Lina Khalid

Software Architecture for Business

Springer

Software Architecture for Business

Lina Khalid

Software Architecture for Business

Springer

Lina Khalid

ISBN 978-3-030-13631-4 ISBN 978-3-030-13632-1 (eBook)
https://doi.org/10.1007/978-3-030-13632-1

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

Software architecture has many axes when you first begin with it: the business goals of the system, the architecture requirements of the system, etc. This book is where you can gather all the knowledge on everything you need to know regarding software architecture.

This book, which mainly focuses on software architecture and its relation to business, is for students who just start their studies in software engineering field and are in the first course on software architecture; it helps them know the main concepts on software architecture and highlights their thoughts on relating this concept with business context. It shows that through building high-quality products, it helps the architects in the business field to think more efficiently in qualities through building the architecture of the products.

I have been trying to gather the sum of my knowledge in the software architecture field in one place to make it simple, short, yet thorough, and all-inclusive, and here it is, right between your hands. This is the perfect guide for the beginners in software architecture. The reason why I am proud of what I managed to put together is not only because of the knowledge contained within this book but also because I believe this is suitable for any student especially the beginners in the field.

So, whether you are taking your first class in software architecture or you are new to a job in this field, this is the book for you.

This book has two main pillars: the first one is software architecture and its relation to quality and the techniques that are used to gather information for quality, such as QAS and QAW, and the second pillar is the business world and how to build high-quality products through software architecture, which would make them competitive in the market.

This will be worth your time.

Good luck!

Lina Khalid

Acknowledgment

First of all, I thank God for answering my prayers and helping me through my journey.

