as they are placed, then the business loses money, and users become very disgruntled (and denying gamblers the opportunity to lose money is not a good thing for anyone). Hence for this scenario, the application must be designed to meet anticipated *peak* throughput, not average. In fact, supporting only average throughput would likely be a “career changing” design error for an architect.

3.2.2 Response Time

This is a measure of the latency an application exhibits in processing a business transaction. Response time is most often (but not exclusively) associated with the time an application takes to respond to some input. A rapid response time allows users to work more effectively, and consequently is good for business. An excellent example is a point-of-sale application supporting a large store. When an item is scanned at the checkout, a fast, second or less response from the system with the item's price means a customer can be served quickly. This makes the customer and the store happy, and that's a good thing for all involved stakeholders.

Again, it's often important to distinguish between guaranteed and average response times. Some applications may need *all* requests to be serviced within a specified time limit. This is a guaranteed response time. Others may specify an average response time, allowing larger latencies when the application is extremely busy. It's also widespread in the latter case for an upper bound response time requirement to be specified. For example, 95% of all requests must be processed in less than 4 s, and no requests must take more than 15 s.

3.2.3 Deadlines

Everyone has probably heard of the weather forecasting system that took 36 h to produce the forecast for the next day! I'm not sure if this is apocryphal, but it's an excellent example of the requirement to meet a performance deadline. Deadlines in

the IT world are commonly associated with batch systems. A social security payment system must complete in time to deposit claimant's payments in their accounts on a given day. If it finishes late, claimants don't get paid when they expect, and this can cause severe disruptions and pain, and not just for claimants. In general, any application that has a limited window of time to complete will have a performance deadline requirement.

These three performance attributes can all be clearly specified and validated. Still, there's a common pitfall to avoid. It lies in the definition of a transaction, request or message, all of which are deliberately used very imprecisely in the above. Essentially this is the definition of an application's workload. The amount of processing required for a given business transaction is an *application specific* measure. Even within an application, there will likely be many different types of requests or transactions, varying perhaps from fast database read operations, to complex updates to multiple distributed databases.

Simply, there is no generic workload measure, it depends entirely on what work the application is doing. So, when agreeing to meet a given performance measure, be precise about the exact workload or *transaction mix*, defined in application-specific terms, that you're signing up for.

3.2.4 Performance for the ICDE System

Performance in the ICDE system is an important quality attribute. One of the key performance requirements pertains to the interactive nature of ICDE. As users perform their work tasks, the client portion of the ICDE application traps key and mouse actions and sends these to the ICDE server for storage. It is consequently extremely important that ICDE users don't experience any delays in using their applications while the ICDE software traps and stores events.

Trapping user and application generated events in the GUI relies on exploiting platform-specific system application programming interface (API) calls. The APIs provide hooks into the underlying GUI and operating system event handling mechanisms. Implementing this functionality is an ICDE client application concern, and hence it is the responsibility of the ICDE client team to ensure this is carried out as efficiently and fast as possible.

Once an event is trapped, the ICDE client must call the server to store the event in the data store. It's vital therefore that this operation does not contribute any delay that the user might experience. For this reason, when an event is detected, it is written to an in-memory queue in the ICDE client. Once the event is stored in the queue, the event detection thread returns immediately and waits to capture the next event. This is a very fast operation and hence introduces no noticeable delay. Another thread running in the background constantly pulls events from the queue and calls the ICDE server to store the data.

This solution within the ICDE client decouples event capture and storage. A delayed write to the server by the background thread cannot delay the GUI

code. From the ICDE server's perspective, this is crucial. The server must of course be designed to store events in the data store as quickly as possible. But the server design can be guaranteed that there will only ever be one client request per user workstation in flight at any instant, as there is only one thread in each client sending the stream of user events to the server.

So for the ICDE server, its key performance requirements were easy to specify. It should provide subsecond average response times to ICDE client requests.

3.3 Scalability

Let's start with a representative definition of scalability²:

How well a solution to some problem will work when the size of the problem increases.

This is useful in an architectural context. It tells us that scalability is about how a design can cope with some aspect of the application's requirements increasing in size. To become a concrete quality attribute requirement, we need to understand exactly what is expected to get bigger. Here are some examples:

3.3.1 Request Load

Based on some defined mix of requests on a given hardware platform, an architecture for a server application may be designed to support 100 tps at peak load, with an average 1 s response time. If this request load were to grow by ten times, can the architecture support this increased load?

In the perfect world and without additional hardware capacity, as the load increases, application throughput should remain constant (i.e., 100 tps), and response time per request should increase only linearly (i.e., 10 s). A scalable solution will then permit additional processing capacity to be deployed to increase throughput and decrease response time. This additional capacity may be deployed in two different ways, one by adding more CPUs³ (and likely memory) to the machine the applications runs on (scale up), the other from distributing the application on multiple machines (scale out). This is illustrated in Fig. 3.1.

Scale up works well if an application is multithreaded, or multiple single threaded process instances can be executed together on the same machine. The latter will of course consume additional memory and associated resources, as processes are heavyweight, resource hungry vehicles for achieving concurrency.

²From <http://www.hyperdictionary.com>

³Adding faster CPUs is never a bad idea either. This is especially true if an application has components or calculations that are inherently single-threaded.

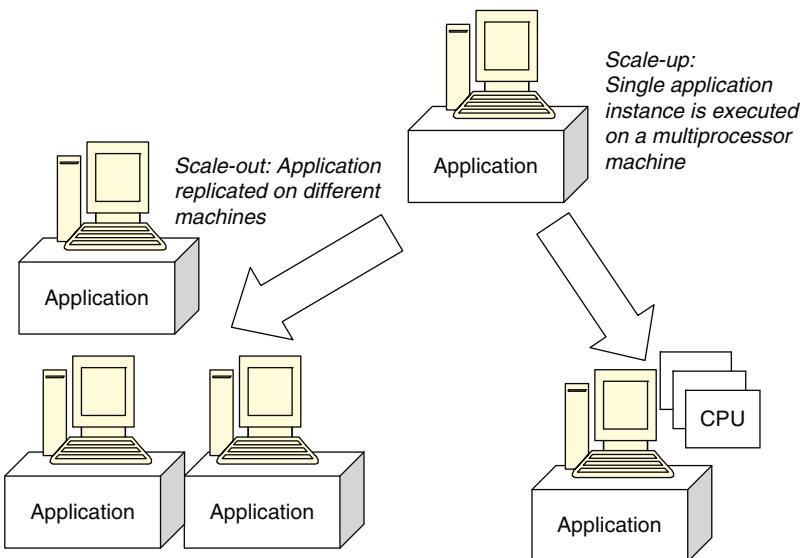


Fig. 3.1 Scale out versus scale up

Scale out works well if there is little or ideally no additional work required managing the distribution of requests amongst the multiple machines. The aim is to keep each machine equally busy, as the investment in more hardware is wasted if one machine is fully loaded and others idle away. Distributing load evenly amongst multiple machines is known as load-balancing.

Importantly, for either approach, scalability should be achieved without modifications to the underlying architecture (apart from inevitable configuration changes if multiple servers are used). In reality, as load increases, applications will exhibit a decrease in throughput and a subsequent exponential increase in response time. This happens for two reasons. First, the increased load causes increased contention for resources such as CPU and memory by the processes and threads in the server architecture. Second, each request consumes some additional resource (buffer space, locks, and so on) in the application, and eventually this resource becomes exhausted and limits scalability.

As an illustration, Fig. 3.2 shows how six different versions of the same application implemented using different JEE application servers perform as their load increases from 100 to 1,000 clients.⁴

⁴The full context for these figures is described in: I.Gorton, A Liu, *Performance Evaluation of Alternative Component Architectures for Enterprise JavaBean Applications*, in *IEEE Internet Computing*, vol.7, no. 3, pages 18-23, 2003. Bear in mind, these results are a snapshot in time and are meant for illustrative purposes. Absolutely no conclusions about the performance of the current versions of these technologies can or should be drawn.

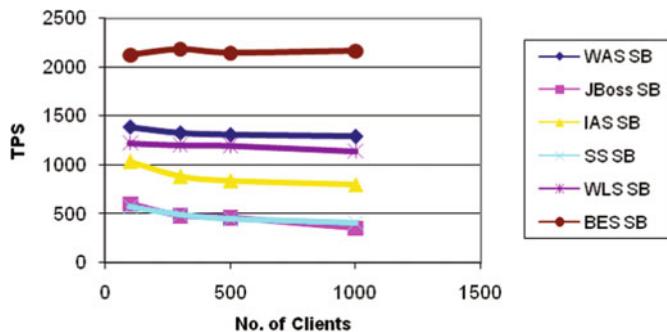


Fig. 3.2 Effects of increasing client request load on JEE platforms

3.3.2 Simultaneous Connections

An architecture may be designed to support 1,000 concurrent users. How does the architecture respond if this number grows significantly? If a connected user consumes some resources, then there will likely be a limit to the number of connections that can be effectively supported.

I encountered a classic example of this problem while performing an architecture review for an Internet Service Provider (ISP). Every time a user connected to the service, the ISP application spawned a new process on their server that was responsible for distributing targeted advertisements to the user. This worked beautifully, but each process consumed considerable memory and processing resources, even when the user simply connected and did nothing. Testing quickly revealed that the ISP's server machines could only support about 2,000 connections before their virtual memory was exhausted and the machines effectively ground to a halt in a disk thrashing frenzy. This made scaling the ISP's operations to support 100,000 users a prohibitively expensive proposition, and eventually, despite frantic redesign efforts, this was a root cause of the ISP going out of business.

3.3.3 Data Size

In a nutshell, how does an application behave as the data it processes increases in size? For example, a message broker application, perhaps a chat room, may be designed to process messages of an expected average size. How well will the architecture react if the size of messages grows significantly? In a slightly different vein, an information management solution may be designed to search and retrieve data from a repository of a specified size. How will the application behave if the size of the repository grows, in terms of raw size and/or number of items? The latter

is becoming such a problem that it has spawned a whole area of research and development known as data intensive computing.⁵

3.3.4 Deployment

How does the effort involved in deploying or modifying an application to an increasing user base grow? This would include effort for distribution, configuration and updating with new versions. An ideal solution would provide automated mechanisms that can dynamically deploy and configure an application to a new user, capturing registration information in the process. This is in fact exactly how many applications are today distributed on the Internet.

3.3.5 Some Thoughts on Scalability

Designing scalable architectures is not easy. In many cases, the need for scalability early in the design just isn't apparent and is not specified as part of the quality attribute requirements. It takes a savvy and attentive architect to ensure inherently nonscalable approaches are not introduced as core architectural components. Even if scalability is a required quality attribute, validating that it is satisfied by a proposed solution often just isn't practical in terms of schedule or cost. That's why it's important for an architect to rely on tried and tested designs and technologies whenever practical.

3.3.6 Scalability for the ICDE Application

The major scalability requirement for the ICDE system is to support the number of users expected in the largest anticipated ICDE deployment. The requirements specify this as approximately 150 users. The ICDE server application should therefore be capable of handling a peak load of 150 concurrent requests from ICDE clients.

3.4 Modifiability

All capable software architects know that along with death and taxes, modifications to a software system during its lifetime are simply a fact of life. That's why taking into account likely changes to the application is a good practice during

⁵A good overview of data intensive computing issues and some interesting approaches is the Special Edition of IEEE Computer from April 2008 – <http://www2.computer.org/portal/web/csdm/magazines/computer#3>

architecture formulation. The more flexibility that can be built into a design upfront, then the less painful and expensive subsequent changes will be. That's the theory anyway.

The modifiability quality attribute is a measure of how easy it may be to change an application to cater for new functional and nonfunctional requirements. Note the use of “may” in the previous sentence. Predicting modifiability requires an *estimate* of effort and/or cost to make a change. You only know for sure what a change will cost after it has been made. Then you find out how good your estimate was.

Modifiability measures are only relevant in the context of a given architectural solution. This solution must be expressed at least structurally as a collection of components, the component relationships and a description of how the components interact with the environment. Then, assessing modifiability requires the architect to assert likely change scenarios that capture how the requirements may evolve. Sometimes these will be known with a fair degree of certainty. In fact the changes may even be specified in the project plan for subsequent releases. Much of the time though, possible modifications will need to be elicited from application stakeholders, and drawn from the architect's experience. There's definitely an element of crystal ball gazing involved.

Illustrative change scenarios are:

- Provide access to the application through firewalls in addition to existing “behind the firewall” access.
- Incorporate new features for self-service check-out kiosks.
- The COTS speech recognition software vendor goes out of business and we need to replace this component.
- The application needs to be ported from Linux to the Microsoft Windows platform.

For each change scenario, the impact of the anticipated change on the architecture can be assessed. This impact is rarely easy to quantify, as more often than not the solution under assessment does not exist. In many cases, the best that can be achieved is a convincing impact analysis of the components in the architecture that will need modification, or a demonstration of how the solution can accommodate the modification without change.

Finally, based on cost, size or effort estimates for the affected components, some useful quantification of the cost of a change can be made. Changes isolated to single components or loosely coupled subsystems are likely to be less expensive to make than those that cause ripple effects across the architecture. If a likely change appears difficult and complex to make, this may highlight a weakness in the architecture that might justify further consideration and redesign.

A word of caution should be issued here. While loosely coupled, easily modifiable architectures are generally “a good thing”, design for modifiability needs to be thought through carefully. Highly modular architectures can become overly complex, incur additional performance overheads and require significantly more design and construction effort. This may be justified in some systems which must be highly configurable perhaps at deployment or run time, but often it's not.

You've probably heard some systems described as "over engineered", which essentially means investing more effort in a system than is warranted. This is often done because architects think they know their system's future requirements, and decide it's best to make a design more flexible or sophisticated, so it can accommodate the expected needs. That sounds reasonable, but requires a reliable crystal ball. If the predictions are wrong, much time and money can be wasted.

I recently was on the peripheral of such a project. The technical lead spent 5 months establishing a carefully designed messaging-based architecture based on the dependency injection pattern.⁶ The aim was to make this architecture extremely robust and create flexible data models for the messaging and underlying data store. With these in place, the theory was that the architecture could be reused over and over again with minimal effort, and it would be straightforward to inject new processing components due to the flexibility offered by dependency injection.

The word *theory* in the previous sentence was carefully chosen however. The system stakeholders became impatient, wondering why so much effort was being expended on such a sophisticated solution, and asked to see some demonstrable progress. The technical lead resisted, insisting his team should not be diverted and continued to espouse the long term benefits of the architecture. Just as this initial solution was close to completion, the stakeholders lost patience and replaced the technical lead with someone who was promoting a much simpler, Web server based solution as sufficient.

This was a classic case of overengineering. While the original solution was elegant and could have reaped great benefits in the long term, such arguments are essentially impossible to win unless you can show demonstrable, concrete evidence of this along the way. Adopting agile approaches is the key to success here. It would have been sensible to build an initial version of the core architecture in a few weeks and demonstrate this addressing a use case/user story that was meaningful to the stakeholder. The demonstration would have involved some prototypical elements in the architecture, would not be fully tested, and no doubt required some throw-away code to implement the use case – all unfortunately distasteful things to the technical lead. Success though would've built confidence with the stakeholders in the technical solution, elicited useful user feedback, and allowed the team to continue on its strategic design path.

The key then is to not let design purity drive a design. Rather, concentrating on known requirements and evolving and refactoring the architecture through regular iterations, while producing running code, makes eminent sense in almost all circumstances. As part of this process, you can continually analyze your design to see what future enhancements it can accommodate (or not). Working closely with stakeholders can help elicit highly likely future requirements, and eliminate those which seem highly unlikely. Let these drive the architecture strategy by all means, but never lose sight of known requirements and short term outcomes.

⁶<http://martinfowler.com/articles/injection.html>

3.4.1 *Modifiability for the ICDE Application*

Modifiability for the ICDE application is a difficult one to specify. A likely requirement would be for the range of events trapped and stored by the ICDE client to be expanded. This would have implication on the design of both the ICDE client and the ICDE server and data store.

Another would be for third party tools to want to communicate new message types. This would have implications on the message exchange mechanisms that the ICDE server supported. Hence both these modifiability scenarios could be used to test the resulting design for ease of modification.

3.5 Security

Security is a complex technical topic that can only be treated somewhat superficially here. At the architectural level, security boils down to understanding the precise security requirements for an application, and devising mechanisms to support them. The most common security-related requirements are:

- *Authentication*: Applications can verify the identity of their users and other applications with which they communicate.
- *Authorization*: Authenticated users and applications have defined access rights to the resources of the system. For example, some users may have read-only access to the application's data, while others have read-write.
- *Encryption*: The messages sent to/from the application are encrypted.
- *Integrity*: This ensures the contents of a message are not altered in transit.
- *Nonrepudiation*: The sender of a message has proof of delivery and the receiver is assured of the sender's identity. This means neither can subsequently refute their participation in the message exchange.

There are well known and widely used technologies that support these elements of application security. The Secure Socket Layer (SSL) and Public Key Infrastructures (PKI) are commonly used in Internet applications to provide authentication, encryption and nonrepudiation. Authentication and authorization is supported in Java technologies using the Java Authentication and Authorization Service (JAAS). Operating systems and databases provide login-based security for authentication and authorization.

Hopefully you're getting the picture. There are many ways, in fact sometimes too many, to support the required security attributes for an application. Databases want to impose their security model on the world. .NET designers happily leverage the Windows operating security features. Java applications can leverage JAAS without any great problems. If an application only needs to execute in one of these security domains, then solutions are readily available. If an application comprises several components that all wish to manage security, appropriate solutions must be

designed that typically localize security management in a single component that leverages the most appropriate technology for satisfying the requirements.

3.5.1 Security for the ICDE Application

Authentication of ICDE users and third party ICDE tools is the main security requirements for the ICDE system. In v1.0, users supply a login name and password which is authenticated by the database. This gives them access to the data in the data store associated with their activities. ICDE v2.0 will need to support similar authentication for users, and extend this to handle third party tools. Also, as third party tools may be executing remotely and access the ICDE data over an insecure network, the in-transit data should be encrypted.

3.6 Availability

Availability is related to an application's reliability. If an application isn't available for use when needed, then it's unlikely to be fulfilling its functional requirements. Availability is relatively easy to specify and measure. In terms of specification, many IT applications must be available at least during normal business hours. Most Internet sites desire 100% availability, as there are no regular business hours online. For a live system, availability can be measured by the proportion of the required time it is useable.

Failures in applications cause them to be unavailable. Failures impact on an application's reliability, which is usually measured by the mean time between failures. The length of time any period of unavailability lasts is determined by the amount of time it takes to detect failure and restart the system. Consequently, applications that require high availability minimize or preferably eliminate single points of failure, and institute mechanisms that automatically detect failure and restart the failed components.

Replicating components is a tried and tested strategy for high availability. When a replicated component fails, the application can continue executing using replicas that are still functioning. This may lead to degraded performance while the failed component is down, but availability is not compromised.

Recoverability is closely related to availability. An application is recoverable if it has the capability to reestablish required performance levels and recover affected data after an application or system failure. A database system is the classic example of a recoverable system. When a database server fails, it is unavailable until it has recovered. This means restarting the server application, and resolving any transactions that were in-flight when the failure occurred. Interesting issues for recoverable applications are how failures are detected and recovery commences (preferably automatically), and how long it takes to recover before full service is reestablished.

During the recovery process, the application is unavailable, and hence the mean time to recover is an important metric to consider.

3.6.1 Availability for the ICDE Application

While high availability for the ICDE application is desirable, it is only crucial that it be available during the business hours of the office environment it is deployed in. This leaves plenty of scope for downtime for such needs as system upgrade, backup and maintenance. The solution should however include mechanisms such as component replication to ensure as close to 100% availability as possible during business hours.

3.7 Integration

Integration is concerned with the ease with which an application can be usefully incorporated into a broader application context. The value of an application or component can frequently be greatly increased if its functionality or data can be used in ways that the designer did not originally anticipate. The most widespread strategies for providing integration are through data integration or providing an API.

Data integration involves storing the data an application manipulates in ways that other applications can access. This may be as simple as using a standard relational database for data storage, or perhaps implementing mechanisms to extract the data into a known format such as XML or a comma-separated text file that other applications can ingest.

With data integration, the ways in which the data is used (or abused) by other applications is pretty much out of control of the original data owner. This is because the data integrity and business rules imposed by the application logic are by-passed. The alternative is for interoperability to be achieved through an API (see Fig. 3.3). In this case, the raw data the application owns is hidden behind a set of functions that facilitate controlled external access to the data. In this manner, business rules and security can be enforced in the API implementation. The only way to access the data and integrate with the application is by using the supplied API.

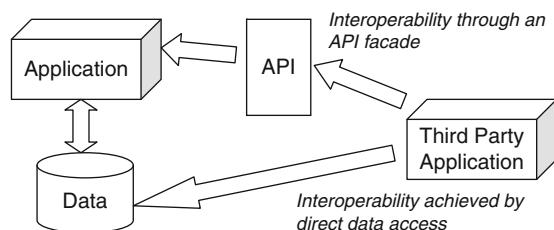


Fig. 3.3 Integration options

The choice of integration strategy is not simple. Data integration is flexible and simple. Applications written in any language can process text, or access relational databases using SQL. Building an API requires more effort, but provides a much more controlled environment, in terms of correctness and security, for integration. It is also much more robust from an integration perspective, as the API clients are insulated from many of the changes in the underlying data structures. They don't break every time the format is modified, as the data formats are not directly exposed and accessed. As always, the best choice of strategy depends on what you want to achieve, and what constraints exist.

3.7.1 Integration for the ICDE Application

The integration requirements for ICDE revolve around the need to support third party analysis tools. There must be a well-defined and understood mechanism for third party tools to access data in the ICDE data store. As third party tools will often execute remotely from an ICDE data store, integration at the data level, by allowing tools direct access to the data store, seems unlikely to be viable. Hence integration is likely to be facilitated through an API supported by the ICDE application.

3.8 Other Quality Attributes

There are numerous other quality attributes that are important in various application contexts. Some of these are:

- *Portability*: Can an application be easily executed on a different software/hardware platform to the one it has been developed for? Portability depends on the choices of software technology used to implement the application, and the characteristics of the platforms that it needs to execute on. Easily portable code bases will have their platform dependencies isolated and encapsulated in a small set of components that can be replaced without affecting the rest of the application.
- *Testability*: How easy or difficult is an application to test? Early design decisions can greatly affect the amount of test cases that are required. As a rule of thumb, the more complex a design, the more difficult it is to thoroughly test. Simplicity tends to promote ease of testing.⁷ Likewise, writing less of your own code by incorporating pretested components reduces test effort.
- *Supportability*: This is a measure of how easy an application is to support once it is deployed. Support typically involves diagnosing and fixing problems that

⁷“There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult”. C.A.R. Hoare.

occur during application use. Supportable systems tend to provide explicit facilities for diagnosis, such as application error logs that record the causes of failures. They are also built in a modular fashion so that code fixes can be deployed without severely inconveniencing application use.

3.9 Design Trade-Offs

If an architect's life were simple, design would merely involve building policies and mechanisms into an architecture to satisfy the required quality attributes for a given application. Pick a required quality attribute, and provide mechanisms to support it.

Unfortunately, this isn't the case. Quality attributes are not orthogonal. They interact in subtle ways, meaning a design that satisfies one quality attribute requirement may have a detrimental effect on another. For example, a highly secure system may be difficult or impossible to integrate in an open environment. A highly available application may trade-off lower performance for greater availability. An application that requires high performance may be tied to a particular platform, and hence not be easily portable.

Understanding trade-offs between quality attribute requirements, and designing a solution that makes sensible compromises is one of the toughest parts of the architect role. It's simply not possible to fully satisfy all competing requirements. It's the architect's job to tease out these tensions, make them explicit to the system's stakeholders, prioritize as necessary, and explicitly document the design decisions.

Does this sound easy? If only this were the case. That's why they pay you the big bucks.

3.10 Summary

Architects must expend a lot of effort precisely understanding quality attributes, so that a design can be conceived to address them. Part of the difficulty is that quality attributes are not always explicitly stated in the requirements, or adequately captured by the requirements engineering team. That's why an architect must be associated with the requirements gathering exercise for system, so that they can ask the right questions to expose and nail down the quality attributes that must be addressed.

Of course, understanding the quality attribute requirements is merely a necessary prerequisite to designing a solution to satisfy them. Conflicting quality attributes are a reality in every application of even mediocre complexity. Creating solutions that choose a point in the design space that adequately satisfies these requirements is remarkably difficult, both technically and socially. The latter involves communications with stakeholders to discuss design tolerances, discovering scenarios

when certain quality requirements can be safely relaxed, and clearly communicating design compromises so that the stakeholders understand what they are signing up for.

3.11 Further Reading

The broad topic of nonfunctional requirements is covered extremely thoroughly in:

L. Chung, B. Nixon, E. Yu, J. Mylopoulos, (Editors). Non-Functional Requirements in Software Engineering Series: The Kluwer International Series in Software Engineering. Vol. 5, Kluwer Academic Publishers. 1999.

An excellent general reference on security and the techniques and technologies an architect needs to consider is:

J. Ramachandran. Designing Security Architecture Solutions. Wiley & Sons, 2002.

An interesting and practical approach to assessing the modifiability of an architecture using architecture reconstruction tools and impact analysis metrics is described in:

I. Gorton, L. Zhu. *Tool Support for Just-in-Time Architecture Reconstruction and Evaluation: An Experience Report*. International Conference on Software Engineering (ICSE) 2005, St Louis, USA, ACM Press.

Chapter 1

Introduction

What Is the UML?

The Unified Modeling Language (UML) is a family of graphical notations, backed by single meta-model, that help in describing and designing software systems, particularly software systems built using the object-oriented (OO) style. That's a somewhat simplified definition. In fact, the UML is a few different things to different people. This comes both from its own history and from the different views that people have about what makes an effective software engineering process. As a result, my task in much of this chapter is to set the scene for this book by explaining the different ways in which people see and use the UML.

Graphical modeling languages have been around in the software industry for a long time. The fundamental driver behind them all is that programming languages are not at a high enough level of abstraction to facilitate discussions about design.

Despite the fact that graphical modeling languages have been around for a long time, there is an enormous amount of dispute in the software industry about their role. These disputes play directly into how people perceive the role of the UML itself.

The UML is a relatively open standard, controlled by the Object Management Group (OMG), an open consortium of companies. The OMG was formed to build standards that supported interoperability, specifically the interoperability of object-oriented systems. The OMG is perhaps best known for the CORBA (Common Object Request Broker Architecture) standards.

The UML was born out of the unification of the many object-oriented graphical modeling languages that thrived in the late 1980s and early 1990s. Since its appearance in 1997, it has relegated that particular tower of Babel to history. That's a service I, and many other developers, am deeply thankful for.

Ways of Using the UML

At the heart of the role of the UML in software development are the different ways in which people want to use it, differences that carry over from other graphical modeling languages. These differences lead to long and difficult arguments about how the UML should be used.

To untangle this, Steve Mellor and I independently came up with a characterization of the three modes in which people use the UML: sketch, blueprint, and programming language. By far the most common of the three, at least to my biased eye, is **UML as sketch**. In this usage, developers use the UML to help communicate some aspects of a system. As with blueprints, you can use sketches in a forward-engineering or reverse-engineering direction. **Forward engineering** draws a UML diagram before you write code, while **reverse engineering** builds a UML diagram from existing code in order to help understand it.

The essence of sketching is selectivity. With forward sketching, you rough out some issues in code you are about to write, usually discussing them with a group of people on your team. Your aim is to use the sketches to help communicate ideas and alternatives about what you're about to do. You don't talk about all the code you are going to work on, only important issues that you want to run past your colleagues first or sections of the design that you want to visualize before you begin programming. Sessions like this can be very short: a 10-minute session to discuss a few hours of programming or a day to discuss a 2-week iteration.

With reverse engineering, you use sketches to explain how some part of a system works. You don't show every class, simply those that are interesting and worth talking about before you dig into the code.

Because sketching is pretty informal and dynamic, you need to do it quickly and collaboratively, so a common medium is a whiteboard. Sketches are also useful in documents, in which case the focus is communication rather than completeness. The tools used for sketching are lightweight drawing tools, and often people aren't too particular about keeping to every strict rule of the UML. Most UML diagrams shown in books, such as my other books, are sketches. Their emphasis is on selective communication rather than complete specification.

In contrast, **UML as blueprint** is about completeness. In forward engineering, the idea is that blueprints are developed by a designer whose job is to build a detailed design for a programmer to code up. That design should be sufficiently complete in that all design decisions are laid out, and the programmer should be able to follow it as a pretty straightforward activity that requires little thought. The designer may be the same person as the programmer, but usually

the designer is a more senior developer who designs for a team of programmers. The inspiration for this approach is other forms of engineering in which professional engineers create engineering drawings that are handed over to construction companies to build.

Blueprinting may be used for all details, or a designer may draw blueprints to a particular area. A common approach is for a designer to develop blueprint-level models as far as interfaces of subsystems but then let developers work out the details of implementing those details.

In reverse engineering, blueprints aim to convey detailed information about the code either in paper documents or as an interactive graphical browser. The blueprints can show every detail about a class in a graphical form that's easier for developers to understand.

Blueprints require much more sophisticated tools than sketches do in order to handle the details required for the task. Specialized CASE (computer-aided software engineering) tools fall into this category, although the term CASE has become a dirty word, and vendors try to avoid it now. Forward-engineering tools support diagram drawing and back it up with a repository to hold the information. Reverse-engineering tools read source code and interpret from it into the repository and generate diagrams. Tools that can do both forward and reverse engineering like this are referred to as **round-trip** tools.

Some tools use the source code itself as the repository and use diagrams as a graphic viewport on the code. These tools tie much more closely into programming and often integrate directly with programming editors. I like to think of these as **tripless** tools.

The line between blueprints and sketches is somewhat blurry, but the distinction, I think, rests on the fact that sketches are deliberately incomplete, highlighting important information, while blueprints intend to be comprehensive, often with the aim of reducing programming to a simple and fairly mechanical activity. In a sound bite, I'd say that sketches are explorative, while blueprints are definitive.

As you do more and more in the UML and the programming gets increasingly mechanical, it becomes obvious that the programming should be automated. Indeed, many CASE tools do some form of code generation, which automates building a significant part of a system. Eventually, however, you reach the point at which all the system can be specified in the UML, and you reach **UML as programming language**. In this environment, developers draw UML diagrams that are compiled directly to executable code, and the UML becomes the source code. Obviously, this usage of UML demands particularly sophisticated tooling. (Also, the notions of forward and reverse engineering don't make any sense for this mode, as the UML and source code are the same thing.)

Model Driven Architecture and Executable UML

When people talk about the UML, they also often talk about **Model Driven Architecture (MDA)** [Kleppe et al.]. Essentially, MDA is a standard approach to using the UML as a programming language; the standard is controlled by the OMG, as is the UML. By producing a modeling environment that conforms to the MDA, vendors can create models that can also work with other MDA-compliant environments.

MDA is often talked about in the same breath as the UML because MDA uses the UML as its basic modeling language. But, of course, you don't have to be using MDA to use the UML.

MDA divides development work into two main areas. Modelers represent a particular application by creating a **Platform Independent Model (PIM)**. The PIM is a UML model that is independent of any particular technology. Tools can then turn a PIM into a **Platform Specific Model (PSM)**. The PSM is a model of a system targeted to a specific execution environment. Further tools then take the PSM and generate code for that platform. The PSM could be UML but doesn't have to be.

So if you want to build a warehousing system using MDA, you would start by creating a single PIM of your warehousing system. If you then wanted this warehousing system to run on J2EE and .NET, you would use some vendor tools to create two PSMs: one for each platform. Then further tools would generate code for the two platforms.

If the process of going from PIM to PSM to final code is completely automated, we have the UML as programming language. If any of the steps is manual, we have blueprints.

Steve Mellor has long been active in this kind of work and has recently used the term **Executable UML** [Mellor and Balcer]. Executable UML is similar to MDA but uses slightly different terms. Similarly, you begin with a platform-independent model that is equivalent to MDA's PIM. However, the next step is to use a **Model Compiler** to turn that UML model into a deployable system in a single step; hence, there's no need for the PSM. As the term *compiler* suggests, this step is completely automatic.

The model compilers are based on reusable archetypes. An archetype describes how to take an executable UML model and turn it into a particular programming platform. So for the warehousing example, you would buy a model compiler and two archetypes (J2EE and .NET). Run each archetype on your executable UML model, and you have your two versions of the warehousing system.

Executable UML does not use the full UML standard; many constructs of UML are considered to be unnecessary and are therefore not used. As a result, Executable UML is simpler than full UML.

All this sounds good, but how realistic is it? In my view, there are two issues here. First is the question of the tools: whether they are mature enough to do the job. This is something that changes over time; certainly, as I write this, they aren't widely used, and I haven't seen much of them in action.

A more fundamental issue is the whole notion of the UML as a programming language. In my view, it's worth using the UML as a programming language only if it results in something that's significantly more productive than using another programming language. I'm not convinced that it is, based on various graphical development environments I've worked with in the past. Even if it is more productive, it still needs to get a critical mass of users for it to make the mainstream. That's a big hurdle in itself. Like many old Smalltalkers, I consider Smalltalk to be much more productive than current mainstream languages. But as Smalltalk is now only a niche language, I don't see many projects using it. To avoid Smalltalk's fate, the UML has to be luckier, even if it is superior.

One of the interesting questions around the UML as programming language is how to model behavioral logic. UML 2 offers three ways of behavioral modeling: interaction diagrams, state diagrams, and activity diagrams. All have their proponents for programming in. If the UML does gain popularity as a programming language, it will be interesting to see which of these techniques become successful.

Another way in which people look at the UML is the range between using it for conceptual and for software modeling. Most people are familiar with the UML used for software modeling. In this **software perspective**, the elements of the UML map pretty directly to elements in a software system. As we shall see, the mapping is by no means prescriptive, but when we use the UML, we are talking about software elements.

With the **conceptual perspective**, the UML represents a description of the concepts of a domain of study. Here, we aren't talking about software elements so much as we are building a vocabulary to talk about a particular domain.

There are no hard-and-fast rules about perspective; as it turns out, there's really quite a large range of usage. Some tools automatically turn source code into the UML diagrams, treating the UML as an alternative view of the source.

That's very much a software perspective. If you use UML diagrams to try and understand the various meanings of the terms *asset pool* with a bunch of accountants, you are in a much more conceptual frame of mind.

In previous editions of this book, I split the software perspective into specification (interface) and implementation. In practice, I found that it was too hard to draw a precise line between the two, so I feel that the distinction is no longer worth making a fuss about. However, I'm always inclined to emphasize interface rather than implementation in my diagrams.

These different ways of using the UML lead to a host of arguments about what UML diagrams mean and what their relationship is to the rest of the world. In particular, it affects the relationship between the UML and source code. Some people hold the view that the UML should be used to create a design that is independent of the programming language that's used for implementation. Others believe that language-independent design is an oxymoron, with a strong emphasis on the moron.

Another difference in viewpoints is what the essence of the UML is. In my view, most users of the UML, particularly sketchers, see the essence of the UML to be the diagrams. However, the creators of the UML see the diagrams as secondary; the essence of the UML is the meta-model. Diagrams are simply a presentation of the meta-model. This view also makes sense to blueprinters and UML programming language users.

So whenever you read anything involving the UML, it's important to understand the point of view of the author. Only then can you make sense of the often fierce arguments that the UML encourages.

Having said all that, I need to make my biases clear. Almost all the time, my use of the UML is as sketches. I find the UML sketches useful with forward and reverse engineering and in both conceptual and software perspectives.

I'm not a fan of detailed forward-engineered blueprints; I believe that it's too difficult to do well and slows down a development effort. Blueprinting to a level of subsystem interfaces is reasonable, but even then you should expect to change those interfaces as developers implement the interactions across the interface. The value of reverse-engineered blueprints is dependent on how the tool works. If it's used as a dynamic browser, it can be very helpful; if it generates a large document, all it does is kill trees.

I see the UML as programming language as a nice idea but doubt that it will ever see significant usage. I'm not convinced that graphical forms are more productive than textual forms for most programming tasks and that even if they are, it's very difficult for a language to be widely accepted.

As a result of my biases, this book focuses much more on using the UML for sketching. Fortunately, this makes sense for a brief guide. I can't do justice to

the UML in its other modes in a book this size, but a book this size makes a good introduction to other books that can. So if you're interested in the UML in its other modes, I'd suggest that you treat this book as an introduction and move on to other books as you need them. If you're interested only in sketches, this book may well be all you need.

How We Got to the UML

I'll admit, I'm a history buff. My favorite idea of light reading is a good history book. But I also know that it's not everybody's idea of fun. I talk about history here because I think that in many ways, it's hard to understand where the UML is without understanding the history of how it got here.

In the 1980s, objects began to move away from the research labs and took their first steps toward the "real" world. Smalltalk stabilized into a platform that people could use, and C++ was born. At that time, various people started thinking about object-oriented graphical design languages.

The key books about object-oriented graphical modeling languages appeared between 1988 and 1992. Leading figures included Grady Booch [Booch, OOAD]; Peter Coad [Coad, OOA], [Coad, OOD]; Ivar Jacobson (Objectory) [Jacobson, OOSE]; Jim Odell [Odell]; Jim Rumbaugh (OMT) [Rumbaugh, insights], [Rumbaugh, OMT]; Sally Shlaer and Steve Mellor [Shlaer and Mellor, data], [Shlaer and Mellor, states]; and Rebecca Wirfs-Brock (Responsibility Driven Design) [Wirfs-Brock].

Each of those authors was now informally leading a group of practitioners who liked those ideas. All these methods were very similar, yet they contained a number of often annoying minor differences among them. The same basic concepts would appear in very different notations, which caused confusion to my clients.

During that heady time, standardization was as talked about as it was ignored. A team from the OMG tried to look at standardization but got only an open letter of protest from all the key methodologists. (This reminds me of an old joke. Question: What is the difference between a methodologist and a terrorist? Answer: You can negotiate with a terrorist.)

The cataclysmic event that first initiated the UML was when Jim Rumbaugh left GE to join Grady Booch at Rational (now a part of IBM). The Booch/Rumbaugh alliance was seen from the beginning as one that could get a critical mass of market share. Grady and Jim proclaimed that "the methods war is over—we won," basically declaring that they were going to achieve

standardization “the Microsoft way.” A number of other methodologists suggested forming an Anti-Booch Coalition.

By OOPSLA ’95, Grady and Jim had prepared their first public description of their merged method: version 0.8 of the *Unified Method* documentation. Even more significant, they announced that Rational Software had bought Objectory and that therefore, Ivar Jacobson would be joining the Unified team. Rational held a well-attended party to celebrate the release of the 0.8 draft. (The highlight of the party was the first public display of Jim Rumbaugh’s singing; we all hope it’s also the last.)

The next year saw a more open process emerge. The OMG, which had mostly stood on the sidelines, now took an active role. Rational had to incorporate Ivar’s ideas and also spent time with other partners. More important, the OMG decided to take a major role.

At this point, it’s important to realize why the OMG got involved. Methodologists, like book authors, like to think that they are important. But I don’t think that the screams of book authors would even be heard by the OMG. What got the OMG involved were the screams of tools vendors, all of which were frightened that a standard controlled by Rational would give Rational tools an unfair competitive advantage. As a result, the vendors energized the OMG to do something about it, under the banner of CASE tool interoperability. This banner was important, as the OMG was all about interoperability. The idea was to create a UML that would allow CASE tools to freely exchange models.

Mary Loomis and Jim Odell chaired the initial task force. Odell made it clear that he was prepared to give up his method to a standard, but he did not want a Rational-imposed standard. In January 1997, various organizations submitted proposals for a methods standard to facilitate the interchange of models. Rational collaborated with a number of other organizations and released version 1.0 of the UML documentation as their proposal, the first animal to answer to the name Unified Modeling Language.

Then followed a short period of arm twisting while the various proposals were merged. The OMG adopted the resulting 1.1 as an official OMG standard. Some revisions were made later on. Revision 1.2 was entirely cosmetic. Revision 1.3 was more significant. Revision 1.4 added a number of detailed concepts around components and profiles. Revision 1.5 added action semantics.

When people talk about the UML, they credit mainly Grady Booch, Ivar Jacobson, and Jim Rumbaugh as its creators. They are generally referred to as the Three Amigos, although wags like to drop the first syllable of the second word. Although they are most credited with the UML, I think it somewhat unfair to give them the dominant credit. The UML notation was first formed in

the Booch/Rumbaugh Unified Method. Since then, much of the work has been led by OMG committees. During these later stages, Jim Rumbaugh is the only one of the three to have made a heavy commitment. My view is that it's these members of the UML committee process that deserve the principal credit for the UML.

Notations and Meta-Models

The UML, in its current state, defines a notation and a meta-model. The **notation** is the graphical stuff you see in models; it is the graphical syntax of the modeling language. For instance, class diagram notation defines how items and concepts, such as class, association, and multiplicity, are represented.

Of course, this leads to the question of what exactly is meant by an association or multiplicity or even a class. Common usage suggests some informal definitions, but many people want more rigor than that.

The idea of rigorous specification and design languages is most prevalent in the field of formal methods. In such techniques, designs and specifications are represented using some derivative of predicate calculus. Such definitions are mathematically rigorous and allow no ambiguity. However, the value of these definitions is by no means universal. Even if you can prove that a program satisfies a mathematical specification, there is no way to prove that the mathematical specification meets the real requirements of the system.

Most graphical modeling languages have very little rigor; their notation appeals to intuition rather than to formal definition. On the whole, this does not seem to have done much harm. These methods may be informal, but many people still find them useful—and it is usefulness that counts.

However, methodologists are looking for ways to improve the rigor of methods without sacrificing their usefulness. One way to do this is to define a **meta-model**: a diagram, usually a class diagram, that defines the concepts of the language.

Figure 1.1, a small piece of the UML meta-model, shows the relationship among features. (The extract is there to give you a flavor of what meta-models are like. I'm not even going to try to explain it.)

How much does the meta-model affect a user of the modeling notation? The answer depends mostly on the mode of usage. A sketcher usually doesn't care too much; a blueprinter should care rather more. It's vitally important to those who use the UML as a programming language, as it defines the abstract syntax of that language.

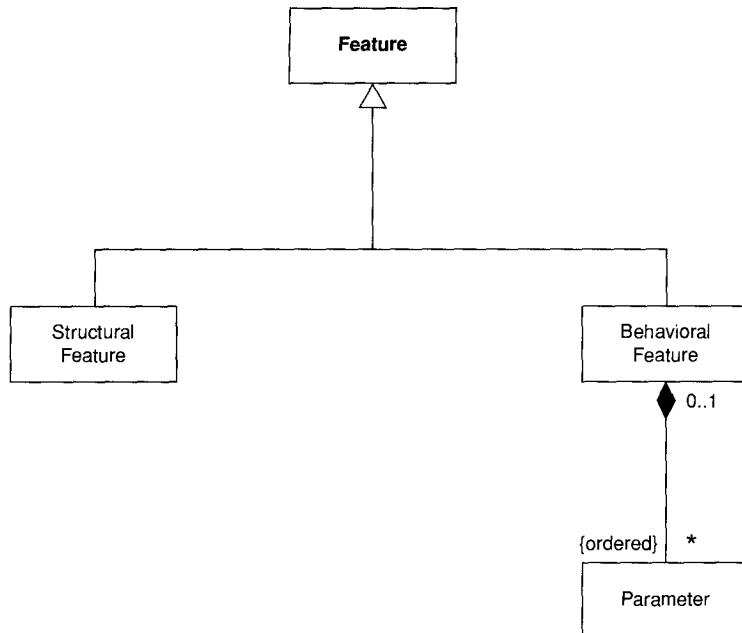


Figure 1.1 *A small piece of the UML meta-model*

Many of the people who are involved in the ongoing development of the UML are interested primarily in the meta-model, particularly as this is important to the usage of the UML and a programming language. Notational issues often run second place, which is important to bear in mind if you ever try to get familiar with the standards documents themselves.

As you get deeper into the more detailed usage of the UML, you realize that you need much more than the graphical notation. This is why UML tools are so complex.

I am not rigorous in this book. I prefer the traditional methods path and appeal mainly to your intuition. That's natural for a small book like this written by an author who's inclined mostly to a sketch usage. If you want more rigor, you should turn to more detailed tomes.

UML Diagrams

UML 2 describes 13 official diagram types listed in Table 1.1 and classified as indicated on Figure 1.2. Although these diagram types are the way many people

Table 1.1 Official Diagram Types of the UML

Diagram	Book Chapters	Purpose	Lineage
Activity	11	Procedural and parallel behavior	In UML 1
Class	3, 5	Class, features, and relationships	In UML 1
Communication	12	Interaction between objects; emphasis on links	UML 1 collaboration diagram
Component	14	Structure and connections of components	In UML 1
Composite structure	13	Runtime decomposition of a class	New to UML 2
Deployment	8	Deployment of artifacts to nodes	In UML 1
Interaction overview	16	Mix of sequence and activity diagram	New to UML 2
Object	6	Example configurations of instances	Unofficially in UML 1
Package	7	Compile-time hierachic structure	Unofficially in UML 1
Sequence	4	Interaction between objects; emphasis on sequence	In UML 1
State machine	10	How events change an object over its life	In UML 1
Timing	17	Interaction between objects; emphasis on timing	New to UML 2
Use case	9	How users interact with a system	In UML 1

approach the UML and how I've organized this book, the UML's authors do not see diagrams as the central part of the UML. As a result, the diagram types are not particularly rigid. Often, you can legally use elements from one diagram type on another diagram. The UML standard indicates that certain elements are typically drawn on certain diagram types, but this is not a prescription.

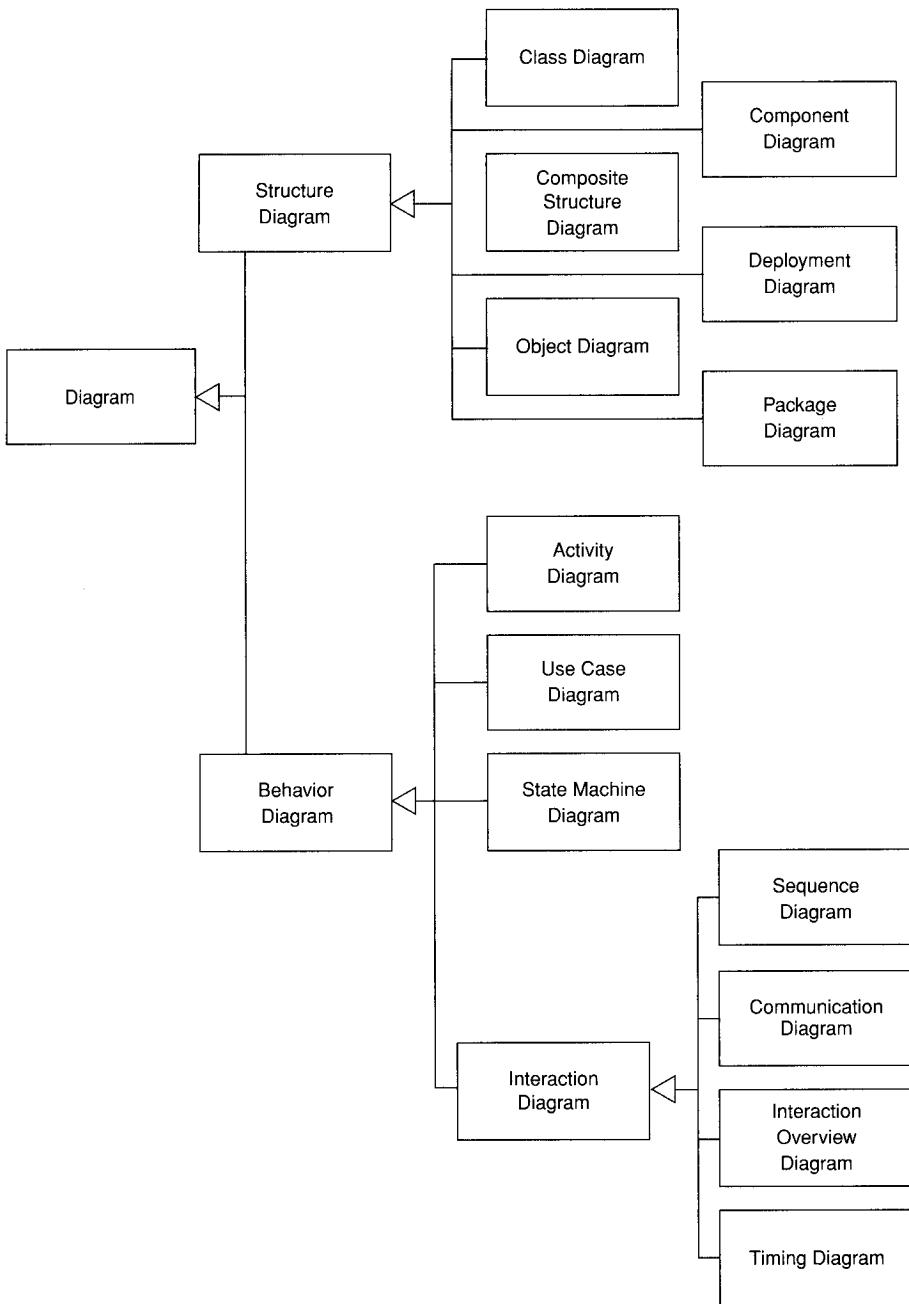


Figure 1.2 Classification of UML diagram types

What Is Legal UML?

At first blush, this should be a simple question to answer: Legal UML is what is defined as well formed in the specification. In practice, however, the answer is a bit more complicated.

An important part of this question is whether the UML has descriptive or prescriptive rules. A language with **prescriptive rules** is controlled by an official body that states what is or isn't legal in the language and what meaning you give to utterances in that language. A language with **descriptive rules** is one in which you understand its rules by looking at how people use the language in practice. Programming languages tend to have prescriptive rules set by a standards committee or dominant vendor, while natural languages, such as English, tend to have descriptive rules whose meaning is set by convention.

UML is quite a precise language, so you might expect it to have prescriptive rules. But UML is often considered to be the software equivalent of the blueprints in other engineering disciplines, and these blueprints are not prescriptive notations. No committee says what the legal symbols are on a structural engineering drawing; the notation has been accepted by convention, similarly to a natural language. Simply having a standards body doesn't do the trick either, because people in the field may not follow everything the standards body says; just ask the French about the Académie Française. In addition, the UML is so complex that the standard is often open to multiple interpretations. Even the UML leaders who reviewed this book would disagree on interpretation of the UML standard.

This issue is important both for me writing this book and for you using the UML. If you want to understand a UML diagram, it's important to realize that understanding the UML standard is not the whole picture. People do adopt conventions, both in the industry widely and within a particular project. As a result, although the UML standard can be the primary source of information on the UML, it can't be the only one.

My attitude is that, for most people, the UML has descriptive rules. The UML standard is the biggest single influence on what UML means, but it isn't the only one. I think that this will become particularly true with UML 2, which introduces some notational conventions that conflict with either UML 1's definition or the conventional usage of UML, as well as adds yet more complexity to the UML. In this book, therefore, I'm trying to summarize the UML as I find it: both the standards and the conventional usage. When I have to make a distinction in this book, I'll use the term **conventional use** to indicate something that isn't in the standard but that I think is widely used. For something that conforms to the standard, I'll use the terms **standard** or **normative**. (Normative

is the term standards people use to mean a statement that you must conform to be valid in the standard. So non-normative UML is a fancy way of saying that something is strictly illegal according to the UML standard.)

When you are looking at a UML diagram, you should bear in mind that a general principle in the UML is that any information may be suppressed for a particular diagram. This suppression can occur either generally—hide all attributes—or specifically—don’t show these three classes. In a diagram, therefore, you can never infer anything by its absence. If a multiplicity is missing, you cannot infer what value it might be. Even if the UML meta-model has a default, such as [1] for attributes, if you don’t see the information on the diagram, it may be because it’s the default or because it’s suppressed.

Having said that, there are some general conventions, such as multivalued properties being sets. In the text, I’ll point out these default conventions.

It’s important to not put too much emphasis on having legal UML if you’re a sketcher or blueprinter. It’s more important to have a good design for your system, and I would rather have a good design in illegal UML than a legal but poor design. Obviously, good and legal is best, but you’re better off putting your energy into having a good design than worrying about the arcana of UML. (Of course, you have to be legal in UML as programming language, or your program won’t run properly!)

The Meaning of UML

One of the awkward issues about the UML is that, although the specification describes in great detail what well-formed UML is, it doesn’t have much to say about what the UML means outside of the rarefied world of the UML meta-model. No formal definition exists of how the UML maps to any particular programming language. You cannot look at a UML diagram and say *exactly* what the equivalent code would look like. However, you can get a *rough idea* of what the code would look like. In practice, that’s enough to be useful. Development teams often form their local conventions for these, and you’ll need to be familiar with the ones in use.

UML Is Not Enough

Although the UML provides quite a considerable body of various diagrams that help to define an application, it’s by no means a complete list of all the useful

diagrams that you might want to use. In many places, different diagrams can be useful, and you shouldn't hesitate to use a non-UML diagram if no UML diagram suits your purpose.

Figure 1.3, a screen flow diagram, shows the various screens on a user interface and how you move between them. I've seen and used these screen flow diagrams for many years. I've never seen more than a very rough definition of what they mean; there isn't anything like it in the UML, yet I've found it a very useful diagram.

Table 1.2 shows another favorite: the decision table. Decision tables are a good way to show complicated logical conditions. You can do this with an activity diagram, but once you get beyond simple cases, the table is both more compact and more clear. Again, many forms of decision tables are out there. Table 1.2 divides the table into two sections: conditions above the double line and consequences below it. Each column shows how a particular combination of conditions leads to a particular set of consequences.

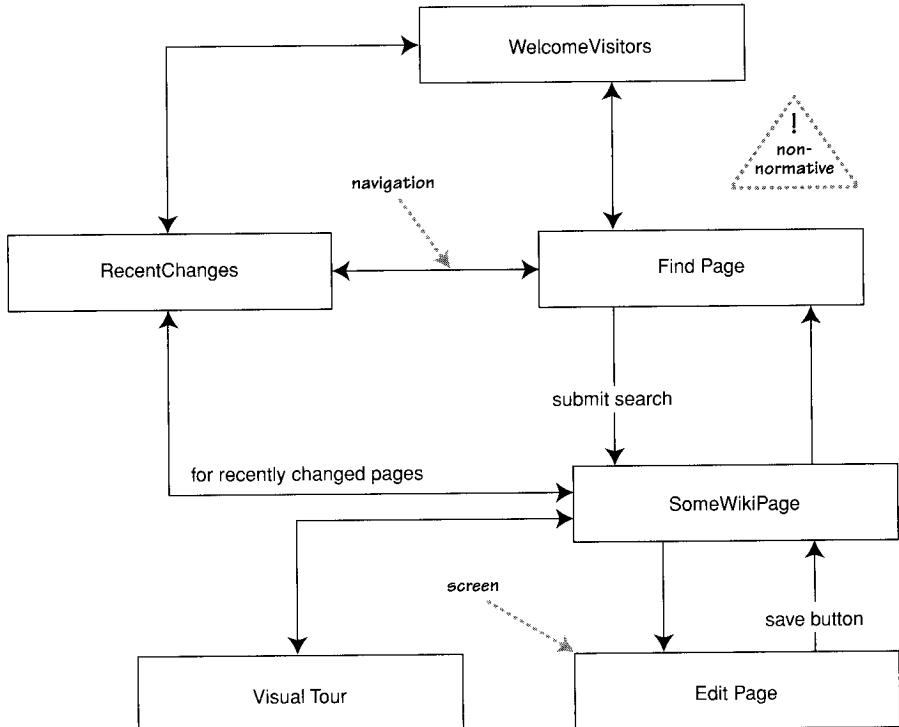


Figure 1.3 An informal screen flow diagram for part of the wiki (<http://c2.com/cgi/wiki>)

Table 1.2 A Decision Table

Premium customer	X	X	Y	Y	N	N
Priority order	Y	N	Y	N	Y	N
International order	Y	Y	N	N	N	N
Fee	\$150	\$100	\$70	\$50	\$80	\$60
Alert rep	•	•	•			

You'll run into various kinds of these things in various books. Don't hesitate to try out techniques that seem appropriate for your project. If they work well, use them. If not, discard them. (This is, of course, the same advice as for UML diagrams.)

Where to Start with the UML

Nobody, not even the creators of the UML, understand or use all of it. Most people use a small subset of the UML and work with that. You have to find the subset of the UML that works for you and your colleagues.

If you are starting out, I suggest that you concentrate first on the basic forms of class diagrams and sequence diagrams. These are the most common and, in my view, the most useful diagram types.

Once you've got the hang of those, you can start using some of the more advanced class diagram notation and take a look at the other diagrams types. Experiment with the diagrams and see how helpful they are to you. Don't be afraid to drop any that don't seem be useful to your work.

Where to Find Out More

This book is not a complete and definitive reference to the UML, let alone OO analysis and design. A lot of words are out there and a lot of worthwhile things to read. As I discuss the individual topics, I also mention other books you should go to for more in-depth information there. Here are some general books on the UML and object-oriented design.

As with all book recommendations, you may need to check which version of the UML they are written for. As of June 2003, no published book uses UML 2.0, which is hardly surprising, as the ink is barely dry on the standard. The books I

suggest are good books, but I can't tell whether or when they will be updated to the UML 2 standard.

If you are new to objects, I recommend my current favorite introductory book: [Larman]. The author's strong responsibility-driven approach to design is worth following.

For the conclusive word on the UML, you should look to the official standards documents; but remember, they are written for consenting methodologists in the privacy of their own cubicles. For a much more digestible version of the standard, take a look at [Rumbaugh, UML Reference].

For more detailed advice on object-oriented design, you'll learn many good things from [Martin].

I also suggest that you read books on patterns for material that will take you beyond the basics. Now that the methods war is over, patterns (page 27) are where most of the interesting material about analysis and design appears.

Chapter 9

Use Cases

Use cases are a technique for capturing the functional requirements of a system. Use cases work by describing the typical interactions between the users of a system and the system itself, providing a narrative of how a system is used.

Rather than describe use cases head-on, I find it easier to sneak up on them from behind and start by describing scenarios. A **scenario** is a sequence of steps describing an interaction between a user and a system. So if we have a Web-based on-line store, we might have a Buy a Product scenario that would say this:

The customer browses the catalog and adds desired items to the shopping basket. When the customer wishes to pay, the customer describes the shipping and credit card information and confirms the sale. The system checks the authorization on the credit card and confirms the sale both immediately and with a follow-up e-mail.

This scenario is one thing that can happen. However, the credit card authorization might fail, and this would be a separate scenario. In another case, you may have a regular customer for whom you don't need to capture the shipping and credit card information, and this is a third scenario.

All these scenarios are different yet similar. The essence of their similarity is that in all these three scenarios, the user has the same goal: to buy a product. The user doesn't always succeed, but the goal remains. This user goal is the key to use cases: A **use case** is a set of scenarios tied together by a common user goal.

In use case-speak, the users are referred to as actors. An **actor** is a role that a user plays with respect to the system. Actors might include customer, customer service rep, sales manager, and product analyst. Actors carry out use cases. A single actor may perform many use cases; conversely, a use case may have several actors performing it. Usually, you have many customers, so many people can be the customer actor. Also, one person may act as more than one actor,

such as a sales manager who does customer service rep tasks. An actor doesn't have to be human. If the system performs a service for another computer system, that other system is an actor.

Actor isn't really the right term; *role* would be much better. Apparently, there was a mistranslation from Swedish, and *actor* is the term the use case community uses.

Use cases are well known as an important part of the UML. However, the surprise is that in many ways, the definition of use cases in the UML is rather sparse. Nothing in the UML describes how you should capture the content of a use case. What the UML describes is a use case diagram, which shows how use cases relate to each other. But almost all the value of use cases lies in the content, and the diagram is of rather limited value.

Content of a Use Case

There is no standard way to write the content of a use case, and different formats work well in different cases. Figure 9.1 shows a common style to use. You begin by picking one of the scenarios as the **main success scenario**. You start the body of the use case by writing the main success scenario as a sequence of numbered steps. You then take the other scenarios and write them as **extensions**, describing them in terms of variations on the main success scenario. Extensions can be successes—user achieves the goal, as in 3a—or failures, as in 6a.

Each use case has a **primary actor**, which calls on the system to deliver a service. The primary actor is the actor with the goal the use case is trying to satisfy and is usually, but not always, the initiator of the use case. There may be other actors as well with which the system communicates while carrying out the use case. These are known as **secondary actors**.

Each step in a use case is an element of the interaction between an actor and the system. Each step should be a simple statement and should clearly show who is carrying out the step. The step should show the intent of the actor, not the mechanics of what the actor does. Consequently, you don't describe the user interface in the use case. Indeed, writing the use case usually precedes designing the user interface.

An extension within the use case names a condition that results in different interactions from those described in the main success scenario (MSS) and states what those differences are. Start the extension by naming the step at which the condition is detected and provide a short description of the condition. Follow the condition with numbered steps in the same style as the main success scenario.

Buy a Product

Goal Level: Sea Level

Main Success Scenario:

1. Customer browses catalog and selects items to buy
2. Customer goes to check out
3. Customer fills in shipping information (address; next-day or 3-day delivery)
4. System presents full pricing information, including shipping
5. Customer fills in credit card information
6. System authorizes purchase
7. System confirms sale immediately
8. System sends confirming e-mail to customer

Extensions:

3a: Customer is regular customer

.1: System displays current shipping, pricing, and billing information

.2: Customer may accept or override these defaults, returns to MSS at step 6

6a: System fails to authorize credit purchase

.1: Customer may reenter credit card information or may cancel

Figure 9.1 Example use case text

Finish these steps by describing where you return to the main success scenario, if you do.

The use case structure is a great way to brainstorm alternatives to the main success scenario. For each step, ask, How could this go differently? and in particular, What could go wrong? It's usually best to brainstorm all the extension conditions first, before you get bogged down working out the consequences. You'll probably think of more conditions this way, which translates to fewer goofs that you have to pick up later.

A complicated step in a use case can be another use case. In UML terms, we say that the first use case **includes** the second. There's no standard way to show an included use case in the text, but I find that underlining, which suggests a hyperlink, works very nicely and in many tools really will be a hyperlink. Thus in Figure 9.1, the first step includes the use case “browse catalog and select items to buy.”

Included use cases can be useful for a complex step that would clutter the main scenario or for steps that are repeated in several use cases. However, don't try to break down use cases into sub-use cases and subsub-use cases using functional decomposition. Such a decomposition is a good way to waste a lot of time.

As well as the steps in the scenarios, you can add some other common information to a use case.

- A **pre-condition** describes what the system should ensure is true before the system allows the use case to begin. This is useful for telling the programmers what conditions they don't have to check for in their code.
- A **guarantee** describes what the system will ensure at the end of the use case. Success guarantees hold after a successful scenario; minimal guarantees hold after any scenario.
- A **trigger** specifies the event that gets the use case started.

When you're considering adding elements, be skeptical. It's better to do too little than too much. Also, work hard to keep the use case brief and easy to read. I've found that long, detailed use cases don't get read, which rather defeats the purpose.

The amount of detail you need in a use case depends on the amount of risk in that use case. Often, you need details on only a few key use cases early on; others can be fleshed out just before you implement them. You don't have to write all the detail down; verbal communication is often very effective, particularly within an iterative cycle in which needs are quickly met by running code.

Use Case Diagrams

As I said earlier, the UML is silent on the content of a use case but does provide a diagram format for showing them, as in Figure 9.2. Although the diagram is sometimes useful, it isn't mandatory. In your use case work, don't put too much effort into the diagram. Instead, concentrate on the textual content of the use cases.

The best way to think of a use case diagram is that it's a graphical table of contents for the use case set. It's also similar to the context diagram used in structured methods, as it shows the system boundary and the interactions with the outside world. The use case diagram shows the actors, the use cases, and the relationships between them:

- Which actors carry out which use cases
- Which use cases include other use cases

The UML includes other relationships between use cases beyond the simple includes, such as «extend». I strongly suggest that you ignore them. I've seen too

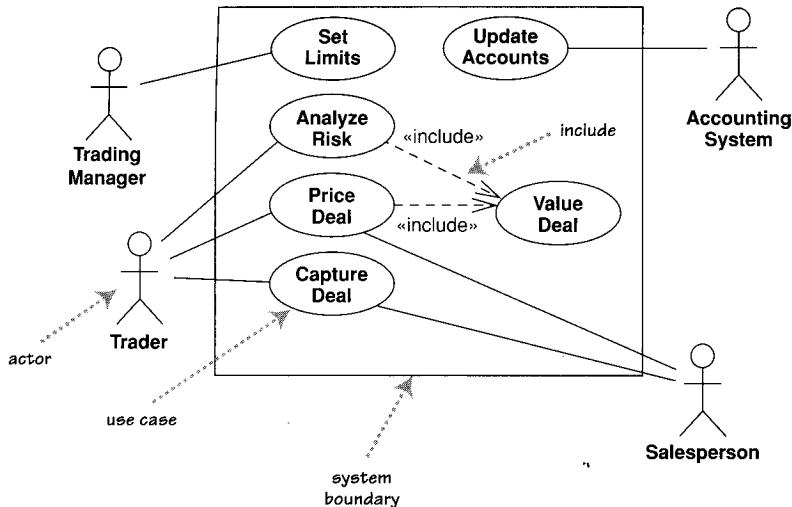


Figure 9.2 Use case diagram

many situations in which teams can get terribly hung up on when to use different use case relationships, and such energy is wasted. Instead, concentrate on the textual description of a use case; that's where the real value of the technique lies.

Levels of Use Cases

A common problem with use cases is that by focusing on the interaction between a user and the system, you can neglect situations in which a change to a business process may be the best way to deal with the problem. Often, you hear people talk about system use cases and business use cases. The terms are not precise, but in general, a **system use case** is an interaction with the software, whereas a **business use case** discusses how a business responds to a customer or an event.

[Cockburn, use cases] suggests a scheme of levels of use cases. The core use cases are at “sea level.” Sea-level use cases typically represent a discrete interaction between a primary actor and the system. Such use cases will deliver something of value to the primary actor and usually take from a couple of minutes to half an hour for the primary actor to complete. Use cases that are there only because they are included by sea-level use cases are **fish level**. Higher, kite-level

use cases show how the sea-level use cases fit into wider business interactions. Kite-level use cases are usually business use cases, whereas sea and fish levels are system use cases. You should have most of your use cases at the sea level. I prefer to indicate the level at the top of the use case, as in Figure 9.1.

Use Cases and Features (or Stories)

Many approaches use features of a system—Extreme Programming calls them user stories—to help describe requirements. A common question is how features and use cases interrelate.

Features are a good way of chunking up a system for planning an iterative project, whereby each iteration delivers a number of features. Use cases provide a narrative of how the actors use the system. Hence, although both techniques describe requirements, their purposes are different.

Although you can go directly to describing features, many people find it helpful to develop use cases first and then generate a list of features. A feature may be a whole use case, a scenario in a use case, a step in a use case, or some variant behavior, such as adding yet another depreciation method for your asset valuations, that doesn't show up in a use case narrative. Usually, features end up being more fine grained than use cases.

When to Use Use Cases

Use cases are a valuable tool to help understand the functional requirements of a system. A first pass at use cases should be made early on. More detailed versions of use cases should be worked just prior to developing that use case.

It is important to remember that use cases represent an *external* view of the system. As such, don't expect any correlations between use cases and the classes inside the system.

The more I see of use cases, the less valuable the use case diagram seems to be. With use cases, concentrate your energy on their text rather than on the diagram. Despite the fact that the UML has nothing to say about the use case text, it is the text that contains all the value in the technique.

A big danger of use cases is that people make them too complicated and get stuck. Usually, you'll get less hurt by doing too little than by doing too much. A couple of pages per use case is just fine for most cases. If you have too little, at

least you'll have a short, readable document that's a starting point for questions. If you have too much, hardly anyone will read and understand it.

Where to Find Out More

Use cases were originally popularized by Ivar Jacobson in [Jacobson, OOSE].

Although use cases have been around for a while, there's been little standardization on their use. The UML is silent on the important contents of a use case and has standardized only the much less important diagrams. As a result, you can find a divergent range of opinions on use cases.

In the last few years, however, [Cockburn, use cases] has become the standard book on the subject. In this chapter, I've followed the terminology and advice of that book for the excellent reason that when we've disagreed in the past, I've usually ended up agreeing with Alistair Cockburn in the end. He also maintains a Web site at <http://usecases.org>. [Constantine and Lockwood] provides a convincing process for deriving user interfaces from use cases; also see <http://foruse.com>.

Chapter 12

Communication Diagrams

Communication diagrams, a kind of interaction diagram, emphasize the data links between the various participants in the interaction. Instead of drawing each participant as a lifeline and showing the sequence of messages by vertical direction as the sequence diagrams does, the communication diagram allows free placement of participants, allows you to draw links to show how the participants connect, and use numbering to show the sequence of messages.

In UML 1.x, these diagrams were called **collaboration diagrams**. This name stuck well, and I suspect that it will be a while before people get used to the new name. (These are different from Collaborations [page 143]; hence the name change.)

Figure 12.1 shows a communication diagram for the same centralized control interaction as in Figure 4.2. With a communication diagram, we can show how the participants are linked together.

As well as showing links that are instances of associations, we can also show transient links, which arise only in the context of the interaction. In this case, the «local» link from Order to Product is a local variable; other transient links are «parameter» and «global». These keywords were used in UML 1 but are missing from UML 2. Because they are useful, I expect them to stay around in conventional use.

The numbering style of Figure 12.1 is straightforward and commonly used, but actually isn't legal UML. To be kosher UML, you have to use a nested decimal numbering scheme, as in Figure 12.2.

The reason for the nested decimal numbers is to resolve ambiguity with self-calls. In Figure 4.2, you can clearly see that `getDiscountInfo` is called within the method `calculateDiscount`. With the flat numbering of Figure 12.1, however, you can't tell whether `getDiscountInfo` is called within `calculateDiscount` or within the overall `calculatePrice` method. The nested numbering scheme resolves this problem.

Despite its illegality, many people prefer a flat numbering scheme. The nested numbers can get very tangled, particularly as calls get rather nested, leading to

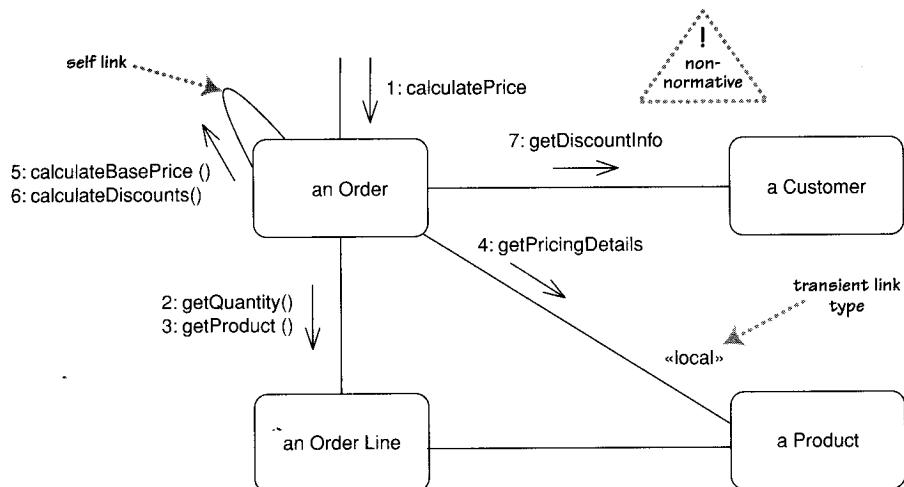


Figure 12.1 Communication diagram for centralized control

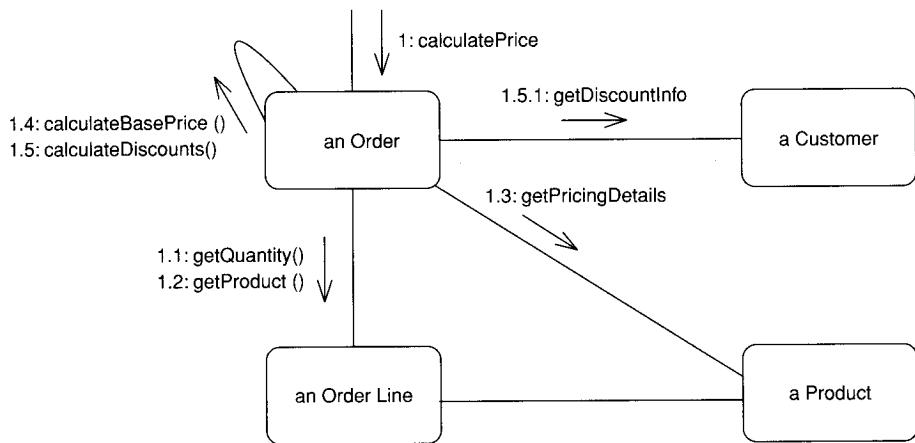


Figure 12.2 Communication diagram with nested decimal numbering

such sequence numbers as 1.1.1.2.1.1. In these cases, the cure for ambiguity can be worse than the disease.

As well as numbers, you may also see letters on messages; these letters indicate different threads of control. So messages A5 and B2 would be in different threads; messages 1a1 and 1b1 would be different threads concurrently nested

within message 1. You also see thread letters on sequence diagrams, although this doesn't convey the concurrency visually.

Communication diagrams don't have any precise notation for control logic. They do allow you to use iteration markers and guards (page 59), but they don't allow you to fully specify control logic. There is no special notation for creating or deleting objects, but the «create» and «delete» keywords are common conventions.

When to Use Communication Diagrams

The main question with communication diagrams is when to use them rather than the more common sequence diagrams. A strong part of the decision is personal preference: Some people like one over the other. Often, that drives the choice more than anything else. On the whole, most people seem to prefer sequence diagrams, and for once, I'm with the majority.

A more rational approach says that sequence diagrams are better when you want to emphasize the sequence of calls and that communication diagrams are better when you want to emphasize the links. Many people find that communication diagrams are easier to alter on a whiteboard, so they are a good approach for exploring alternatives, although in those cases, I often prefer CRC cards.

Chapter 14

Component Diagrams

A debate that's always ranged large in the OO community is what the difference is between a component and any regular class. This is not a debate that I want to settle here, but I can show you the notation the UML uses to distinguish between them.

UML 1 had a distinctive symbol for a component (Figure 14.1). UML 2 removed that icon but allows you to annotate a class box with a similar-looking icon. Alternatively, you can use the «component» keyword.

Other than the icon, components don't introduce any notation that we haven't already seen. Components are connected through implemented and required interfaces, often using the ball-and-socket notation (page 71) just as for class diagrams. You can also decompose components by using composite structure diagrams.

Figure 14.2 shows an example component diagram. In this example, a sales till can connect to a sales server component, using a sales message interface. Because the network is unreliable, a message queue component is set up so the till can talk to the server when the network is up and talk to a queue when the network is down; the queue will then talk to the server when the network becomes available. As a result, the message queue both supplies the sales message interface to talk with the till and requires that interface to talk with the server. The server is broken down into two major components. The transaction processor realizes the sales message interface, and the accounting driver talks to the accounting system.

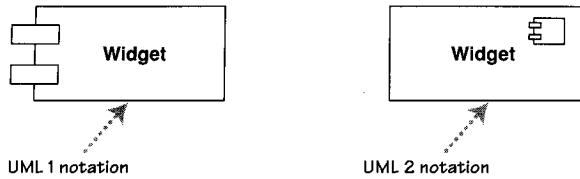


Figure 14.1 Notation for components

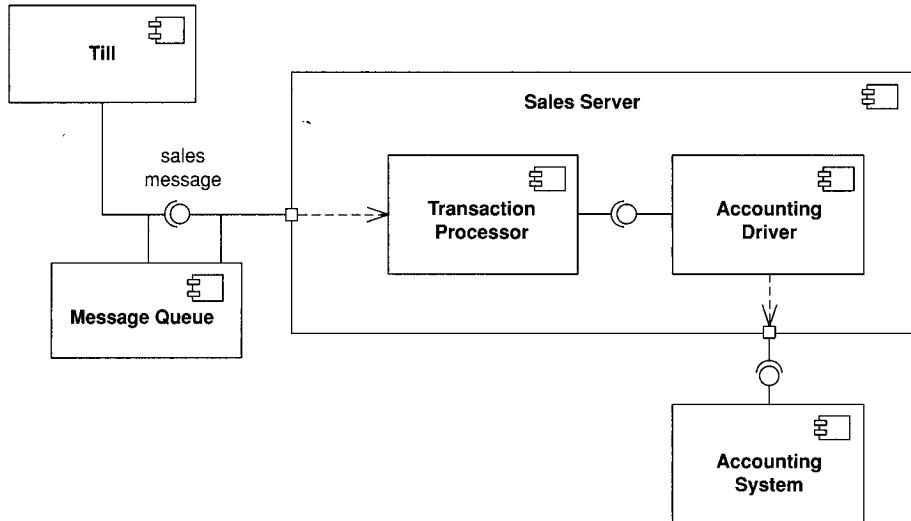


Figure 14.2 An example component diagram

As I've already said, the issue of what is a component is the subject of endless debate. One of the more helpful statements I've found is this:

Components are not a technology. Technology people seem to find this hard to understand. Components are about how customers want to relate to software. They want to be able to buy their software a piece at a time, and to be able to upgrade it just like they can upgrade their stereo. They want new pieces to work seamlessly with their old pieces, and to be able to upgrade on their own schedule, not the manufacturer's schedule. They want to be able to mix and match pieces from various manufacturers. This is a very reasonable requirement. It is just hard to satisfy.

Ralph Johnson, <http://www.c2.com/cgi/wiki?DoComponentsExist>

The important point is that components represent pieces that are independently purchasable and upgradeable. As a result, dividing a system into components is as much a marketing decision as it is a technical decision, for which [Hohmann] is an excellent guide. It's also a reminder to beware of overly fine-grained components, because too many components are hard to manage, especially when versioning rears its ugly head, hence "DLL hell."

In earlier versions of the UML, components were used to represent physical structures, such as DLLs. That's no longer true; for this task, you now use artifacts (page 97).

When to Use Component Diagrams

Use component diagrams when you are dividing your system into components and want to show their interrelationships through interfaces or the breakdown of components into a lower-level structure.

Chapter 3

What Are Microservices?

Section 1.1 provided an initial definition of the term microservice. However, there are a number of different ways to define microservices. The different definitions are based on different aspects of microservices. They also show for which reasons the use of microservices is advantageous. At the end of the chapter the reader should have his or her own definition of the term microservice—depending on the individual project scenario.

The chapter discusses the term microservice from different perspectives:

- Section 3.1 focuses on the size of microservices.
- Section 3.2 explains the relationship between microservices, architecture, and organization by using the Conway’s Law.
- Section 3.3 presents a domain architecture of microservices based on domain-driven design (DDD) and bounded context.
- Section 3.5 explains why microservices should contain a user interface (UI).

3.1 Size of a Microservice

The name “microservices” conveys the fact that the size of the service matters; obviously, microservices are supposed to be small.

One way to define the size of a microservice is to count the lines of code (LOC).¹ However, such an approach has a number of problems:

- It depends on the programming language used. Some languages require more code than others to express the same functionality—and microservices are explicitly not supposed to predetermine the technology stack. Therefore, defining microservices based on this metric is not very useful.
- Finally, microservices represent an architecture approach. Architectures, however, should follow the conditions in the domain rather than adhering to technical metrics such as LOC. Also for this reason attempts to determine size based on code lines should be viewed critically.

In spite of the voiced criticism, LOC can be an indicator for a microservice. Still, the question as to the ideal size of a microservice remains. How many LOC may a microservice have? Even if there are no absolute standard values, there are nevertheless influencing factors, which may argue for larger or smaller microservices.

Modularization

One factor is modularization. Teams develop software in modules to be better able to deal with its complexity; instead of having to understand the entire software package, developers only need to understand the module(s) they are working on as well as the interplay between the different modules. This is the only way for a team to work productively in spite of the enormous complexity of a typical software system. In daily life there are often problems as modules get larger than originally planned. This makes them hard to understand and hard to maintain, because changes require an understanding of the entire module. Thus it is very sensible to keep microservices as small as possible. On the other hand, microservices, unlike many other approaches to modularization, have an overhead.

Distributed Communication

Microservices run within independent processes. Therefore, communication between microservices is distributed communication via the network. For this type of system, the “First Rule of Distributed Object Design”² applies. This rule states that systems should not be distributed if it can be avoided. The reason for this is that

1. <http://yobriefca.se/blog/2013/04/28/micro-service-architecture/>

2. <http://martinfowler.com/bliki/FirstLaw.html>

a call on another system via the network is orders of magnitude slower than a direct call within the same process. In addition to the pure latency time, serialization and deserialization of parameters and results are time consuming. These processes not only take a long time, but also cost CPU capacity.

Moreover, distributed calls might fail because the network is temporarily unavailable or the called server cannot be reached—for instance due to a crash. This increases complexity when implementing distributed systems, because the caller has to deal with these errors in a sensible manner.

Experience³ teaches us that microservice-based architectures work in spite of these problems. When microservices are designed to be especially small, the amount of distributed communication increases and the overall system gets slower. This is an argument for larger microservices. When a microservice contains a UI and fully implements a specific part of the domain, it can operate without calling on other microservices in most cases, because all components of this part of the domain are implemented within one microservice. The desire to limit distributed communication is another reason to build systems according to the domain.

Sustainable Architecture

Microservices also use distribution to design architecture in a sustainable manner through distribution into individual microservices: it is much more difficult to use a microservice than a class. The developer has to deal with the distribution technology and has to use the microservice interface. In addition, he or she might have to make preparations for tests to include the called microservice or replace it with a stub. Finally, he has to contact the team responsible for the respective microservice.

To use a class within a deployment monolith is much simpler—even if the class belongs to a completely different part of the monolith and falls within the responsibility of another team. However, because it is so simple to implement a dependency between two classes, unintended dependencies tend to accumulate within deployment monoliths. In the case of microservices dependencies are harder to implement, which prevents the creation of unintended dependencies.

Refactoring

However, the boundaries between microservices also create challenges, for instance during refactoring. If it becomes apparent that a piece of functionality does not fit well within its present microservice, it has to be moved to another microservice. If the target microservice is written in a different programming language, this transfer

3. <http://martinfowler.com/articles/distributed-objects-microservices.html>

inevitably leads to a new implementation. Such problems do not arise when functionalities are moved within a microservice. This consideration may argue for larger microservices, and this topic is the focus of section 7.3.

Team Size

The independent deployment of microservices and the division of the development effort into teams result in an upper limit for the size of an individual microservice. A team should be able to implement features within a microservice and deploy those features into production independently of other teams. By ensuring this, the architecture enables the scaling of development without requiring too much coordination effort between the teams.

A team has to be able to implement features independently of the other teams. Therefore, at first glance it seems like the microservice should be large enough to enable the implementation of different features. When microservices are smaller, a team can be responsible for several microservices, which together enable the implementation of a domain. A lower limit for the microservice size does not result from the independent deployment and the division into teams.

However, an upper limit does result from it: when a microservice has reached a size that prevents its further development by a single team, it is too large. For that matter a team should have a size that is especially well suited for agile processes, which is typically three to nine people. Thus a microservice should never grow so large that a team of three to nine people cannot develop it further by themselves. In addition to the sheer size, the number of features to be implemented in an individual microservice plays an important role. Whenever a large number of changes is necessary within a short time, a team can rapidly become overloaded. Section 12.2 highlights alternatives that enable several teams to work on the same microservice. However, in general a microservice should never grow so large that several teams are necessary to work on it.

Infrastructure

Another important factor influencing the size of a microservice is the infrastructure. Each microservice has to be able to be deployed independently. It must have a continuous delivery pipeline and an infrastructure for running the microservice, which has to be present not only in production but also during the different test stages. Also databases and application servers might belong to infrastructure. Moreover, there has to be a build system for the microservice. The code for the microservice has to be versioned independently of that for other microservices. Thus a project within version control has to exist for the microservice.

Depending on the effort that is necessary to provide the required infrastructure for a microservice, the sensible size for a microservice can vary. When a small microservice size is chosen, the system is distributed into many microservices, thus requiring more infrastructure. In the case of larger microservices, the system overall contains fewer microservices and consequently requires less infrastructure.

Build and deployment of microservices should anyhow be automated. Nevertheless, it can be laborious to provide all necessary infrastructure components for a microservice. Once setting up the infrastructure for new microservices is automated, the expenditure for providing infrastructures for additional microservices decreases. This automation enables further reduction of the microservice size. Companies that have been working with microservices for some time usually simplify the creation of new microservices by providing the necessary infrastructure in an automated manner.

Additionally, some technologies enable reduction of the infrastructure overhead to such an extent that substantially smaller microservices are possible—however, with a number of limitations in such cases. Such nanoservices are discussed in Chapter 14, “Technologies for Microservices.”

Replaceability

A microservice should be as easy to replace as possible. Replacing a microservice can be sensible when its technology becomes outdated or if the microservice code is of such bad quality that it cannot be developed any further. The replaceability of microservices is an advantage when compared to monolithic applications, which can hardly be replaced at all. When a monolith cannot be reasonably maintained anymore, its development has either to be continued in spite of the associated high costs or a similarly cost-intensive migration has to take place. The smaller a microservice is, the easier it is to replace it with a new implementation. Above a certain size a microservice may be difficult to replace, for it then poses the same challenges as a monolith. Replaceability thus limits the size of a microservice.

Transactions and Consistency

Transactions possess the so-called ACID characteristics:

- **Atomicity** indicates that a given transaction is either executed completely or not at all. In case of an error, all changes are reversed.
- **Consistency** means that data is consistent before and after the execution of a transaction—database constraints, for instance, are not violated.

- **Isolation** indicates that the operations of transactions are separated from each other.
- **Durability** indicates permanence: changes to the data are stored and are still available after a crash or other interruption of service.

Within a microservice, changes to a transaction can take place. Moreover, the consistency of data in a microservice can be guaranteed very easily. Beyond an individual microservice, this gets difficult, and overall coordination is necessary. Upon the rollback of a transaction all changes made by all microservices would have to be reversed. This is laborious and hard to implement, for the delivery of the decision that changes have to be reversed has to be guaranteed. However, communication within networks is unreliable. Until it is decided whether a change may take place, further changes to the data are barred. If additional changes have taken place, it might no longer be possible to reverse a certain change. However, when microservices are kept from introducing data changes for some time, system throughput is reduced.

However, when communications occur via messaging systems, transactions are possible (see section 8.4). With this approach, transactions are also possible without a close link between the microservices.

Consistency

In addition to transactions, data consistency is important. An order, for instance, also has to be recorded as revenue. Only then will revenue and order data be consistent. Data consistency can be achieved only through close coordination. Data consistency can hardly be guaranteed across microservices. This does not mean that the revenue for an order will not be recorded at all. However, it will likely not happen exactly at the same point of time and maybe not even within one minute of order processing because the communication occurs via the network—and is consequently slow and unreliable.

Data changes within a transaction and data consistency are only possible when all data being processed is part of the same microservice. Therefore, data changes determine the lower size limit for a microservice: when transactions are supposed to encompass several microservices and data consistency is required across several microservices, the microservices have been designed too small.

Compensation Transactions across Microservices

At least in the case of transactions there is an alternative: if a data change has to be rolled back in the end, compensation transactions can be used for that.

A classic example for a distributed transaction is a travel booking, which consists of a hotel, a rental car, and a flight. Either everything has to be booked together or nothing at all. Within real systems and also within microservices, this functionality is divided into three microservices because the three tasks are very different. Inquiries are sent to the different systems whether the desired hotel room, rental car, and flight are available. If all are available, everything is reserved. If, for instance, the hotel room suddenly becomes unavailable, the reservations for the flight and the rental car have to be cancelled. However, in the real world the concerned companies will likely demand a fee for the booking cancellation. Due to that, the cancellation is not only a technical event happening in the background like a transaction rollback but also a business process. This is much easier to represent with a compensation transaction. With this approach, transactions across several elements in microservice environments can also be implemented without the presence of a close technical link. A compensation transaction is just a normal service call. Technical as well as business reasons can lead to the use of mechanisms such as compensation transactions for microservices.

Summary

In conclusion, the following factors influence the size of a microservice (see Figure 3.1):

- The team size sets an upper limit; a microservice should never be so large that one very large team or several teams are required to work on it. Eventually, the teams are supposed to work and bring software into production independently of each other. This can only be achieved when each team works on a separate deployment unit—that is, a separate microservice. However, one team can work on several microservices.
- Modularization further limits the size of a microservice: The microservice should preferably be of a size that enables a developer to understand all its aspects and further develop it. Even smaller is of course better. This limit is below the team size: whatever one developer can still understand, a team should still be able to develop further.
- Replaceability reduces with the size of the microservice. Therefore, replaceability can influence the upper size limit for a microservice. This limit lies below the one set by modularization: when somebody decides to replace a microservice, this person has first of all to be able to understand the microservice.
- A lower limit is set by infrastructure: if it is too laborious to provide the necessary infrastructure for a microservice, the number of microservices should be kept rather small; consequently the size of each microservice will be larger.

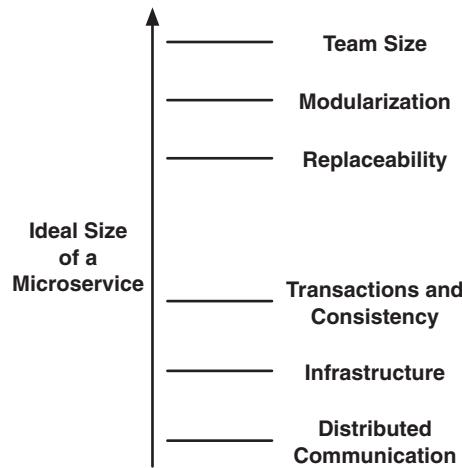


Figure 3.1 Factors Influencing the Size of a Microservice

- Similarly, distributed communication overhead increases with the number of microservices. For this reason, the size of microservices should not be set too small.
- Consistency of data and transactions can only be ensured within a microservice. Therefore, microservices should not be so small that consistency and transactions must be ensured across several microservices.

These factors not only influence the size of microservices but also reflect a certain idea of microservices. According to this idea, the main advantages of microservices are independent deployment and the independent work of the different teams, along with the replaceability of microservices. The optimal size of a microservice can be deduced from these desired features.

However, there are also other reasons for microservices. When microservices are, for instance, introduced because of their independent scaling, a microservice size has to be chosen that ensures that each microservice is a unit, which has to scale independently.

How small or large a microservice can be, cannot be deduced solely from these criteria. This also depends on the technology being used. Especially the effort necessary for providing infrastructure for a microservice and the distributed communication depends on the utilized technology. Chapter 14 looks at technologies, which make the development of very small services possible—denoted as nanoservices. These nanoservices have different advantages and disadvantages to microservices, which, for instance, are implemented using technologies presented in Chapter 13, “Example of a Microservice-based Architecture.”

Thus, there is no ideal size. The actual microservice size will depend on the technology and the use case of an individual microservice.

Try and Experiment

How great is the effort required for the deployment of a microservice in your language, platform, and infrastructure?

- Is it just a simple process? Or is it a complex infrastructure containing application servers or other infrastructure elements?
- How can the effort for the deployment be reduced so that smaller microservices become possible?

Based on this information you can define a lower limit for the size of a microservice. Upper limits depend on team size and modularization, so you should also think of appropriate limits in those terms.

3.2 Conway's Law

Conway's Law⁴ was coined by the American computer scientist Melvin Edward Conway and indicates the following:

Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

It is important to know that this law is meant to apply not only to software but to any kind of design. The communication structures that Conway mentions, do not have to be identical to the organization chart. Often there are informal communication structures, which also have to be considered in this context. In addition, the geographical distribution of teams can influence communication. After all it is much simpler to talk to a colleague who works in the same room or at least in the same office than with one working in a different city or even in a different time zone.

4. <http://www.melconway.com/research/committees.html>

Reasons for the Law

Conway's Law derives from the fact that each organizational unit designs a specific part of the architecture. If two architectural parts have an interface, coordination in regards to this interface is required—and, consequently, a communication relationship between the organizational units that are responsible for the respective parts of the architecture.

From Conway's Law it can also be deduced that design modularization is sensible. Via such a design, it is possible to ensure that not every team member has to constantly coordinate with every other team member. Instead the developers working on the same module can closely coordinate their efforts, while team members working on different modules only have to coordinate when they develop an interface—and even then only in regards to the specific design of the external features of this interface.

However, the communication relationships extend beyond that. It is much easier to collaborate with a team within the same building than with a team located in another city, another country, or even within a different time zone. Therefore, architectural parts having numerous communication relationships are better implemented by teams that are geographically close to each other, because it is easier for them to communicate with each other. In the end, the Conway's Law focuses not on the organization chart but on the real communication relationships.

By the way, Conway postulated that a large organization has numerous communication relationships. Thus communication becomes more difficult or even impossible in the end. As a consequence, the architecture can be increasingly affected and finally break down. In the end, having too many communication relationships is a real risk for a project.

The Law as Limitation

Normally Conway's Law is viewed as a limitation, especially from the perspective of software development. Let us assume that a project is modularized according to technical aspects (see Figure 3.2). All developers with a UI focus are grouped into one team, the developers with backend focus are put into a second team, and data bank experts make up the third team. This distribution has the advantage that all three teams consist of experts for the respective technology. This makes it easy and transparent to create this type of organization. Moreover, this distribution also appears logical. Team members can easily support each other, and technical exchange is also facilitated.

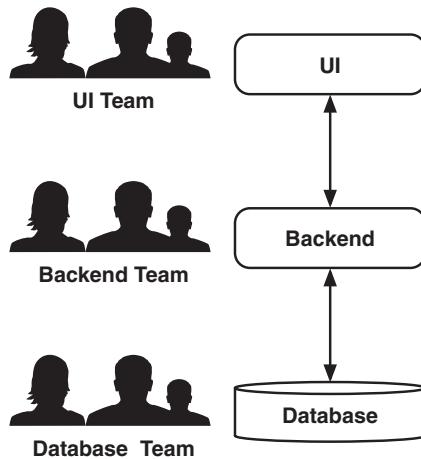


Figure 3.2 Technical Project Distribution

According to Conway's Law, it follows from such a distribution that the three teams will implement three technical layers: a UI, a backend, and a database. The chosen distribution corresponds to the organization, which is in fact sensibly built. However, this distribution has a decisive disadvantage: a typical feature requires changes to UI, backend, and database. The UI has to render the new features for the clients, the backend has to implement the logic, and the database has to create structures for the storage of the respective data. This results in the following disadvantages:

- The person wishing to have a feature implemented has to talk to all three teams.
- The teams have to coordinate their work and create new interfaces.
- The work of the different teams has to be coordinated in a manner that ensures that their efforts temporally fit together. The backend, for instance, cannot really work without getting input from the database, and the UI cannot work without input from the backend.
- When the teams work in sprints, these dependencies cause time delays: The database team generates in its first sprint the necessary changes, within the second sprint the backend team implements the logic, and in the third sprint the UI is dealt with. Therefore, it takes three sprints to implement a single feature.

In the end this approach creates a large number of dependencies as well as a high communication and coordination overhead. Thus this type of organization does not make much sense if the main goal is to implement new features as rapidly as possible.

Many teams following this approach do not realize its impact on architecture and do not consider this aspect further. This type of organization focuses instead on the notion that developers with similar skills should be grouped together within the organization. This organization becomes an obstacle to a design driven by the domain like microservices, whose development is not compatible with the division of teams into technical layers.

The Law as Enabler

However, Conway's Law can also be used to support approaches like microservices. If the goal is to develop individual components as independently of each other as possible, the system can be distributed into domain components. Based on these domain components, teams can be created. Figure 3.3 illustrates this principle: There are individual teams for product search, clients, and the order process. These teams work on their respective components, which can be technically divided into UI, back-end, and database. By the way, the domain components are not explicitly named in the figure, for they are identical to the team names. Components and teams are synonymous. This approach corresponds to the idea of so-called cross-functional teams, as proposed by methods such as Scrum. These teams should encompass different roles so that they can cover a large range of tasks. Only a team designed along such principles can be in charge of a component—from engineering requirements via implementation through to operation.

The division into technical artifacts and the interface between the artifacts can then be settled within the teams. In the easiest case, developers only have to talk to developers sitting next to them to do so. Between teams, coordination is more complex. However, inter-team coordination is not required very often, since features are ideally implemented by independent teams. Moreover, this approach creates thin interfaces between the components. This avoids laborious coordination across teams to define the interface.

Ultimately, the key message to be taken from Conway's Law is that architecture and organization are just two sides of the same coin. When this insight is cleverly put to use, the system will have a clear and useful architecture for the project. Architecture and organization have the common goal to ensure that teams can work in an unobstructed manner and with as little coordination overhead as possible.

The clean separation of functionality into components also facilitates maintenance. Since an individual team is responsible for individual functionality and component, this distribution will have long-term stability, and consequently the system will remain maintainable.

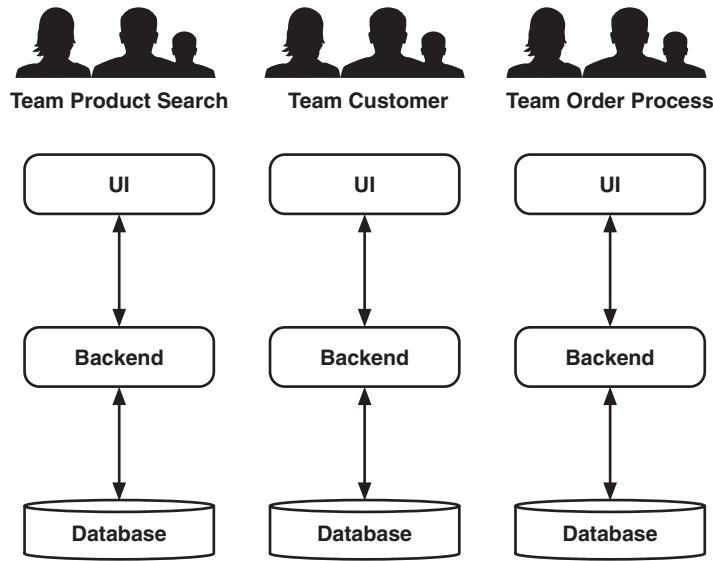


Figure 3.3 Project by Domains

The teams need requirements to work upon. This means that the teams need to contact people who define the requirements. This affects the organization beyond the projects, for the requirements come from the departments of the enterprise, and these also according to Conway's Law have to correspond to the team structures within the project and the domain architecture. Conway's Law can be expanded beyond software development to the communication structures of the entire organization, including the users. To put it the other way round: the team structure within the project and consequently the architecture of a microservice system can follow from the organization of the departments of the enterprise.

The Law and Microservices

The previous discussion highlighted the relationship between architecture and organization of a project only in a general manner. It would be perfectly conceivable to align the architecture along functionalities and devise teams, each of which are in charge for a separate functionality without using microservices. In this case the project would develop a deployment monolith within which all functionalities are implemented. However, microservices support this approach. Section 3.1 already discussed that microservices offer technical independence. In conjunction with the division by domains, the teams become even more independent of each other and have even less need to coordinate their work. The technical coordination as well as the coordination concerning the domains can be reduced to the absolute minimum. This makes it far easier to work in parallel on numerous features and also to bring the features in production.

Microservices as a technical architecture are especially well suited to support the approach to devise a Conway's Law-based distribution of functionalities. In fact, exactly this aspect is an essential characteristic of a microservices-based architecture.

However, orienting the architecture according to the communication structures entails that a change to the one also requires a change of the other. This makes architectural changes between microservices more difficult and makes the overall process less flexible. Whenever a piece of functionality is moved from one microservice to another, this might have the consequence that another team has to take care of this functionality from that point on. This type of organizational change renders software changes more complex.

As a next step this chapter will address how the distribution by domain can best be implemented. Domain-driven design (DDD) is helpful for that.

Try and Experiment

Have a look at a project you know:

- What does the team structure look like?
 - Is it technically motivated, or is it divided by domain?
 - Would the structure have to be changed to implement a microservices-based approach?
 - How would it have to be changed?
- Is there a sensible way to distribute the architecture onto different teams? Eventually each team should be in charge of independent domain components and be able to implement features relating to them.
 - Which architectural changes would be necessary?
 - How laborious would the changes be?

3.3 Domain-Driven Design and Bounded Context

In his book of the same title, Eric Evans formulated domain-driven design (DDD)⁵ as pattern language. It is a collection of connected design patterns and supposed to support software development especially in complex domains. In the following text, the names of design patterns from Evan's book are written in *italics*.

5. Eric Evans. 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison-Wesley.

Domain-driven design is important for understanding microservices, for it supports the structuring of larger systems according to domains. Exactly such a model is necessary for the division of a system into microservices. Each microservice is meant to constitute a domain, which is designed in such a way that only one microservice has to be changed in order to implement changes or to introduce new features. Only then is the maximal benefit to be derived from independent development in different teams, as several features can be implemented in parallel without the need for extended coordination.

Ubiquitous Language

DDD defines a basis for how a model for a domain can be designed. An essential foundation of DDD is *Ubiquitous Language*. This expression denotes that the software should use exactly the same terms as the domain experts. This applies on all levels: in regards to code and variable names as well as for database schemas. This practice ensures that the software really encompasses and implements the critical domain elements. Let us assume for instance that there are express orders in an e-commerce system. One possibility would be to generate a Boolean value with the name “fast” in the order table. This creates the following problem: domain experts have to translate the term “express order,” which they use on a daily basis, into “order with a specific Boolean value.” They might not even know what Boolean values are. This renders any discussion of the model more difficult, for terms have to be constantly explained and related to each other. The better approach is to call the table within the database scheme “express order.” In that case it is completely transparent how the domain terms are implemented in the system.

Building Blocks

To design a domain model, DDD identifies basic patterns:

- *Entity* is an object with an individual identity. In an e-commerce application, the customer or the items could be examples for *Entities*. *Entities* are typically stored in databases. However, this is only the technical implementation of the concept *Entity*. An *Entity* belongs in essence to the domain modeling like the other DDD concepts.
- *Value Objects* do not have their own identity. An address can be an example of a *Value Object*, for it makes only sense in the context of a specific customer and therefore does not have an independent identity.
- *Aggregates* are composite domain objects. They facilitate the handling of invariants and other conditions. An order, for instance, can be an *Aggregate* of order lines. This can be used to ensure that an order from a new customer does not exceed a certain value. This is a condition that has to be fulfilled by calculating values from the order lines so that the order as *Aggregate* can control these conditions.

- *Services* contain business logic. DDD focuses on modeling business logic as *Entities*, *Value Objects*, and *Aggregates*. However, logic accessing several such objects cannot be sensibly modeled using these objects. For these cases there are *Services*. The order process could be such a *Service*, for it needs access to items and customers and requires the *Entity* order.
- *Repositories* serve to access all *Entities* of a type. Typically, there is a persistence technology like a database behind a *Repository*.
- *Factories* are mostly useful to generate complex domain objects. This is especially the case when these contain for instance many associations.

Aggregates are of special importance in the context of microservices: within an *Aggregate* consistency can be enforced. Because consistency is necessary, parallel changes have to be coordinated in an *Aggregate*. Otherwise two parallel changes might endanger consistency. For instance, when two order positions are included in parallel into an order, consistency can be endangered. The order has already a value of €900 and is maximally allowed to reach €1000. If two order positions of €60 each are added in parallel, both might calculate a still acceptable total value of €960 based on the initial value of €900. Therefore, changes have to be serialized so that the final result of €1020 can be controlled. Accordingly, changes to *Aggregates* have to be serialized. For this reason, an *Aggregate* cannot be distributed across two microservices. In such a scenario consistency cannot be ensured. Consequently, *Aggregates* cannot be divided between microservices.

Bounded Context

Building blocks such as *Aggregate* represent for many people the core of DDD. DDD describes, along with strategic design, how different domain models interact and how more complex systems can be built up this way. This aspect of DDD is probably even more important than the building blocks. In any case it is the concept of DDD, which influences microservices.

The central element of strategic designs is the *Bounded Context*. The underlying reasoning is that each domain model is only sensible in certain limits within a system. In e-commerce, for instance, number, size, and weight of the ordered items are of interest in regards to delivery, for they influence delivery routes and costs. For accounting on the other hand prices and tax rates are relevant. A complex system consists of several *Bounded Contexts*. In this it resembles the way complex biological organisms are built out of individual cells, which are likewise separate entities with their own inner life.

Bounded Context: An Example

The customer from the e-commerce system shall serve as an example for a *Bounded Context* (see Figure 3.4). The different *Bounded Contexts* are Order, Delivery, and Billing. The component Order is responsible for the order process. The component Delivery implements the delivery process. The component Billing generates the bills.

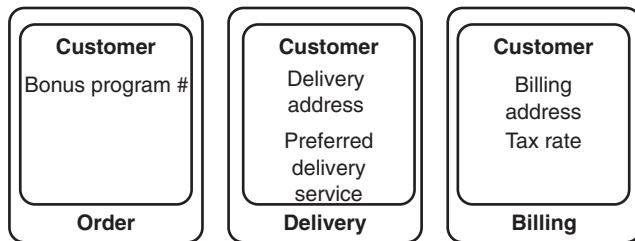


Figure 3.4 Project by Domains

Each of these Bounded Contexts requires certain customer data:

- Upon ordering the customer is supposed to be rewarded with points in a bonus program. In this Bounded Context the number of the customer has to be known to the bonus program.
- For Delivery the delivery address and the preferred delivery service of the customer are relevant.
- Finally, for generating the bill the billing address and the tax rate of the customer have to be known.

In this manner each *Bounded Context* has its own model of the customer. This renders it possible to independently change microservices. If for instance more information regarding the customer is necessary for generating bills, only changes to the *Bounded Context* billing are necessary.

It might be sensible to store basic information concerning the customer in a separate *Bounded Context*. Such fundamental data is probably sensible in many *Bounded Contexts*. To this purpose the *Bounded Contexts* can cooperate (see below).

(continued)

A universal model of the customer, however, is hardly sensible. It would be very complex since it would have to contain all information regarding the customer. Moreover, each change to customer information, which is necessary in a certain context, would concern the universal model. This would render such changes very complicated and would probably result in permanent changes to the model.

To illustrate the system setup in the different *Bounded Contexts* a *Context Map* can be used (see section 7.2). Each of the *Bounded Contexts* then can be implemented within one or several microservices.

Collaboration between *Bounded Contexts*

How are the individual *Bounded Contexts* connected? There are different possibilities:

- In case of a *Shared Kernel* the domain models share some common elements; however, in other areas they differ.
- *Customer/Supplier* means that a subsystem offers a domain model for the caller. The caller in this case is the client who determines the exact setup of the model.
- This is very different in the case of *Conformist*: The caller uses the same model as the subsystem, and the other model is thereby forced upon him. This approach is relatively easy, for there is no need for translation. One example is a standard software for a certain domain. The developers of this software likely know a lot about the domain since they have seen many different use cases. The caller can use this model to profit from the knowledge from the modeling.
- The *Anticorruption Layer* translates a domain model into another one so that both are completely decoupled. This enables the integration of legacy systems without having to take over the domain models. Often data modeling is not very meaningful in legacy systems.

- *Separate Ways* means that the two systems are not integrated, but stay independent of each other.
- In the case of *Open Host Service*, the *Bounded Context* offers special services everybody can use. In this way everybody can assemble their own integration. This is especially useful when an integration with numerous other systems is necessary and when the implementation of these integrations is too laborious.
- *Published Language* achieves similar things. It offers a certain domain modeling as a common language between the *Bounded Contexts*. Since it is widely used, this language can hardly be changed anymore afterwards.

Bounded Context and Microservices

Each microservice is meant to model one domain so that new features or changes have only to be implemented within one microservice. Such a model can be designed based on *Bounded Context*.

One team can work on one or several *Bounded Contexts*, which each serve as a foundation for one or several microservices. Changes and new features are supposed to concern typically only one *Bounded Context*—and thus only one team. This ensures that teams can work largely independently of each other. A *Bounded Context* can be divided into multiple microservices if that seems sensible. There can be technical reasons for that. For example, a certain part of a *Bounded Context* might have to be scaled up to a larger extent than the others. This is simpler if this part is separated into its own microservice. However, designing microservices that contain multiple *Bounded Contexts* should be avoided, for this entails that several new features might have to be implemented in one microservice. This interferes with the goal to develop features independently.

Nevertheless, it is possible that a special requirement comprises many *Bounded Contexts*—in that case additional coordination and communication will be required.

The coordination between teams can be regulated via different collaboration possibilities. These influence the independence of the teams as well: *Separate Ways*, *Anticorruption Layer* or *Open Host Service* offer a lot of independence. *Conformist* or *Customer/Supplier* on the other hand tie the domain models very closely together. For *Customer/Supplier* the teams have to coordinate their efforts closely: the supplier needs to understand the requirements of the customer. For *Conformist*, however, the teams do not need to coordinate: one team defines the model that the other team just uses unchanged (see Figure 3.5).

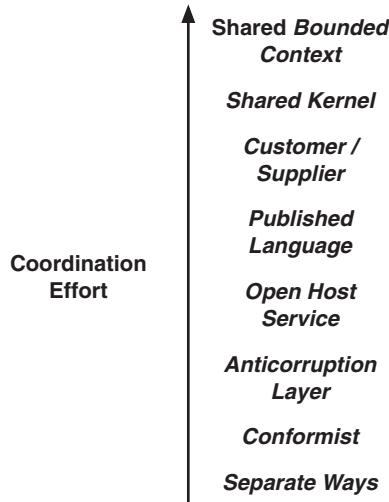


Figure 3.5 Communication Effort of Different Collaborations

As in the case of Conway's Law from section 3.2, it becomes very apparent that organization and architecture are very closely linked. When the architecture enables a distribution of the domains in which the implementation of new features only requires changes to a defined part of the architecture, these parts can be distributed to different teams in such a way that these teams can work largely independently of each other. DDD and especially *Bounded Context* demonstrate what such a distribution can look like and how the parts can work together and how they have to coordinate.

Large-Scale Structure

With large-scale structure, DDD also addresses the question how the system in its entirety can be viewed from the different *Bounded Contexts* with respect to microservices.

- A *System Metaphor* can serve to define the fundamental structure of the entire system. For example, an e-commerce system can orient itself according to the shopping process: the customer starts out looking for products, then he/she will compare items, select one item, and order it. This can give rise to three microservices: search, comparison, and order.
- A *Responsibility Layer* divides the system into layers with different responsibilities. Layers can call other layers only if those are located below them. This does not refer to a technical division into database, UI and logic. In an

e-commerce system, domain layers might be (for example) the catalog, the order process, and billing. The catalog can call on the order process, and the order process can call on the generation of the bill. However, calls into the other direction are not permitted.

- *Evolving Order* suggests it is best not to determine the overall structure too rigidly. Instead, the order should arise from the individual components in a stepwise manner.

These approaches can provide an idea how the architecture of a system, which consists of different microservices, can be organized (see also Chapter 7, “Architecture of Microservice-based Systems”).

Try and Experiment

Look at a project you know:

- Which *Bounded Contexts* can you identify?
- Generate an overview of the *Bounded Contexts* in a *Context Map*. Compare section 7.2.
- How do the *Bounded Contexts* cooperate? (Anticorruption Layer Customer/Supplier etc.). Add this information to the *Context Map*.
- Would other mechanisms have been better at certain places? Why?
- How could the *Bounded Contexts* be sensibly distributed to teams so that features are implemented by independent teams?

These questions might be hard to answer because you need to get a new perspective on the system and how the domains are modeled in the system.

3.4 Why You Should Avoid a Canonical Data Model (Stefan Tilkov)

by Stefan Tilkov, innoQ

In recent times, I've been involved in a few architecture projects on the enterprise level again. If you've never been in that world, that is, if you've been focusing on

individual systems so far, let me give you the gist of what this kind of environment is like. There are lots of meetings, more meetings, and even more meetings; there's an abundance of slide decks, packed with text and diagrams—none of that Presentation Zen nonsense, please. There are conceptual architecture frameworks, showing different perspectives; there are guidelines and reference architectures, enterprise-wide layering approaches, a little bit of SOA and EAI and ESB and portals and (lately) API talk thrown in for good measure. Vendors and system integrators and (of course) consultants all see their chance to exert influence on strategic decisions, making their products or themselves an integral part of the company's future strategy. It can be a very frustrating but (at least sometimes) also very rewarding experience: those wheels are very big and really hard to turn, but if you manage to turn them, the effect is significant.

It's also amazing to see how many of the things that cause problems when building large systems are repeated on the enterprise level. (We don't often make mistakes ... but if we do, we make them big!) My favorite one is the idea of establishing a canonical data model (CDM) for all of your interfaces.

If you haven't heard of this idea before, a quick summary is: Whatever kind of technology you're using (an ESB, a BPM platform, or just some assembly of services of some kind), you standardize the data models of the business objects you exchange. In its extreme (and very common) form, you end up with having just one kind of Person, Customer, Order, Product, etc., with a set of IDs, attributes, and associations everyone can agree on. It isn't hard to understand why that might seem a very compelling thing to attempt. After all, even a nontechnical manager will understand that the conversion from one data model to another whenever systems need to talk to each other is a complete waste of time. It's obviously a good idea to standardize. Then, anyone who happens to have a model that differs from the canonical one will have to implement a conversion to and from it just once, new systems can just use the CDM directly, and everyone will be able to communicate without further ado!

In fact, it's a horrible, horrible idea. Don't do it.

In his book on domain-driven design, Eric Evans gave a name to a concept that is obvious to anyone who has actually successfully built a larger system: the *Bounded Context*. This is a structuring mechanism that avoids having a single huge model for all of your application, simply because that (a) becomes unmanageable and (b) makes no sense to begin with. It recognizes that a Person or a Contract are different things in different contexts on a *conceptual level*. This is not an implementation problem—it's reality.

If this is true for a large system—and trust me, it is—it's infinitely more true for an enterprise-wide architecture. Of course you can argue that with a CDM, you're

only standardizing the interface layer, but that doesn't change a thing. You're still trying to make everyone agree what a concept means, and my point is that you should recognize that not every single system has the same needs.

But isn't this all just pure theory? Who cares about this, anyway? The amazing thing is that organizations are excellent in generating a huge amount of work based on bad assumptions. The CDM (in the form I've described it here) requires coordination between all the parties that use a particular object in their interfaces (unless you trust that people will be able to just design the right thing from scratch on their own, which you should never do). You'll have meetings with some enterprise architect and a few representatives for specific systems, trying to agree what a customer is. You'll end up with something that has tons of optional attributes because all the participants insisted theirs need to be there, and with lots of things that are kind of weird because they reflect some system's internal restrictions. Despite the fact that it'll take you ages to agree on it, you'll end up with a zombie interface model will be universally hated by everyone who has to work with it.

So is a CDM a universally bad idea? Yes, unless you approach it differently. In many cases, I doubt a CDM's value in the first place and think you are better off with a different and less intrusive kind of specification. But if you want a CDM, here are a number of things you can do to address the problems you'll run into:

- Allow independent parts to be specified independently. If only one system is responsible for a particular part of your data model, leave it to the people to specify what it looks like canonically. Don't make them participate in meetings. If you're unsure whether the data model they create has a significant overlap with another group's, it probably hasn't.
- Standardize on formats and possibly fragments of data models. Don't try to come up with a consistent model of the world. Instead, create small building blocks. What I'm thinking of are e.g. small XML or JSON fragments, akin to microformats, that standardize small groups of attributes (I wouldn't call them business objects).
- Most importantly, don't push your model from a central team downwards or outwards to the individual teams. Instead, it should be the teams who decide to "pull" them into their own context when they believe they provide value. It's not you who's doing the really important stuff (even though that's a common delusion that's attached to the mighty Enterprise Architect title). Collect the data models the individual teams provide in a central location, if you must, and make them easy to browse and search. (Think of providing a big elastic search index as opposed to a central UML model.)

What you actually need to do as an enterprise architect is to get out of people's way. In many cases, a crucial ingredient to achieve this is to create as little centralization as possible. It shouldn't be your goal to make everyone do the same thing. It should be your goal to establish a minimal set of rules that enable people to work as independently as possible. A CDM of the kind I've described above is the exact opposite.

3.5 Microservices with a UI?

This book recommends that you equip microservices with a UI. The UI should offer the functionality of the microservice to the user. In this way, all changes in regards to one area of functionality can be implemented in one microservice—regardless of whether they concern the UI, the logic, or the database. However, microservice experts so far have different opinions in regards to the question of whether the integration of UI into microservices is really required. Ultimately, microservices should not be too large. And when logic is supposed to be used by multiple frontends, a microservice consisting of pure logic without a UI might be sensible. In addition, it is possible to implement the logic and the UI in two different microservices but to have them implemented by one team. This enables implementation of features without coordination across teams.

Focusing on microservices with a UI puts the main emphasis on the distribution of the domain logic instead of a distribution by technical aspects. Many architects are not familiar with the domain architecture, which is especially important for microservices-based architectures. Therefore, a design where the microservices contain the UI is helpful as a first approach in order to focus the architecture on the domains.

Technical Alternatives

Technically the UI can be implemented as Web UI. When the microservices have a RESTful-HTTP interface, the Web-UI and the RESTful-HTTP interface are very similar—both use HTTP as a protocol. The RESTful-HTTP interface delivers JSON or XML, the Web UI HTML. If the UI is a Single-Page Application, the JavaScript code is likewise delivered via HTTP and communicates with the logic via RESTful HTTP. In case of mobile clients, the technical implementation is more complicated. Section 8.1 explains this in detail. Technically a deployable artifact can deliver via an HTTP interface, JSON/XML, and HTML. In this way it implements the UI and allows other microservices to access the logic.

Self-Contained System

Instead of calling this approach “Microservice with UI” you can also call it “Self-Contained System” (SCS).⁶ SCS define microservices as having about 100 lines of code, of which there might be more than one hundred in a complete project.

An SCS consists of many of those microservices and contains a UI. It should communicate with other SCSs asynchronously, if at all. Ideally each functionality should be implemented in just one SCS, and there should be no need for SCSs to communicate with each other. An alternative approach might be to integrate the SCSs at the UI-level.

In an entire system, there are then only five to 25 of these SCS. An SCS is something one team can easily deal with. Internally the SCS can be divided into multiple microservices.

The following definitions result from this reasoning:

- SCS is something a team works on and which represents a unit in the domain architecture. This can be an order process or a registration. It implements a sensible functionality, and the team can supplement the SCS with new features. An alternative name for a SCS is a vertical. The SCS distributes the architecture by domain. This is a vertical design in contrast to a horizontal design. A horizontal design would divide the system into layers, which are technically motivated—for instance UI, logic, or persistence.
- A microservice is a part of a SCS. It is a technical unit and can be independently deployed. This conforms with the microservice definition put forward in this book. However, the size given in the SCS world corresponds to what this book denotes as nanoservices (see Chapter 14).
- This book refers to nanoservices as units that are still individually deployable but make technical trade-offs in some areas to further reduce the size of the deployment units. For that reason, nanoservices do not share all technical characteristics of microservices.

SCS inspired the definition of microservices as put forward in this book. Still there is no reason not to separate the UI into a different artifact in case the microservice gets otherwise too large. Of course, it is more important that the microservice is small and thus maintainable than to integrate the UI. But the UI and logic should at least be implemented by the same team.

6. <http://scs-architecture.org>

3.6 Conclusion

Microservices are a modularization approach. For a deeper understanding of microservices, the different perspectives discussed in this chapter are very helpful:

- Section 3.1 focuses on the size of microservices. But a closer look reveals that the size of microservices itself is not that important, even though size is an influencing factor. However, this perspective provides a first impression of what a microservice should be. Team size, modularization, and replaceability of microservices each determine an upper size limit. The lower limit is determined by transactions, consistency, infrastructure, and distributed communication.
- Conway’s Law (section 3.2) shows that the architecture and organization of a project are closely linked—in fact, they are nearly synonymous. Microservices can further improve the independence of teams and thus ideally support architectural designs that aim at the independent development of functionalities. Each team is responsible for a microservice and therefore for a certain part of a domain, so that the teams are largely independent concerning the implementation of new functionalities. Thus, in regards to domain logic there is hardly any need for coordination across teams. The requirement for technical coordination can likewise be reduced to a minimum because of the possibility for technical independence.
- In section 3.3 domain-driven design provides a very good impression as to what the distribution of domains in a project can look like and how the individual parts can be coordinated. Each microservice can represent a *Bounded Context*. This is a self-contained piece of domain logic with an independent domain model. Between the *Bounded Contexts* there are different possibilities for collaboration.
- Finally, section 3.5 demonstrates that microservices should contain a UI to be able to implement the changes for functionality within an individual microservice. This does not necessarily have to be a deployment unit; however, the UI and microservice should be in the responsibility of one team.

Together these different perspectives provide a balanced picture of what constitutes microservices and how they can function.

Essential Points

To put it differently: A successful project requires three components:

- an organization (This is supported by Conway's Law.)
- a technical approach (This can be microservices.)
- a domain design as offered by DDD and *Bounded Context*

The domain design is especially important for the long-term maintainability of the system.

Try and Experiment

Look at the three approaches for defining microservices: size, Conway's Law, and domain-driven design.

- Section 1.2 showed the most important advantages of microservices. Which of the goals to be achieved by microservices are best supported by the three definitions? DDD and Conway's Law lead, for instance, to a better time-to-market.
- Which of the three aspects is, in your opinion, the most important? Why?

Chapter 13

Example of a Microservices-Based Architecture

This chapter provides an example of an implementation of a microservices-based architecture. It aims at demonstrating concrete technologies in order to lay the foundation for experiments. The example application has a very simple domain architecture containing a few compromises. Section 13.1 deals with this topic in detail.

For a real system with a comparable low complexity as in the presented example application, an approach without microservices would be better suited. However, the low complexity makes the example application easy to understand and simple to extend. Some aspects of a microservice environment, such as security, documentation, monitoring, or logging are not illustrated in the example application—but these aspects can be relatively easily addressed with some experiments.

Section 13.2 explains the technology stack of the example application. The build tools are described in section 13.3. Section 13.4 deals with Docker as a technology for the deployment. Docker needs to run in a Linux environment. Section 13.5 describes Vagrant as a tool for generating such environments. Section 13.6 introduces Docker Machine as alternative tool for the generation of a Docker environment, which can be combined with Docker Compose for the coordination of several Docker containers (section 13.7). The implementation of Service Discovery is discussed in section 13.8. The communication between the microservices and the user interface is the main topic of section 13.9. Thanks to resilience other microservices are not affected if a single microservice fails. In the example application resilience is implemented with Hystrix (section 13.10). Load Balancing (section 13.11), which can distribute the load onto several instances of a microservice, is closely related to that. Possibilities for the integration of non-Java-technologies are detailed in section 13.12, and testing is discussed in section 13.13.

The code of the example application can be found at <https://github.com/ewolff/microservice>. It is Apache-licensed, and can, accordingly, be used and extended freely for any purpose.

13.1 Domain Architecture

The example application has a simple web interface, with which users can submit orders. There are three microservices (see Figure 13.1):

- “Catalog” keeps track of products. Items can be added or deleted.
- “Customer” performs the same task in regards to customers: It can register new customers or delete existing ones.
- “Order” can not only show orders but also create new orders.

For the orders the microservice “Order” needs access to the two other microservices, “Customer” and “Catalog.” The communication is achieved via REST. However, this interface is only meant for the internal communication between the microservices. The user can interact with all three microservices via the HTML-/HTTP-interface.

Separate Data Storages

The data storages of the three microservices are completely separate. Only the respective microservice knows the information about the business objects. The microservice “Order” saves only the primary keys of the items and customers, which are necessary for the access via the REST interface. A real system should use

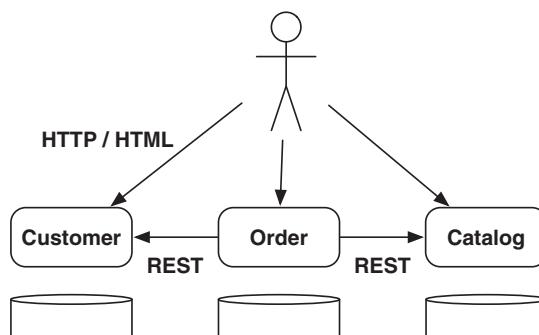


Figure 13.1 Architecture of the Example Application

artificial keys as the internal primary keys so that they do not become visible to the outside. These are internal details of the data storage that should be hidden. To expose the primary keys, the class `SpringRestDataConfig` within the microservices configures Spring Data REST accordingly.

Lots of Communication

Whenever an order needs to be shown, the microservice “Customer” is called for the customer data and the microservice “Catalog” for each line of the order in order to determine the price of the item. This can have a negative influence on the response times of the application as the display of the order cannot take place before all requests have been answered by the other microservices. As the requests to the other services take place synchronously and sequentially, latencies will add up. This problem can be solved by using asynchronous parallel requests.

In addition, a lot of computing power is needed to marshal the data for sending and receiving. This is acceptable in case of such a small example application. When such an application is supposed to run in production, alternatives have to be considered.

This problem can, for instance, be solved by caching. This is relatively easy as customer data will not change frequently. Items can change more often—still, not so fast that caching would pose a problem. Only the amount of data can interfere with this approach. The use of microservices has the advantage that such a cache can be implemented relatively simply at the interface of the microservices, or even at the level of HTTP, if this protocol is used. An HTTP cache, like the one used for websites, can be added to REST services in a transparent manner and without much programming effort.

Bounded Context

Caching will solve the problem of too long response times technically. However, very long response times can also be a sign of a fundamental problem. Section 3.3 argued that a microservice should contain a *Bounded Context*. A specific domain model is only valid in a *Bounded Context*. The modularization into microservices in this example contradicts this idea: The domain model is used to modularize the system into the microservices “Order” for orders, “Catalog” for items, and “Customer” for customers. In principle the data of these entities should be modularized in different *Bounded Contexts*.

The described modularization implements, in spite of low domain complexity, a system consisting of three microservices. In this manner the example application is easy to understand while still having several microservices and demonstrating the

communication between microservices. In a real system the microservice “Order” can also handle information about the items that is relevant for the order process such as the price. If necessary, the service can replicate the data from another microservice into its own database in order to access it efficiently. This is an alternative to the aforementioned caching. There are different possibilities how the domain models can be modularized into the different *Bounded Contexts* “Order,” “Customer,” and “Catalog.”

This design can cause errors: when an order has been put into the system and the price of the item is changed afterwards, the price of the order changes as well, which should not happen. In case the item is deleted, there is even an error when displaying the order. In principle the information concerning the item and the customer should become part of the order. In that case the historical data of the orders including customer and item data would be transferred into the service “Order.”

Don’t Modularize Microservices by Data!

It is important to understand the problem inherent in architecting a microservices system by domain model. Often the task of a global architecture is misunderstood: The team designs a domain model, which comprises, for instance, objects such as customers, orders, and items. Based on this model microservices are defined. That is how the modularization into microservices could have come about in the example application, resulting in a huge amount of communication. A modularization based on processes such as ordering, customer registration, and product search might be more advantageous. Each process could be a *Bounded Context* that has its own domain model for the most important domain objects. For product search the categories of items might be the most relevant, while for the ordering process, data like weight and size might matter more.

The modularization by data can also be advantageous in a real system. When the microservice “Order” gets too big in combination with the handling of customer and product data, it is sensible to modularize data handling. In addition, the data can be used by other microservices. When devising the architecture for a system, there is rarely a single right way of doing things. The best approach depends on the system and the properties the system should have.

13.2 Basic Technologies

Microservices in the example application are implemented with Java. Basic functionalities for the example application are provided by the Spring Framework.¹ This

1. <http://projects.spring.io/spring-framework/>

framework offers not only dependency injection, but also a web framework, which enables the implementation of REST-based services.

HSQL Database

The database HSQLDB handles and stores data. It is an in-memory database, which is written in Java. The database stores the data only in RAM so that all data is lost upon restarting the application. In line with this, this database is not really suited for production use, even if it can write data to a hard disk. On the other hand, it is not necessary to install an additional database server, which keeps the example application easy. The database runs in the respective Java application.

Spring Data REST

The microservices use Spring Data REST² in order to provide the domain objects with little effort via REST and to write them into the database. Handing objects out directly means that the internal data representation leaks into the interface between the services. Changing the data structures is very difficult as the clients need to be adjusted as well. However, Spring Data REST can hide certain data elements and can be configured flexibly so that the tight coupling between the internal model and the interface can be decoupled if necessary.

Spring Boot

Spring Boot³ facilitates Spring further. Spring Boot makes the generation of a Spring system very easy: with Spring Boot starters predefined packages are available that contain everything that is necessary for a certain type of application. Spring Boot can generate WAR files, which can be installed on a Java application or web server. In addition, it is possible to run the application without an application or web server. The result of the build is a JAR file in that case, which can be run with a Java Runtime Environment (JRE). The JAR file contains everything for running the application and also the necessary code to deal with HTTP requests. This approach is by far less demanding and simpler than the use of an application server (<https://jaxenter.com/java-application-servers-dead-112186.html>).

A simple example for a Spring Boot application is shown in Listing 13.1. The main program `main` hands control over to Spring Boot. The class is passed in as a parameter so that the application can be called. The annotation `@SpringBootApplication`

2. <http://projects.spring.io/spring-data-rest/>

3. <http://projects.spring.io/spring-boot/>

makes sure that Spring Boot generates a suitable environment. For example, a web server is started, and an environment for a Spring web application is generated as the application is a web application. Because of `@RestController` the Spring Framework instantiates the class and calls methods for the processing of REST requests. `@RequestMapping` shows which method is supposed to handle which request. Upon request of the URL “/” the method `hello()` is called, which returns as result the sign chain “hello” in the HTTP body. In an `@RequestMapping` annotation, URL templates such as “/customer/{id}” can be used. Then a URL like “/customer/42” can be cut into separate parts and the 42 bound to a parameter annotated with `@PathVariable`. As dependency the application uses only `spring-boot-starter-web` pulling all necessary libraries for the application along—for instance the web server, the Spring Framework, and additional dependent classes. Section 13.3 will discuss this topic in more detail.

Listing 13.1 *A simple Spring Boot REST Service*

```
@RestController
@SpringBootApplication
public class ControllerAndMain {

    @RequestMapping("/")
    public String hello() {
        return "hello";
    }

    public static void main(String[] args) {
        SpringApplication.run(ControllerAndMain.class, args);
    }
}
```

Spring Cloud

Finally, the example application uses Spring Cloud⁴ to gain easy access to the Netflix Stack. Figure 13.2 shows an overview.

Spring Cloud offers via the Spring Cloud Connectors access to the PaaS (platform as a service) Heroku and Cloud Foundry. Spring Cloud for Amazon Web Services offers an interface for services from the Amazon Cloud. This part of Spring Cloud is responsible for the name of the project but is not helpful for the implementation of microservices.

4. <http://projects.spring.io/spring-cloud/>

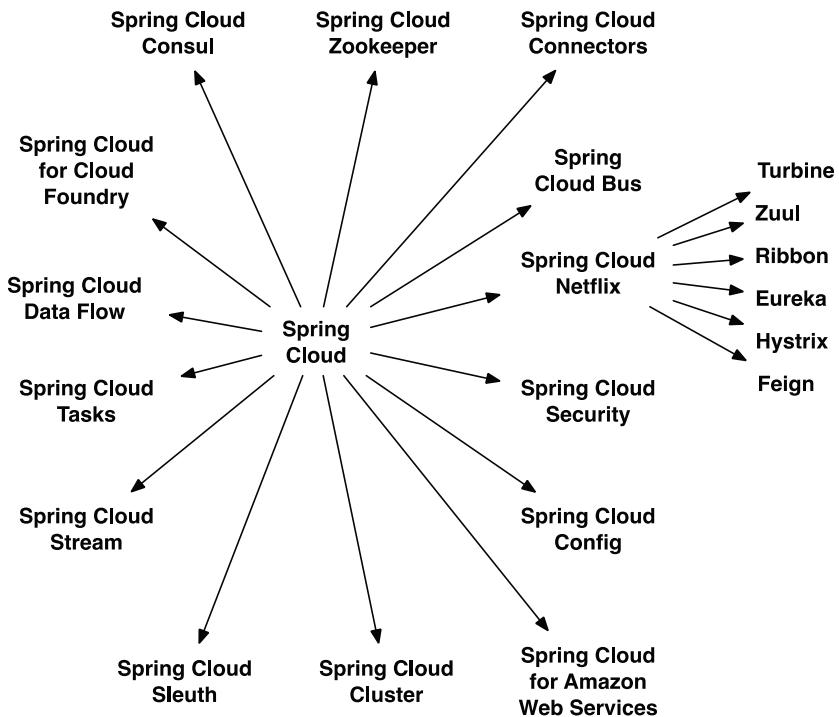


Figure 13.2 Overview of Spring Cloud

However, the other sub-projects of Spring Cloud provide a very good basis for the implementation of microservices:

- **Spring Cloud Security** supports the implementation of security mechanisms as typically required for microservices, among those single sign on into a microservices environment. That way a user can use each of the microservices without having to log in anew every time. In addition, the user token is transferred automatically for all calls to other REST services to ensure that those calls can also work with the correct user rights.
- **Spring Cloud Config** can be used to centralize and dynamically adjust the configuration of microservices. Section 11.4 already presented technologies, which configure microservices during deployment. To be able to reproduce the state of a server at any time, a new server should be started with a new microservice instance in case of a configuration change instead of dynamically adjusting an existing server. If a server is dynamically adjusted, there is no guarantee that new servers are generated with the right configuration as they

are configured in a different way. Because of these disadvantages the example application refrains from using this technology.

- **Spring Cloud Bus** can send dynamic configuration changes for Spring Cloud Config. Moreover, the microservices can communicate via Spring Cloud Bus. However, the example application does not use this technology because Spring Cloud Config is not used, and the microservices communicate via REST.
- **Spring Cloud Sleuth** enables distributed tracing with tools like Zipkin or Htrace. It can also use a central log storage with ELK (see section 11.2).
- **Spring Cloud Zookeeper** supports Apache Zookeeper (see section 7.10). This technology can be used to coordinate and configure distributed services.
- **Spring Cloud Consul** facilitates Services Discovery using Consul (see section 7.11).
- **Spring Cloud Cluster** implements leader election and stateful patterns using technologies like Zookeeper or Consul. It can also use the NoSQL data store Redis or the Hazelcast cache.
- **Spring Cloud for Cloud Foundry** provides support for the Cloud Foundry PaaS. For example, single sign on (SSO) and OAuth2 protected resources are supported as well as creating managed service for the Cloud Foundry service broker.
- **Spring Cloud Connectors** support access to services provided by PaaS like Heroku or Cloud Foundry.
- **Spring Cloud Data Flow** helps with the implementation of applications and microservices for Big Data analysis.
- **Spring Cloud Tasks** provides features for short lived microservices.
- Finally, **Spring Cloud Stream** supports messaging using Redis, Rabbit, or Kafka.

Spring Cloud Netflix

Spring Cloud Netflix offers simple access to Netflix Stack, which has been especially designed for the implementation of microservices. The following technologies are part of this stack:

- **Zuul** can implement routing of requests to different services.
- **Ribbon** serves as a load balancer.

- **Hystrix** assists with implementing resilience in microservices.
- **Turbine** can consolidate monitoring data from different Hystrix servers.
- **Feign** is an option for an easier implementation of REST clients. It is not limited to microservices. It is not used in the example application.
- **Eureka** can be used for Service Discovery.

These technologies are the ones that influence the implementation of the example application most.

Try and Experiment

For an introduction into Spring it is worthwhile to check out the Spring Guides at <https://spring.io/guides/>. They show in detail how Spring can be used to implement REST services or to realize messaging solutions via JMS. An introduction into Spring Boot can be found at <https://spring.io/guides/gs/spring-boot/>. Working your way through these guides provides you with the necessary know-how for understanding the additional examples in this chapter.

13.3 Build

The example project is built with the tool Maven.⁵ The installation of the tool is described at <https://maven.apache.org/download.cgi>. The command **mvn package** in the directory **microservice/microservice-demo** can be used to download all dependent libraries from the Internet and to compile the application.

The configuration of the projects for Maven is saved in files named **pom.xml**. The example project has a Parent-POM in the directory **microservice-demo**. It contains the universal settings for all modules and in addition a list of the example project modules. Each microservice is such a module, and some infrastructure servers are modules as well. The individual modules have their own **pom.xml**, which contains the module name among other information. In addition, they contain the dependencies, i.e., the Java libraries they use.

5. <http://maven.apache.org/>

Listing 13.2 Part of pom.xml Including Dependencies

```
...
<dependencies>

    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>
            spring-boot-starter-data-jpa
        </artifactId>
    </dependency>
```

Listing 13.2 shows a part of a **pom.xml**, which lists the dependencies of the module. Depending on the nature of the Spring Cloud features the project is using, additional entries have to be added in this part of the **pom.xml** usually with the **groupId** `org.springframework.cloud`.

The build process results in one JAR file per microservice, which contains the compiled code, the configuration, and all necessary libraries. Java can directly start such JAR files. Although the microservices can be accessed via HTTP, they do not have to be deployed on an application or web server. This part of the infrastructure is also contained in the JAR file.

As the projects are built with Maven, they can be imported into all usual Java IDEs (integrated development environment) for further development. IDEs simplify code changes tremendously.

Try and Experiment

- Download and compile the example:

Download the example provided at <https://github.com/ewolff/microservice>. Install Maven; see <https://maven.apache.org/download.cgi>. In the subdirectory **microservices-demo** execute the command **mvn package**. This will build the complete project.

- Create a continuous integration server for the project:

<https://github.com/ewolff/user-registration-V2> is an example project for a continuous delivery project. This contains in subdirectory **ci-setup** a

setup for a continuous integration server (Jenkins) with static code analysis (Sonarqube) and Artifactory for the handling of binary artifacts. Integrate the microservices project into this infrastructure so that a new build is triggered upon each change.

The next section (13.4) will discuss Vagrant in more detail. This tool is used for the continuous integration servers. It simplifies the generation of test environments greatly.

13.4 Deployment Using Docker

Deploying microservices is very easy:

- Java has to be installed on the server.
- The JAR file, which resulted from the build, has to be copied to the server.
- A separate configuration file **application.properties** can be created for further configurations. It is automatically read out by Spring Boot and can be used for additional configurations. An **application.properties** containing default values is comprised in the JAR file.
- Finally, a Java process has to start the application out of the JAR file.

Each microservice starts within its own Docker container. As discussed in section 11.7, Docker uses Linux containers. In this manner the microservice cannot interfere with processes in other Docker containers and has a completely independent file system. The Docker image is the basis for this file system. However, all Docker containers share the Linux kernel. This saves resources. In comparison to an operating system process a Docker container has virtually no additional overhead.

Listing 13.3 Dockerfile for a Microservice Used in the Example Application

```
FROM java
CMD /usr/bin/java -Xmx400m -Xms400m \
-jar /microservice-demo/microservice-demo-catalog\
/target/microservice-demo-catalog-0.0.1-SNAPSHOT.jar
EXPOSE 8080
```

A file called **Dockerfile** defines the composition of a Docker container. Listing 13.3 shows a Dockerfile for a microservice used in the example application:

- **FROM** determines the base image used by the Docker container. A Dockerfile for the image `java` is contained in the example project. It generates a minimal Docker image with only a JVM installed.
- **CMD** defines the command executed at the start of the Docker container. In the case of this example it is a simple command line. This line starts a Java application out of the JAR file generated by the build.
- Docker containers are able to communicate with the outside via network ports. **EXPOSE** determines which ports are accessible from outside. In the example the application receives HTTP requests via port 8080.

13.5 Vagrant

Docker runs exclusively under Linux, because it uses Linux containers. However, there are solutions for other operating systems, which start a virtual Linux machine and thus enable the use of Docker. This is largely transparent so that the use is practically identical to the use under Linux. But in addition all Docker containers need to be built and started.

To make installing and handling Docker as easy as possible, the example application uses Vagrant. Figure 13.3 shows how Vagrant works:

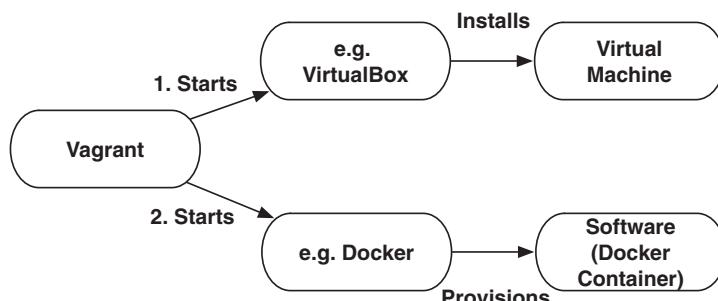


Figure 13.3 How Vagrant Works

To configure Vagrant a single file is necessary, the Vagrantfile. Listing 13.4 shows the Vagrantfile of the example application:

Listing 13.4 *Vagrantfile from the Example Application*

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/trusty64"
  config.vm.synced_folder "./microservice-demo",
    "/microservice-demo", create: true
  config.vm.network "forwarded_port",
    guest: 8080, host: 18080
  config.vm.network "forwarded_port",
    guest: 8761, host: 18761
  config.vm.network "forwarded_port",
    guest: 8989, host: 18989

  config.vm.provision "docker" do |d|
    d.build_image "--tag=java /vagrant/java"
    d.build_image "--tag=eureka /vagrant/eureka"
    d.build_image
      "--tag=customer-app /vagrant/customer-app"
    d.build_image "
      "--tag=catalog-app /vagrant/catalog-app"
    d.build_image "--tag=order-app /vagrant/order-app"
    d.build_image "--tag=turbine /vagrant/turbine"
    d.build_image "--tag=zuul /vagrant/zuul"
  end
  config.vm.provision "docker", run: "always" do |d|
    d.run "eureka",
      args: "-p 8761:8761"+
        "-v /microservice-demo:/microservice-demo"
    d.run "customer-app",
      args: "-v /microservice-demo:/microservice-demo"+
        "--link eureka:eureka"
    d.run "catalog-app",
      args: "-v /microservice-demo:/microservice-demo"+
        "--link eureka:eureka"
    d.run "order-app",
      args: "-v /microservice-demo:/microservice-demo"+
        "--link eureka:eureka"
```

```

d.run "zuul",
  args: "-v /microservice-demo:/microservice-demo"+
    " -p 8080:8080 --link eureka:eureka"
d.run "turbine",
  args: "-v /microservice-demo:/microservice-demo"+
    " --link eureka:eureka"
end

end

```

- `config.vm.box` selects a base image—in this case an Ubuntu-13.04 Linux installation (Trusty Tahr).
- `config.vm.synced_folder` mounts the directory containing the results of the Maven build into the virtual machine. In this manner the Docker containers can directly make use of the build results.
- The ports of the virtual machine can be linked to the ports of the computer running the virtual machine. The `config.vm.network` settings can be used for that. In this manner applications in the Vagrant virtual machine become accessible as if running directly on the computer.
- `config.vm.provision` starts the part of the configuration that deals with the software provisioning within the virtual machine. Docker serves as provisioning tool and is automatically installed within the virtual machine.
- `d.build_image` generates the Docker images using Dockerfiles. First the base image `java` is created. Images for the three microservices `customer-app`, `catalog-app` and `order-app` follow. The images for the Netflix technologies servers belong to the infrastructure: Eureka for Service Discovery, Turbine for monitoring, and Zuul for routing of client requests.
- Vagrant starts the individual images using `d.run`. This step is not only performed when provisioning the virtual machine, but also when the system is started anew (`run: "always"`). The option `-v` mounts the directory `/microservice-demo` into each Docker container so that the Docker container can directly execute the compiled code. `-p` links a port of the Docker container to a port of virtual machine. This link provides access to the Docker container Eureka under the host name `eureka` from within the other Docker containers.

In the Vagrant setup the JAR files containing the application code are not contained in the Docker image. The directory `/microservice-demo` does not belong to the Docker container. It resides on the host running the Docker containers, that is, the Vagrant VM. It would also be possible to copy these files into the Docker image. Afterwards the resulting image could be copied on a repository server and

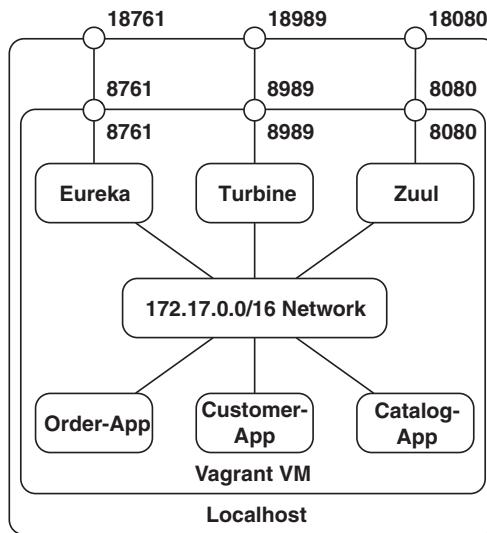


Figure 13.4 Network and Ports of the Example Application

downloaded from there. Then the Docker container would contain all necessary files to run the microservice. A deployment in production then only needs to start the Docker images on a production server. This approach is used in the Docker Machine setup (see section 13.6).

Networking in the Example Application

Figure 13.4 shows how the individual microservices of the example application communicate via the network. All Docker containers are accessible in the network via IP addresses from the 172.17.0.0/16 range. Docker generates such a network automatically and connects all Docker containers to the network. Within the network all ports are accessible that are defined in the Dockerfiles using EXPOSE. The Vagrant virtual machine is also connected to this network. Via the Docker links (see Listing 13.4) all Docker containers know the Eureka container and can access it under the host name **eureka**. The other microservices have to be looked up via Eureka. All further communication takes place via the IP address.

In addition, the `-p` option in the `d.run` entries for the Docker containers in Listing 13.4 has connected the ports to the Vagrant virtual machine. These containers can be accessed via these ports of the Vagrant virtual machine. To reach them also from the computer running the Vagrant virtual machine there is a port mapping that links the ports to the local computer. This is accomplished via the `config.vm.network` entries in `Vagrantfile`. The port 8080 of the Docker container “zuul” can, for instance, be accessed via the port 8080 in the Vagrant virtual machine.

This port can be reached from the local computer via the port 18080. So the URL `http://localhost:18080/` accesses this Docker container.

Try and Experiment

- **Run the Example Application**

The example application does not need much effort to make it run. A running example application lays the foundation for the experiments described later in this chapter.

One remark: The **Vagrantfile** defines how much RAM and how many CPUs the virtual machines gets. The settings `v.memory` and `v.cpus`, which are not shown in the listing, deal with this. Depending on the computer used, the values should be increased if a lot of RAM or many CPUs are present. Whenever the values can be increased, they should be elevated in order to speed the application up.

The installation of Vagrant is described in <https://www.vagrantup.com/docs/installation/index.html>. Vagrant needs a virtualization solution like VirtualBox. The installation of VirtualBox is explained at <https://www.virtualbox.org/wiki/Downloads>. Both tools are free.

The example can only be started once the code has been compiled. Instructions how to compile the code can be found in the experiment described in section 13.3. Afterwards you can change into the directory **docker-vagrant** and start the example demo using the command `vagrant up`.

To interact with the different Docker containers, you have to log into the virtual machine via the command `vagrant ssh`. This command has to be executed within the subdirectory **docker-vagrant**. For this to be possible an ssh client has to be installed on the computer. On Linux and Mac OS X such a client is usually already present. In Windows installing git will bring an ssh client along as described at <http://git-scm.com/download/win>. Afterwards `vagrant ssh` should work.

- **Investigate Docker Containers**

Docker contains several useful commands:

- `docker ps` provides an overview of the running Docker containers.
- The command `docker log "name of Docker container"` shows the logs.

- `docker log -f "name of Docker Container"` provides incessantly the up-to-date log information of the container.
- `docker kill "name of the Docker Container"` terminates a Docker container.
- `docker rm "name of the Docker Container"` deletes all data. For that all containers first needs to be stopped. After starting the application, the log files of the individual Docker containers can be looked at.

- **Update Docker Containers**

A Docker container can be terminated (`docker kill`) and the data of the container deleted (`docker rm`). The commands have to be executed inside the Vagrant virtual machine. `vagrant provision` starts the missing Docker containers again. This command has to be executed on the host running Vagrant. If you want to change the Docker container, simply delete it, compile the code again and generate the system anew using `vagrant provision`. Additional Vagrant commands include the following:

- `vagrant halt` terminates the virtual machine.
- `vagrant up` starts it again.
- `vagrant destroy` destroys the virtual machine and all saved data.

- **Store Data on Disk**

Right now the Docker container does not save the data so that it is lost upon restarting. The used HSQLDB database can also save the data into a file. To achieve that a suitable HSQLDB URL has to be used, see http://hsqldb.org/doc/guide/dbproperties-chapt.html#dpc_connection_url. Spring Boot can read the JDBC URL out of the `application.properties` file; see <http://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-sql.html#boot-features-connect-to-production-database>. Now the container can be restarted without data loss. But what happens if the Docker container has to be generated again? Docker can save data also outside of the container itself; compare <https://docs.docker.com/userguide/dockervolumes/>. These options provide a good basis for further experiments. Also another database than HSQLDB can be used, such as MySQL. For that purpose another Docker container has to be installed that contains the database.

(continued)

In addition to adjusting the JDBC URL, a JDBC driver has to be added to the project.

- **How is the Java Docker Image Built?**

The Docker file is more complex than the ones discussed here. <https://docs.docker.com/reference/builder/> demonstrates which commands are available in Dockerfiles. Try to understand the structure of the Dockerfiles.

13.6 Docker Machine

Vagrant serves to install environments on a developer laptop. In addition to Docker, Vagrant can use simple shell scripts for deployment. However, for production environments this solution is unsuitable. Docker Machine⁶ is specialized in Docker. It supports many more virtualization solutions as well as some cloud providers.

Figure 13.5 demonstrates how Docker Machine builds a Docker environment: First, using a virtualization solution like VirtualBox, a virtual machine is installed. This virtual machine is based on boot2docker, a very lightweight version of Linux designed specifically as a running environment for Docker containers. On that Docker Machine installs a current version of Docker. A command like `docker-machine create --driver virtualbox dev` generates, for instance, a new environment with the name dev running on a VirtualBox computer.

The Docker tool now can communicate with this environment. The Docker command line tools use a REST interface to communicate with the Docker server. Accordingly, the command line tool just has to be configured in a way that enables

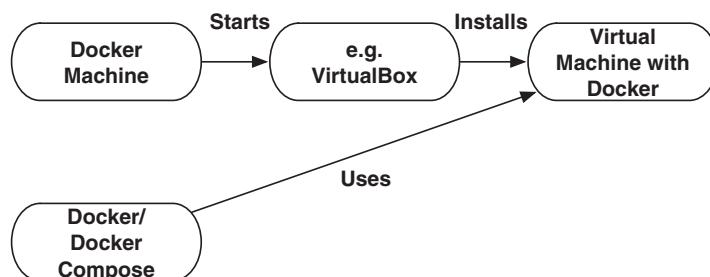


Figure 13.5 Docker Machine

6. <https://docs.docker.com/machine/>

it to communicate with the server in a suitable manner. In Linux or Mac OS X, the command `eval "$(docker-machine env dev)"` is sufficient to configure the Docker appropriately. For Windows PowerShell, the command `docker-machine.exe env --shell powershell dev` must be used and in Windows cmd `docker-machine.exe env --shell cmd dev`.

Docker Machine thus renders it very easy to install one or several Docker environments. All the environments can be handled by Docker Machine and accessed by the Docker command line tool. As Docker Machine also supports technologies like Amazon Cloud or VMware vSphere, it can be used to generate production environments.

Try and Experiment

The example application can also run in an environment created by Docker Machine.

The installation of Docker Machine is described at <https://docs.docker.com/machine/#installation>. Docker Machine requires a virtualization solution like VirtualBox. How to install VirtualBox can be found at <https://www.virtualbox.org/wiki/Downloads>. Using `docker-machine create --virtualbox-memory "4096" --driver virtualbox dev` a Docker environment called **dev** can now be created on a Virtual Box. Without any further configuration the storage space is set to 1 GB, which is not sufficient for a larger number of Java Virtual Machines.

`docker-machine` without parameters displays a help text, and `docker-machine create` shows the options for the generation of a new environment. <https://docs.docker.com/machine/get-started-cloud/> demonstrates how Docker Machine can be used in a Cloud. This means that the example application can also easily be started in a cloud environment.

At the end of your experiments, `docker-machine rm` deletes the environment.

13.7 Docker Compose

A microservice-based system comprises typically several Docker containers. These have to be generated together and need to be put into production simultaneously.

This can be achieved with Docker Compose.⁷ It enables the definition of Docker containers, which each house one service. YAML serves as format.

7. <http://docs.docker.com/compose/>

Listing 13.5 Docker Compose Configuration for the Example Application

```
version: '2'
services:
  eureka:
    build: ../microservice-demo/microservice-demo-eureka-server
    ports:
      - "8761:8761"
  customer:
    build: ../microservice-demo/microservice-demo-customer
    links:
      - eureka
  catalog:
    build: ../microservice-demo/microservice-demo-catalog
    links:
      - eureka
  order:
    build: ../microservice-demo/microservice-demo-order
    links:
      - eureka
  zuul:
    build: ../microservice-demo/microservice-demo-zuul-server
    links:
      - eureka
    ports:
      - "8080:8080"
  turbine:
    build: ../microservice-demo/microservice-demo-turbine-
server
    links:
      - eureka
    ports:
      - "8989:8989"
```

Listing 13.5 shows the configuration of the example application. It consists of the different services. `build` references the directory containing the Dockerfile. The Dockerfile is used to generate the image for the service. `links` defines which additional Docker containers the respective container should be able to access. All containers can access the Eureka container under the name `eureka`. In contrast to the Vagrant configuration there is no Java base image, which contains only a Java installation. This is because Docker Compose supports only containers that really offer a service. Therefore, this base image has to be downloaded from the Internet. Besides, in case of the Docker Compose containers the JAR files are copied into the Docker images so that the images contain everything for starting the microservices.

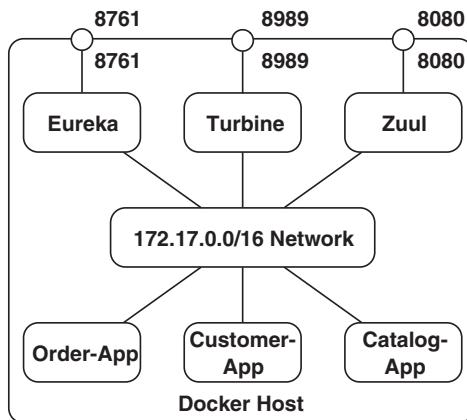


Figure 13.6 Network and Ports of the Example Application

The resulting system is very similar to the Vagrant system (see Figure 13.6). The Docker containers are linked via their own private network. From the outside, only Zuul can be accessed for the processing of requests and Eureka for the dashboard. They are running directly on a host that then can be accessed from the outside.

Using `docker-compose build` the system is created based on the Docker Compose configuration. Thus the suitable images are generated. `docker-compose up` then starts the system. Docker Compose uses the same settings as the Docker command line tool so it can also work together with Docker Machine. Thus it is transparent whether the system is generated on a local virtual machine or somewhere in the Cloud.

Try and Experiment

- **Run the Example with Docker Compose**

The example application possesses a suitable Docker Compose configuration. Upon the generation of an environment with Docker Machine, Docker Compose can be used to create the Docker containers. `README.md` in the directory `docker` describes the necessary procedure.

- **Scale the Application**

Have a look at the `docker-compose scale` command. It can scale the environment. Services can be restarted and logs can be analyzed and

(continued)

finally stopped. Once you have started the application, you can test these functionalities.

- **Cluster Environments for Docker**

Mesos (<http://mesos.apache.org/>) together with Mesosphere (<http://mesosphere.com/>), Kubernetes (<http://kubernetes.io/>), or CoreOS (<http://coreos.com/>) offers similar options as Docker Compose and Docker Machine. However they are meant for servers and server clusters. The Docker Compose and Docker Machine configurations can provide a good basis for running the application on these platforms.

13.8 Service Discovery

Section 7.11 introduced the general principles of Service Discovery. The example application uses Eureka⁸ for Service Discovery.

Eureka is a REST-based server, which enables services to register themselves so that other services can request their location in the network. In essence, each service can register a URL under its name. Other services can find the URL by the name of the service. Using this URL other services can then send REST messages to this service.

Eureka supports replication onto several servers and caches on the client. This makes the system fail-safe against the failure of individual Eureka servers and enables rapid answer requests. Changes to data have to be replicated to all servers. Accordingly, it can take some time until they are really updated everywhere. During this time the data is inconsistent: Each server has a different version of the data.

In addition, Eureka supports Amazon Web Services because Netflix uses it in this environment. Eureka can, for instance, quite easily be combined with Amazon's scaling.

Eureka monitors the registered services and removes them from the server list if they cannot be reached anymore by the Eureka server.

Eureka is the basis for many other services of the Netflix Stack and for Spring Cloud. Through a uniform Service Discovery, other aspects such as routing can easily be implemented.

Eureka Client

For a Spring Boot application to be able to register with a Eureka server and to find other microservices, the application has to be annotated with `@EnableDiscoveryClient` or `@EnableEurekaClient`. In addition, a dependency from

8. <https://github.com/Netflix/Eureka>

spring-cloud-starter-eureka has to be included in the file `pom.xml`. The application registers automatically with the Eureka server and can access other microservices. The example application accesses other microservices via a load balancer. This is described in detail in section 13.11.

Configuration

Configuring the application is necessary to define, for instance, the Eureka server to be used. The file `application.properties` (see Listing 13.6) is used for that. Spring Boot reads it out automatically in order to configure the application. This mechanism can also be used to configure one's own code. In the example application the values serve to configure the Eureka client:

- The first line defines the Eureka server. The example application uses the Docker link, which provides the Eureka server under the host name “eureka.”
- `leaseRenewalIntervalInSeconds` determines how often data is updated between client and server. As the data has to be held locally in a cache on each client, a new service first needs to create its own cache and replicate it onto the server. Afterwards the data is replicated onto the clients. Within a test environment it is important to track system changes rapidly so that the example application uses five seconds instead of the preset value of 30 seconds. In production with many clients, this value should be increased. Otherwise the updates of information will use a lot of resources, even though the information remains essentially unchanged.
- `spring.application.name` serves as the name for the service during the registration at Eureka. During registration the name is converted into capital letters. This service would thus be known by Eureka under the name “CUSTOMER.”
- There can be several instances of each service to achieve fail over and load balancing. The `instanceId` has to be unique for each instance of a service. Because of that it contains a random number, which ensures unambiguousness.
- `preferIpAddress` makes sure that microservices register with their IP addresses and not with their host names. Unfortunately in a Docker environment host names are not easily resolvable by other hosts. This problem is circumvented by the use of IP addresses.

Listing 13.6 Part of `application.properties` with Eureka Configuration

```
eureka.client.serviceUrl.defaultZone=http://eureka:8761/eureka/  
eureka.instance.leaseRenewalIntervalInSeconds=5  
spring.application.name=catalog  
eureka.instance.metadataMap.instanceId=${random.value}  
eureka.instance.preferIpAddress=true
```

Eureka Server

The Eureka server (Listing 13.7) is a simple Spring Boot application, which turns into a Eureka server via the `@EnableEurekaServer` annotation. In addition, the server needs a dependency on `spring-cloud-starter-eureka-server`.

Listing 13.7 *Eureka Server*

```
@EnableEurekaServer
@EnableAutoConfiguration
public class EurekaApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaApplication.class, args);
    }
}
```

The Eureka server offers a dashboard that shows the registered services. In the example application, this can be found at <http://localhost:18761/> (Vagrant) or on Docker host under port 8761 (Docker Compose). Figure 13.7 shows a screenshot of

The screenshot shows a web browser window titled "Eureka" with the URL "192.168.99.100:8761". The page has a dark header with the "spring Eureka" logo and navigation links for "HOME" and "LAST 1000 SINCE STARTUP".

System Status

Environment	test	Current time	2016-08-12T12:58:29 +0000
Data center	default	Uptime	00:07
		Lease expiration enabled	true
		Renews threshold	10
		Renews (last min)	56

DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CATALOG	n/a (1)	(1)	UP (1) - 128ef58bfe14:catalog:8080
CUSTOMER	n/a (1)	(1)	UP (1) - f91ce0465104:customer:8080
ORDER	n/a (1)	(1)	UP (1) - 0f8ba497b30e:order:8080
TURBINE	n/a (1)	(1)	UP (1) - c293d899d2dd:turbine:8989
ZUUL	n/a (1)	(1)	UP (1) - e1deb522a9f1:zuul:8080

Figure 13.7 *Eureka Dashboard*

the Eureka dashboards for the example application. The three microservices and the Zuul-Proxy, which is discussed in the next section, are present on the dashboard.

13.9 Communication

Chapter 8, “Integration and Communication,” explains how microservices communicate with each other and can be integrated. The example application uses REST for internal communication. The REST end points can be contacted from outside; however, the web interface the system offers is of far greater importance. The REST implementation uses HATEOAS. The list containing all orders, for instance, contains links to the individual orders. This is automatically implemented by Spring Data REST. However, there are no links to the customer and the items of the order.

Using HATEOAS can go further: the JSON can contain a link to an HTML document for each order—and vice versa. In this way a JSON-REST-based service can generate links to HTML pages to display or modify data. Such HTML code can, for instance, present an item in an order. As the “Catalog” team provides the HTML code for the item, the catalog team itself can introduce changes to the presentation—even if the items are displayed in another module.

REST is also of use here: HTML and JSON are really only representations of the same resource that can be addressed by a URL. Via Content Negotiation the right resource representation as JSON or HTML can be selected (see section 8.2).

Zuul: Routing

The Zuul⁹ proxy transfers incoming requests to the respective microservices. The Zuul proxy is a separate Java process. To the outside only one URL is visible; however, internally the calls are processed by different microservices. This enables the system to internally change the structure of the microservices while still offering a URL to the outside. In addition, Zuul can provide web resources. In the example in Figure 13.8, Zuul provides the first HTML page viewed by the user.

Zuul needs to know which requests to transfer to which microservice. Without additional configuration Eureka will forward a request to a URL starting with “/customer” to the microservice called CUSTOMER. This renders the internal microservice names visible to the outside. However, this routing can also be configured differently. Moreover, Zuul filters can change the requests in order to implement general aspects in the system. There is, for instance, an integration with Spring Cloud Security to pass on security tokens to the microservices. Such filters can also be used to pass on certain requests to specific servers. This makes it possible, for instance,

9. <https://github.com/Netflix/zuul>

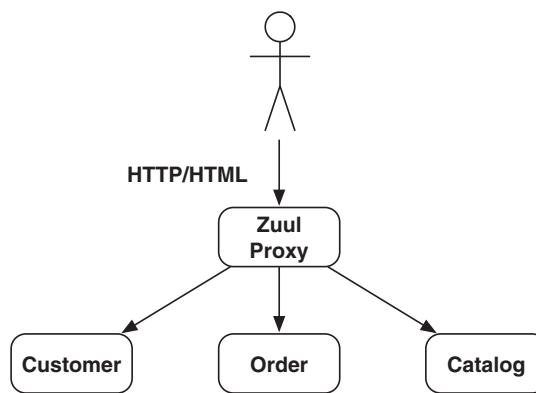


Figure 13.8 Zuul Proxy in the Example Application

to transfer requests to servers having additional analysis options for investigating error situations. In addition, a part of a microservice functionality can be replaced by another microservice.

Implementing the Zuul proxy server with Spring Cloud is very easy and analogous to the Eureka server presented in Listing 13.7. Instead of `@EnableEurekaServer` it is `@EnableZuulProxy`, which activates the Zuul-Proxy. As an additional dependency, `spring-cloud-starter-zuul` has to be added to the application, for instance, within the Maven build configuration, which then integrates the remaining dependencies of Zuul into the application.

A Zuul server represents an alternative to a Zuul proxy. It does not have routing built in, but uses filters instead. A Zuul server is activated by `@EnableZuulServer`.

Try and Experiment

- **Add Links to Customer and Items**

Extend the application so that an order contains also links to the customer and to the items and thus implements HATEOAS better. Supplement the JSON documents for customers, items, and orders with links to the forms.

- **Use the “Catalog” Service to Show Items in Orders**

Change the order presentation so that HTML from the “Catalog” service is used for items. To do so, you have to insert suitable JavaScript code into the order component, which loads HTML code from the “Catalog.”

- **Implement Zuul Filters**

Implement your own Zuul filter (see <https://github.com/Netflix/zuul/wiki/Writing-Filters>). The filter can, for instance, only release the requests. Introduce an additional routing to an external URL. For instance, /google could redirect to <http://google.com/>. Compare the Spring Cloud reference documentation.¹⁰

- **Authentication and Authorization**

Insert an authentication and authorization with Spring Cloud Security. Compare <http://cloud.spring.io/spring-cloud-security/>.

10. <http://projects.spring.io/spring-cloud/docs/1.0.3/spring-cloud.html>

13.10 Resilience

Resilience means that microservices can deal with the failure of other microservices. Even if a called microservice is not available, they will still work. Section 9.5 presented this topic.

The example application implements resilience with Hystrix.¹¹ This library protects calls so that no problems arise if a system fails. When a call is protected by Hystrix, it is executed in a different thread than the call itself. This thread is taken from a distinct thread pool. This makes it comparatively easy to implement a timeout during a call.

Circuit Breaker

In addition, Hystrix implements a *Circuit Breaker*. If a call causes an error, the *Circuit Breaker* will open after a certain number of errors. In that case subsequent calls are not directed to the called system anymore, but generate an error immediately. After a sleep window the *Circuit Breaker* closes so that calls are directed to the actual system again. The exact behavior can be configured.¹² In the configuration the error threshold percentage can be determined. That is the percentage of calls that have to cause an error within the time window for the circuit breaker to open. Also the sleep window can be defined, in which the *Circuit Breaker* is open and not sending calls to the system.

11. <https://github.com/Netflix/Hystrix/>

12. <https://github.com/Netflix/Hystrix/wiki/Configuration>

Hystrix with Annotations

Spring Cloud uses Java annotations from the project `hystrix-javanica` for the configuration of Hystrix. This project is part of `hystrix-contrib`.¹³ The annotated methods are protected according to the setting in the annotation. Without this approach Hystrix commands would have to be written, which is a lot more effort than just adding some annotations to a Java method.

To be able to use Hystrix within a Spring Cloud application, the application has to be annotated with `@EnableCircuitBreaker` respectively `@EnableHystrix`. Moreover, the project needs to contain a dependency to `spring-cloud-starter-hystrix`.

Listing 13.8 shows a section from the class `CatalogClient` of the “Order” microservice from the example application. The method `findAll()` is annotated with `@HystrixCommand`. This activates the processing in a different thread and the *Circuit Breaker*. The *Circuit Breaker* can be configured—in the example the number of calls, which have to cause an error in order to open the *Circuit Breaker*, is set to 2. In addition, the example defines a `fallbackMethod`. Hystrix calls this method if the original method generates an error. The logic in `findAll()` saves the last result in a cache, which is returned by the `fallbackMethod` without calling the real system. In this way a reply can still be returned when the called microservice fails, however this reply might no longer be up-to-date.

Listing 13.8 Example for a Method Protected by Hystrix

```
@HystrixCommand(
    fallbackMethod = "getItemsCache",
    commandProperties = {
        @HystrixProperty(
            name = "circuitBreaker.requestVolumeThreshold", value = "2")
    }
)
public Collection findAll() {
    this.itemsCache = ...
    ...
    return pagedResources.getContent();
}

private Collection getItemsCache() {
    return itemsCache;
}
```

13. <https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib>

Monitoring with the Hystrix Dashboard

Whether a *Circuit Breaker* is currently open or closed gives an indication of how well a system is running. Hystrix offers data to monitor this. A Hystrix system provides such data as a stream of JSON documents via HTTP. The Hystrix Dashboard can visualize the data in a web interface. The dashboard presents all *Circuit Breakers* along with the number of requests and their state (open/closed) (see Figure 13.9). In addition, it displays the state of the thread pools.

A Spring Boot Application needs to have the annotation `@EnableHystrixDashboard` and a dependency to `spring-cloud-starter-hystrix-dashboard` to be able to display a Hystrix Dashboard. That way any Spring Boot application might in addition show a Hystrix Dashboard, or the dashboard can be implemented in an application by itself.

Turbine

In a complex microservices environment it is not useful that each instance of a microservice visualizes the information concerning the state of its Hystrix *Circuit Breaker*. The state of all *Circuit Breakers* in the entire system should be summarized on a single dashboard. To visualize the data of the different Hystrix systems on one

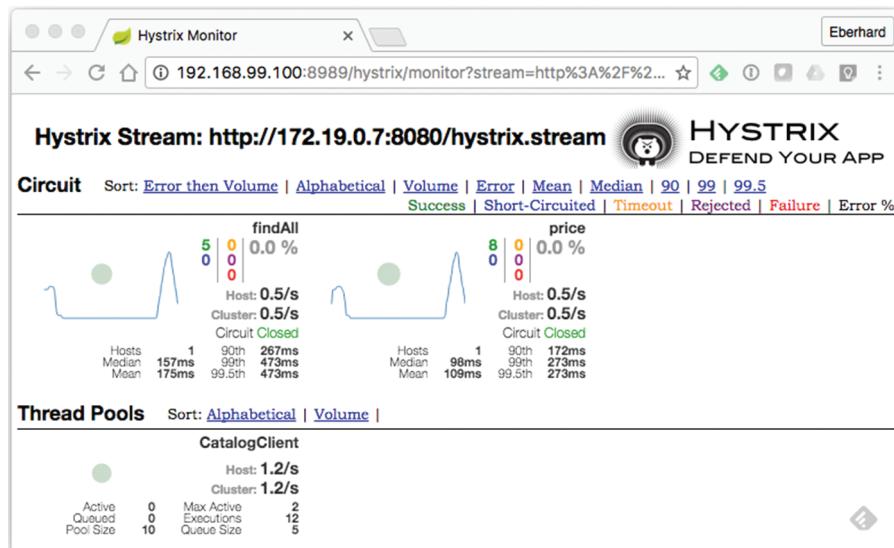


Figure 13.9 Example for a Hystrix Dashboard

dashboard, there is the Turbine project. Figure 13.10 illustrates the approach Turbine takes: the different streams of the Hystrix enabled microservices are provided at URLs like `http://<host:port>/hystrix.stream`. The Turbine server requests them and provides them in a consolidated manner at the URL `http://<host:port>/turbine.stream`. This URL can be used by the dashboard in order to display the information of all *Circuit Breakers* of the different microservice instances.

Turbine runs in a separate process. With Spring Boot the Turbine server is a simple application, which is annotated with `@EnableTurbine` and `@EnableEurekaClient`. In the example application it has the additional annotation `@EnableHystrixDashboard` so that it also displays the Hystrix Dashboard. It also needs a dependency on `spring-cloud-starter-turbine`.

Which data is consolidated by the Turbine server is determined by the configuration of the application. Listing 13.9 shows the configuration of the Turbine servers of the example project. It serves as a configuration for a Spring Boot application just like `application.properties` files but is written in YAML. The configuration sets the value `ORDER` for `turbine.aggregator.clusterConfig`. This is the application name in Eureka. `turbine.aggregator.appConfig` is the name of the data stream in the Turbine server. In the Hystrix Dashboard a URL like `http://172.17.0.10:8989/turbine.stream?cluster=ORDER` has to be used in visualize the data stream. Part of the URL is the IP address of the Turbine server, which can be found in the Eureka Dashboard. The dashboard accesses the Turbine server via the network between the Docker containers.

Listing 13.9 Configuration application.yml

```
turbine:
  aggregator:
    clusterConfig: ORDER
    appConfig: order
```

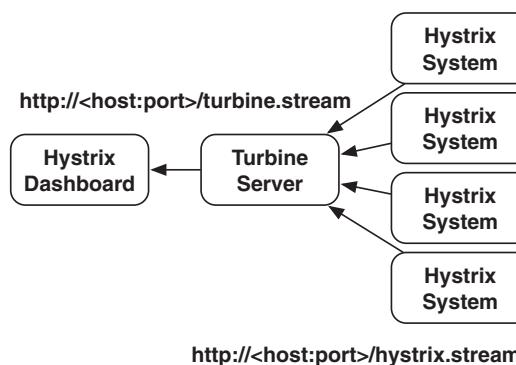


Figure 13.10 Turbine Consolidates Hystrix Monitoring Data

Try and Experiment

- **Terminate Microservices**

Using the example application generate a number of orders. Find the name of the “Catalog” Docker container using `docker ps`. Stop the “Catalog” Docker container with `docker kill`. This use is protected by Hystrix.

What happens? What happens if the “Customer” Docker container is terminated as well? The use of this microservice is not protected by Hystrix.

- **Add Hystrix to “Customer” Microservice**

Protect the use of the “Customer” Docker container with Hystrix also. In order to do so change the class `CustomerClient` from the “Order” project. `CatalogClient` can serve as a template.

- **Change Hystrix Configuration**

Change the configuration of Hystrix for the “Catalog” microservice. There are several configuration options.¹⁴ Listing 13.8 (CatalogClient from the “Order” Project) shows the use of the Hystrix annotations. Other time intervals for opening and closing of the circuit breakers are, for instance, a possible change.

14. <https://github.com/Netflix/Hystrix/wiki/Configuration>

13.11 Load Balancing

For Load Balancing the example application uses Ribbon.¹⁵ Many load balancers are proxy based. In this model the clients send all calls to a Load Balancer. The Load Balancer runs as a distinct server and forwards the request to a web server—often depending on the current load of the web servers.

Ribbon implements a different model called client-side load balancing: The client has all the information to communicate with the right server. The client calls the server directly and distributes the load by itself to different servers. In the architecture there is no bottleneck as there is no central server all calls would have to pass. In conjunction with data replication by Eureka, Ribbon is quite resilient: As long as the client runs, it can send requests. The failure of a proxy load balancer would stop all calls to the server.

15. <https://github.com/Netflix/ribbon/wiki>

Dynamic scaling is very simple within this system: A new instance is started, enlists itself at Eureka, and then the Ribbon Clients redirect load to the instance.

As already discussed in the section dealing with Eureka (section 13.8), data can be inconsistent over the different servers. Because data is not up to date, servers can be contacted, which really should be left out by the Load Balancing.

Ribbon with Spring Cloud

Spring Cloud simplifies the use of Ribbon. The application has to be annotated with `@RibbonClient`. While doing so, a name for the application can be defined. In addition, the application needs to have a dependency on `spring-cloud-starter-ribbon`. In that case an instance of a microservice can be accessed using code like that in Listing 13.10. For that purpose, the code uses the Eureka name of the microservice.

Listing 13.10 Determining a Server with Ribbon Load Balancing

```
ServiceInstance instance
= loadBalancer.choose("CATALOG");
String url = "http://" + instance.getHost() + ":" +
instance.getPort() + "/catalog/";
```

The use can also be transparent to a large extent. To illustrate this Listing 13.11 shows the use of `RestTemplate` with Ribbon. This is a Spring class, which can be used to call REST services. In the Listing the `RestTemplate` of Spring is injected into the object as it is annotated with `@Autowired`. The call in `callMicroservice()` looks like it is contacting a server called “stores.” In reality this name is used to search a server at Eureka, and the REST call is sent to this server. This is done via Ribbon so that the load is also distributed across the available servers.

Listing 13.11 Using Ribbon with `RestTemplate`

```
@RibbonClient(name = "ribbonApp")
... // Left out other Spring Cloud / Boot Annotations
public class RibbonApp {

    @Autowired
    private RestTemplate restTemplate;

    public void callMicroservice() {
        Store store = restTemplate.
            getForObject("http://stores/store/1", Store.class);
    }
}
```

Try and Experiment

- **Load Balance to an Additional Service Instance**

The “Order” microservice distributes the load onto several instances of the “Customer and Catalog” microservice—if several instances exist. Without further measures, only a single instance is started. The “Order” microservice shows in the log which “Catalog” or “Customer” microservice it contacts. Initiate an order and observe which services are contacted.

Afterwards start an additional “Catalog” microservice. You can do that using the command: `docker run -v /microservice-demo:/microservice-demo --link eureka:eureka catalog-app` in Vagrant. For Docker Compose `docker-compose scale catalog=2` should be enough. Verify whether the container is running and observe the log output.

For reference: “Try and Experiment” in section 13.4 shows the main commands for using Docker. Section 13.7 shows how to use Docker Compose.

- **Create Data**

Create a new dataset with a new item. Is the item always displayed in the selection of items? Hint: The database runs within the process of the microservice—that is, each microservice instance possesses its own database.

13.12 Integrating Other Technologies

Spring Cloud and the entire Netflix Stack are based on Java. Thus, it seems impossible for other programming languages and platforms to use this infrastructure. However, there is a solution: the application can be supplied with a sidecar. The sidecar is written in Java and uses Java libraries to integrate into a Netflix-based infrastructure. The sidecar, for instance, takes care of registration and finding other microservices in Eureka. Netflix itself offers for this purpose the Prana project.¹⁶ The Spring Cloud solution is explained in the documentation.¹⁷ The sidecar runs in a distinct process and serves as an interface between the microservice itself and the microservice infra-

16. <http://github.com/Netflix/Prana/>

17. http://cloud.spring.io/spring-cloud-static/Brixton.SR5/#_polyglot_support_with_sidecar

structure. In this manner other programming languages and platforms can be easily integrated into a Netflix or Spring Cloud environment.

13.13 Tests

The example application contains test applications for the developers of microservices. These do not need a microservice infrastructure or additional microservices—in contrast to the production system. This enables developers to run each microservice without a complex infrastructure.

The class `orderTestApp` in the “Order” project contains such a test application. The applications contain their own configuration file `application-test.properties` with specific settings within the directory `src/test/resources`. The settings prevent that the applications register with the Service Discovery Eureka. Besides, they contain different URLs for the dependent microservices. This configuration is automatically used by the test application as it uses a Spring profile called “test.” All JUnit tests use these settings as well so that they can run without dependent services.

Stubs

The URLs for the dependent microservices in the test application and the JUnit tests point to stubs. These are simplified microservices, which only offer a part of the functionalities. They run within the same Java process as the real microservices or JUnit tests. Therefore, only a single Java process has to be started for the development of a microservice, analogous to the usual way of developing with Java. The stubs can be implemented differently—for instance, using a different programming language or even a web server, which returns certain static documents representing the test data (see section 10.6). Such approaches might be better suited for real-life applications.

Stubs facilitate development. If each developer needs to use a complete environment including all microservices during development, a tremendous amount of hardware resources and a lot of effort to keep the environment continuously up to date would be necessary. The stubs circumvent this problem as no dependent microservices are needed during development. Due to the stubs the effort to start a microservice is hardly bigger than the one for a regular Java application.

In a real project the teams can implement stubs together with the real microservices. The “Customer” team can implement a stub for the “Customer” microservice in addition to the real service, which is used by the other microservices for

development. This ensures that the stub largely resembles the microservice and is updated if the original service is changed. The stub can be taken care of in a different Maven projects, which can be used by the other teams.

Consumer-Driven Contract Test

It has to be ensured that the stubs behave like the microservices they simulate. In addition, a microservice has to define the expectations regarding the interface of a different microservice. This is achieved by consumer-driven contract tests (see section 10.7). These are written by the team that uses the microservices. In the example this is the team that is responsible for the “Order” microservice. In the “Order” microservice the consumer-driven contract tests are found in the classes `CatalogConsumerDrivenContractTest` and `CustomerConsumerDrivenContractTest`. They run there to test the stubs of the “Customer and Catalog” microservice for correctness.

Even more important than the correct functioning of the stubs is the correct functioning of the microservices themselves. For that reason, the consumer-driven contract tests are also contained in the “Customer and Catalog” project. There they run against the implemented microservices. This ensures that the stubs as well as the real microservices are in line with this specification. In case the interface is supposed to be changed, these tests can be used to confirm that the change does not break the calling microservice. It is up to the used microservices—“Customer and Catalog” in the example—to comply with these tests. In this manner the requirements of the “Order” microservice in regard to the “Customer and Catalog” microservice can be formally defined and tested. The consumer-driven contract tests serve in the end as formal definition of the agreed interface.

In the example application the consumer-driven contract tests are part of the “Customer and Catalog” projects in order to verify that the interface is correctly implemented. Besides they are part of the “Order” project for verifying the correct functioning of the stubs. In a real project copying the tests should be prevented. The consumer-driven contract tests can be located in one project together with the tested microservices. Then all teams need to have access to the microservice projects to be able to alter the tests. Alternatively, they are located within the projects of the different teams that are using the microservice. In that case the tested microservice has to collect the tests from the other projects and execute them.

In a real project it is not really necessary to protect stubs by consumer-driven contract tests, especially as it is the purpose of the stubs to offer an easier implementation than the real microservices. Thus the functionalities will be different and conflict with consumer-driven contract tests.

Try and Experiment

- Insert a field into “Catalog” or “Customer” data. Is the system still working? Why?
- Delete a field in the implementation of the server for “Catalog” or “Customer.” Where is the problem noticed? Why?
- Replace the home-grown stubs with stubs, that use a tool from Section 10.6.
- Replace the consumer-driven contract tests with tests that use a tool from Section 10.7.

13.14 Experiences with JVM-Based Microservices in the Amazon Cloud (Sascha Möllering)

By Sascha Möllering, zanox AG

During the last months zanox has implemented a lightweight microservices architecture in Amazon Web Services (AWS), which runs in several AWS regions. Regions divide the Amazon Cloud into sections like US-East or EU-West, which each have their own data centers. They work completely independently of each other and do not exchange any data directly. Different AWS regions are used because latency is very important for this type of application and is minimized by latency-based routing. In addition, it was a fundamental aim to design the architecture in an event-driven manner. Furthermore, the individual services were intended not to communicate directly but rather to be separated by message queues respectively bus systems. An Apache Kafka cluster as message bus in the zanox data center serves as central point of synchronization for the different regions. Each service is implemented as a stateless application. The state is stored in external systems like the bus systems, Amazon ElastiCache (based on the NoSQL database Redis), the data stream processing technology Amazon Kinesis, and the NoSQL database Amazon DynamoDB. The JVM serves as basis for the implementation of the individual services. We chose Vert.x and the embedded web server Jetty as frameworks. We developed all applications as self-contained services so that a Fat JAR, which can easily be started via `java -jar`, is generated at the end of the build process.

There is no need to install any additional components or an application server. Vert.x serves as basis framework for the HTTP part of the architecture. Within the application work is performed almost completely asynchronously to achieve high

performance. For the remaining components we use Jetty as framework: These act either as Kafka/Kinesis consumer or update the Redis cache for the HTTP layer. All called applications are delivered in Docker containers. This enables the use of a uniform deployment mechanism independent of the utilized technology. To be able to deliver the services independently in the different regions, an individual Docker Registry storing the Docker images in a S3 bucket was implemented in each region. S3 is a service that enables the storage of large file on Amazon server.

If you intend to use Cloud Services, you have to address the question of whether you want to use the managed services of a cloud provider or develop and run the infrastructure yourself. zanox decided to use the managed services of a cloud provider because building and administrating proprietary infrastructure modules does not provide any business value. The EC2 computers of the Amazon portfolio are pure infrastructure. IAM, on the other hand, offers comprehensive security mechanisms. In the deployed services the AWS Java SDK is used, which enables it, in combination with IAM roles for EC2,¹⁸ to generate applications that are able to access the managed services of AWS without using explicit credentials. During initial bootstrapping an IAM role containing the necessary permissions is assigned to an EC2 instance. Via the Metadata Service¹⁹ the AWS SDK is given the necessary credentials. This enables the application to access the managed services defined in the role. Thus, an application can be that sends metrics to the monitoring system Amazon Cloud Watch and events to the data streaming processing solution Amazon Kinesis without having to roll out explicit credentials together with the application.

All applications are equipped with REST interfaces for heartbeats and health checks so that the application itself as well as the infrastructure necessary for the availability of the application can be monitored at all times: Each application uses health checks to monitor the infrastructure components it uses. Application scaling is implemented via Elastic Load Balancing (ELB) and AutoScaling²⁰ to be able to achieve a fine-grained application depending on the concrete load. AutoScaling starts additional EC2 instances if needed. ELB distributes the load between the instances. The AWS ELB service is not only suitable for web applications working with HTTP protocols but for all types of applications. A health check can also be implemented based on a TCP protocol without HTTP. This is even simpler than an HTTP healthcheck.

Still the developer team decided to implement the ELB healthchecks via HTTP for all services to achieve the goal that they all behave exactly the same, independent of the implemented logic, the used frameworks, and the language. It is also quite

18. <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/iam-roles-for-amazon-ec2.html>

19. <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-metadata.html>

20. <https://docs.aws.amazon.com/AutoScaling/latest/DeveloperGuide/as-add-elb-healthcheck.html>

possible that in the future applications that do not run on JVM and, for instance, use Go or Python as programming languages, are deployed in AWS.

For the ELB healthcheck zanox uses the application heartbeat URL. As a result, traffic is only directed to the application respectively potentially necessary infrastructure scaling operations are only performed once the EC2 instance with the application has properly been started and the heartbeat was successfully monitored.

For application monitoring Amazon CloudWatch is a good choice as CloudWatch alarms can be used to define scaling events for the AutoScaling Policies, that is, the infrastructure scales automatically based on metrics. For this purpose, EC2 basis metrics like CPU can be used, for instance. Alternatively, it is possible to send your own metrics to CloudWatch. For this purpose, this project uses a fork of the project jmxtrans-agent,²¹ which uses the CloudWatch API to send JMX metrics to the monitoring system. JMX (Java Management Extension) is the standard for monitoring and metrics in the Java world. Besides metrics are sent from within the application (i.e., from within the business logic) using the library Coda Hale Metrics²² and a module for the CloudWatch integration by Blacklocus.²³

A slightly different approach is chosen for the logging: In a cloud environment it is never possible to rule out that a server instance is abruptly terminated. This often causes the sudden loss of data that are stored on the server. Log files are an example for that. For this reason, a logstash-forwarder²⁴ runs in parallel to the core application on the server for sending the log entries to our ELK-Service running in our own data center. This stack consists of Elasticsearch for storage, Logstash for parsing the log data, and Kibana for UI-based analysis. ELK is an acronym for Elasticsearch, Logstash, und Kibana. In addition, a UUID is calculated for each request respectively each event in our HTTP layer so that log entries can still be assigned to events after EC2 instances have ceased to exist.

Conclusion

The pattern of microservices architectures fits well to the dynamic approach of Amazon Cloud if the architecture is well designed and implemented. The clear advantage over implementing in your own data center is the infrastructure flexibility. This makes it possible to implement a nearly endlessly scalable architecture, which is, in addition, very cost efficient.

21. <https://github.com/SaschaMoellering/jmxtrans-agent>

22. <https://dropwizard.github.io/metrics/>

23. <https://github.com/blacklocus/metrics-cloudwatch>

24. <https://github.com/elastic/logstash-forwarder>

13.15 Conclusion

The technologies used in the example provide a very good foundation for implementing a microservices architecture with Java. Essentially, the example is based on the Netflix Stack, which has demonstrated its efficacy for years already in one of the largest websites.

The example demonstrates the interplay of different technologies for Service Discovery, Load Balancing, and resilience—as well as an approach for testing microservices and for their execution in Docker containers. The example is not meant to be directly useable in a production context but is first of all designed to be very easy to set up and get running. This entails a number of compromises. However, the example serves very well as the foundation for further experiments and the testing of ideas.

In addition, the example demonstrates a Docker-based application deployment, which is a good foundation for microservices.

Essential Points

- Spring, Spring Boot, Spring Cloud, and the Netflix Stack offer a well-integrated stack for Java-based microservices. These technologies solve all typical challenges posed during the development of microservices.
- Docker-based deployment is easy to implement, and in conjunction with Docker Machine and Docker Compose, can be used for deployment in the Cloud, too.
- The example application shows how to test microservices using consumer-driven contract tests and stubs without special tools. However, for real-life projects tools might be more useful.

Try and Experiment

Add Log Analysis

The log analysis of all log files is important for running a microservice system. At <https://github.com/ewolff/user-registration-V2> an example project is provided. The subdirectory **log-analysis** contains a setup for an ELK (Elasticsearch, Logstash und Kibana) stack-based log analysis. Use this approach to add a log analysis to the microservice example.

(continued)

Add Monitoring

In addition, the example project from the continuous delivery book contains graphite an installation of Graphite for monitoring in the subdirectory. Adapt this installation for the microservice example.

Rewrite a Service

Rewrite one of the services in a different programming language. Use the consumer-driven contract tests (see sections 13.13 and 10.7) to protect the implementation. Make use of a sidecar for the integration into the technology stack (see section 13.12).

Username: Iowa State University Library **Book:** Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

4.1 Business Case

In 2006, a large telecommunications company wanted to expand its Internet Protocol (IP) network to support “carrier-class services”, and more specifically high-quality voice over IP (VOIP) systems. One important aspect to achieve this goal was synchronization of the VOIP servers and other equipment. Poor synchronization results in low quality of service (QoS), degraded performance, and unhappy customers. To achieve the required level of synchronization, the company wanted to deploy a network of time servers that support the Network Time Protocol (NTP). Time servers are formed into groups that typically correspond to geographical regions. Within these regions, time servers are organized hierarchically in levels or *strata*, where time servers placed in the upper level of the hierarchy (stratum 1) are equipped with hardware (e.g., Cesium Oscillator, GPS signal) that provides precise time. Time servers that are lower in the hierarchy use NTP to request time from servers in the upper levels or from their peers.

Many pieces of equipment depend on the time provided by time servers in the network, so one priority for the company was to correct any problems that occur on the time servers. Such problems may require dispatching a technician to perform physical maintenance on the time servers, such as rebooting. Another priority for the company was to collect data from the time servers to monitor the performance of the synchronization framework.

In the initial deployment plans, the company wanted to field 100 time servers of a particular model. Besides NTP, time servers support the Simple Network Management Protocol (SNMP), which provides three basic operations:

- `set()` operations: change configuration variables (e.g., connected peers)
- `get()` operations: retrieve configuration variables or performance data
- `trap()` operations: notifications of exceptional events such as the loss or restoration of the GPS signal or changes in the time reference

To achieve the company's goals, a management system for the time servers needed to be developed. This system needed to conform to the FCAPS model, which is a standard model for network management. The letters in the acronym stand for:

- *Fault management.* The goal of fault management is to recognize, isolate, correct, and log faults that occur in the network. In this case, these faults correspond to traps generated by time servers or other problems such as loss of communication between the management system and the time servers.
- *Configuration management.* This includes gathering and storing configurations from network devices, thereby simplifying the configuration of devices and tracking changes that are made to device configurations. In this system, besides changing individual configuration variables, it is necessary to be able to deploy a specific configuration to several time servers.
- *Accounting.* The goal here is to gather device information. In this context, this includes tracking device hardware and firmware versions, hardware equipment, and other components of the system.
- *Performance management.* This category focuses on determining the efficiency of the current network. By collecting and analyzing performance data, the network health can be monitored. In this case, delay, offset, and jitter measures are collected from the time servers.
- *Security management.* This is the process of controlling access to assets in the network. In this case, there are two important types of users: technicians and administrators. Technicians can visualize trap information and configurations but cannot make changes; administrators are technicians who can visualize the same information but can also make changes to configurations, including adding and removing time servers from the network.

Once the initial network was deployed, the company planned to extend it by adding time servers from newer models that might potentially support management protocols other than SNMP.

The remainder of this chapter describes a design for this system, created using ADD 3.0.

Username: Iowa State University Library **Book:** Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

4.2 System Requirements

Requirement elicitation activities had been previously performed, and the following is a summary of the most relevant requirements collected.

4.2.1 Use Case Model

The use case model in [Figure 4.1](#) presents the most relevant use cases that support the FCAPS model in the system. Other use cases are not shown.

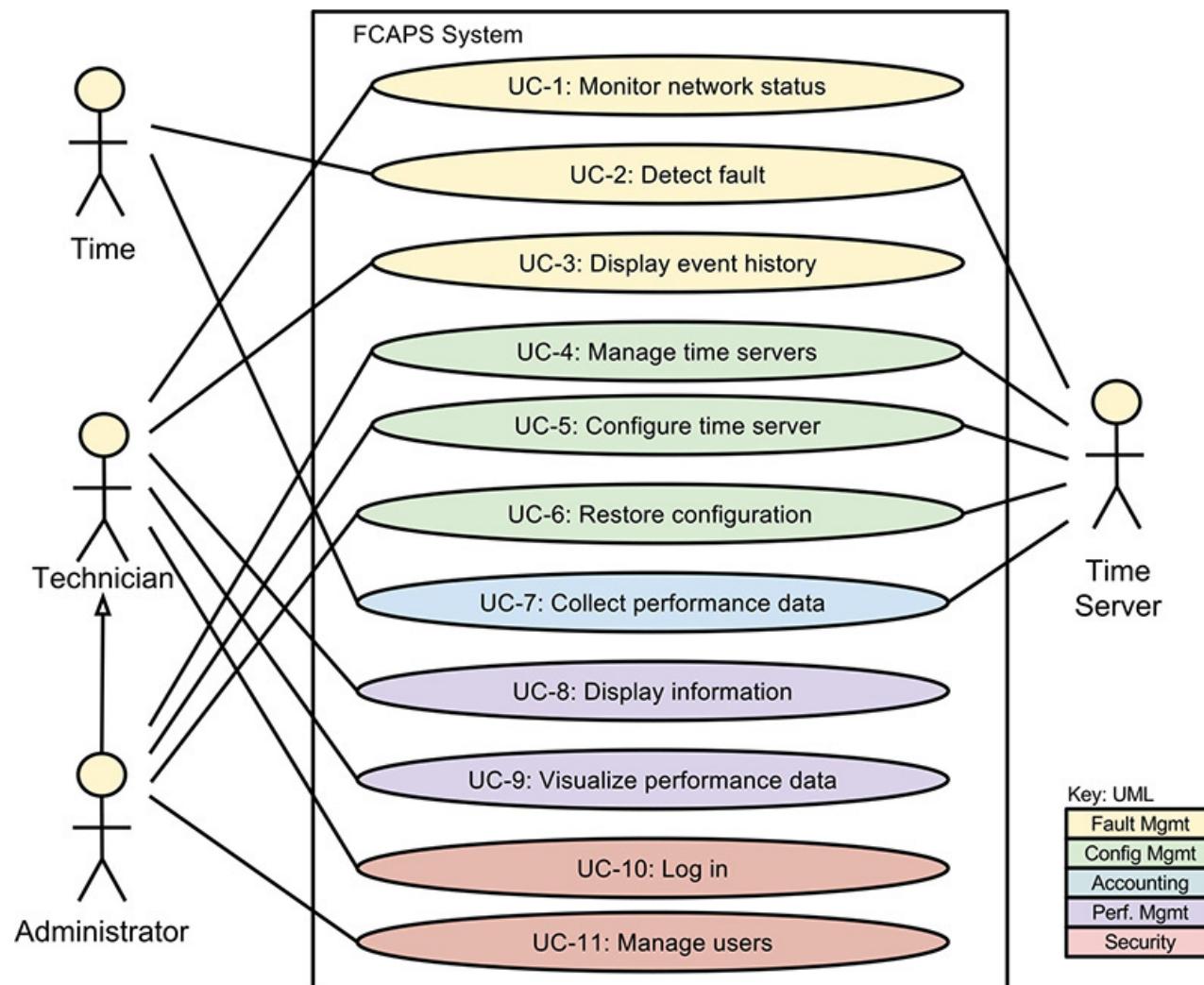


FIGURE 4.1 Use case model for the FCAPS system

Each of these use cases is described in the following table:

Use Case	Description
UC-1: Monitor network status	A user monitors the time servers in a hierarchical representation of the whole network. Problematic devices are highlighted, along with the logical regions where they are grouped. The user can expand and collapse the network representation. This representation is updated continuously as faults are detected or repaired.
UC-2: Detect fault	Periodically the management system contacts the time servers to see if they are “alive”. If a time server does not respond, or if a trap that signals a problem or a return to a normal state of operation is received, the event is stored and the network representation observed by the users is updated accordingly.
UC-3: Display event history	Stored events associated with a particular time server or group of time servers are displayed. These can be filtered by various criteria such as type or severity.
UC-4: Manage time servers	The administrator adds a time server to, or removes a time server from, the network.
UC-5: Configure time server	An administrator changes configuration parameters associated with a particular time server. The parameters are sent to the device and are also stored locally.
UC-6: Restore configuration	A locally stored configuration is sent to one or more time servers.
UC-7: Collect performance data	Network performance data (delay, offset, and jitter) is collected periodically from the time servers.
UC-8: Display information	The user displays stored information about the time server—configuration values and other parameters such as the server name.
UC-9: Visualize performance data	The user displays network performance measures (delay, offset, jitter) in a graphical way to view and analyze network performance.
UC-10: Log in	A user logs into the system through a login/password screen. Upon successful login, the user is presented with different options according to their role.
U-11: Manage users	The administrator adds or removes a user or modifies user permissions.

4.2.2 Quality Attribute Scenarios

In addition to these use cases, a number of quality attribute scenarios were elicited and documented. The six most relevant ones are presented in the following table. For each scenario, we also identify the use case that it is associated with.

ID	Quality Attribute	Scenario	Associated Use Case
QA-1	Performance	Several time servers send traps to the management system at peak load; 100% of the traps are successfully processed and stored.	UC-2
QA-2	Modifiability	A new time server management protocol is introduced to the system as part of an update. The protocol is added successfully without any changes to the core components of the system.	UC-5
QA-3	Availability	A failure occurs in the management system during normal operation. The management system resumes operation in less than 30 seconds.	All
QA-4	Performance	The management system collects performance data from a time server during peak load. The management system collects all performance data within 5 minutes, while processing all user requests, to ensure no loss of data due to CON-5.	UC-7
QA-5	Performance, usability	A user displays the event history of a particular time server during normal operation. The list of events from the last 24 hours is displayed within 1 second.	UC-3
QA-6	Security	A user performs a change in the system during normal operation. It is possible to know who performed the operation and when it was performed 100% of the time.	All

4.2.3 Constraints

Finally, a set of constraints on the system and its implementation were collected. These are presented in the following table.

ID	Constraint
CON-1	A minimum of 50 simultaneous users must be supported.
CON-2	The system must be accessed through a web browser (Chrome V3.0+, Firefox V4+, IE8+) in different platforms: Windows, OSX, and Linux.
CON-3	An existing relational database server must be used. This server cannot be used for other purposes than hosting the database.
CON-4	The network connection to user workstations can have low bandwidth but is generally reliable.
CON-5	Performance data needs to be collected in intervals of no more than 5 minutes, as higher intervals result in time servers discarding data.
CON-6	Events from the last 30 days must be stored.

4.2.4 Architectural Concerns

Given that this is greenfield development, only a few general architectural concerns are identified initially, as shown in the following table.

ID	Concern
CRN-1	Establishing an overall initial system structure.
CRN-2	Leverage the team's knowledge about Java technologies, including Spring, JSF, Swing, Hibernate, Java Web Start and JMS frameworks, and the Java language.
CRN-3	Allocate work to members of the development team.

Given these sets of inputs, we are now ready to proceed to describe the design process, as described in [Section 3.2](#). In this chapter, we present only the final results of the requirements collection process. The job of collecting these requirements is nontrivial, but is beyond the scope of this chapter.

Username: Iowa State University Library **Book:** Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

5.1 Business Case

This case study involves an Internet company that provides popular content and online services to millions of web users. Besides providing information externally, the company collects and analyzes massive logs of data that are generated from its infrastructure (e.g., application and server logs, system metrics). Such an approach of dealing with computer-generated log messages is also called *log management* (http://en.wikipedia.org/wiki/Log_management_and_intelligence).

Because of very fast infrastructure growth, the company's IT department realizes that the existing in-house systems can no longer process the required log data volume and velocity. Moreover, requests for a new system are coming from other company stakeholders, including product managers and data scientists, who would like to leverage the various kinds of data that can be collected from multiple data sources, not just logs.

The *marketecture* diagram (informal depiction of the system's structure) shown in [Figure 5.1](#) represents the desired solution from a functional perspective for three major groups of users.

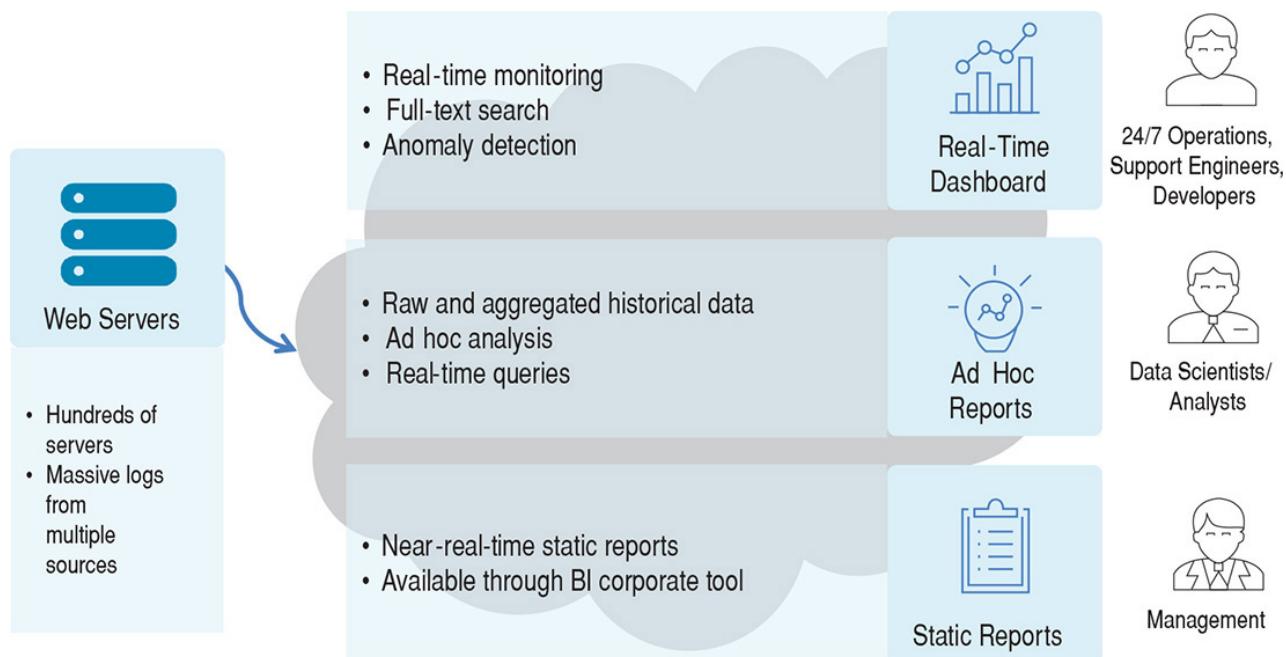


FIGURE 5.1 Marketecture diagram for the Big Data system

Username: Iowa State University Library **Book:** Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

5.2 System Requirements

Requirement elicitation activities have been previously performed. The most important requirements collected are summarized here. They comprise a set of primary use cases, a set of quality attribute scenarios, a set of constraints, and a set of architectural concerns.

5.2.1 Use Case Model

The primary use cases for the system are described in the following table.

Use Case	Description
UC-1: Monitor online services	On-duty operations staff can monitor the current state of services and IT infrastructure (such as web server load, user activities, and errors) through a real-time operational dashboard, which enables them to quickly react to issues.
UC-2: Troubleshoot online service issues	Operations, support engineers, and developers can do troubleshooting and root-cause analysis on the latest collected logs by searching log patterns and filtering log messages.
UC-3: Provide management reports	Corporate users, such as IT and product managers, can see historical information through predefined (static) reports in a corporate BI (business intelligence) tool, such as those showing system load over time, product usage, service level agreement (SLA) violations, and quality of releases.
UC-4: Support data analytics	Data scientists and analysts can do ad hoc data analysis through SQL-like queries to find specific data patterns and correlations to improve infrastructure capacity planning and customer satisfaction.
UC-5: Anomaly detection	The operations team should be notified 24/7 about any unusual behavior of the system. To support this notification plan, the system shall implement real-time anomaly detection and alerting (future requirement).
UC-6: Provide security reports	Security analysts should be provided with the ability to investigate potential security and compliance issues by exploring audit log entries that include destination and source addresses, a time stamp, and user login information (future requirement).

5.2.2 Quality Attribute Scenarios

The most relevant quality attribute (raw) scenarios are presented in the following table. For each scenario, we also identify the use case that it is associated with.

ID	Quality Attribute	Scenario	Associated Use Case
QA-1	Performance	The system shall collect up to 15,000 events/second from approximately 300 web servers.	UC-1, 2, 5
QA-2	Performance	The system shall automatically refresh the real-time monitoring dashboard for on-duty operations staff with < 1 min latency.	UC-1
QA-3	Performance	The system shall provide real-time search queries for emergency troubleshooting with < 10 seconds query execution time, for the last 2 weeks of data.	UC-2
QA-4	Performance	The system shall provide near-real-time static reports with per-minute aggregation for business users with < 15 min latency, < 5 seconds report load.	UC-3, 6
QA-5	Performance	The system shall provide ad hoc (i.e., non-predefined) SQL-like human-time queries for raw and aggregated historical data, with < 2 minutes query execution time. Results should be available for query in < 1 hour.	UC-4
QA-6	Scalability	The system shall store raw data for the last 2 weeks available for emergency troubleshooting (via full-text search through logs).	UC-2
QA-7	Scalability	The system shall store raw data for the last 60 days (approximately 1 TB of raw data per day, approximately 60 TB in total).	UC-4
QA-8	Scalability	The system shall store per-minute aggregated data for 1 year (approximately 40 TB) and per-hour aggregated data for 10 years (approximately 50 TB).	UC-3, 4, 6
QA-9	Extensibility	The system shall support adding new data sources by just updating a configuration, with no interruption of ongoing data collection.	UC-1, 2, 5
QA-10	Availability	The system shall continue operating with no downtime if any single node or component fails.	All use cases
QA-11	Deployability	The system deployment procedure shall be fully automated and support a number of environments: development, test, and production.	All use cases

5.2.3 Constraints

The constraints associated with the system are presented in the following table.

ID	Constraint
CON-1	The system shall be composed primarily of open source technologies (for cost reasons). For those components where the value/cost of using proprietary technology is much higher, proprietary technology may be used.
CON-2	The system shall use the corporate BI tool with a SQL interface for static reports (e.g., MicroStrategy, QlikView, Tableau).
CON-3	The system shall support two specific deployment environments: private cloud (with VMware vSphere Hypervisor) and public cloud (Amazon Web Services). Architecture and technology decisions should be made to keep deployment vendor as agnostic as possible.

5.2.4 Architectural Concerns

The initial architectural concerns that are considered are shown in the following table.

ID	Concern
CRN-1	Establishing an initial overall structure as this is a greenfield system.
CRN-2	Leverage the team's knowledge of the Apache Big Data ecosystem.

Username: Iowa State University Library **Book:** Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

6.1 Business Case

In 2010, the government of a Latin American country issued a regulation that required banking institutions to digitally sign bank statements. To comply with the regulation, "ACME Bank" decided to commission the development of a software system, which we will call BankStat, whose main purpose was the generation of digitally signed bank statements.

[Figure 6.1](#) presents a context diagram that illustrates how the BankStat system works. At its core, the system executes a batch process, which retrieves raw bank statement information from a data source (an external database) and then performs a series of validations on this data to generate the bank statements and prepare them for digital signature by an external provider. The statements are sent to the provider, which returns the signed bank statements. These statements are then stored by BankStat for further processing, including sending the statements to customers. This batch process is triggered automatically once a month and, during its execution, approximately 2 million bank statements are processed.

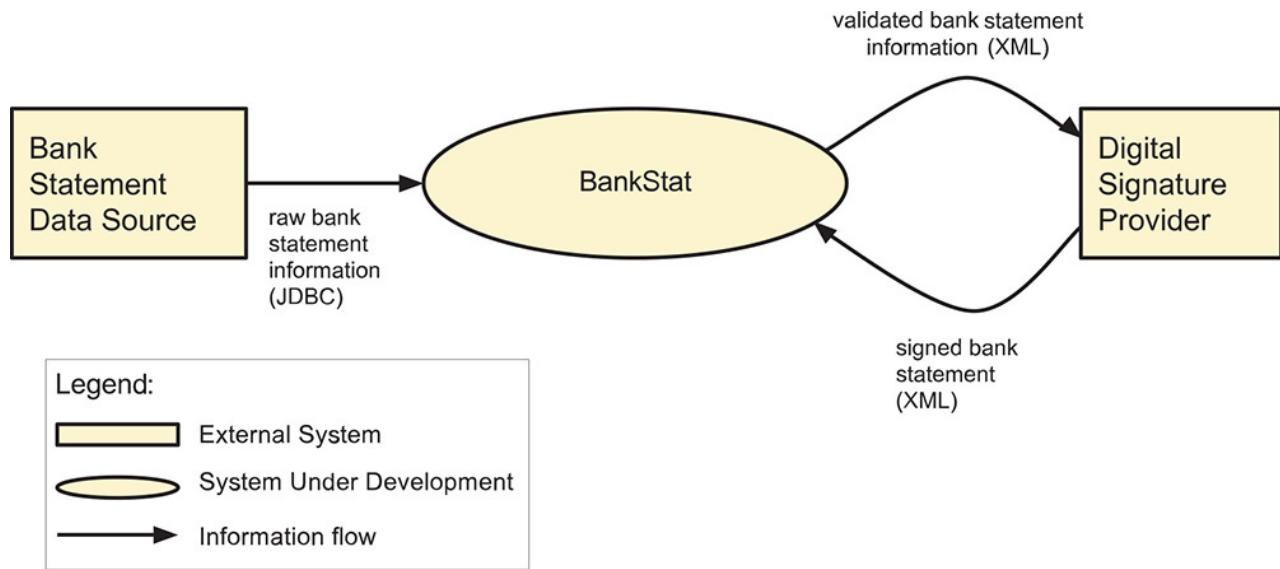


FIGURE 6.1 Context diagram for the BankStat system

The following quality attributes scenarios are primary for this system:

- **Reliability:** Under normal operating conditions, the batch process is executed in its entirety 100% of the time.
- **Performance:** Under normal operating conditions, when the batch process starts, 2 million bank statements are read, processed, and sent to the signing provider in at most one hour.
- **Availability:** During normal processing, a failure may occur when reading information from the data source or when sending information for digital signature. A notification is then sent to the administrator, who manually restarts the process. When it is restarted, only the information that had not already been processed is treated.

Due to time constraints imposed by the government, only the core batch process for the system was developed and put into production. This initial release, however, did not provide a friendly interface with the system, which is necessary to monitor the state of the bank statement processing, to request the reprocessing of incorrect statements, and to generate reports. In the first release, the process could only be started or stopped manually from a console. For a second release of the system, the ACME Bank requested an extension of the BankStat system to better address these shortcomings.

The following subsections present the drivers for this second release of the system.

6.1.1 Use Case Model

[Figure 6.2](#) presents the use case model for the second release of BankStat.

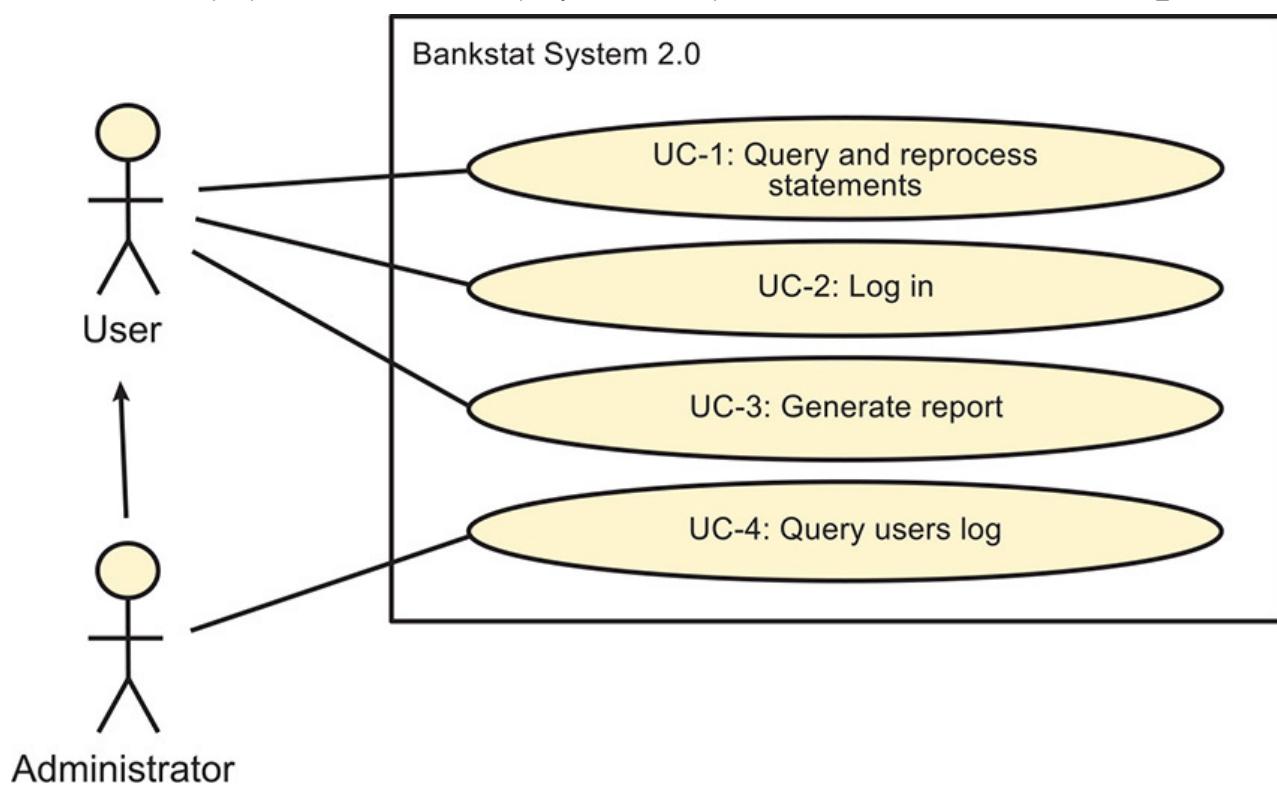


FIGURE 6.2 Use cases for the BankStat system (Key: UML)

These use cases are described in more detail here:

Use Case	Description
UC-1: Query and reprocess statements	The user manually requests the reprocessing of a number of statements. The user specifies criteria to query and select the statements that must be reprocessed. The user can, for example, select a period of interest or status of the statements that he is interested in (e.g., processed, signed, non-signed).
UC-2: Log in	The user logs in to the system.
UC-3: Generate report	The user generates reports regarding the process.
UC-4: Query users log	The administrator queries user logs to display the activities of a particular user or groups of users. Information can be filtered using criteria such as dates or types of operations.

6.1.2 Quality Attribute Scenarios

The following table presents the new quality attribute scenario that is considered for this extension of the system.

ID	Quality Attribute	Scenario	Associated Use Case
QA-1	Security	A user performs any operation on the system, at any moment, and 100% of the operations performed by the user are recorded by the system in the operations log.	UC-4

6.1.3 Constraints

The following table presents the constraints that are considered for this extension of the system.

ID	Constraint
CON-1	The user's accounts and permissions are handled by an existing user directory server that is used by various applications in the bank.
CON-2	Communication with the data source must be realized using JDBC.
CON-3	Communication with the digital signature provider system is performed using web services. These web services receive and return the information in an XML format that adheres to specifications established by the government.
CON-4	The system must be accessed from a web browser, although the access is available only from the bank's intranet.

6.1.4 Architectural Concerns

The following table presents the concerns that are initially considered for this extension of the system.

ID	Concern
CRN-1	The system shall be programmed using Java and Java-related technologies to leverage the expertise of the development team.
CRN-2	The introduction of new functionality must, as far as possible, avoid modifications to the existing batch processing core.

Username: Iowa State University Library **Book:** Designing Software Architectures: A Practical Approach. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

6.2 Existing Architectural Documentation

This section presents a simplified version of the system's views, which provide relevant information for the changes in the architecture.

6.2.1 Module View

The package diagram shown in [Figure 6.3](#) depicts the system layers and the modules that they contain.

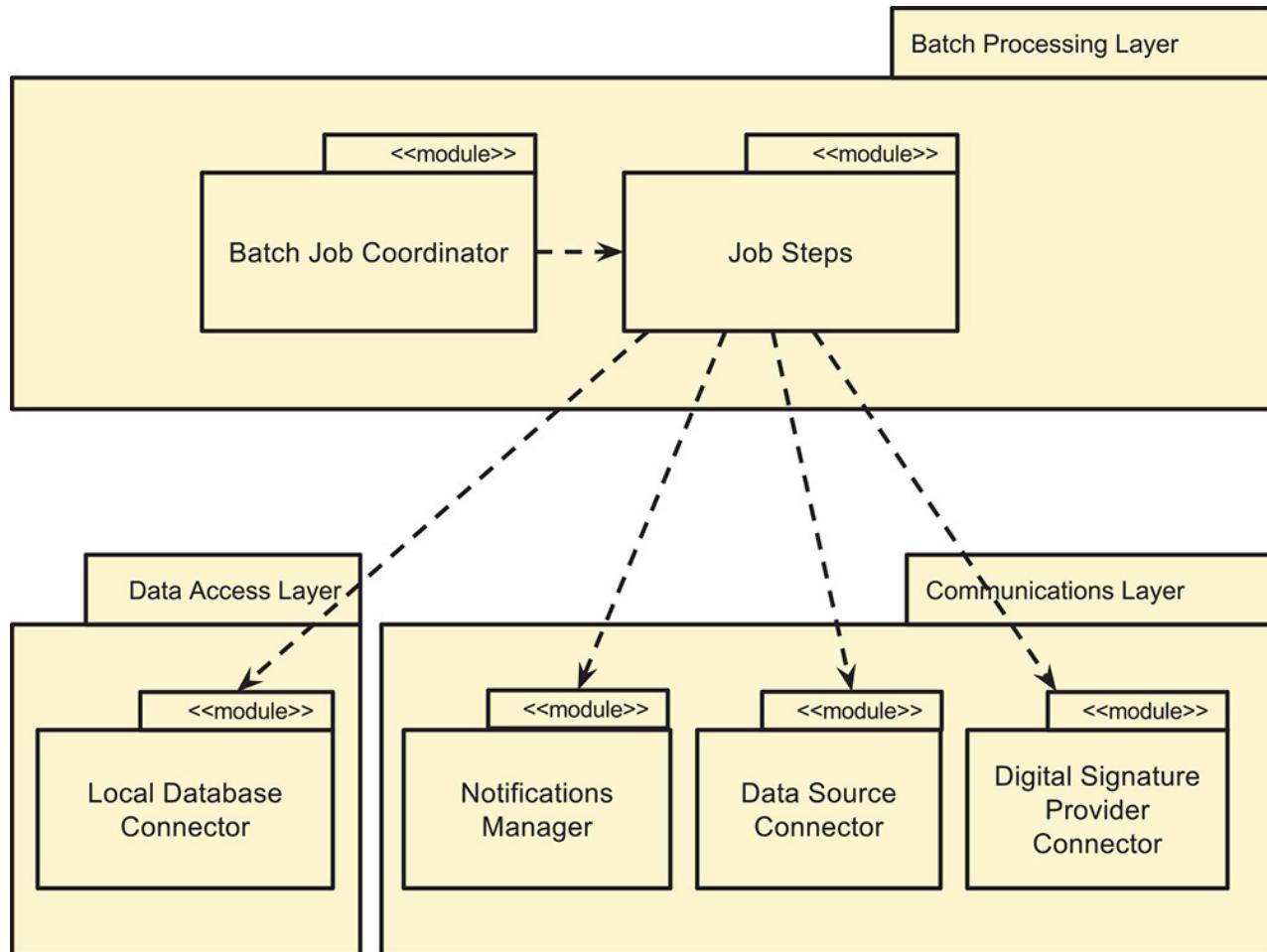


FIGURE 6.3 Existing modules and layers in the BankStat system (Key: UML)

The responsibilities of the elements depicted in the diagram are described in the following table.

Element	Responsibility
Batch Processing Layer	This layer contains modules that perform the batch process. These components are developed using the Spring Batch framework.
Data Access Layer	This layer contains modules that store and retrieve data from a local database, which is used by the modules in the Batch Processing Layer.
Communications Layer	This layer contains modules that support communication with the external digital signature provider and the bank statement data source.
Batch Job Coordinator	This module is responsible for coordinating the execution of the batch process, including launching the process and invoking the different steps associated with it.
Job Steps	This module contains the “steps” that are part of the batch job. These steps perform activities such as validating the information retrieved from the data source and generating the bank statements. Such steps generally read, process, and write data. Data is read from and written to the local database.
Local Database Connector	This module is responsible for accessing a local database used by the job steps to exchange information while performing the batch process. We refer to this database as “local” to differentiate it from the external data source; this database is used only locally (i.e., internally) by the application, even if it is deployed in a different node (see the next section).
Notifications Manager	This module manages logs and sends notifications in case of issues such as a communication failure with the external system.
Data Source Connector	This module is responsible for connecting with the external database that provides the raw bank statement information.
Digital Signature Provider Connector	This module is responsible for accessing the external system that performs the digital signing of the bank statements.

6.2.2 Allocation View

The deployment diagram shown in [Figure 6.4](#) presents an allocation view consisting of nodes and their relationships.

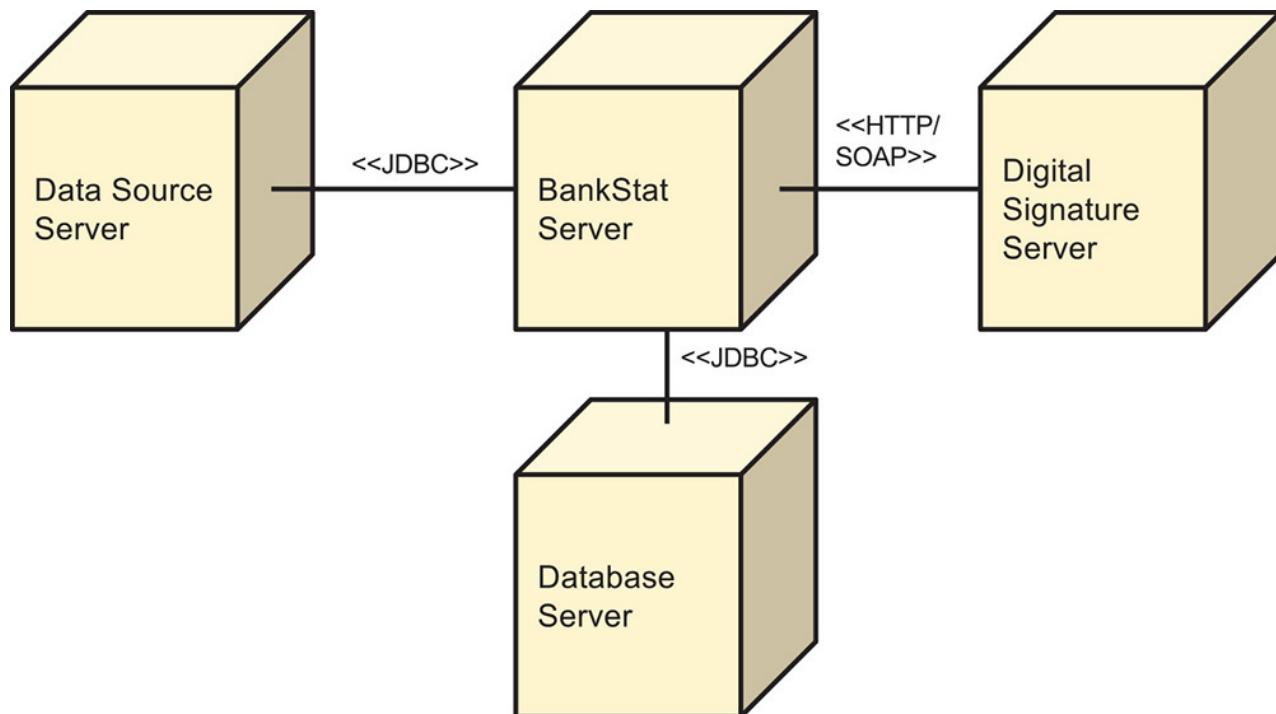


FIGURE 6.4 Existing deployment diagram for the BankStat system (Key: UML)

The responsibilities of the elements depicted in the diagram are described in the following table.

Element	Responsibility
Data Source Server	This server hosts a database that contains the raw data used to produce the bank statements.
BankStat Server	This server hosts the main batch process that is responsible for retrieving information from the Data Source Server, validating the information, and sending the information to the Digital Signature Server for signing.
Database Server	This server hosts a database that is used locally by the batch process in the BankStat Server to hold the state and information used in the execution of the batch process.
Digital Signature Server	This server, which is provided by an external entity, is responsible receiving, digitally signing, and returning the bank statements. The server exposes web services that receive and produce XML information.

Towards an Evidence-Based Understanding of Emergence of Architecture Through Continuous Refactoring in Agile Software Development

Lianping Chen¹², Muhammad Ali Babar³

¹Lero - The Irish Software Engineering Research Center, University of Limerick, Limerick

²Paddy Power PLC, Dublin, Republic of Ireland

³The University of Adelaide, Adelaide, Australia

lianping.chen@lero.ie, ali.babar@adelaide.edu.au

Abstract—The proponents of Agile software development approaches claim that software architecture emerges from continuous small refactoring, hence, there is not much value in spending upfront effort on architecture related issues. Based on a large-scale empirical study involving 102 practitioners who had worked with agile and architecture approaches, we have found that whether or not architecture emerges through continuous refactoring depends upon several contextual factors. Our study has identified 20 factors that have been categorized into four elements: project, team, practices, and organization. These empirically identified contextual factors are expected to help practitioners to make informed decisions about their architecture practices in agile software development.

Keywords—software architecture; agile software development component; empirical study

I. INTRODUCTION

Agile software development approaches have been widely adopted in the industry [1-2] for improving a software development team's ability to easily accommodate changes and delivery working software continuously and incrementally. Despite the increasing popularity of Agile approaches, there is huge scepticism about developing large scale critical software systems because of lack of attention paid to the issues related to software architecture by Agile followers [3]. This situation has caused intense debate among software development practitioners and researchers [2].

One of the most fundamental points of the debate is “whether or not a satisfactory architecture emerges from continuous small refactoring in agile software development” [4]? A satisfactory architecture is one that satisfies the architecturally significant requirements [5] of the software system. This issue has been so fiercely debated because many agile approaches encourage their followers not to worry about software architecture related issues upfront under the assumption that “a satisfactory architecture can emerge from continuous small refactoring.”

The proponents of architecture-centric approaches warn against this assumption and quote stories of disastrous consequences for not paying sufficient attention to architecture-centric issues through a system's lifecycle. Hence, there has been significant interest in and support for finding a middle ground by integrating sound architectural practices and principles in agile approaches [2-3, 6].

However, there is not much empirical evidence in support or against the key assumption of agile approaches “*a satisfactory architecture emerges from continuous refactoring*”; and what are the factors that facilitate or inhibit the emergence of a satisfactory architecture through continuous refactoring? We assert that it is important to systematically gather and rigorously analyse empirical data to provide an evidence-based understanding about the emergence of a satisfactory architecture through continuous refactoring.

We have conducted a large-scale empirical study that explored the observations and experiences of 102 experienced practitioners with regards to the claim that “*a satisfactory architecture emerges from continuous small refactoring*.” We carried out email-based semi-structured interviews and analyzed the collected data using descriptive statistics and techniques from Grounded Theory (GT) [7].

The results reveal that whether or not a satisfactory architecture can emerge depends upon contextual factors. Our study has identified 20 key factors that can support or inhibit the emergence of a satisfactory architecture through continuous refactoring in agile software development. We have classified those 20 factors into four elements that have been presented as a framework in this paper. Those four elements are: project, team, practices, and organization. This four elements framework and its 20 factors have enabled us to discuss the contexts in which a satisfactory architecture is likely to emerge through continuous small refactoring or otherwise. The findings are expected to help practitioners to make informed decisions about their architectural practices in Agile software development. We believe that this evidence-based understanding is important to address the commonly observed lack of attention paid to architecture-centric activities in Agile software development [8].

II. BACKGROUND AND MOTIVATION

Agile approaches (such as Extreme Programming (XP) [9], Crystal Clear [10] and Scrum [11]) are based on certain assumptions [12] and Agile manifesto¹. Since Agile movement was originally promoted as a way of moving away from formalism and bureaucratic centralized way of developing software, a large number of agile software development followers started playing down the role and importance of software architecture related processes, activities, artefacts, and

¹ <http://www.agilemanifesto.org>

roles in software development projects. For example, Thapparambil argues that “*no agile methods discuss Architecture in any length.*” Many practitioners have been led to believe that agile methods consider architectural design to be an optional matter. The description of the agile methods provide very little discussion on common architectural design activity types [13] such as architectural analysis, architectural synthesis and architectural evaluation, as well as the artifact types [13] associated with these activities.

Scrum, one of the most popular project management approaches, claims to support architectural practices through frequent oral communication. According to Scrum, the architecture of one-project application can always be refactored and repackaged to components for a higher level of reuse after the release to production to implement a walking skeleton, a small end-to-end functionality of the system, at the beginning of the project.

In the Incremental Re-architecture strategy of Crystal Clear [10], the team starts from the working architectural skeleton and incrementally evolves the architecture or infrastructure in stages and in parallel with the system functionality. Two architectural work products are almost certainly needed to be produced within a Crystal Clear project: system architecture and common domain model.

Agile proponents reason that “*Refactoring is the primary method to develop Architecture in the Agile world*”. The Incremental Design practice of XP [9] claims that architecture can emerge in daily design. The emergent design means that architecture relies on looking for potentially poor architectural solutions in the implemented code and making a better architecture when needed in design cycles of hours and days. According to this approach, the architecture emerges from the system rather than being imposed by some direct structuring force.

These positions of agile advocates create an impression that “*Refactoring is the primary method to develop architecture in the Agile world*” [14]. Abrahamsson and colleagues have also concluded that the proponents of Agile methods believe that architecture should emerge from continuous small refactoring [3]. The advocates of architecture-centric approaches consider software architecture design as a very important activity to be conducted in the early stage of software development life cycle [15-17]. Software architecture researchers and practitioners are very sceptical about the claim that software architecture can emerge from continuous, small refactoring without upfront thinking and appropriate design efforts. Their scepticism has been reinforced by several stories where large scale software projects failed to succeed as a result of failure in paying sufficient attention to software architecture related issues [8].

The widespread adoption of agile approaches and importance of software architecture in a large scale software development projects have been promoting the importance of paying more attention to architectural aspects in agile approaches. There is a growing interest in identifying the mechanics and prerequisites of integrating agile and architectural approaches [2]. An increased number of efforts are focusing on devising appropriate ways of incorporating architecture-centric activities into Agile software development

practices [3, 6]. For example, Ali Babar conducted a case study to identify and understand architectural practices and challenges of teams using Agile approaches [18]. Falessi and his colleagues surveyed the perceptions of software developers about the potential co-existence of Agile development and software architecture [19]. Nord and Tomayko [6] have proposed a few ways of integrating some of the SEI architecture-centric methods into the XP framework.

There are some other proposals for combining the strengths of the core elements of agile and architecture-centric approaches. For example, [20-21] combine the strengths of the core elements of the risk-driven, architecture-centric Rational Unified Process (RUP) and the XP [9] process. The combinations were enabled by the facts that RUP and XP share the cornerstone of the iterative, incremental and evolutionary development [22] and that most of the core elements of RUP and XP are complementary.

None of these efforts purport to explore the role and/nature of a particular architecture activity in agile approaches. That means there has been no significant empirical effort aimed at investigating one of the most significant point of debate between agile proponents and architecture followers: *can a satisfactory architecture emerge from continuous small refactoring* [4]?

Boehm and his colleagues have reported a set of guidelines for deciding how much agility and architecting are enough in a software development project [23]. Their set of guidelines was derived from an analysis model that considers three factors: project size, criticality, and volatility. Their effort highlights the importance of considering the contextual factors in deciding the level of architectural efforts required in projects using agile approaches.

We assert that the contextual factors identified by Boehm and his colleagues can also play important role in understand whether or not a satisfactory architecture can emerge from small, continuous refactoring. At the same time, we also assert that there can be many more contextual factors whose interplay can impact the emergence of “*satisfactory architecture*” through continuous refactoring. There has been no significant effort aimed at systematically identifying the factors that characterise the context in which a satisfactory architecture emerges or otherwise.

Hence, the main goal of this research is to empirically explore the perceptions and experiences of practitioners for identifying and building a taxonomy of the factors that can influence the emergence of a satisfactory architecture through continuous refactoring. In order to achieve the research goal, we carried out a large-scale study for empirically investigating two key aspects of the role of refactoring in achieving a satisfactory architecture when using agile approaches:

- Whether or not a satisfactory architecture can emerge from continuous small refactoring;
- What can be the contextual circumstances in which a satisfactory architecture emerges or not?

Both of these aspects were considered important for our research as we were not only interested in empirically finding

out whether or not a satisfactory architecture emerges from continuous refactoring based on practitioners' experiences but also intended to discover the factors that can play a role in the emergence of a satisfactory architecture through refactoring.

III. METHODOLOGICAL DETAILS

In this section, we describe and discuss the research methodology, data gathering approach, and data analysis method used for the reported research.

A. Survey

We decided to use survey research method to explore the perceptions and opinions of practitioners about their perceptions, observations, and experiences of the role of small continuous refactoring in the emergence of satisfactory architecture. A survey research method is considered suitable for gathering self-reported quantitative and qualitative data from a large number of respondents [24]. The objective of our study was to gather self-reported qualitative data. Our survey design was a cross-sectional, case control study. Survey research can use one or a combination of several data gathering techniques such as interviews and self-administered questionnaires [25]. We decided to use interviews as the data collection instrument as it appears to be more appropriate for gathering the detailed information required to find the answers to the questions that motivated our research.

We recruited the participants from different sources based on our pre-defined criteria for inviting the potential participants in our study; some of the criteria elements were industrial experiences of working in software development using agile and architecture approaches, experience of working as software architecture or requirements engineer, and represent different regions and domains. We sought the participants through the primary researcher's personal and professional networks as well as through a professional networking site, LinkedIn.

B. Data Collection Approach

We conducted semi-structured interviews using open-ended questions via email. Each email thread started from a brief description of the background and objective of the study, and a few questions for kicking off the required discussion. The email thread was continued through the primary researcher's enquiring and the participants' responses for more details until it was felt that an exhaustive amount of information had been gathered from each participant. Our email-based data gathering approach not only avoided the difficulty for the participant to find a sufficient chunk of free time to fit the researcher's schedule, but also provided participants with more time for preparing and sending reflective answers.

The interview questions focused on the participants' observations and experiences on architecture practices in Agile software development. In particular, we asked about their experiences and observations of cases where a satisfactory architecture emerged or did not emerge as expected from continuous small refactoring, and their reflections on why a satisfactory architecture emerged or did not emerge in those cases. Each email thread resulted in one to five pages (A4 size)

document of communication log. The communication logs with all the participants form a rich set of qualitative data.

C. Data Analysis

For data analysis, we used mixed methods approach. The answers to the first question needed to be analysed by quantifying the qualitative data from the participants' responses based on their experiences and observations of cases where a satisfactory architecture emerged or did not emerge as expected. The quantification simply counted the number of participants who answered "Yes" or "No" to the two particular questions we asked related to the emergence of a satisfactory architecture through refactoring. We then summarized the quantified data using descriptive statistics.

Then we analyzed the data to find answer to the second aspect of our inquiry (i.e., underlying reasons for the observed phenomenon). For this analysis, we followed the techniques described in Grounded Theory (GT) [7]. Recently, other researchers have also used GT to study different aspects of Agile software development [26]. The data analysis procedure in GT involves three types of coding: open coding, axial coding, and selective coding. During the open coding, we analyzed the data line by line for creating and assigning codes to phrases, sentences, or paragraphs. A code is a phrase that summarizes the key point concisely. During the axial coding, we went through the codes, and related them to each other, via a combination of inductive and deductive thinking [7]. The data analysis was not a linear process. We went through several iterations to refine, adjust the codes, and their relationships. The codes emerged directly from the data, which is in turn collected directly from the field. Thus, the resulting findings are grounded within the context of the real world experiences and observations.

For our finding, we included only those concepts that were mentioned by at least two participants. Once the concepts had made their way into the findings, we did not discriminate between them based on their frequencies; rather, we focused on the logical relationships among the concepts as recommended by GT. We report the findings in the following sections.

IV. PARTICIPANTS' DEMOGRAPHIC INFORMATION

We interviewed 102 experienced software professionals across the world. Each of them has experience in both architecture design and Agile approaches. Due to limited space, we cannot provide the complete details of the participants' demographic details, hence, we just provide a summary of their backgrounds and work experiences.

The participants were located in 13 countries from 6 continents. The distribution across these continents is summarized in Figure 1. A majority of them were from North America. They had accumulated more than 1,600 years of work experience of software development (See Table I) in over 700 companies from nearly 40 different domains. All of them had experiences in both architecture design and Agile approaches. The experience they had in architecture, and Agile approaches was: 9 years and 7 years on average respectively; 2.5 years and 2 years minimum respectively; 20 years and 30 years maximum respectively. It is worth noting that a few of

them reported that they had been using Agile approaches before the term “Agile” was coined.

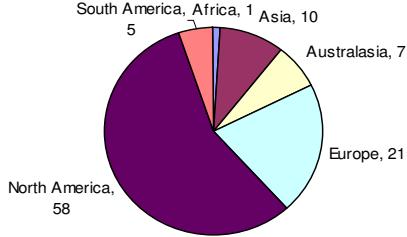


Fig. 1. Geographical distribution of participants

The Agile approaches that the participants had been using included Scrum [11], XP, Lean (Lean software development) [27], FDD (Feature Driven Development) [28], Kanban [29], and Crystal Clear [10]. The distribution over these approaches is summarized in Table II. Scrum and XP are the mostly widely used ones among the participants. It is worth noting that many of them used multiple approaches. The participants had worked for various organizations. On average, a participant had worked for 8 organizations at different stages of his/her career. The participants had worked for 737 different companies that ranged from very small (<10), small (10-49), medium (50-249), large (250-10,000), and to very large (>10,000). The ten top domains of the participants’ companies are presented in Fig. 2; the domains include automotive, telecom, finance, and web-based socio-technical systems. In order to protect our participants’ privacy, we refer to them by numbers P1 to P102 when presenting their quotations.

TABLE I. SUMMARY OF PARTICIPANTS’ EXPERIENCE

	Minimum	Maximum	Average
Software Industry	6	37	16
Architecture ²	2.5	20	9
Agile	2	30	7

TABLE II. AGILE APPROACHES USED BY THE PARTICIPANTS

Scrum	XP	Lean	FDD	Kanban	C. C.
82	68	10	6	4	2

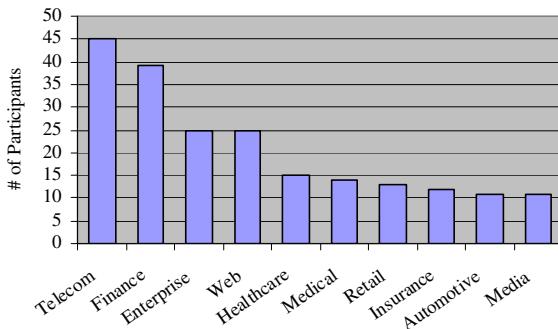


Fig. 2. Distribution of participants based on domains

² It was measured by the number of years that a participant has been taking the position of architect.

V. ON THE EMERGENCE OF ARCHITECTURE FROM CONTINUOUS REFACTORING

To gather evidence regarding whether a satisfactory architecture can emerge through continuous small refactoring, we particularly asked every participant two questions:

- Q1: Have you experienced or observed cases where a satisfactory architecture emerged from continuous small refactoring as expected?
- Q2: Have you experienced or observed cases where the team expected a satisfactory architecture to emerge from continuous small refactoring, but it did not?

Figure 3 shows the numbers of participants who said “YES” or “NO”. A majority of the participants (68%, 69 out of 102) had experienced or observed cases where a satisfactory architecture emerged from continuous small refactoring. A slightly less than a third (32%, 33 out of 102) of the participants had not experienced or observed any such cases. It indicates that in many cases a satisfactory architecture can emerge from continuous small refactoring.

Does this mean that, in most cases, a team can go for implementation directly and let the architecture emerge? Participants’ replies to Q2 do not support this claim. Figure 4 presents the participants’ replies to Q2. A large majority of them (82%, 84 out of 102) had experienced the cases where a satisfactory architecture did not emerge from continuous small refactoring as expected. This result indicates that in many cases a satisfactory architecture cannot emerge from continuous small refactoring.

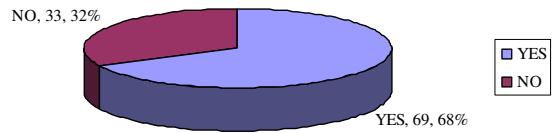


Fig. 3. Q1: Have you observed cases where a satisfactory architecture emerged as expected?

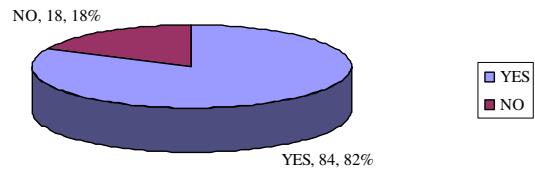


Fig. 4. Q2: Have you observed cases where a satisfactory architecture did not emerge as expected?

To gain more insights into the data, we cross-tabulated participants’ answers to Q1 and Q2. Table III shows the result. The number in each cell represents the number of participants who gave the answers indicated by the column and row headings. We can see that a majority of the participants (59%, 60 out of 102) have experienced or observed both types of cases. Slightly less than a quarter of the participants (23%, 23

out of 102) have only experienced cases where a satisfactory architecture did not emerge as expected. Nearly one in ten participants (9%, 9 out of 102) has only experienced or observed cases where a satisfactory architecture emerged. A similar number of participants (9%, 9 out of 102) have not experienced or observed either type of cases. This is because they have not tried to follow the advice that architecture should emerge from continuous small refactoring. We will discuss these numbers in Section VII.

TABLE III. Q1 VS. Q2 CROSS TABULATION

	Q2.Yes	Q2.No
Q1.Yes	60 (59%)	9 (9%)
Q1.No	16 (23%)	9 (9%)

The descriptive statistical summary of the participants' answers to Q1 and Q2, as presented above, shows that a satisfactory architecture emerged from continuous small refactoring in many cases, but did not emerge in many cases as well. Thus, the answer for whether a satisfactory architecture

can emerge, suggested by the data, is: It depends. But, what does it depend on? We present the findings gained from our qualitative data analysis in the next section.

VI. FACTORS THAT IMPACT THE EMERGING OF ARCHITECTURE THROUGH REFACTORY

The second key aspect of this study was to identify the contextual factors that may support or inhibit the emergence of a satisfactory architecture through continuous refactoring. For this part of the data, we applied the data analysis techniques from GT as previously stated. Our data analysis identified twenty contextual factors that perceived to have positive or negative impact on the emergence of architecture through refactoring. We systematically analysed the identified factors and place them into categories to form a framework of contextual factors that can help predict whether or not a satisfactory architecture would emerge through continuous refactoring. Figure 5 shows the framework that consists of factors: project, team, practices, and organization.

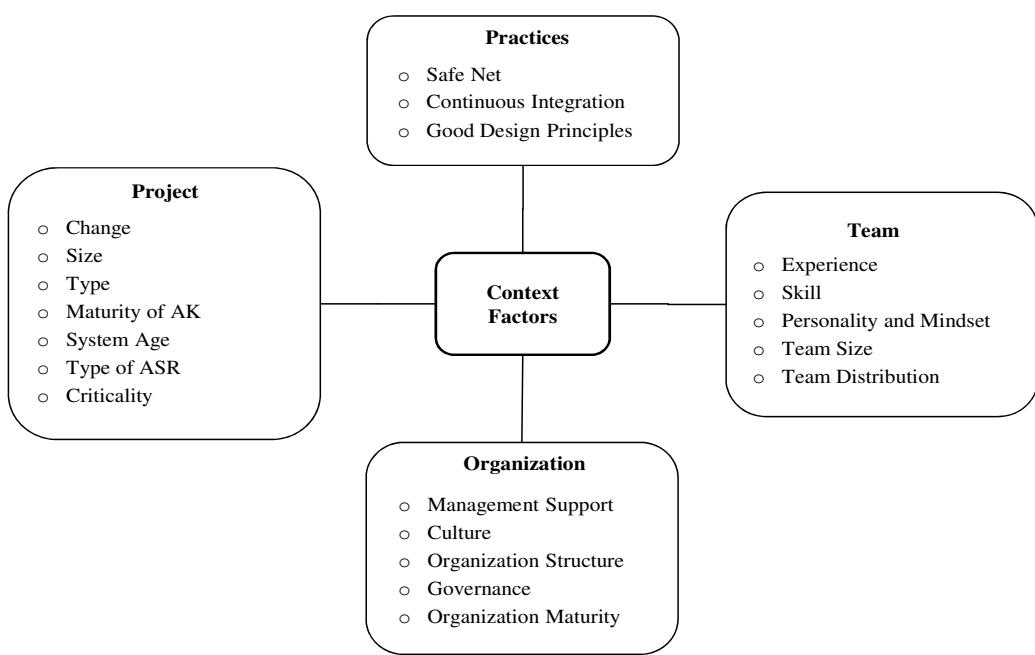


Fig. 5. A framework of factors that influence the emerging of a satisfactory architecture from continuous small refactoring

A. Project

This category of factors is related to different aspects of a project that can impact the emergence of architecture from refactoring. Our analysis revealed that around 35% of the participants indicated the factors that have been placed under this category. Many other studies have also mentioned that the implementation of agile approaches is customized based on a project's need. This category has 7 factors.

Rate of Change has impact on the emergence of architecture through continuous refactoring, as mentioned by many participants. If the rate of change in the requirements is

quite high or a significant implementation has been done before getting to know the key non/functional requirements, it is quite unlikely that a satisfactory architecture emerges through small continuous refactoring: “*Many times string non-functional (business) requirements may appear after the software started to be built, and that may impact on the half-built solution you've got*”, p41. On the other hand, if the rate of change is very low, continuous refactoring may become an unnecessary overhead rather than helping architecture to emerge: “*When unlike #1 [where rate of change is high]. I have seen where due to the constant changes and updates that the end goal either gets clouded, lost all together or is severely changed then what was first thought*”, p92.

Size of A Project also plays an important role in emergence of architecture through refactoring. Like many sceptics of the claim about the emergence of architecture through refactoring, most of the participants reported that most of the time this happens for smaller projects. Interestingly, however, a few participants have experienced cases where a satisfactory architecture emerged for large projects: “*Larger projects--involving several groups-- are more prone to architectural issues, but some of this can be mitigated by focusing on loosely coupled interactions between the software components*”, p3.

Type of Project is also an important consideration for this matter as for some types of projects, such as those that have only one feature and are algorithmically complex, and those that do not allow small releases, a satisfactory architecture is hard to emerge, “*sometimes you need a base with a set of minimum of functionality*” and this “*minimum*” can be “*a fairly large critical mass. Then it may not be possible to work in small steps*”, p11.

Maturity of Architectural Knowledge (AK) means the amount of experience and knowledge of reusable architectural artefacts such as reference architectures, design patterns, and tactics. If a project team is mature in AK and its applications in different situations and contexts, it is likely that their knowledge and experience can lead to implicitly disciplined and mindful refactoring that takes into consideration of well-known design principles and patterns. Hence, such refactoring usually leads to the emergence of a satisfactory architecture, compared with a team consisting of architecturally immature people. With a matured AK, they are able to know where to start the architecture and have a relative clear vision on potential evolutionary paths of the architecture they started with. Without the existence of mature AK, a satisfactory architecture is unlikely to emerge from continuous refactoring.

System Age also plays an important role in supporting the emergence of architecture through refactoring. Software systems designed based on contemporary design principles and technologies are likely to be more amenable to gain structural and behavioural integrity through refactoring. Generations old systems have the tendency of being monolithic and unnecessarily interdependent that can make it difficult (or impossible) for them to provide a coherent and conceptual integral architecture through refactoring. “*Failure [architecture did not emerge] is also more prevalent in cases of large pre-existing products which either had no discernable architecture to begin with or had their architecture erode away over years of maintenance and haphazard addition of new features by persons who didn't understand the pre-existing architecture or ignored it for one reason or another*”, p85.

Type of ASRs (Architecturally Significant Requirements) and Their Criticality are known to be the key driver of architecture design activities. For refactoring, ASRs are also the key motivators. However, it all depends upon the nature of ASRs whether or not a highly coherent and integral architecture is achieved through refactoring. For example, design time ASRs such understandability may be

achieved through continuous small refactoring by applying well known design principles and patterns. However, this may not be the case for another type of ASR such as security, which needs to be proactively addressed at design stage. “*A good example is security, which in all of my years of experience, should be designed in (but can be implemented later). Especially in terms of cascading Web Services and such, [an] impedance mismatch between services and framework architecture is costly for rework*”, p33. That means the criticality of achieving a particular ASR also plays an important role in a decision whether to do detailed design upfront or wait to get it fixed through refactoring. A project with critical ASRs hardly progresses in an expectation that a satisfactory architecture will emerge from refactoring.

B. Team

The category incorporates those factors that are directly or indirectly related to different dimensions of a project’s team. Nearly half of the participants (48%, 49 out of 102) mentioned the team related factors having an impact on the emergence of architecture through continuous refactoring. “*You need to have a good Agile team that know how to go about this process [emerging of architecture] well. Otherwise this is a disaster and produces a lot of waste*”, p99.

Experience and Skill of designing similar systems can also help support architecture through refactoring as mentioned by several participants. Researchers in other disciplines have also reported significant differences in understanding a design problem and devising solutions based on the amount of design experience [30]. That means significant experience with similar projects carried out using similar technologies in similar domains can enable designers to gain and maintain conceptual integrity of the software design through refactoring of the structure of the software being developed or evolved. “*An inexperienced developer may not have as much success since he/she may not know what architectural goals need to be achieved, which could result in absolutely messy unusable code*”, p37. Skillset is another related factor mentioned by many respondents. The nature of a skill set and the amount of experience usually go together. That is why our analysis revealed that when participants indicated experience they were referring to experience of a particular skillset directly or indirectly. Lack of sufficient skills usually results in refactoring not making any noticeable contribution to have a sufficient architecture rather there may be risk of having the source code broken at various places: “*Those [teams] not having high skill in this area tend to simply make the codebase ‘different’, not better. Worst case they retard progress with continual rework*”, p85.

Personalities and Mindset of team members is important to succeed in getting a satisfactory architecture through continuous refactoring. “*Change should be acknowledged as a part of the development process*” (p3), be “*willing to learn technology and try to adapt to them*” (p9), have “*passion*” (p9), and need “*dedication throughout the team (75% or more of its members, in my experience) to good quality design*” (p13). “*Pedantic developer personalities who are more focused on consistency of micro details rather than the overall readability, understandability, obviousness*

of the code" (P28), "*Unwillingness from some team members to look outside of their own code (keep constant look at overall system's architecture)*" (P30), and "*personalities not inclined to work in groups*" (P55) can lead to failure.

Team Size and Distribution are also mentioned as factors. Team size is usually driven by project size, but it may not always be the case, e.g., a small project may use a large team with the expectation to finish the project faster. The participants of our study were of the view that a satisfactory architecture may be achieved through refactoring if the team size is relatively small. It was also revealed that a team's distribution nature may also impact as a satisfactory architecture can usually be achieved through refactoring in collocated teams. "*Proximity between co-workers (being able to listen and talk freely...no cubicles) [is essential]*", P24. "*2 teams working across geo (in different [time zones]) without an integrated approach has resulted in failure [architecture did not emerge]*", P65.

C. Practices

Since software development practices interact with each other, the participants of our study have also indicated several practices which can potentially support or inhibit the achievement of a satisfactory architecture through continuous refactoring. Due to limited space, we describe the three most frequently mentioned practices that can impact on the emergence of a satisfactory architecture.

Safe Net refers to automated testing with good coverage to form a safe net for refactoring, so that the team can have confidence that refactoring does not break a system. Many participants were of the view that continuous refactoring may fail to return a satisfactory architecture because of low coverage through automated testing. "*Your team must use constant automated unit testing and measure code coverage before practicing 'wide' refactoring. Code coverage in testing should be above 90% to give [the] team some kind of psychological confidence that changing the architecture will not break the system. Without this confidence team members will be resistant to change [the] architecture much and will rather focus on small local refactoring.*", P30. "*Without automated tests, architectural changes become more risky. They tend to take longer to implement. And so they tend to occur less often, and you don't see incremental movement toward a good architecture -- more typically you see incremental degradation of the architecture into chaos*", P13.

Continuous Integration is a key supporting practice for architecture to emerge from continuous small refactoring. "*Re-factoring without continuous integration tests to verify re-factoring hasn't broken, or re-broken the product is very, VERY dangerous [...] Way before Agile and integrated tests I worked on a rule-based system [with more] than 10,000 rules. One team member would always re-factor something [...]. He knew he was so good [that] his changes were 'correct'. He broke almost every build and by the time we found out he was on the road driving three hours away*", P14.

Good Design Principles are expected to lead to high quality architecture that can ensure the achievement of desired ASRs – Agile approaches also support the use of

good design principles, albeit implicitly, for example, refactoring is one way improving the internal structure of an application. Many of the participants mentioned the use of several principles: DRY (Don't Repeat Yourself) [31], SOLID (the Single responsibility principle, the Open closed principle, the Liskov substitution principle, the Interface segregation principle, and the Dependency inversion principle), and KISS (Keep It Simple and Straightforward). "*...all these practices complement each other...while eliminating the pitfalls that otherwise might be introduced by other practices. For example, unit test without refactoring will definitely introduce rigidity in architecture compounded with light up-front design could contribute to severe and rapid architectural deterioration thus compromise in quality and productivity*", P60.

D. Organization

Software development practices and their outcomes are expected to be influenced by organizational cultures and practices. The participants of our study also identified organizational factors that can support or inhibit the emergence of a satisfactory architecture through continuous refactoring. The organizational factors were identified by almost 33% of the participants.

Management support and commitment to agile and architecture is important. When lack of management support, a satisfactory architecture did not emerge. "*Of course when an architecture transformation is not a priority target and a team is not given enough time to work on the necessary changes then the new architecture will not emerge*", P5. "*You[r] team should have enough time to perform [a] 'wider' look at the system and change its architecture when local refactoring comes into conflict with overall architecture. Working under constant pressure from management to deliver releases, [the] team may not feel it's in the best interest to spend days improving architecture... even realizing it may significantly improve [the] product's maintenance and reliability. Management must be supportive on team's decision to refactor code and invest time, if necessary*", P30.

Culture can also play an important role in achieving architecture through refactoring. The cultural aspects mentioned by the participants include: good communication channels, encouragement for people to take ownership and commitment, open, and blame-free. If suitable culture is not there, a satisfactory architecture is unlikely to emerge. "*Team members must have good communication channels and discuss overall changes with each other all the time [...], so everyone would know about system-wide changes [...]. Bad communication between team members is another strong contributor to fail producing new architecture from small local changes*", P30. A participant referred to a failure case: "*They [the team] were used to work heads down in their cubicles for months without speaking to anybody. After that time they simply deposit hundreds of pages of useless diagrams and felt good about it. [...] the sense of responsibility that comes with Agile is not there*", P24.

Organization Structure can also help or hinder in gaining a satisfactory architecture through refactoring. When describing the reasons for cases where a satisfactory architecture did not emerge, participants said: “*When the organizational structure prevents [emerging of architecture], for example, enterprise organizations often make it difficult to do continuous small refactoring.*” p29. “*Their organizational structure was based by role (architects, analysts and developers). They were not embracing the openness of the Agile approach*”, p24.

TABLE IV. A CHARACTERIZATION OF CONTEXTS IN WHICH A SATISFACTORY ARCHITECTURE IS LIKELY TO EMERGE

	Factor	Success Condition
Project	<i>Change</i>	Medium to high rate of change
	<i>Size</i>	Small
	<i>Type</i>	Support small releases
	<i>Maturity of AK</i>	Mature Architecture Knowledge (AK)
	<i>System Age</i>	Green field
	<i>Type of ASR</i>	No demanding ASR that cannot be satisfied by refactoring
Team	<i>Criticality</i>	Low criticality
	<i>Experience</i>	Experienced
	<i>Skill</i>	Skilled
	<i>Personality and Mindset</i>	Willing to make change, learn, have passion, with dedication to good design
	<i>Team Size</i>	Small
Practices	<i>Distribution</i>	Collocated
	<i>Safe Net</i>	Automated testing with good coverage
	<i>Continuous Integration</i>	Continuous integration
	<i>Good Design Principles</i>	Applying good design principles such as DRY, SOLID, KISS
Organization	<i>Management</i>	Management support and commitment
	<i>Culture</i>	Good communication channels, encouraging for taking ownership and commitment, open, blame-free
	<i>Structure</i>	Embraces the openness of Agile approaches.
	<i>Governance</i>	Proper architecture governance
	<i>Maturity</i>	Certain Level of Maturity

Organizational Governance and Maturity are important factors in ensuring architecture can be achieved through refactoring. A participant said: “*Without any kind of governance, 5 different development teams implemented different code that impacted the performance of everyone on the platform*”, p32. When answering the reasons for cases where a satisfactory architecture emerged, a participant noted: “*It is a success because there is good architecture governance*”, p76. Moreover, it is commonly known that sound architecture are designed upfront or achieved through refactoring in relative mature organizations. When explaining why a satisfactory architecture did not emerge as expected, p63 replied: “*CMMI level one task[s] are only accomplished by the heroic act of individuals.*”

E. A Characterization of Contexts

The findings from our study have resulted in a characterization of contexts in which a satisfactory architecture is likely to emerge from continuous small refactoring (see Table IV). For the projects in contexts that are different from those characterized in Table IV, relying on a satisfactory architecture to emerge instead of doing an adequate upfront architecture design is risky. For these projects, a proper upfront design by adopting architecture-centric practices is highly recommended.

Due to the qualitative and exploratory nature of this study, giving an accurate measure for each factor is beyond the scope of this study. Characterizing a factor accurately (e.g., what is the threshold for project size, what are the specific types of ASR, how mature an organization should be, how informal or formal the governance should be) would require a separate empirical program consisting of defining appropriate metrics and then validating them. Though the characterization given in Table IV could not provide accurate measurement for each of these factors, it provides useful information to assist practitioners in deciding appropriate architecture practices to be used in their particular projects, rather than following the advice that “*architecture should emerge from continuous small refactoring*” on its face value.

VII. DISCUSSION

The findings of this study has shown that like any other software development practices, the contextual factors play a significant role in supporting or inhibit the emergence of a satisfactory architecture through continuous refactoring. However, many instances have been reported when contextual aspects of a particular practice or belief may be ignored by practitioners [8]. Interestingly, our study has also unearth a reason for this phenomenon. Table III (see Section V) shows that people, who are only aware of cases where a satisfactory architecture emerged and have not observed cases where a satisfactory architecture did not emerge, do exist (9%, 9 out of 102 participants). Because they have only experienced or observed successful cases, they tend to have strong belief in their practices. Thus, they tend to say with strong conviction to other practitioners: “*it worked for me, so I do not see why it should not work for you*” [8].

The findings of our study can provide some important insights with regards to the importance and role of contextual factors that should be taken into consideration when devising strategies to deal with architectural aspects, otherwise, it can harmful. If other practitioners, who work in totally different contexts, listened and followed their generic advice, they may run into difficulties, or even cause project failures. Clarifying or even just recognizing the confusion caused by such context-regardless preaching of practices is not easy. This is evidenced by the many “*blind bigots, sometimes rabid bigots*” among followers of Agile advocates [8].

In such a situation, empirical evidence is particularly valuable. Thus, we suggest that more empirical studies should be conducted to establish an evidence-based understanding of the context factors and their impact on the

effectiveness of agile practices in general and combining architecture and agile approaches. Such understanding is essential for practitioners to select and tailor their process to a particular project according to the specific contexts at hand. As described in Section II, Boehm et al. [23] and Abrahamsson et al. [3] have also indicated a set of factors that determine the needs for architecture-centric activities. Compared to Boehm et al. [23], we have identified all the three factors they considered, i.e., project size, criticality, and volatility (corresponding to change). We have also identified many more factors. This answers the question we asked in Section II about the guidance [23] they provided. The answer is: many more factors need to be considered in their guidance. So we suggest practitioners to carefully consider the extra factors we have identified when using their three-factor based guidance.

The context model presented by Kruchten [32] provides the most comprehensive list of factors. Our results have confirmed Kruchten's experience-based insights with empirical evidence. Our study has also identified several factors (e.g., experience and skill) that are not included in the Kruchten's model. Our study only focuses on a particular practice in Agile software development (i.e., emergence of architecture through continuous small refactoring), we also identified and highlighted several supporting practices as factors. So our study results corroborate and complement Kruchten's work.

VIII. THREATS TO VALIDITY

Given the qualitative nature of our research, the usual threats to validity are inconsistent. An investigator's bias is not considered a threat in GT, but a required attribute. The investigator is expected to select the participants, refine the questions, and develop the theory. When evaluating the validity of a qualitative research, the terms like quality and credibility are used. Quality concerns the question: are the findings useful [33]? Credibility concerns the question: are the findings trustworthy and do they reflect the participants' experiences with a phenomenon [33]? We used three criteria to evaluate the quality and credibility of this study.

Fit concerns with questions like, “*do the findings fit/resonate with the professionals for whom the research was intended?*” We kept all traces from each finding to the participants' replies throughout the data analysis. We can link a finding to the replies from the participants.

Applicability or Usefulness concerns with question such as “*do the findings offer new insights? Can they be used to develop policy or change practice?*” On average, each of the participants mentioned three out of the 20 factors we have identified. At maximum, an individual participant mentioned six factors. This indicates that the list of 20 factors as a whole is probably beyond an individual's knowledge. We believe that these 20 factors together with the characterization of contexts can offer new insights. The findings can also cause changes in practice. These findings can help practitioners to realize the importance of context factors, and help them to make informed decisions. Hence, they may decide to spend sufficient time and resources on

architecture design in the contexts where a satisfactory architecture is unlikely to emerge.

Variation concerns with questions like, “*has variation been built into the findings?*” That means if a phenomenon is complex, the findings accurately represent that complexity. The study participants had diverse backgrounds as noted in the demographic information of the participants. Hence, the findings are expected to show that diversity.

The inherent limitation of studies like ours is that the results can only be explained in the specifically explored contexts. The identified factors are not exhaustive. They only represent those that have been experienced and observed by our participants. The selection of participants through social connections could potentially result in selection bias. We have carefully excluded persons who are likely to advocate one side of issue due to their vested interest. A very high number of variables that affect a real software engineering project make it difficult to conclusively identify the impact that one factor may have on a project.

IX. CONCLUSION

Despite continuously increasing popularity and adoption of Agile approaches, there is an increasing perplexity about the software architecture's role and importance in Agile software development [3]. One of the most fundamental points of the perplexity is the question: *Can a satisfactory architecture emerge from continuous small refactoring* [4]? Based on a large scale interview based empirical study, it can be concluded that different participants had different views about whether or not a satisfactory architecture emerges through continuous refactoring. Our study has identified twenty contextual factors that have been placed in four elements of framework: project, team, practices, and organization. We have further characterized the contexts in which a satisfactory architecture is likely to “*emerge*”. These findings can be used by practitioners to make informed decisions on their architecture practices in Agile software development.

Particularly, we hope the empirical evidence reported in this study can help eliminate the commonly observed phenomenon [8]: some practitioners usually ignore architecture-centric activities with the justification that “*we are doing Agile, architecture should emerge from continuous small refactoring*.”

ACKNOWLEDGMENTS

The authors are thankful to all the participants. The first author also acknowledges the help of Bashar Nuseibeh, Klaas-Jan Stol, Soo Ling Lim, and Sajid Ibrahim Hashmi for improving an initial draft of this paper.

REFERENCES

- [1] D. West and T. Grant, "Agile Development: Mainstream Adoption Has Changed Agility - Trends In Real-World Adoption Of Agile Methods," Forrester Research, Inc.2010.

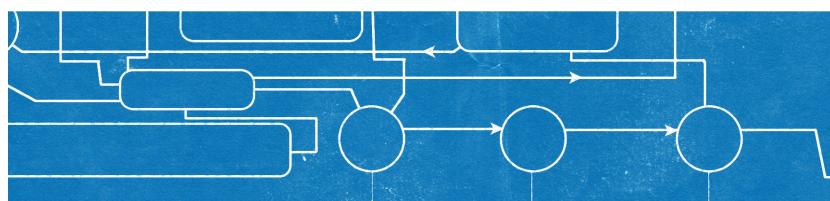
- [2] M. A. Babar, "Making Software Architecture and Agile Approaches Work Together: Foundations and Approaches," in *Aigle Software Architecture: Aligning Agile Processes and Software Architectures*, M. A. Babar, Brown, A. W., Mistrik, I., , Ed., ed: Morgan Kaufmann, 2014, pp. 1-22.
- [3] P. Abrahamsson, *et al.*, "Agility and Architecture: Can They Coexist?," *IEEE Software*, vol. 27, pp. 16-22, 2010.
- [4] H. Erdoganmus, "Architecture Meets Agility," *IEEE Software*, vol. 26, pp. 2-4, 2009.
- [5] L. Chen, *et al.*, "Characterizing Architecturally Significant Requirements," *Software, IEEE*, vol. 30, pp. 38-45, 2013.
- [6] R. L. Nord and J. E. Tomayko, "Software architecture-centric methods and agile development," *Software, IEEE*, vol. 23, pp. 47-53, 2006.
- [7] J. Corbin and A. Strauss, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*, 3 ed.: Sage Publications, 2007.
- [8] P. Kruchten, "Voyage in the Agile Memeplex," *ACM Queue*, vol. 5, pp. 38-44, 2007.
- [9] K. Beck, *Extreme Programming Explained: Embrace Change*: Addison Wesley Longman, Inc., Reading, MA. USA, 2000.
- [10] A. Cockburn, *Crystal Clear: A Human-Powered Methodology for Small Teams*: Addison-Wesley, 2004.
- [11] K. Schwaber, *Agile Project Management with Scrum*: Microsoft Press, 2004.
- [12] D. Turk, *et al.*, "Assumptions underlying agile software-development processes," *Journal of Database Management*, vol. 16, pp. 62-87, 2005.
- [13] C. Hofmeister, *et al.*, "A general model of software architecture design derived from five industrial approaches," *Journal of System and Software*, vol. 80, pp. 106-126, 2007.
- [14] P. Thapparambil. (2005) Agile architecture: pattern or oxymoron? *Agile Times*. 43-48.
- [15] C. Hofmeister, *et al.*, *Applied Software Architecture*: Addison-Wesley, 1999.
- [16] L. Bass, *et al.*, *Software Architecture in Practice*, 2nd ed.: Addison-Wesley, 2003.
- [17] P. Kruchten, *The Rational Unified Process: An Introduction*: Addison-Wesley, 2003.
- [18] M. A. Babar, "An exploratory study of architectural practices and challenges in using agile software development approaches," in *Joint Working IEEE/IFIP Conf. on Software Architecture & European Conf. on Software Architecture*, 2009, pp. 81-90.
- [19] D. Falessi, *et al.*, "Peaceful Coexistence: Agile Developer Perspectives on Software Architecture," *IEEE Software*, vol. 27, pp. 23-25, 2010.
- [20] S. W. Ambler, *Agile modeling: effective practices for eXtreme Programming and the Unified Process*. New York: John Wiley & Sons, Inc., 2002.
- [21] IBM. (2004, *RUP for Extreme Programming (XP) Plug-Ins*. Available: <http://www-128.ibm.com/developerworks/rational/library/4156.html>
- [22] C. Larman, *Agile and iterative development: a manager's guide*. Boston, MA: Addison Wesley Professional, 2003.
- [23] B. Boehm, *et al.*, "Architected Agile Solutions for Software-Reliant Systems," in *Agile Software Development: Current Research and Future Directions* T. Dingsøyr, *et al.*, Eds., 1 ed: Springer, 2010, pp. 165-184.
- [24] B. Kitchenham and S. L. Pfleeger, "Principles of Survey Research, Parts 1 to 6," *Software Engineering Notes*, 2001-2002.
- [25] T. C. Lethbridge, "Studying Software Engineers: Data Collection Techniques for Software Field Studies," *Empirical Software Engineering*, vol. 10, pp. 311-341, 2005.
- [26] R. Hoda, *et al.*, "Organizing self-organizing teams," presented at the Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, Cape Town, South Africa, 2010.
- [27] M. Poppendieck and T. Poppendieck, *Lean Software Development: An Agile Toolkit*: Addison-Wesley, 2003.
- [28] S. R. Palmer and J. M. Felsing, *A Practical Guide to Feature-Driven Development*: Prentice Hall, 2002.
- [29] D. J. Anderson, *Kanban*: Blue Hole Press, 2010.
- [30] S. Ahmed, *et al.*, "Understanding the Difference Between How Novice and Experienced Designers Approach Design Tasks," *Journal of Research in Engineering Design*, vol. 14, pp. 1-11, 2003.
- [31] A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*: Addison-Wesley Professional, 1999.
- [32] P. Kruchten, "Contextualizing agile software development," *Journal of Software: Evolution and Process*, vol. 25, pp. 351-361, 2013.
- [33] J. W. Creswell, *Qualitative Inquiry and Research Design: Choosing among Five Approaches*, 2 ed.: Sage Publications, 2006.

Toward Agile Architecture

Insights from 15 Years of ATAM Data

Stephany Bellomo, Ian Gorton, and Rick Kazman,
Software Engineering Institute

// To gain insight into which quality attributes are of greatest concern to agile practitioners, researchers analyzed stakeholder concerns from 15 years of Architecture Trade-Off Analysis Method data across 31 projects. Dominant concerns include modifiability, performance, availability, interoperability, and deployability. //



CONCERNs ABOUT LONG-TERM deterioration in large-scale agile projects' quality (called technical debt¹) have renewed the agile community's interest in software architecture.²⁻⁴ This interest indicates a growing consensus that a strong architectural foundation lets teams rapidly evolve complex software systems using iterative, incremental development. This position is consistent with arguments in *IEEE Software*'s Mar./Apr. 2010 special issue on Agility and Architecture.

One major challenge agile teams face in building an architectural

foundation is balancing two competing concerns:

- delivering near-term functional requirements (based on the agile philosophy of delivering user value early and often) and
- meeting near- and long-term quality attribute goals (without which the project can grind to a halt because system complexity makes efficient modifications impossible).

Quality attribute prioritization in particular can be difficult in early increments, and a wrong decision can

result in hard-to-modify, unreliable, slow, or insecure systems.⁵

So, we sought to obtain empirical evidence on the most common quality attribute concerns that projects must address and their relative importance. We performed a meta-analysis on the results of several Software Engineering Institute analyses carried out over the past 15 years using the Architecture Trade-Off Analysis Method (ATAM). The ATAM leverages quality attribute scenarios that project stakeholders have identified and prioritized and uses this information to evaluate the architecture and identify architectural risk. (For more on the ATAM, see the related sidebar.) Here, we examine our analysis results and describe how agile teams can use them to prioritize and allocate quality attributes.

ATAM Study Results

We present results from two studies of ATAM data collected from 31 projects. In Study 1, Ipek Ozkaya and her colleagues analyzed ATAM scenario data from 1999 to 2006.⁶ In Study 2, we analyzed ATAM scenario data from 2006 to 2013. The quality attribute scenario data examined in both studies was generated during ATAM phase 1 and collected from the elicited utility trees.⁷ (For more on the two ATAM phases and utility trees, see the ATAM sidebar.)

The sidebar "Our Analysis Methodology" describes our method. SEI's ATAM analyses cover a range of project types across government and industry domains. Table 1 categorizes the data into IT versus embedded systems, and government versus nongovernment projects. Ozkaya and her colleagues' paper detailed Study 1's project profile.⁶ Study 2 included nine projects and

348 quality attribute scenarios; these projects were more heavily weighted toward IT (78 versus 42 percent).

Figure 1 compares the two studies' results, which don't include categories with fewer than three scenarios. (We didn't include them because their scenarios tended to be more context specific or less generalizable.) The top quality attribute concerns were consistent. For example, modifiability was the most common quality attribute in both studies. Performance was the second-most-important attribute in Study 1 and third (by a small margin) in Study 2. This suggests that although the technical environment for software-intensive systems has evolved over the years, the most important stakeholder and architectural concerns haven't changed much.

Table 2 lists the top 20 quality attribute concerns of Study 2, ranked from highest to lowest according to their frequency in the dataset. This view reinforces the importance of modifiability, performance, interoperability, availability, and (to a lesser extent) usability. However, it also suggests that project teams prioritize specific quality attribute concerns in a finer-grained way than what Figure 1 shows. For example, consider the top three quality attribute concerns: "modifiability: reducing coupling," "performance: latency," and "interoperability: upgrading and integrating with other system components."

Data Analysis and Discussion

We now look at this data's implications.

Modifiability Leads

As we mentioned before, modifiability ranked highest in both studies (24 percent in all scenarios for Study 1, and 26 percent for Study 2). As Table

THE ARCHITECTURE TRADE-OFF ANALYSIS METHOD

The Architecture Trade-Off Analysis Method (ATAM) lets developers analyze software and system architectures with respect to quality goals.¹ The ATAM has a long pedigree—government and commercial organizations have been using it for more than 10 years—and substantial supporting documentation, including books, papers, and training courses. The ATAM normally involves two phases. Phase 1 elicits information about the architecture from the architecture team; phase 2 elicits project stakeholder needs.

The ATAM's component techniques have been developed and refined over the years. One such technique is *quality-attribute-scenario analysis*, which captures architecturally focused requirements from stakeholders. Such analysis follows a structured format defined in six parts: an explicit stimulus, a stimulus source, an explicit response, a response measure, the environment, and the artifact the scenario affects.

Another technique is a *utility tree*, which is a hierarchically organized affinity grouping of quality attribute concerns and scenarios derived from a particular project.¹

Finally, *quality attribute tactics* are architectural design primitives that ATAM evaluators look for in the architecture to quickly assess the architects' strategies.¹

The structured nature of these techniques helps ATAM teams perform elicitation and analysis of quality attribute information in a consistent manner.

Reference

1. P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, 2001.

OUR ANALYSIS METHODOLOGY

We conducted our analysis in four stages using three independent coders—analysts who assign tags (categories) to scenarios.

In stage 1, we collected Architecture Trade-Off Analysis Method reports for 2006 through 2013 and consolidated them into one dataset.

In stage 2, we conducted independent coding of a sample of the data (100 records) using a shared set of tags to assess and address variability across analysts. This stage culminated in a session reconciling coding discrepancies.

In stage 3, we conducted independent coding of all remaining records in the dataset and carried out final reconciliation.

In stage 4, we performed the categorization, summation, and analysis of the full dataset, producing the results we summarize in the main article.

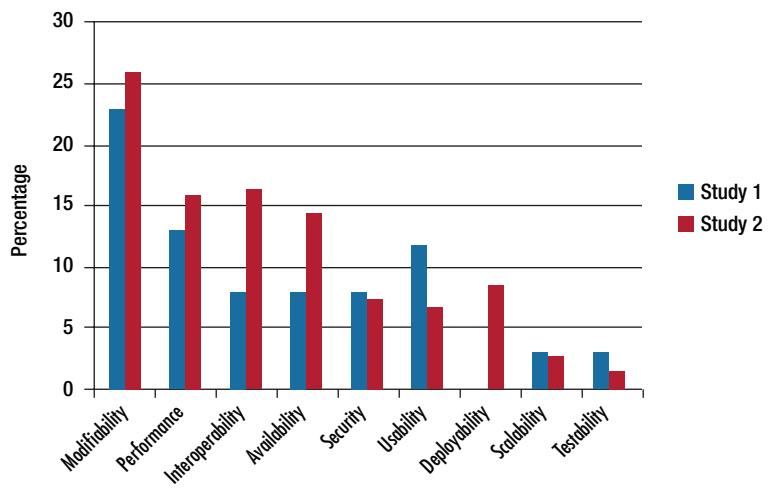


FIGURE 1. The distribution of quality attributes covered in the two studies. The top quality attribute concerns were consistent across both studies.

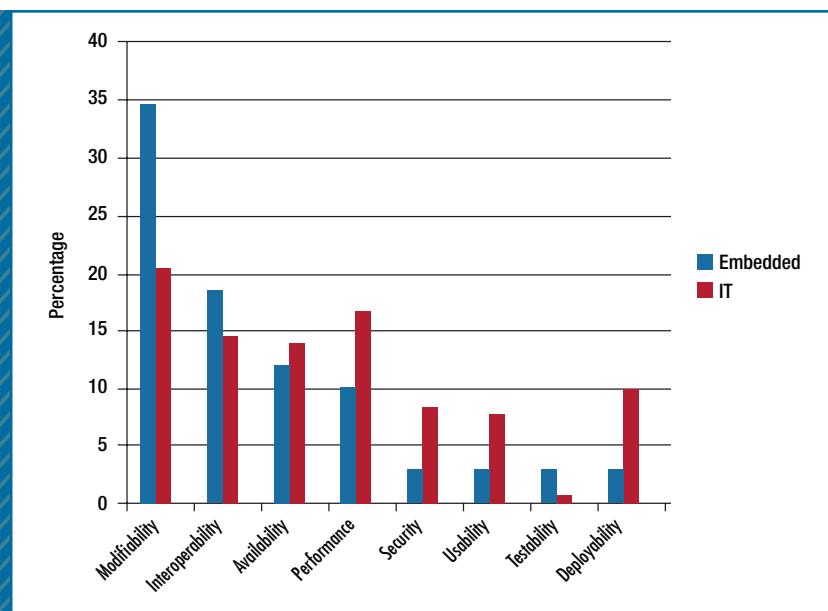


FIGURE 2. The distribution of quality attribute concerns for IT and embedded-systems projects from Study 2. Modifiability concerns were much more prominent in embedded-systems projects.

TABLE 1

Architecture Trade-Off Analysis Method project profiles.

Study	Time frame	Project type (%)			
		IT	Embedded	Government	Nongovernment
1	1999–2006	58	42	88	12
2	2006–2013	78	22	66	34

2 shows, five of the top 20 quality attribute concerns were modifiability related. So, we performed a detailed analysis of these concerns.

Table 3 summarizes and ranks the modifiability concerns. The top two concerns in Table 3 comprise 47 percent and focus on anticipating software changes. A variety of system types are represented in the ATAM data; anticipated changes included adding or replacing third-party components, adding new sensor types, replacing algorithms, and changing UI content. However, stakeholder concerns related to change weren't limited to the software application alone. A significant number of the scenarios focused on changes external to the software, such as OS replacement, network modifications, new technology integration, and hardware platform upgrades. A list such as this, derived from real project data, can help teams reason about future changes they should consider and avoid being caught off guard by an impactful, unanticipated change.

Performance, Interoperability, and Availability

Performance, interoperability, and availability concerns also occurred frequently. Because of space limitations and because the interoperability concerns found in the dataset were often similar to the modifiability concerns, we focus here on performance and availability.

Tables 4 and 5 summarize and rank the performance and availability concerns. A common theme emerges. Most scenarios focused on maintaining critical capability and consistent response times in the face of partial or major system failures (80 percent of the performance scenarios and 64 percent of

TABLE 2

The top 20 quality attribute concerns from Study 2.

Rank (by no. of scenarios)	Quality attribute	Concern
1	Modifiability	Reducing coupling
2	Performance	Latency
3	Interoperability	Upgrading and integrating with other system components
4	Modifiability	Designing for portability
5	Usability	Ease of operation
6	Availability	Fault detection
7	Interoperability	Ease of interfacing with other systems
8	Modifiability	Designing for extensibility
9	Availability	Fault recovery
10	Performance	Resource management
11	Deployability	Minimizing build, test, and release duration
12	Modifiability	Reusability
13	Availability	Preventing faults
14	Scalability	Increased processing demands
15	Security	Authorization
16	Interoperability	Resource and data sharing
17	Security	Resisting attacks
18	Deployability	Configuration or dependency management
19	Modifiability	Configurability and composability
20	Deployability	Backward compatibility or a rollback strategy

the availability scenarios). Failure scenarios included loss of hardware, software failures, network outages or failures, dramatic load increases, and synchronization failures. We suggest that besides modifiability, teams should consider these types of failure scenarios when reasoning about work they might need to plan for.

IT versus Embedded Systems

Figure 2 shows the differences in quality attribute concerns between IT and embedded-systems projects from Study 2. Modifiability accounted for almost 35 percent of the embedded-system concerns—15

percent more than any other quality attribute category. In contrast, although modifiability was the most important concern for IT projects, there wasn't as much of a gap between it and the other quality attributes. Most IT project quality attribute counts, including modifiability, ranged from 14 to 20 percent.

These numbers could reflect either the system's underlying quality attribute requirements or the design's current perceived problems. If the former is the case, this finding suggests that for embedded projects, developers must consider requirements that promote portability and

encapsulation to reduce the impact that changes have on external subsystems (such as hardware or networks) over time. For IT projects, the more even spread across quality attributes suggests a need to balance requirements considerations more broadly. Although no one-size-fits-all list of quality attribute concerns exists, we propose that these project-specific views can be useful when developers are considering how to prioritize in an IT-versus-embedded context.

Deployability: An Emerging Concern

Deployability represented 10 percent of the quality attribute concerns for

TABLE 3**A summary of the modifiability concerns.**

Concern	Percentage of scenarios
Adding, replacing, or modifying a functionality or component (for example, a third-party component, new sensor, adaptor, or algorithm)	32
Configurable software (order throughput, UI content, display, feature toggle, reporting, or simulation)	15
Operating system or registry changes	11
Replacing or changing the networked environment (for example, a new protocol or network platform)	10
Changing, adding, or replacing a model or tool capability (run-time and non-run-time)	8
Software component composition or reuse (from the core asset base)	7
Porting or integrating a new technology, device, or hardware platform (a new vehicle, a device, sensors, a dual core, and so on)	6
Changing message content and formats (translation might be required)	4
Adding or replacing services (such as messaging)	3
User- or context-driven adaptation (for example, ActiveX not allowed)	3
Replacing or upgrading middleware, a webserver, or a database	2
Minimizing merge complexity	2

TABLE 4**A summary of the performance concerns.**

Concern	Percentage of scenarios
Maintaining response time for a system capability (simulation, display, backup, run report, and so on)	25
Maintaining response time for critical capabilities with limited or intermittent resources	25
Maintaining response time while the hardware or platform changes (for example, a different machine or new CPU architecture)	21
Maintaining response time while the load or usage increases (increased load, increased service demands, increased fidelity, and so on)	9
Maintaining start-up or shut-down response time threshold	8
Monitoring system or network performance	6
Processing transactions with an increased load within an acceptable degraded range (for example, increased transactions or larger files)	4
System capability working as expected	2

the Study 2 IT projects. This quality attribute has emerged only recently (it was only in the 2006–2013 data), so we looked more closely at the related scenarios.

Table 6 lists the deployability-related scenarios from the ATAM

data. Although these concerns cross-cut many existing quality attribute areas, such as modifiability, we found that the goals driving deployability were different, as were the response measures. Scenarios 1 through 4 aimed to reduce

complexity and reduce deployment and testing cycle times. Scenarios 5 and 6 emphasized minimizing ripple effects due to change; however, the overarching stakeholder concern was change management for a complex distributed deployment environment.

TABLE 5



A summary of the availability concerns.

Concern	Percentage of scenarios
Critical operational capabilities (server, network card, and so on) remaining intact after a hardware failure (or blocked access)	34
Detecting faults, updating logs, or monitoring outage trends (for example, a software component failure or bad node)	19
Critical operational capabilities remaining intact after a software failure or bad data input	15
Restarting after a failure without losing data or state information (rollback, resynchronize, and so on)	11
Critical operational capabilities remaining intact after a network or messaging failure	9
Critical operational capabilities remaining intact after a synchronization failure (for example, clock time or data sources)	6
Visibly notifying the user if hardware or software components fail	6

TABLE 6



Deployability-related scenarios.

Scenario	Description
1	A new release is being planned; the final upgrade plan must be developed in a week (considerable time is spent on configuring specific installations).
2	An upgrade is being pushed out; the software is robust enough to handle 99 percent of errors, leaving 1 percent for manual handling.
3	All full controller builds must be completed in less than 5 minutes (even under the build farm's peak load).
4	The average time to create or build a new system-level test case is one day.
5	One team of developers modifies release version x , while another team makes architectural changes to versions x and $x + 1$ that aren't synched with the first team's changes. The first team integrates the modifications into version $x + 1$ within eight weeks.
6	A new version of the spec isn't compatible with previous versions. The software application and adapter should be backward-compatible, besides supporting the new version.

Reducing deployment and automated testing cycle time is a focus for books on this topic, such as Jez Humble and David Farley's *Continuous Delivery*.⁸ These scenarios could give teams a starting place for reasoning about ways to enable their deployment goals.

Using the Data

The data and concrete examples we present here can help agile teams determine which qualities to focus on when working on early increments and planning future increments. Teams can incorporate high-ranking

quality attributes and quality attribute concerns (including examples) as checklists at designated points during the incremental life cycle.⁹ This will help them quickly assess whether they need to consider a specific quality attribute concern to support an anticipated business need.

Evolving a system's architecture in an agile project presents unique challenges. We suggest weaving quality attribute concerns into the life cycle that a project team is using; this provides natural opportunities to incorporate the considerations we present here. For example,

during agile retrospective meetings, the team might brainstorm about quality-attribute-related deficiencies that surfaced in the prior increment. During sprint planning, the architect could propose addressing a quality attribute concern and, if the product owner agrees, add it to the sprint backlog. This data also provides ammunition for the agile team lead in making the business case to the product owner by justifying the importance of dealing with such concerns architecturally.

Table 7 can also serve as a checklist for team members reflecting on

TABLE 7

Quality attribute concerns grouped by quality attribute and ranked according to the number of scenarios in which they appeared.

Quality attribute	Concern
Modifiability	Designing for portability
	Designing for extensibility
	Reusability
	Configurability and compositability
	Increasing cohesion
	Adding or modifying functionality
Performance	Latency
	Resource management
	Throughput
	Performance monitoring
	Initialization
	Accuracy
Interoperability	Upgrading and integrating with other system components
	Ease of interfacing with other systems or components
	Resource and data sharing
	Data integrity
	Compliance with standards and protocols
Availability	Fault detection
	Fault recovery
	Preventing faults
	Transaction auditing and logging
	Graceful degradation

quality attribute concerns. Quality attribute concern data can help agile developers create review questions for reasoning about design trade-offs, such as, “Did we negatively impact another architecturally significant quality attribute concern with this design decision?” Jungwo Ryoo and his colleagues described an example of applying quality-attribute-concern and design-tactic data in this way.¹⁰

Our ultimate aim is to provide systematic support for iteratively addressing architectural concerns in agile projects. Techniques for efficiently identifying, prioritizing, and allocating architecture tasks to iterations and measuring their ongoing effects on project velocity are fundamental to this work’s success. Regarding future applicability, data such as this

could also serve as input to decision-support-system models¹¹ that provide automated support for reasoning about quality attribute tradeoffs. 

Acknowledgments

This article is based on research funded and supported by the US Department of Defense under contract FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. DM-0002047

References

- N. Brown et al., “Managing Technical Debt in Software-Reliant Systems,” *Proc. FSE/SDP Workshop Future of Software Eng. Research*, 2010, pp. 47–52.
- S. Ambler, *Disciplined Agile Delivery: A Practitioner’s Guide to Agile Software Delivery in the Enterprise*, IBM Press, 2012.
- D. Leffingwell, *Scaling Software Agility: Best Practices for Large Enterprises*, Addison-Wesley, 2007.
- S. Bain, *Emergent Design: The Evolutionary Nature of Professional Software Development*, Addison-Wesley, 2008.
- S. Bellomo, R. Nord, and I. Ozkaya, “A Study of Enabling Factors for Rapid Fielding: Combined Practices to Balance Tension between Speed and Stability,” *Proc. 2013 Int’l Conf. Software Eng.*, 2013, pp. 982–991.
- I. Ozkaya et al., “Making Practical Use of Quality Attribute Information,” *IEEE Software*, vol. 25, no. 2, 2008, pp. 25–33.
- P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, 2001.
- J. Humble and D. Farley, *Continuous Delivery*, 1st ed., Addison-Wesley, 2010.
- F. Bachmann et al., “Integrate End to End Early and Often,” *IEEE Software*, vol. 30, no. 4, 2013, pp. 9–14.
- J. Ryoo, P. Laplante, and R. Kazman, “In Search of Architectural Patterns for Software Security,” *Computer*, vol. 42, no. 6, 2009, pp. 98–100.
- N. Ernst and I. Gorton, “Using AI to Model Quality Attribute Tradeoffs,” *Proc. IEEE 1st Int’l Workshop Artificial Intelligence for Requirements Eng. (AIRE 14)*, 2014, pp. 51–52.



Selected CS articles and columns
are also available for free at
<http://ComputingNow.computer.org>

IEEE  computer society
NEWSLETTERS
Stay Informed on
Hot Topics

COMPUTING NOW
TRAINING SPOTLIGHT
TRANSACTIONS CONNECTION
WHAT'S NEW BUILD YOUR CAREER
IN COMPUTER LIBRARY NEWS FLASH
CS CONNECTION
DIGITAL LIBRARY NEWS FLASH
CONFERENCE CONNECTION
WHAT'S NEW IN COMPUTER
TRANSACTIONS CONNECTION
S MEMBER CONNECTION
COMPUTING NOW
TRAINING SPOTLIGHT
CS MEMBER CONNECTION



computer.org/newsletters

ABOUT THE AUTHORS



STEPHANY BELLOMO is a senior member of the technical staff on the Software Engineering Institute's Architecture Practices team. Her research interests include architecting for agility, technical debt, and DevOps and continuous delivery. Bellomo received an MS in software engineering from George Mason University. She's a senior member of the IEEE Computer Society. Contact her at sbellomo@sei.cmu.edu.



IAN GORTON is a senior member of the technical staff on the Software Engineering Institute's Architecture Practices team, where he investigates issues related to software architecture at scale. His research includes designing large-scale data management and analytics systems and understanding the inherent connections and tensions between software, data, and deployment architectures. Gorton received a PhD in computer science from Sheffield Hallam University. He's a senior member of the IEEE Computer Society. Contact him at i.gorton@neu.edu.



RICK KAZMAN is a professor at the University of Hawaii and a principal researcher at the Software Engineering Institute. His primary research interests are software architecture, design and analysis tools, software visualization, and software-engineering economics. Kazman received a PhD in computational linguistics from Carnegie Mellon University. He's a senior member of the IEEE Computer Society. Contact him at kazman@sei.cmu.edu.

Call for Articles

IEEE Software seeks practical, readable articles that will appeal to experts and nonexperts alike. The magazine aims to deliver reliable information to software developers and managers to help them stay on top of rapid technology change. Submissions must be original and no more than 4,700 words, including 200 words for each table and figure.

Author guidelines: www.computer.org/software/author.htm
Further details: software@computer.org
www.computer.org/software

IEEE Software

Quiz 2

- 1- What are the metrics used to validate and measure the performance of an application?
 - Quality attribute are part of an application's nonfunctional requirements. These requirements capture the many sides of how the functional requirements of an application are achieved. In order to achieve a specific need, a detailed quality attribute requirement must be specified.
- 2- Is it sufficient for an application to meet the average throughput to guarantee efficiency? Justify your answer.
 - No, Throughput is a measure of the amount of work an application must perform in unit time. Therefore, it will be ineffective for an application to meet average throughput. Moreover, this will cause the application to do little work so it would not really be sufficient.
- 3- Why is security considered as an important quality attribute? What are the main requirements?
 - A quality attribute is defined as "an application's requirements in terms of scalability, availability, ease of change, portability, usability, performance, and so on." One of the general software quality attributes is security, security helps in the scalability of the system which relates to different application characteristics; also, it provides authentication for users which extends to handling third party tools.
 - A functional security requirement is something that describes functional behavior that enforces security. It can be directly tested and observed. Requirements that have things to do with access control, data integrity, authentication, and wrong password lockouts fall under functional requirements.
- 4- Distinguish between scale up and scale out mechanisms. Which approach delivers better performance?
 - Scale up works well if an application is multithreaded, or multiple single threaded process instances can be executed together on the same machine. However, towards the end it will consume additional memory and some resources, the reason is that processes sometimes need multiple resources at once.
 - Scale out works well if there is little or ideally no additional work required managing the distribution of requests amongst the multiple machines. Our purpose is to keep every and each machine equally busy because if one of the machines if loaded to its fullest, more hardware would be wasted.
 - If there is little work then scale out will provide better performance; however, if there is a lot of work to be done in an application (multithreaded application), then the scale up approach will deliver a better performance. In my opinion, I would choose scale out since there is a better chance for success and productivity.
- 5- How are design trade-offs made in the case of conflicting quality attributes?
 - A design that satisfies one quality attribute may have some effect on another attribute. in order to achieve greater availability, an available application may have to trade-off lower performance. It isn't plausible to fully satisfy all requirements. Sometimes, an architect ay need to trade-off something for the higher quality of another thing.

Quiz #3

1- Define Architectural drivers.

- Architectural drivers drive and shape the architecture. These drivers include a design purpose, quality attributes, primary functionality, architectural concerns, and constraints. The drivers are critical for the success of the system.

2- How does the primary functionality influence the overall functioning of the system?

- Primary functionality is the functionality that is critical to achieve the business goals that motivate the development of the system. It manages to influence the overall system is that it implies a high level of technical difficulty. It requires the interaction of many architectural elements; also, it promotes modifiability and reusability.

3- Explain the techniques used to specify and prioritize quality attributes.

- The best way to specify and prioritize quality attributes is by using a set of scenarios. By using a set of scenarios an architect would save more time on multiple tasks such as defining terms and would be able to pair a stimulus with a response for his/her project.

4- What are the types of Architectural concerns?

- **General concerns:** These are “broad” issues that one deals with in creating the architecture.
- **Specific concerns:** These are more detailed system-internal issues such as exception management, dependency management, configuration, logging, authentication, authorization, caching, and so forth that are common across large numbers of applications
- **Internal requirements:** These requirements are usually not specified explicitly in traditional requirement documents, as customers usually seldom express them.
- **Issues:** These result from analysis activities, such as a design review.

5- Give examples of technical and business constraints.

- A technical constraint is the use of open source technologies; however, a business constraint is that the system must obey the Sarbanes-Oxley Act that is must be delivered by December 15.

Quiz #4

1. What are the levels of design?

- Architectural Design.
- Element Interaction Design.
- Element Internals Design.

2. Define Architectural Design Concepts.

- A decision is architectural if it has non-local consequences and those consequences matter to the achievement of an architectural driver. Architectural Design involves making decisions, working with available skills and materials, to satisfy requirements and constraints. In architectural design, we make decisions to transform our design purpose, requirements, constraints, and architectural concerns. Thus, Architectural Design is a key step to achieving your product and project goals

3. Differentiate between Reference Architectures and Architectural styles.

- Reference architectures are blueprints that provide an overall logical structure for particular types of applications or a reference model mapped onto one or more architectural patterns. However, architectural styles are types of components and connectors in a specified topology that are useful for structuring an application either logically or physically which tend to be popular in academia.

4. What criteria should be considered while selecting externally developed families?

- The criteria that we should consider are:
 - Problem that it addresses.
 - Cost.
 - Type of license.
 - Support.
 - Learning curve.
 - Maturity.
 - Popularity.
 - Compatibility and ease of integration.
 - Support for critical quality group.
 - Size.

5. How are patterns and tactics implemented considering their abstract nature?

- Patterns are implemented by providing models on how to physically structure the system to deploy it which are used to satisfy the quality attributes such as availability, performance, and security. However, tactics are implemented by providing or categorizing design decisions that influence and mediate the control of quality attributes response while focusing of a single quality attributes response because tactics provide a top-down way of thinking about design patterns.

1. How does a Rich Client Application differ from a regular web application?
 - rich client application is decomposed into three layers: the presentation layer, business layer and data layer. The presentation layer usually contains UI and presentation logic components; the business layer usually contains business logic, business workflow and business entity components; and the data layer usually contains data access and service agent components. While a regular web application may or may not have professionalism in its components.
2. In which scenarios is the distributed deployment pattern preferred over nondistributed?
 - In front of a large audience. The distributed deployment pattern is preferred if what we need is to separate the layers of an application on different physical tiers because it allows us to configure the application servers that host the various layers in order to best meet the requirements of each layer.
3. Which kind of reference architecture must be used when you aim to have high portability of UI?
 - Layered reference architecture such as Java EE because its structures and respective elements and relations provide templates for concrete architectures in a particular domain of UI
4. What are the limitations of Rich Internet application?
 - Access to local resources can be limited, because the application may run in a sandbox.
 - Loading time is non-negligible.
 - Plug-in execution environments may not be available in all platforms.
5. Explain Load-Balanced Cluster Pattern.
 - In the Load-Balanced Cluster pattern, the application is deployed on multiple servers that share the workload. Client requests are received by a load balancer, which redirects them to the various servers according to their current load. The different application servers can process several requests concurrently, which results in performance improvements.

1. What are the differences between the layers pattern and domain object pattern?

- The layers pattern defines two or more layers for the software under development, where each layer has a distinct and specific responsibility. To make the layering more effective, the interactions between the layers should be highly constrained. While the domain object pattern encapsulates each distinct, nontrivial piece of application functionality in self-contained building block called a domain object.

2. What are the commonalities between data mapper and proxy patterns?

- The proxy frees both the client and the subjects from implementing component-specific housekeeping functionality. It also transparent to clients whether they are connected with the real subject component or its proxy, because both publish an identical interface. A data mapper is a mediator that moves data between an object-oriented domain model and relational database. A client can use the data mapper to store or retrieve application data in the database. The data mapper performs any needed data transformations and maintains consistency between the two representations. Both need to insulate applications, transfer data, may be impractical to access the services of a component directly.

3. What problem does interface partitioning solve?

- It separates the explicit interface of a module from its implementation, and exports the explicit interface to the clients of the module, but keep its implementation private.

4. What are the differences between increase cohesion and reduce coupling tactics?

- In increase cohesion, If the responsibilities A and B in a module do not serve the same purpose, they should be placed in different modules. This may involve creating a new module or moving a responsibility to an existing module. While in reduce coupling, it can either encapsulate, use an intermediary, restrict dependencies, refactor, or abstract common services.

5. What strategy would you use in your architecture to recover from attacks?

- I would use the Maintain Audit Trail, which will help in keeping a record of user and system actions and their effects, to help trace the actions of, and to identify, an attacker.



SE 339 - SOFTWARE ARCHITECTURE

MICROSERVICES

Lotfi ben Othmane

Iowa State University

Fall 2017

LAST LECTURE

What is a software architecture?

1. Partition the system into components considering requirements and constraints
2. Assigning responsibility to the components of the system
3. Addressing structural issues of composing elements of a system

GOAL

What is a microservice?

RESOURCES

- Microservices – Flexible software architecture

by Eberhard Wolff

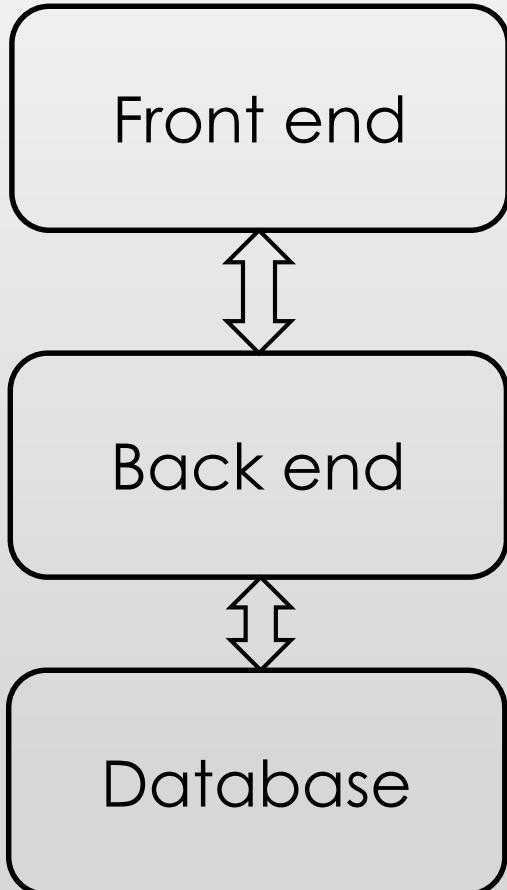
<http://microservices-book.com/content.html>

- Microservices -- A definition of this new architectural term

by James Lewis and Martin Fowler

<https://www.martinfowler.com/articles/microservices.html>

MONOLITHIC APPLICATIONS



The code may be organized into modules

1. Dependency problem - Adding new features or even bug fixes require change to many components and redeployment of all the application
2. Interoperability problem - Organization is based on technology
 - Different teams work with different technologies
3. Scalability problem – Should applies to all the application
4. Resilience problem – Fail affects all the application
5. Consistency problem – Shared data needs to be consistent using transactions management protocol.

MONOLITHIC APPLICATIONS

There has been development of architecture styles and techniques to address the problems

- Web services for interoperability
- Transaction management with EJB
- Load balancing
- Etc.

Microservices was first discussed in a workshop of software architects, Venice, 2011

MICROSERVICES

An approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.

These services are built around business capabilities and independently deployable by fully automated deployment machinery.

UNIX PHILOSOPHY

- One program should fulfill only one task
- Programs should be able to work together
- A universal interface should be used—e.g., text stream

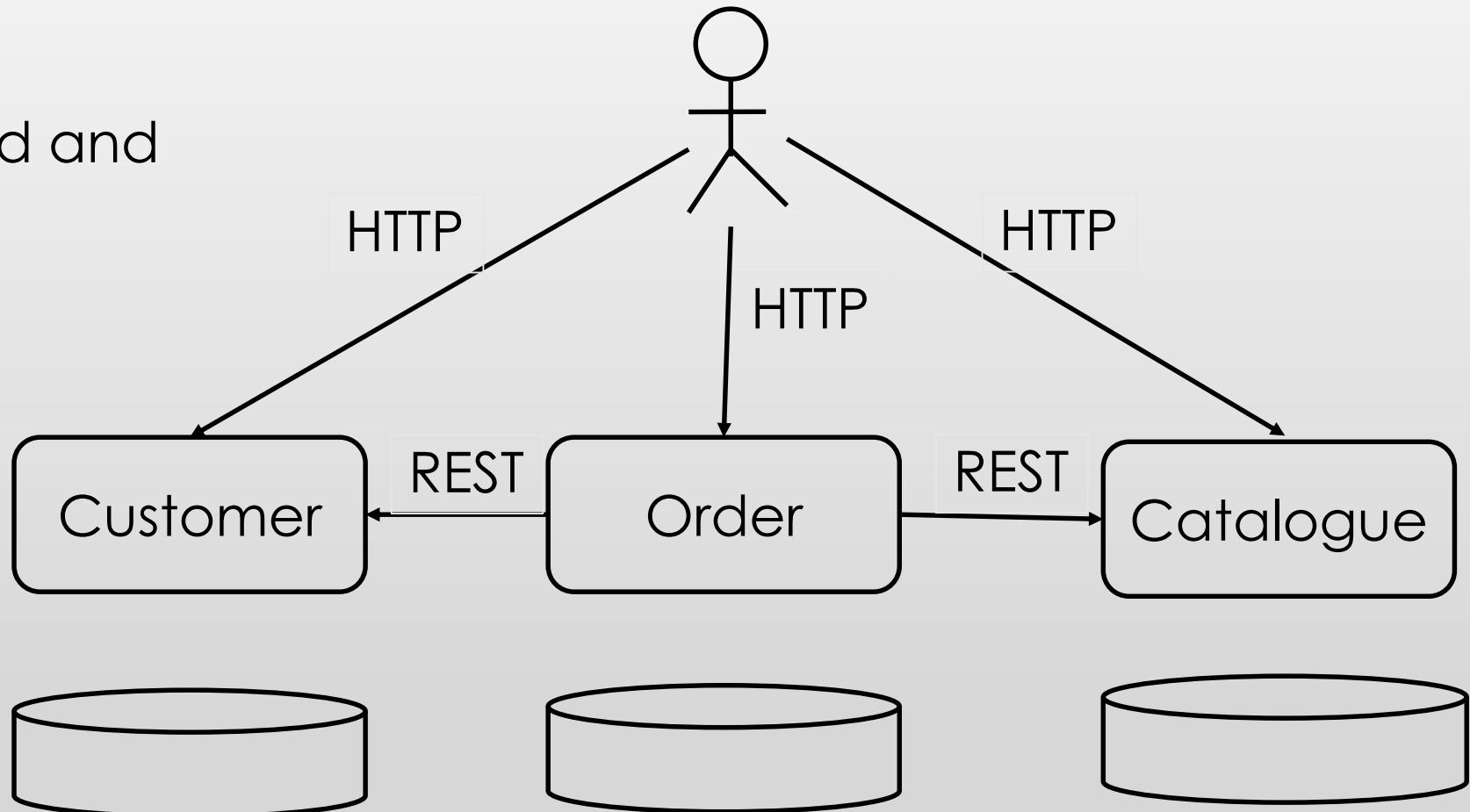
DEPENDENCY PROBLEM

Solution

- The boundaries is based on business context not technology
 - No separation between front end and back end
- Orchestration is implemented in microservices in not in infrastructure or communication
 - Threads and workflows are managed by microservices,
- Each microservice has a clear interface
- Each microservice manages its data
- Microservices run on independent processes
 - Could be **deployed independently**

DEPENDENCY PROBLEM

No separation
between front end and
back end



Chapter 13 of book microservices

DEPENDENCY PROBLEM

- One of the main challenges in architecture is to identify cut-points
 - Identify independent components
- Principle 1: Split is based on business capabilities boundaries
- Principle 2: Future changes should require updates to one microservice—minimize propagation

CRITERIA FOR CREATING NEW MICRO SERVICES

- Introduction of different data models
- Mixing of synchronous and asynchronous communication
- Incorporating additional services
- Different load scenarios for different aspects of the service

INTEROPERABILITY PROBLEM

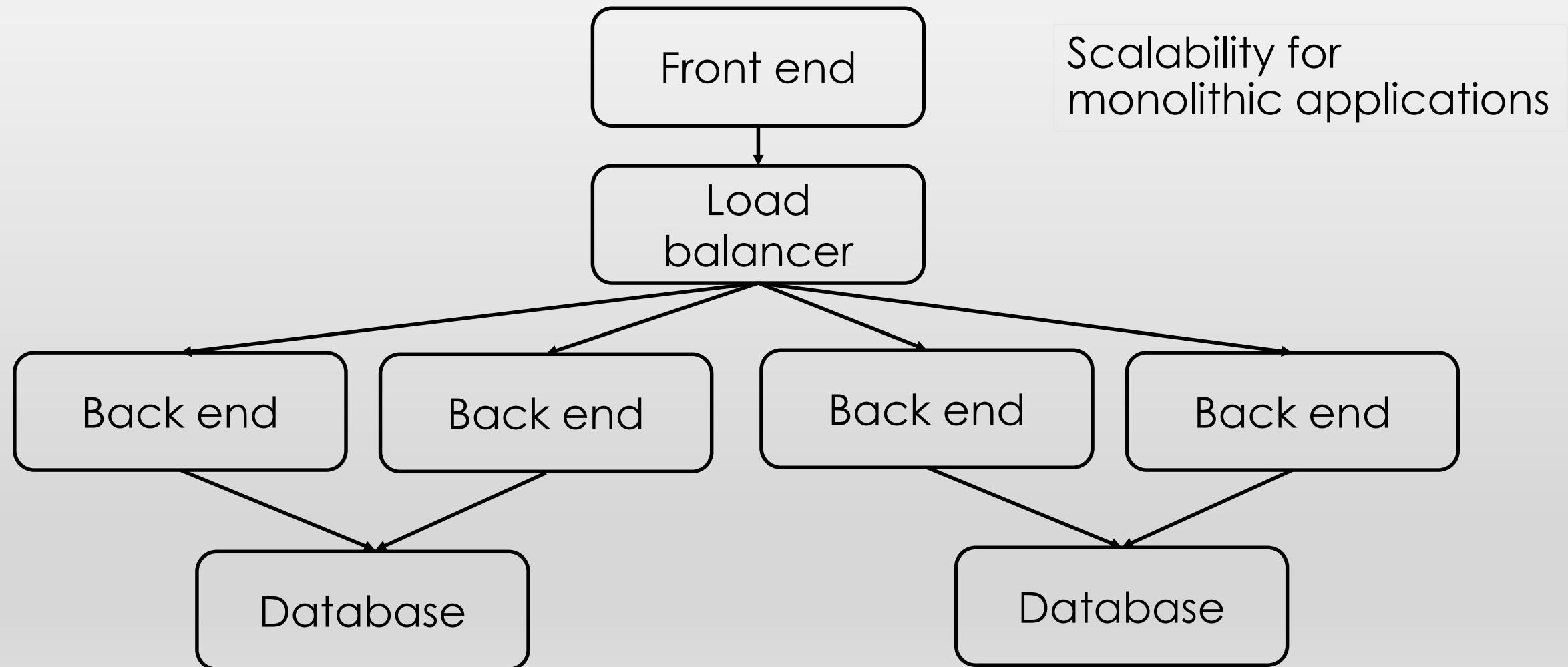
Web service solution

- Web services addresses this by allowing communication between web services using SOAP or Jason
- Web services support transactions for consistency – May be needed in some contexts
- Web services run on one process

Microservice solution

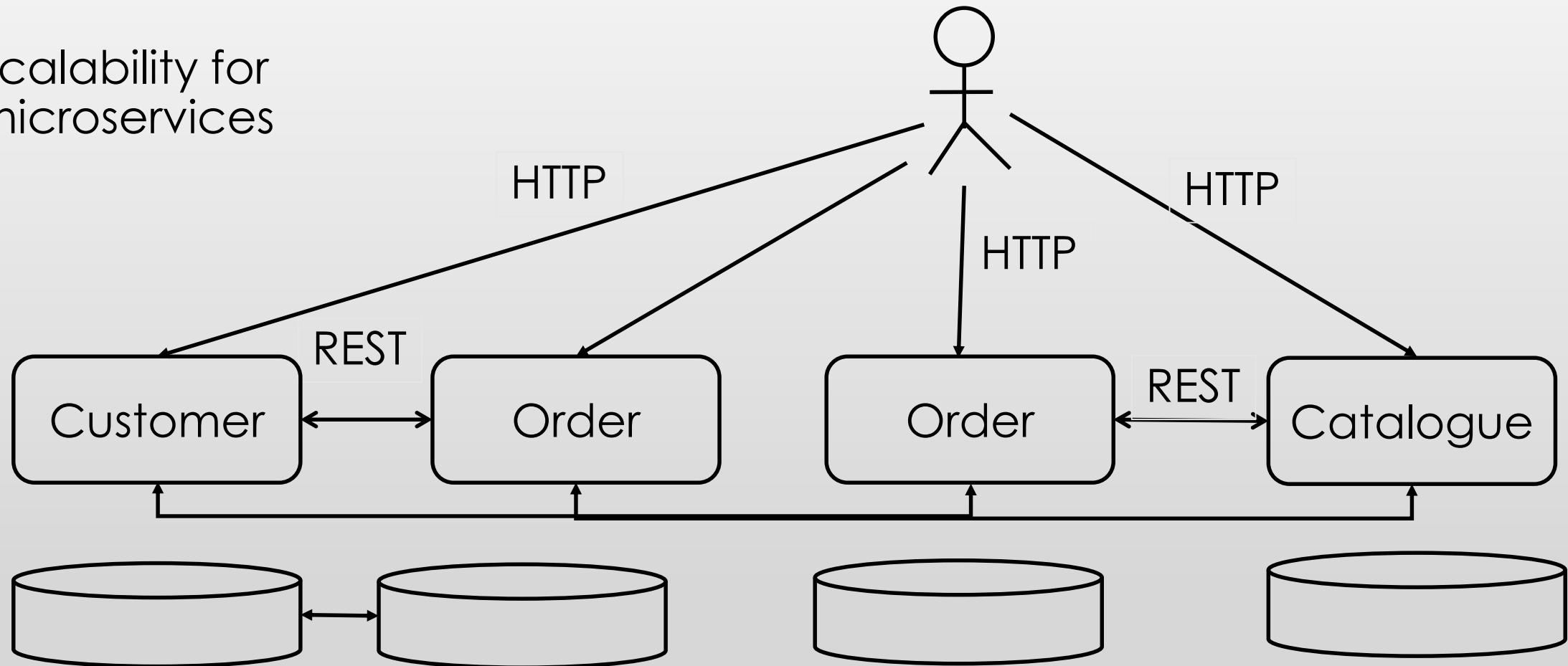
- Use lightweight communication mechanisms such as REST and RPC
- Compensation operation for inconsistency

SCALABILITY PROBLEM



SCALABILITY PROBLEM

Scalability for
microservices



RESILIENCE PROBLEM

Potential failure

- Service might have bugs – crash
- Service may become unavailable due to hardware or network problem
- Service may become slow to respond

→ Plan for eventual failure

- How should/must the microservice behave in the case of failure of each of the dependencies?
- Use of circuit breaker to handle failure: monitor microservices and trigger correction in case of failure

CONSISTENCY PROBLEM

Microservices related-characteristics

- Asynchronous communication
- No shared data – or minimum shared
- No management of service states for consistency

→ Implement logic to detect inconsistency and to trigger corrective operations

- Do not use central system for consistency

CHARACTERISTICS

- Organized around business capabilities
- Products not Projects – developers support the product
- Smart endpoints and dumb pipes – request-logic-response
- Decentralized governance – technology choices
- Decentralized data Management
- Infrastructure automation – continuous development
- Design for failure – consider failure
- Evolutionary design – rewriting a component without affecting its collaborators

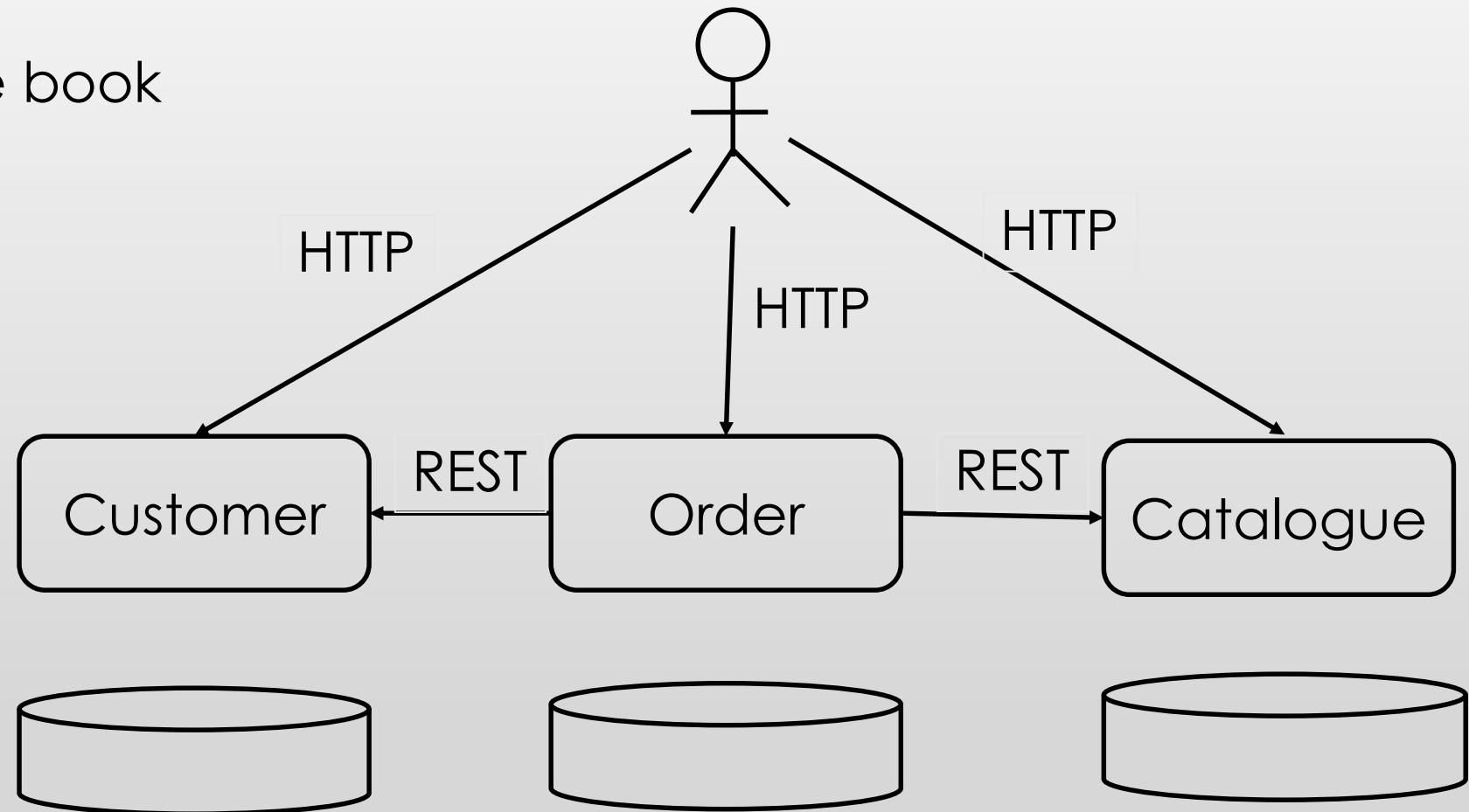
<https://www.martinfowler.com/articles/microservices.html#footnote-etymology>

BUT

Krutchén says: “The life of a software architect is a long (and sometimes painful) succession of sub-optimal decisions made partly in the dark.”

ASSIGNMENT

Chapter 13 of the book



TECHNOLOGIES

- Use of Spring Framework – Java
- Use of HSQL database
- Spring Cloud Netflix for routing, discovery, resilience, etc.
- Deployment using Docker

MICROSERVICES EXAMPLE

Download the example from:

<https://github.com/ewolff/microservice>

We are working on an assignment based on the example.

Thank you.

Next class is on UML



SE 339 - SOFTWARE ARCHITECTURE

UNDERSTANDING SOFTWARE ARCHITECTURE

Lotfi ben Othmane

Iowa State University

Fall 2017

I will be in a conference next week

I am checking options for replacement

GOAL

Define software architecture

What does software architecture mean? (15 min)

1. Partition the system into components considering requirements and constraints
2. Assigning responsibility to the components of the system
3. Addressing structural issues of composing elements of a system
4. All the above
5. I have better idea

Quiz - 4 questions (15 + 10 min)

The criteria for partitioning a system are:

1. Consider requirements and constraints
2. Minimize dependencies between components
3. Assign clear responsibilities for each component of the software
4. I have another idea

Components communicate through

1. Method calls
2. Synchronization mechanisms, e.g., in threads
3. Triggered when an event is received
4. None of the above
5. All of the above

Which ones of the characteristics of client/server pattern below is correct?

1. Asynchronous request-reply communication
2. Support one or many clients
3. Provide mechanisms for high performance
4. Maintain a state of connected clients
5. Provide mechanisms for discovering services
6. Provide mechanisms for authorizations
7. Provide mechanisms for error handling
8. Provide mechanisms for interoperability

Architecture views:

1. Describe the architecture from given viewpoints—e.g., stakeholders
2. Are: logical view, process view, physical view, and development view
3. Are: module view, components and connector view, and allocation view

- Can you execute the abstract description of architecture?
- What do architects do with the multitude of frameworks/COTS that implement the patterns they need to make their architecture concrete?
- Why abstract architecture needs to be rectified as concrete implementation progresses?

10 min each

Questions

What is a software architect? (10 min)

1. The team manager
2. The senior developer who make decision about the use of frameworks
3. A person in the team that has the following skills: liaison, software engineering, has deep technology knowledge, and cautious about risks.

Krutchén says: “The life of a software architect is a long (and sometimes painful) succession of sub-optimal decisions made partly in the dark.” (10 min)

1. Do you agree with the statement?
2. Why the decision are sub-optimal?
3. Why the decision are made in the dark?

INTRODUCTION TO UML

Lotfi ben Othmane

Iowa State University

Fall 2017

Some slides are taken from a
lecture of Majid Ali Khan

Spring 2005

Several diagrams are taken
from UML Distilled
By Martin Fowler

GOAL

Learn what is UML and
how to use the diagrams that describe the architecture

MODELING

- Describing a system at a high level of abstraction
- Is it necessary to model software systems?

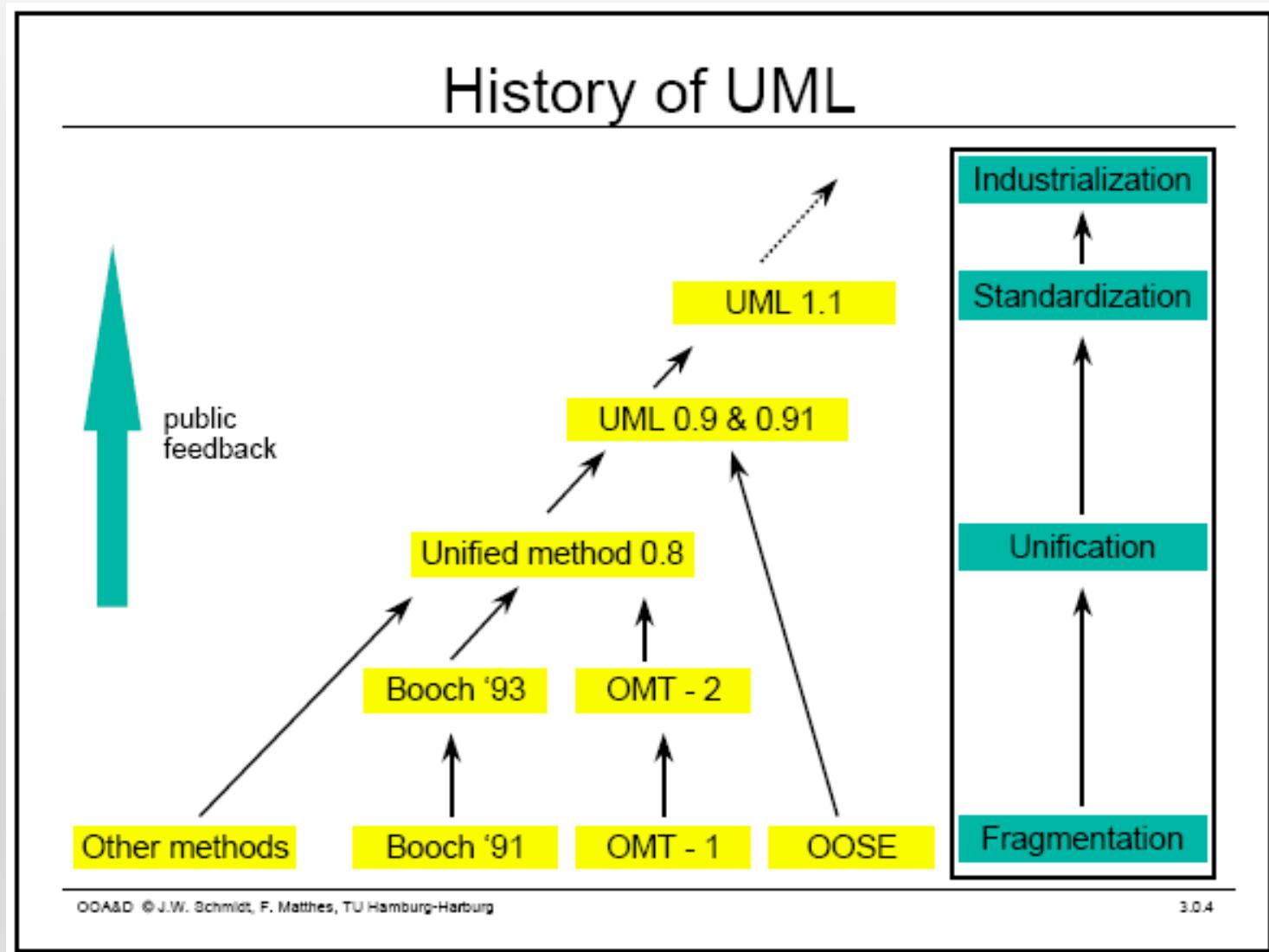
WHAT IS UML?

- UML stands for “Unified Modeling Language”
- It is a industry-standard graphical language for specifying, visualizing, constructing, and documenting the artifacts of software systems
- The UML uses mostly graphical notations to express the OO analysis and design of software projects
- Simplifies the complex process of software design

WHY UML FOR MODELING?

- Use graphical notation to communicate more clearly than natural language (imprecise) and code (too detailed)
- Help acquire an overall view of a system
- UML is *not* dependent on any one language or technology
- UML moves us from fragmentation to standardization

HISTORY OF UML



TYPES OF UML DIAGRAMS

Most used diagram types:

- Use Case Diagram
- Class Diagram
- Sequence Diagram
- Components diagram
- Collaboration Diagram
- State Diagram
- Activity diagram
- Deployment diagram

USE CASE DIAGRAM

- Mainly used to capture user requirements
- Provides an external view of the system
- Used to describe user scenarios
- Capture a generalized description of how a system will be used.
- Provides an overview of the intended functionality of the system

LEVELS OF USE CASES

- Goal of use case: How the actors use the system (from customer perspective)
- System use case –interactions with the system
- Business use cases – how business responds to events
- Feature set shall not be mixed with use cases
 - A feature could be a use case, a set in a use case, or a variant behaviors

USE CASE DIAGRAM

- **Actors:** A role that a user plays with respect to the system.
- **Use case:** A set of scenarios that describe the interactions of the actors with the system.



- **System boundary:** Rectangle shape representing the boundary of the system.

USE CASE DIAGRAM

- Association: communication between an actor and a use case.



- Generalization: relationship between a general use case and a special use case



USE CASE DIAGRAM

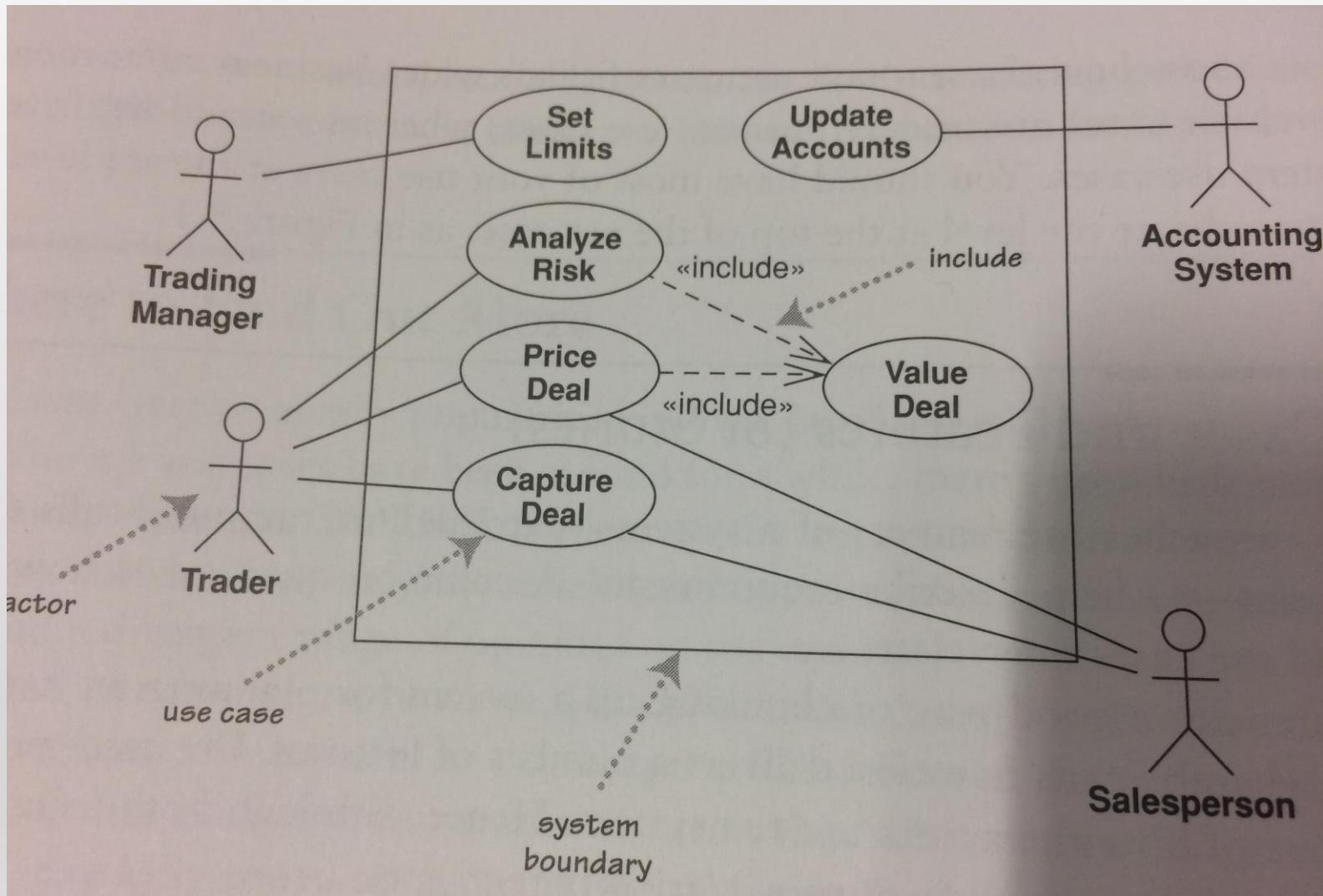
- Include: a chunk of behavior is similar across more than one use case.

<<include>>
----->

- Extend: use case add behavior to the base use case.

<<extend>>
----->

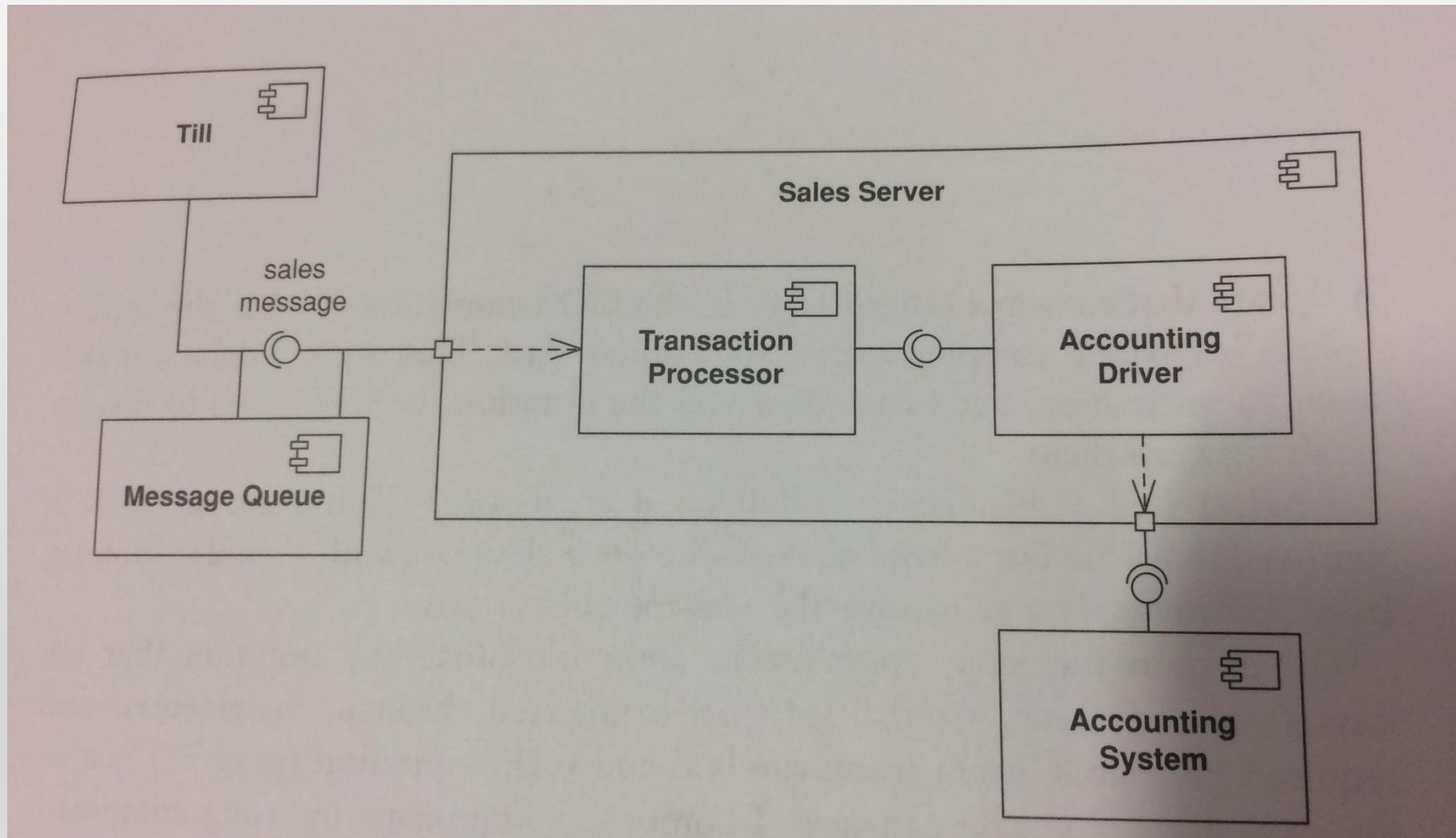
EXAMPLE OF USE CASE



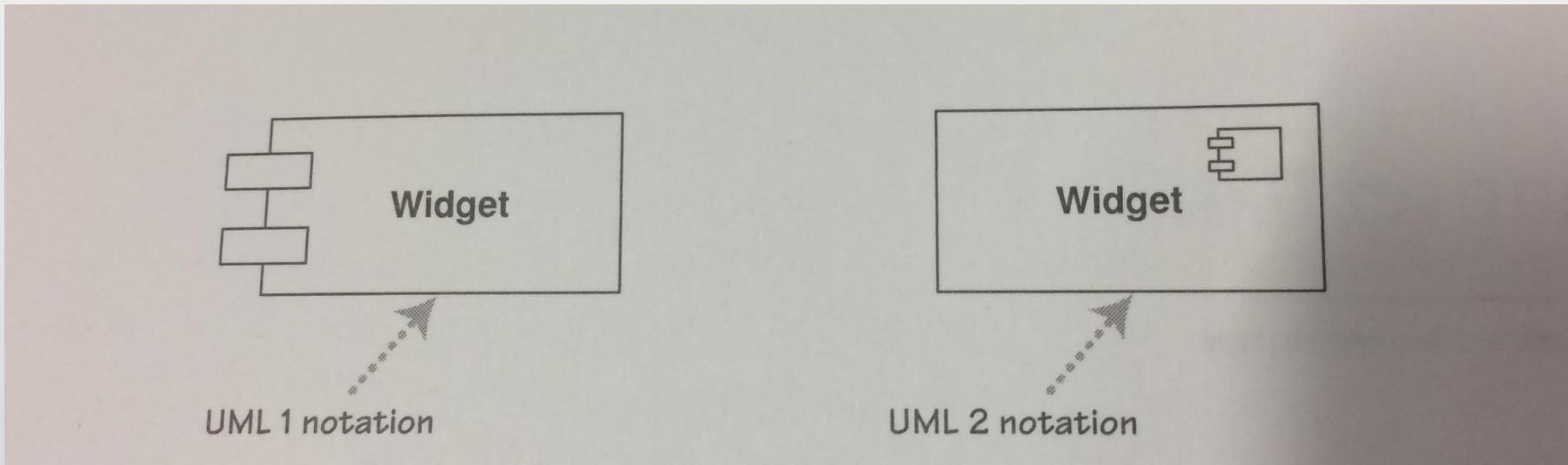
COMPONENT DIAGRAM

- Components are independent pieces that compose the software from customers perspectives
 - Customers can upgrade each component separately
 - Old components can work with new components seamlessly
 - Support mix and match components of different providers
- A component could be a class or a set of classes
- Your goal is to identify the interfaces of and data exchanges among the independent components of software

COMPONENT DIAGRAM



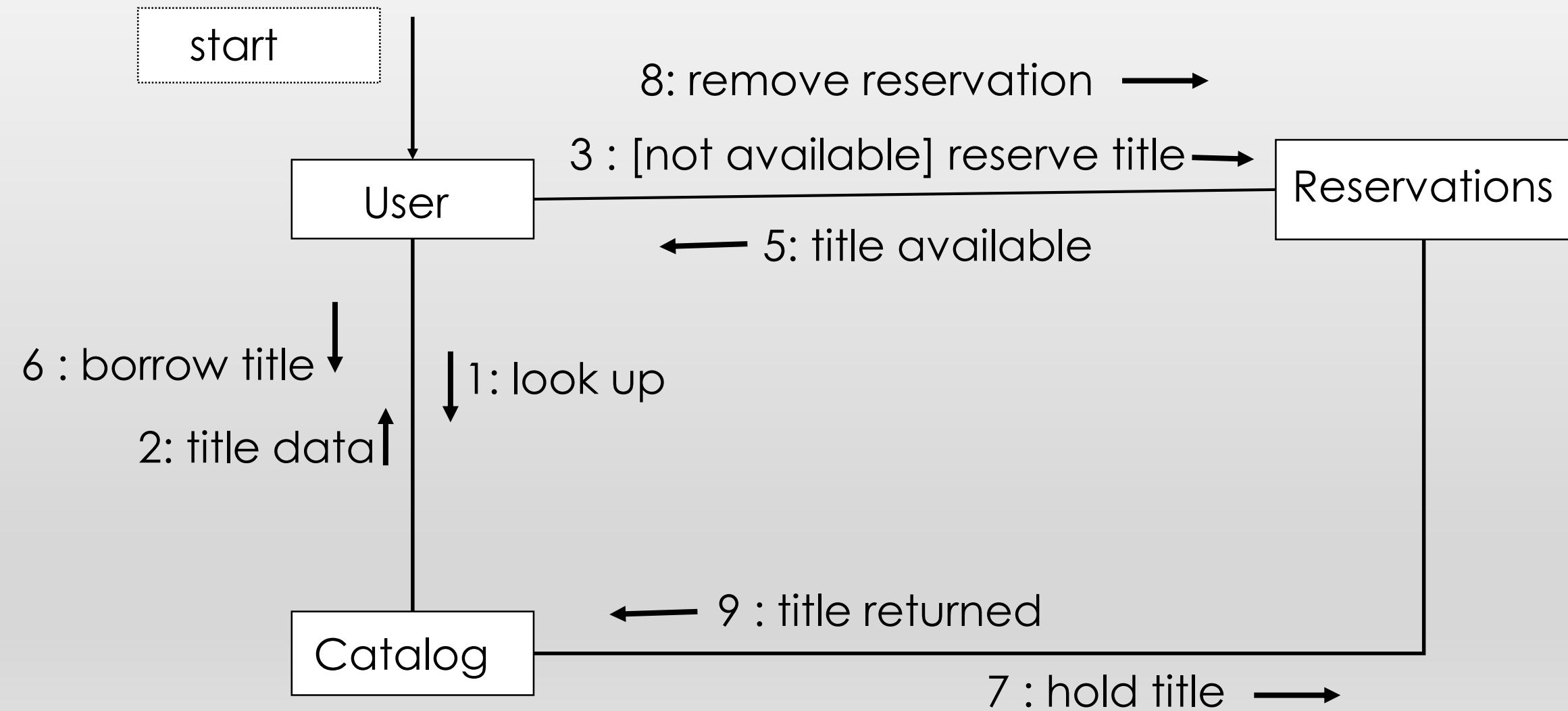
COMPONENT DIAGRAM



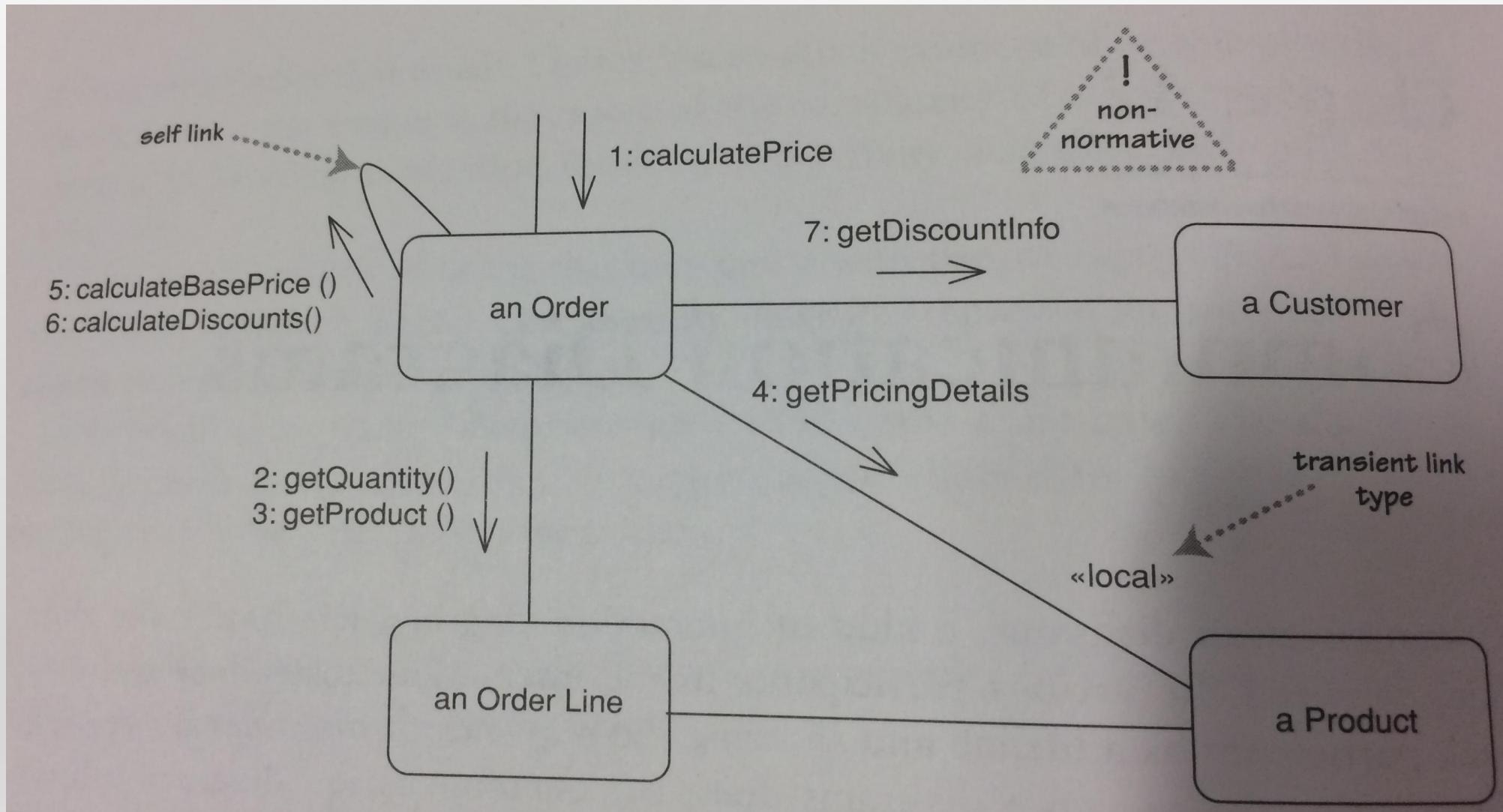
COMMUNICATION DIAGRAMS

- Shows the data links between the various participants in the interaction
- The objects are listed as rectangles and the arrows indicate the messages being passed
- The numbers next to the messages show the sequence of the messages as they are passed between the objects
- The diagram is used to show how the components cooperate for a given use case -> It could be used to validate the component diagram

EXAMPLE OF COMMUNICATION DIAGRAM



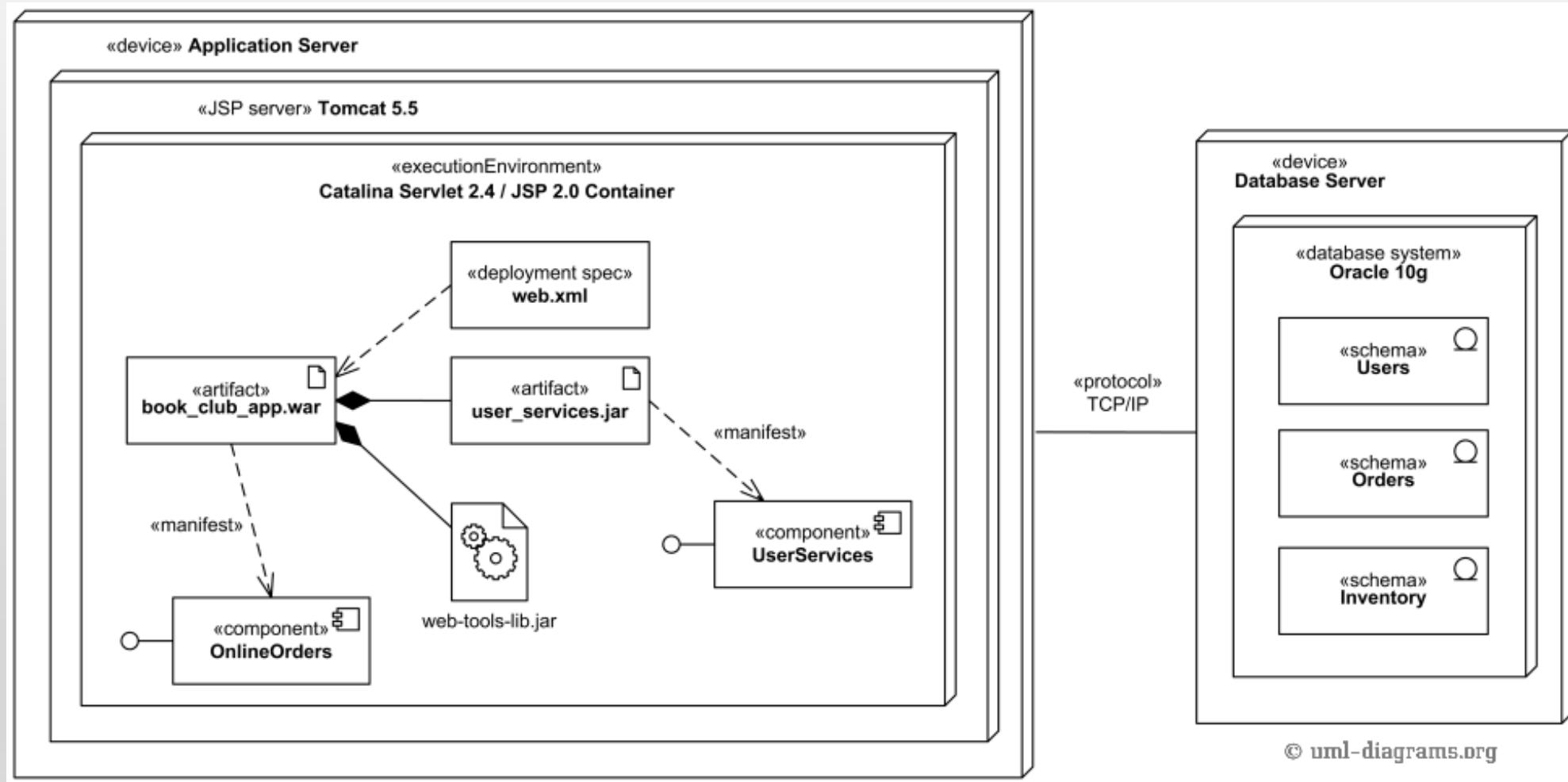
EXAMPLE OF COMMUNICATION DIAGRAM



DEPLOYMENT DIAGRAM

- Deployment diagram models the **allocation** of artifacts such as software and files to **nodes** such as devices. It also models the communication method such as RMI, REST, SOAP, HTTP.

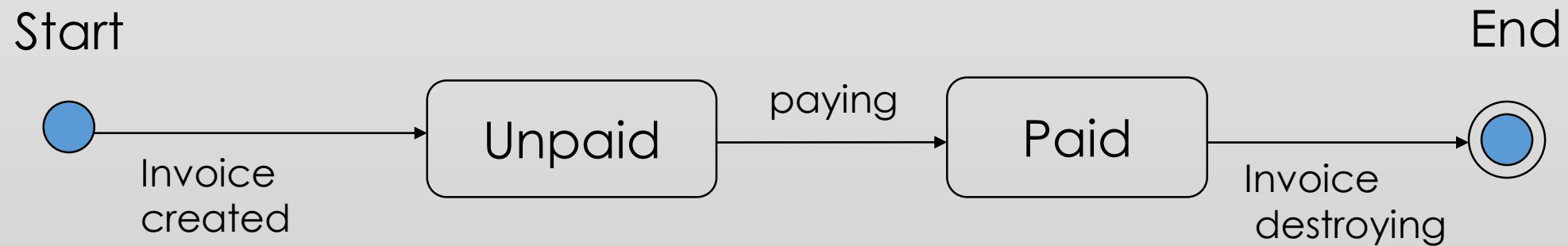
DEPLOYMENT DIAGRAM



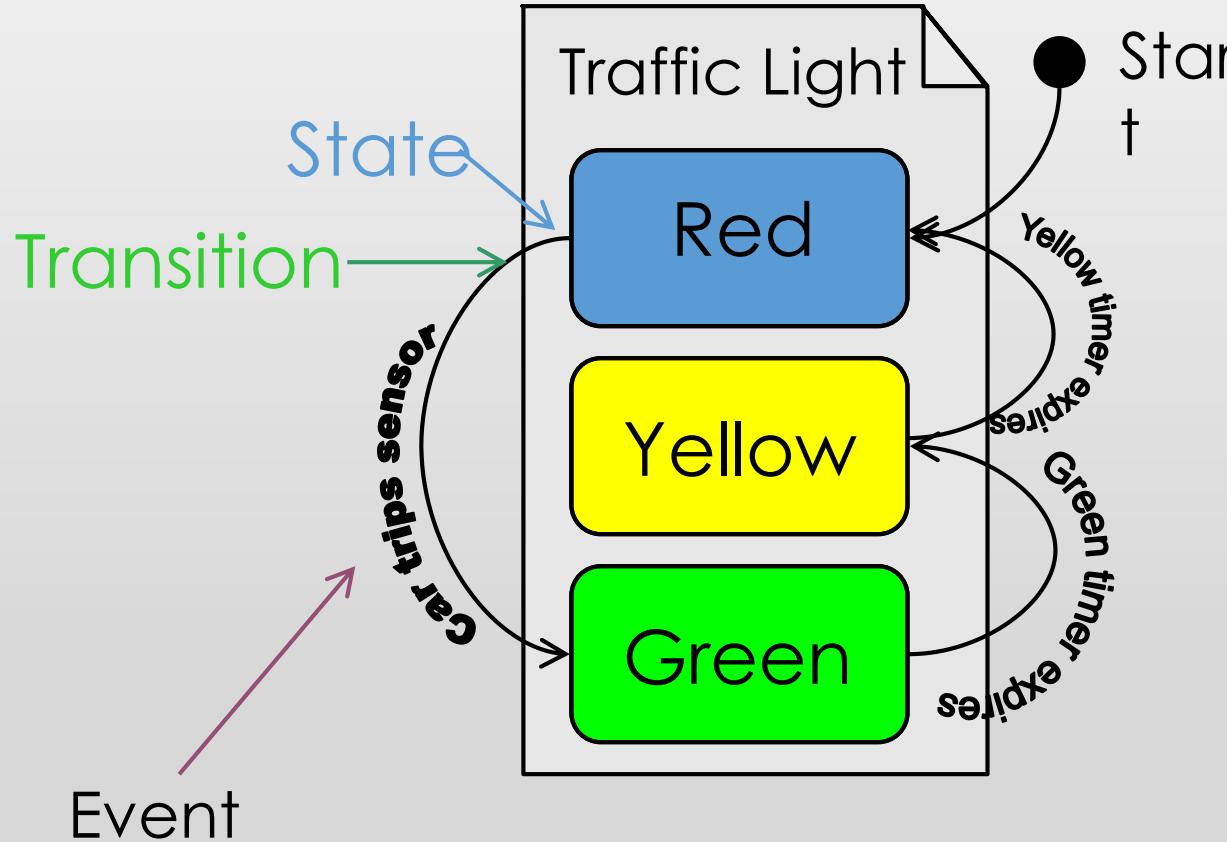
© uml-diagrams.org

STATE DIAGRAMS (BILLING EXAMPLE)

State Diagrams show the sequence of states an object goes through during its life cycle in response to stimuli.



STATE DIAGRAMS (TRAFFIC LIGHT EXAMPLE)



SUMMARY

- UML offers a graphical notation language to model system structure and behavior
- By using standard notations architecture could be communicated easier.
- The language has multiple diagrams, including the use case, class, state, activity, sequence, interaction, and deployment diagrams.
- Architect selects the diagram based on the aspects that they want to model: structure, interaction, message sequence, object-state change....

EXERCISE

Description: The customer browses the catalog and adds desired items to the shopping basket. When the customer wishes to pay, they describe the shipping and credit card information and confirm the sale. The system checks the authorization on the credit card and confirms the sale both immediately and with a follow-up e-mail.

Work: Develop

- 1- use case diagram
- 2- component diagram
- 3- communication diagram



EMERGENCE OF ARCHITECTURE

Lotfi ben Othmane

Iowa State University

Fall 2017

MAIN QUESTION

- What is your position on the following

Can a satisfactory architecture emerge from continuous small refactoring in agile development?

- a) Agree
- b) Disagree

- What is the position of agile advocate?
- Why some people do not agree with the statement?

1. What are the factors that impact the emergence of architecture through refactoring?
2. Do you agree with the finding of research?

1. What are the success conditions for the emergence of architecture through refactoring?
2. What is your position on the findings?

MAIN QUESTION

- Now what is your position?

Can a satisfactory architecture emerge from continuous small refactoring in agile development?

- a) Agree
- b) Disagree

UML IN-CLASS EXERCISES

Lotfi ben Othmane

Iowa State University

Fall 2017

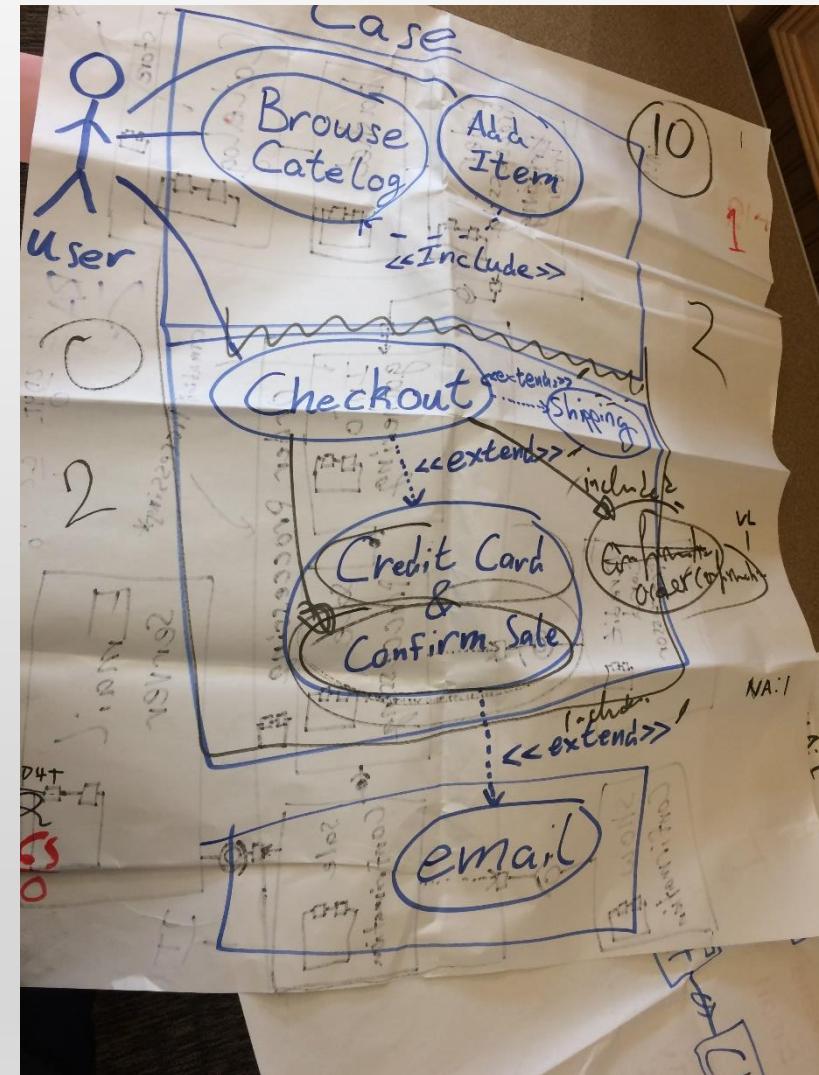
EXERCISE

Description: The customer browses the catalog and adds desired items to the shopping basket. When the customer wishes to pay, they describe the shipping and credit card information and confirm the sale. The system checks the authorization on the credit card and confirms the sale both immediately and with a follow-up e-mail.

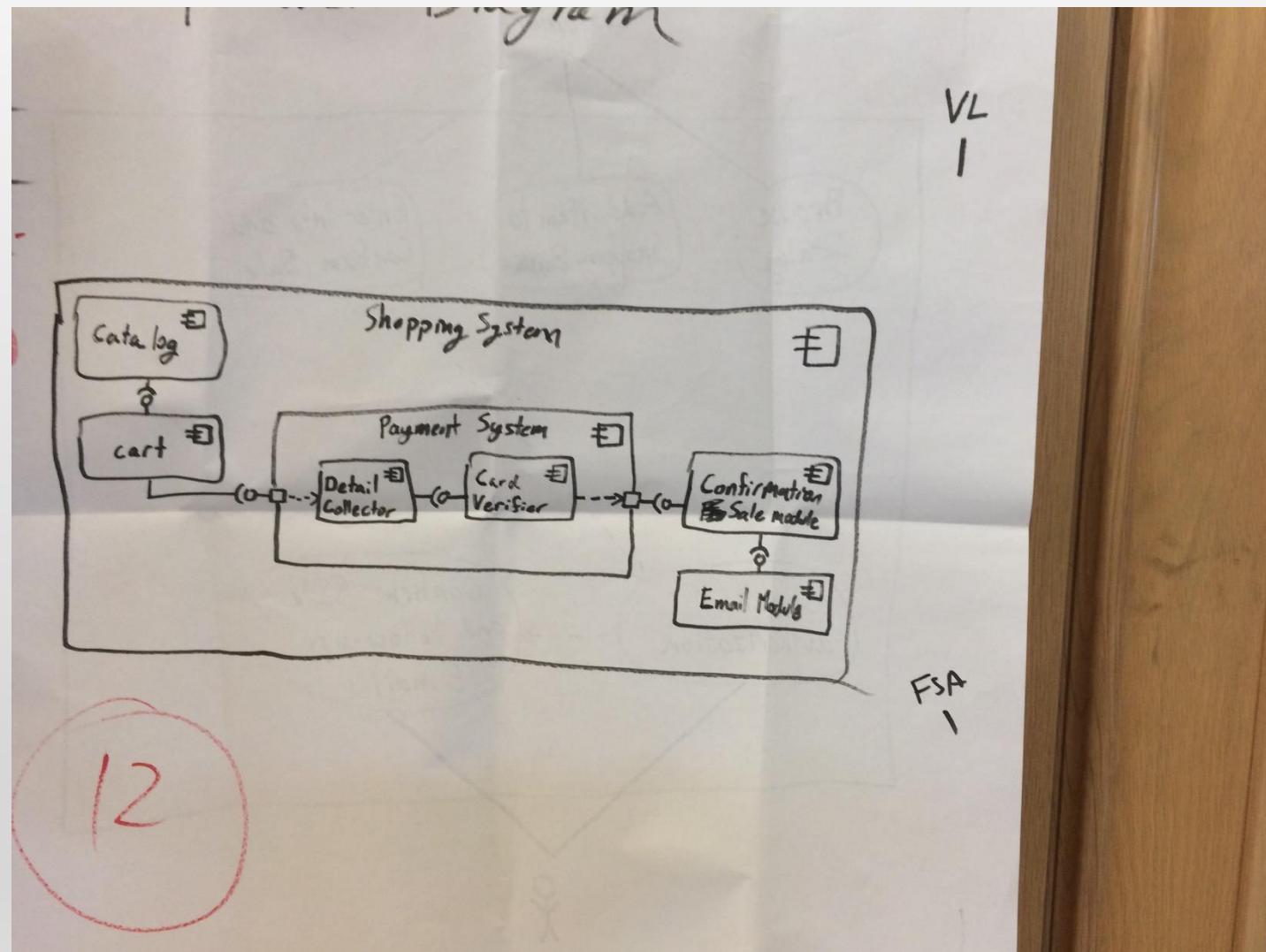
Work: Develop

- 1- use case diagram
- 2- component diagram
- 3- communication diagram

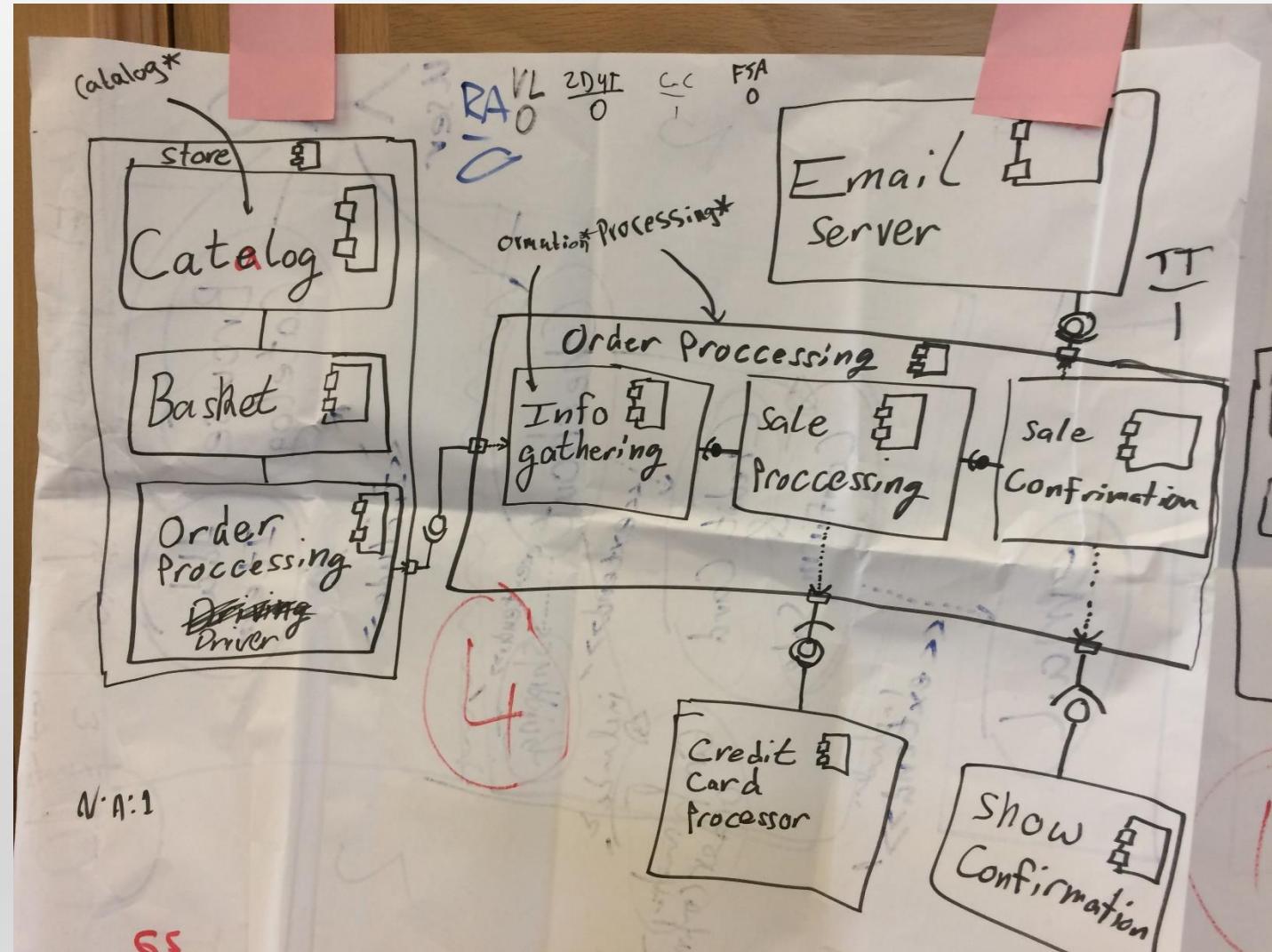
SELECTED USE-CASE DIAGRAM



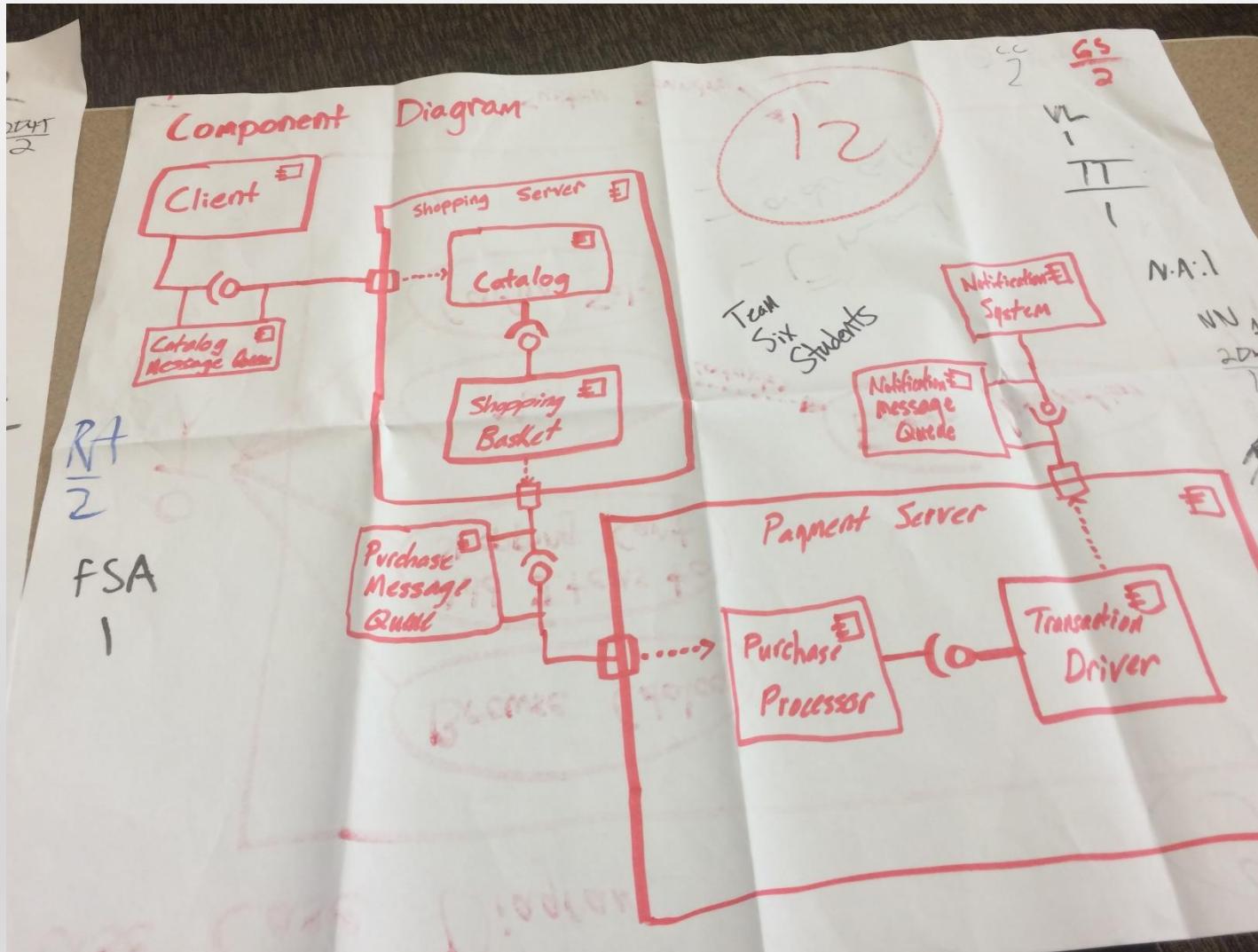
SELECTED COMPONENT DIAGRAM



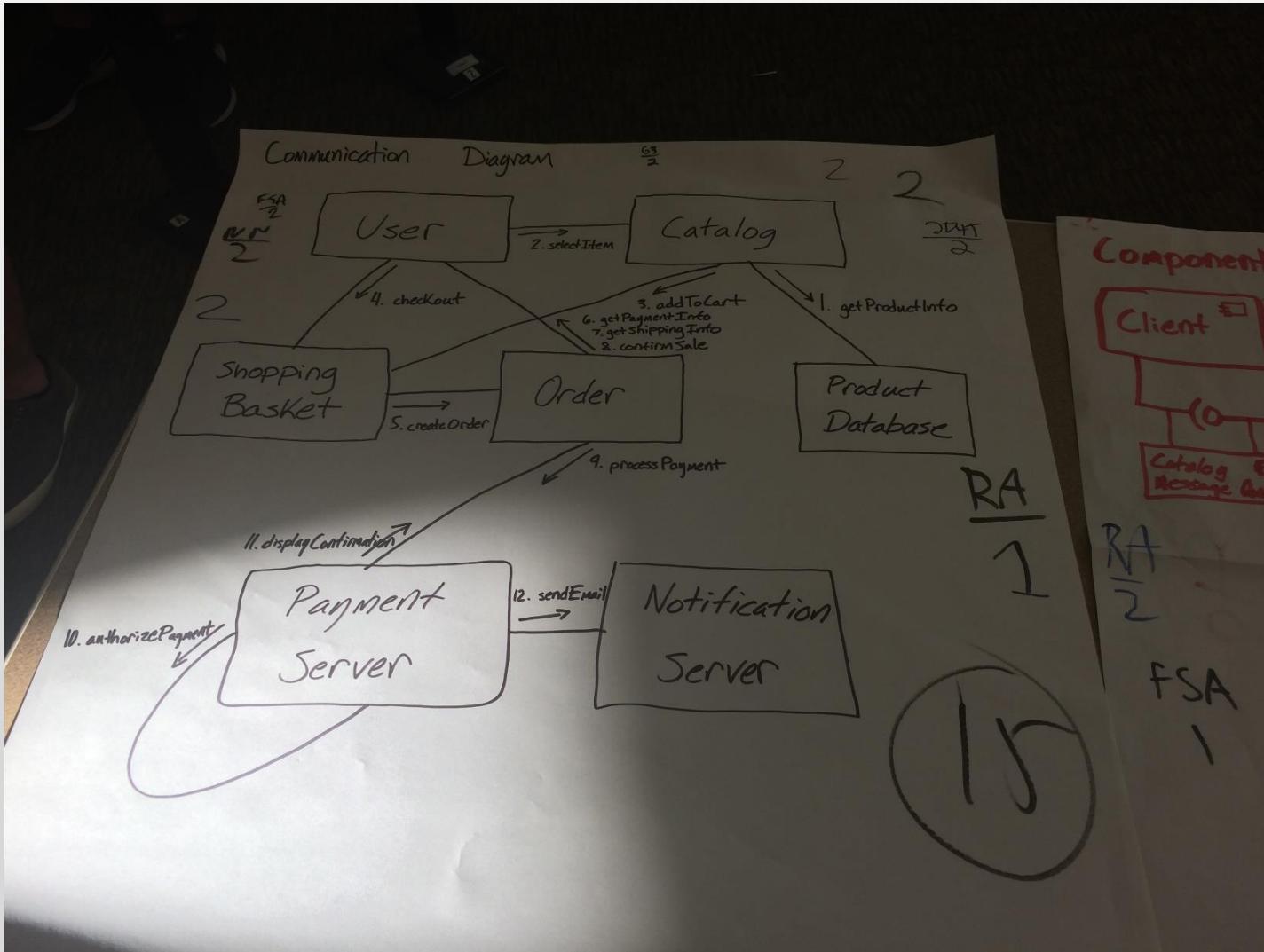
SELECTED COMPONENT DIAGRAM



SELECTED COMPONENT DIAGRAM



SELECTED COMMUNICATION DIAGRAM



ARCHITECTURAL DRIVERS

Lotfi ben Othmane

Iowa State University

Fall 2017

DESIGN OF ARCHITECTURE

- Designing is making decisions to achieve goals and satisfy requirements and constraints.
- Architecture design means making decisions to transform design purpose, requirements, constraints and concerns to structures and relationships between them.
- Architectural drivers are what and why of the design
- Architectural drivers include design purpose, quality attributes, primary functionalities, concerns and constraints

SPECIFYING QUALITY ATTRIBUTE REQUIREMENTS

- Use of scenario approach
- The parts of the scenario are
 1. **Stimulus** – event that trigger the scenario
 2. **Stimulus source** – who initiates the stimulus
 3. **Response** – what the system do with the stimulus
 4. **Response measure** – a metric that measures whether the response is satisfactory or not
 5. **Environment** – conditions for the response
 6. **Artifact** – parts of the system that contribute to the scenario

SPECIFYING QUALITY ATTRIBUTE REQUIREMENTS

Performance scenario for submitting applications

1. Stimulus: Regular application
2. Stimulus source: User
3. Response: Save the application, process the payment, return a confirmation to the user
4. Response measure: < 5 sec
5. Environment: normal and overload conditions
6. Artifact: submission, payment, save file, save database

EXERCISE

System: You are assigned to a project to develop a system water-consumption management. Customers have a device that opens/closes the water pipe and measures the water consumption. The device communicates with the central office through cellular network.

- Simulate a quality attribute workshop to get the architecture driver
- Use the utility tree to get the architecture driver



ARCHITECTURAL DESIGN

Lotfi ben Othmane

Iowa State University

Fall 2017

Check correct answers

Architecture design is:

1. Making decisions to satisfy project goals and constraints
2. Using design concepts to produce architecture
3. Creating design concepts that satisfy project goals and constraints
4. Identifying the optimum solution to satisfy quality attributes

Check correct answers

Architecture design is

- A process that includes identification of candidate solutions, evaluation of the solutions, prototype using the solutions
- A process that includes identification of candidate solutions, evaluation of the solution, selection of the “best” solution
- A process that includes identification of candidate solutions, prototype using the solutions, selection of the “best” solution

Architecture design is needed for:

Cross incorrect answers

1. Project proposals
2. Enable agility
3. Selection of tools for the development environment
4. Get the stockholders agree on the project
5. Estimate the budget for a project
6. Identify the resources needed for the project
7. Select the project manager
8. Write product leaflet

cross incorrect answers

Architectural decisions:

- have non-local consequences
- have consequences that matter to the achievement of architectural drivers
- have consequences that matter to the achievement of an architectural driver
- are a combination of those (give details)

Cross wrong statements

- Architecture is a set of design decisions that influence the control of a quality attribute response
- Design patterns provide an overall logical structure for a particular types of applications
- Tactics are conceptual solutions for recurring problems
- Deployment diagrams provide models to physically structure software



ARCHITECTURAL DESIGN

Lotfi ben Othmane

Iowa State University

Fall 2017

SELECTION OF EXTERNAL COMPONENTS

- Which of the criteria of page 36-38 you would not use for your project? Why?

BIG DATA ANALYTICS CATALOG

- Where can you find these catalogs?
- Do you need to build these for your projects?
- Can you use the catalog of page 37 for Web applications?



SERVICE –ORIENTED ARCHITECTURE AND TECHNOLOGIES

Lotfi ben Othmane

Iowa State University

Fall 2017

- Which problem does Service-Oriented Architecture (SOA) address?
 - 1) Performance
 - 2) Security
 - 3) Modifiability
 - 4) Interoperability
 - 5) Reliability
 - 6) System complexity

- Which of the following technologies does not support: client applications can discover servers, find out what services they are offering, and invoke the functions they provide
 1. Webservices
 2. CORBA
 3. RMI
 4. J2EE
 5. DCOM
 6. Plug and play
 7. Microservices

Which one of the following does not apply to SOA?

- 1) Remote call of APIs
- 2) Autonomy of components
- 3) Compatibility is based on policy
- 4) Worsen the performance of a system
- 5) Offer better security
- 6) Could be developed using several programming languages

Which of the following is not supported by WS standards?

- 1) Security
- 2) Interoperability
- 3) Transaction
- 4) Reliability
- 5) Availability
- 6) payment

ARCHITECTURAL DRIVERS

Lotfi ben Othmane

Iowa State University

Fall 2017

DESIGN OF ARCHITECTURE

- Designing is making decisions to achieve goals and satisfy requirements and constraints.
- Architecture design means making decisions to transform design purpose, requirements, constraints and concerns to structures and relationships between them.
- Architectural drivers are what and why of the design
- Architectural drivers include design purpose, quality attributes, primary functionalities, concerns and constraints

SPECIFYING QUALITY ATTRIBUTE REQUIREMENTS

- Use of scenario approach
- The parts of the scenario are
 1. **Stimulus** – event that trigger the scenario
 2. **Stimulus source** – who initiates the stimulus
 3. **Response** – what the system do with the stimulus
 4. **Response measure** – a metric that measures whether the response is satisfactory or not
 5. **Environment** – conditions for the response
 6. **Artifact** – parts of the system that contribute to the scenario

SPECIFYING QUALITY ATTRIBUTE REQUIREMENTS

Performance scenario for submitting applications

1. Stimulus: Regular application
2. Stimulus source: User
3. Response: Save the application, process the payment, return a confirmation to the user
4. Response measure: < 5 sec
5. Environment: normal and overload conditions
6. Artifact: submission, payment, save file, save database

EXERCISE

System: You are assigned to a project to develop a system water-consumption management. Customers have a device that opens/closes the water pipe and measures the water consumption. The device communicates with the central office through cellular network.

- Simulate a quality attribute workshop to get the architecture driver
- Use the utility tree to get the architecture driver

ARCHITECTURAL DRIVERS

Lotfi ben Othmane

Iowa State University

Fall 2017

FCAPS SYSTEM (CH4)

1. What is the system about?
2. What kind of services does the NTP system support?
3. What are FCAPS?
4. What is greenfield development?

FCAPS SYSTEM (CH4)

What are FCAPS? <https://en.wikipedia.org/wiki/FCAPS>

FCAPS SYSTEM (CH4) - ACTIVITY 1

1. Do the architectural drivers consider natural disaster?
2. Do the architectural drivers cover Service Level Agreement (SLA)?
3. Can the system be used for cloud services.

<http://www.pool.ntp.org/en/>

FCAPS SYSTEM (CH4) - ACTIVITY 2

An insider plugged a server of his own and connected it to the network. Extend the architectural drivers to address this.

BIG DATA SYSTEM (CH5)

- What is the system about?

BIG DATA SYSTEM – ACTIVITY 1

- Do the architectural drivers consider that the services are in different locations?

BIG DATA SYSTEM – ACTIVITY 2

- Extend the system to support that the customers use their own analytics software with the data.
- How useful is the utility tree for this case?

BAKING SYSTEM

- What is the system about?
- What is brownfield development?

BAKING SYSTEM-ACTIVITY 1

- Extend the architectural drivers so the system supports investigation of fraud

CPRE 339 Software Architecture and Design

Mid-Term Examination - Solutions Spring 2017

Section 1 (50 points)

The high score of each of the 5 questions below is 10 pts. The answers will be evaluated based on correctness and completeness.

Question 1: What is software architecture and why do we need it?

Answer: Software Architecture is a set of structures that describes the system. Software Architecture is needed for understanding relationships (interactions, communications, collaborations) among components and their properties.

Question 2: What decisions does each of the 3 structure category embodies? Give examples.

Answer:

1. Module – assigned specific computational responsibilities to modules. For example: decomposing the code into structures and classes.
2. Component and connector – focus on the way elements interact with each other at runtime to carry out the system functions. Example: services for synchronization, interaction between client and server.
3. Allocation – describes the mapping from software structure to system environment. Example: assigning modules for developments, implementation, integration and testing. Hardware assignments such as file management, processor core allocation, hardware assignment.

Questions 3: What are the 4 techniques used to extract architecturally significant requirements and how they differ? Based on your experience, which one is best?

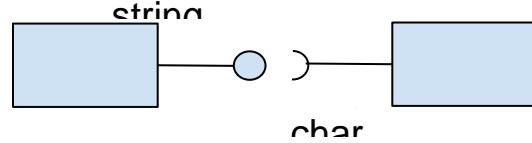
Answer:

1. Requirement documents: this document has more information about quality attributes rather than information of functionality.
2. Scenarios: this approach handles use case situations between system and users. It provides information about tradeoff between quality attributes and functionality, risks and security cases.
3. Interview with stakeholders. Best way to directly and quickly set quality attributes and architecturally significant requirements.
4. Understanding business goals: this approach identifies business goals that affect architecturally significant requirements. This is more abstract and might not be clear enough.

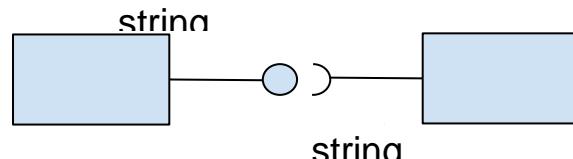
Question 4: Describe the 3 types of interface incompatibility using examples?

Answer:

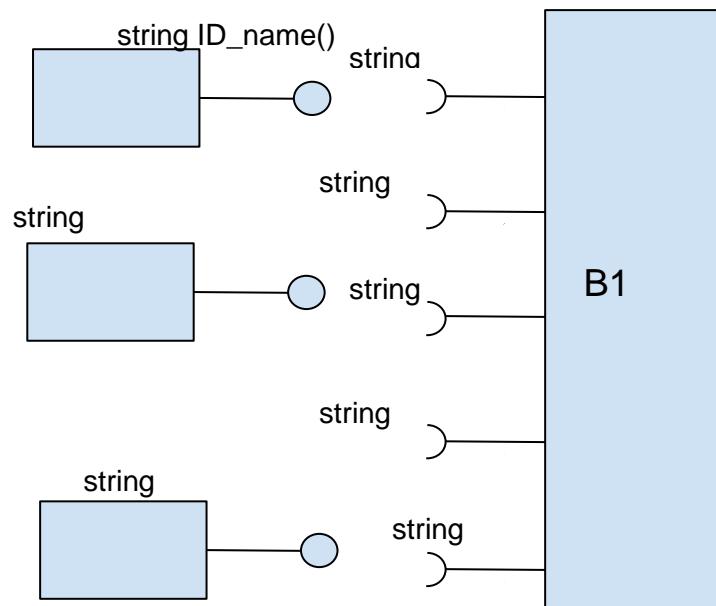
1. Parameter incompatibility – operations have same name but different types.



2. Operation incompatibility – names of the operations in the composed interface are different although they have same functionality.



3. Operation incompleteness – where the provides interface of one component is the subset of the required interface of another.



Question 5: Formulate a design pattern for handling exceptions?

Answer:

(The format of a design pattern is “Problem:, Solution:,”)

Problem: how exceptions need to be handled?

Solution: System components should not handle exceptions themselves. Therefore, exceptions must be handled externally elsewhere. It follows language structures for throwing and catching.
So the system must have a module which handles the exception externally. This exception handler should have the following -

- a condition to check if exception is true
- try - throw- catch sequence
- return statement which continues the execution from the point where it was interrupted.
- include a safe state.

Example:

```
Condition (exception is true)
    Throw exception;
    ....
    Program code
    ....
    Catch exception;
    Safe state return;
```

Section 2 (Total of 50 pts)

Your night flight was cancelled and you were offered to spend the night in the airport. You met in the dinner an old man who turned out to be a business development manager. You introduced yourself as software architect. The man talked proudly about the automations he has at home. He can control the AC/heater and adjust the temperatures in the rooms. He also has a security system (to detect thefts) that uses motion detection sensors. He has a problem though; he reduced the temperature of the house before traveling to reduce the use of energy. Unfortunately, the temperature of the house will be very low when he is back. He did not find a system that adjusts the temperatures of the rooms based on whether there are people inside or not. He asked you if you would team with him to develop a system that addresses the need.

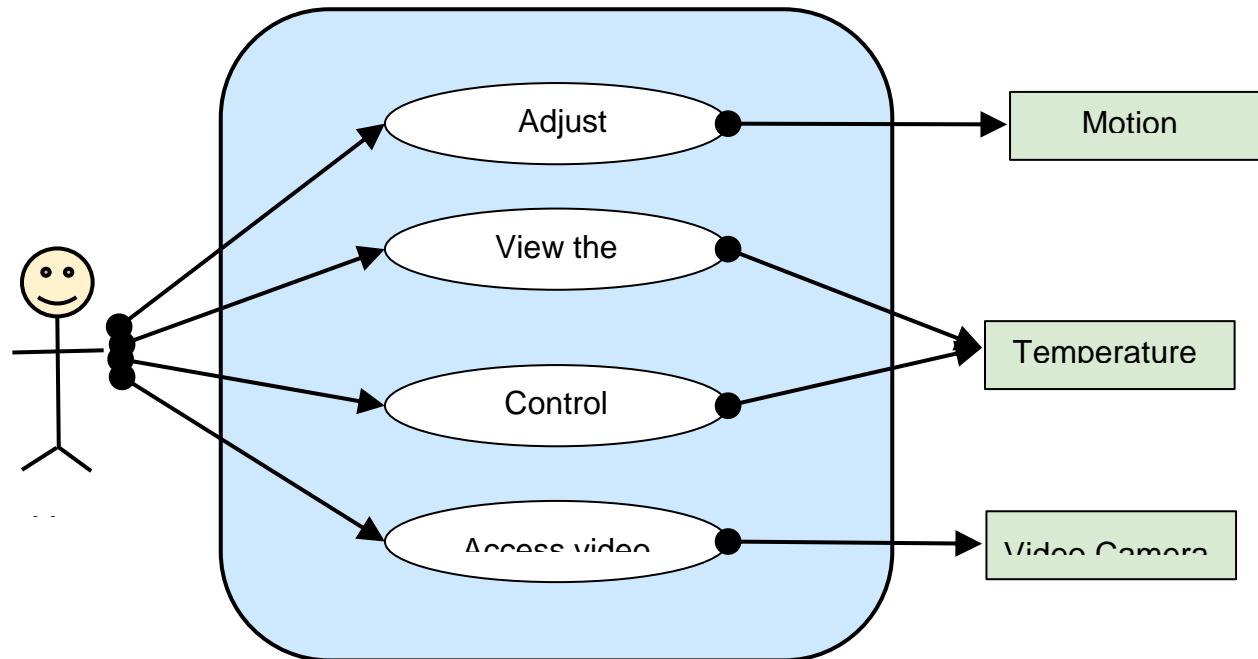
The man described further his vision about the future system. He explained that the system should use the motion detectors to adjust the temperature in the rooms. The temperature of the rooms should be accessible through a mobile App. He also wants the system to streamline videos from the cameras in the house. The App should allow to control the temperature of the house remotely.

You accepted the proposal. Your new partner asked you to prepare an architecture for the system so you may present it to potential investors that he will contact. Use the challenges described below to develop the blueprints of your software architecture. (Justify your choices.)

(This question has multiple approaches so there is no fixed correct answer. As long as your answer is relevant and captures the functionality of the system, it is considered correct. This is only one possible solution.)

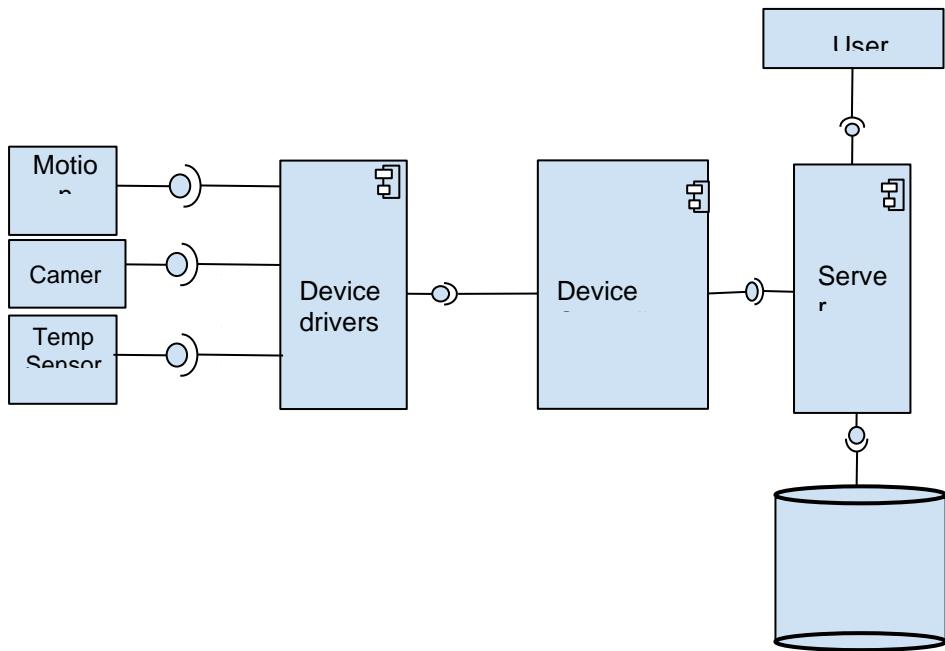
Challenge 1 (10 pts)

Develop a use case diagram for the software.



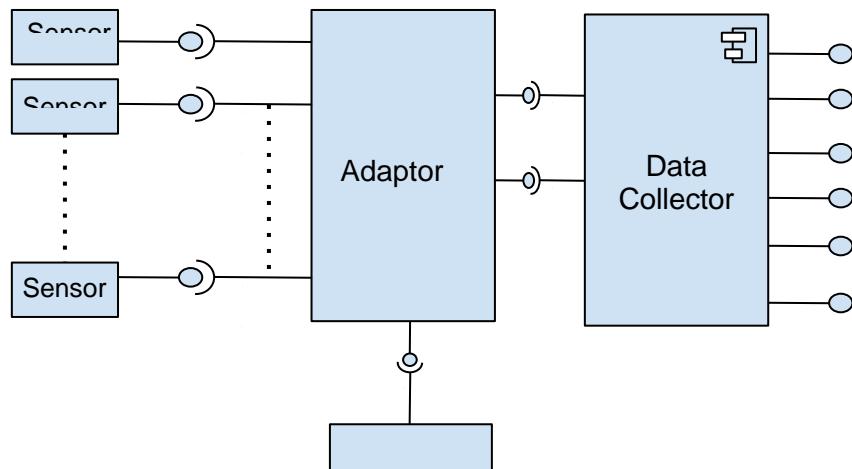
Challenge 2 (15 pts)

Develop a component diagram that shows how the software will fulfill the requirements.



Challenge 3 (10 pts)

The system should communicate with the different types of motion sensors that future customers may have. Develop a revision of your components diagram that addresses the requirements and discuss your options.



Challenge 4 (10 pts)

Some of the products your partners worked on had limited success because the customers had often to restart the systems after crashes. Developers sometimes mishandle exceptions which causes crashes and infinite loops. Propose a solution for exception handling that better handles errors and justify your choices.

Solution:

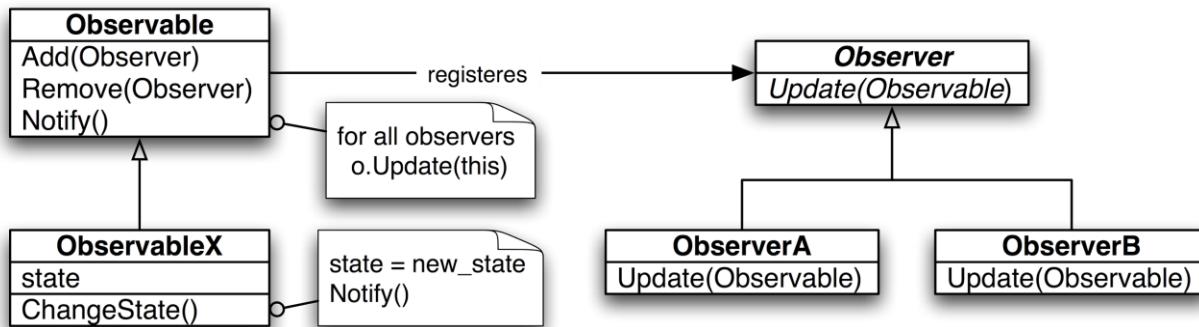
You need to describe a solution in the form of a specific problem and a solution that addresses it. You must be explicit in discussing the solution and not suggesting options or solution idea.

Below is an example of answer.

Problem: Mishandling exceptions will lead to infinite loops and system crashes. Current exception handler requires system restart after system crash. The problem is how to handle exceptions so there is no system crashes?

Solution

- Use observer pattern for error reporting i.e When the observed object (subject) encounters an exception, all dependents get notified and updated automatically.
- Use a condition to check if exception is true
- Use try - throw- catch sequence
- The return statement which continues the execution from the point where it was interrupted.
- Include a safe state



Challenge 5 (5 pts)

Your partner pointed also that the customers will not accept that anyone use the application to access the devices of their home. Propose a solution to ensure only appropriate people can use the application. You may draw a component diagram that visualizes your solution.

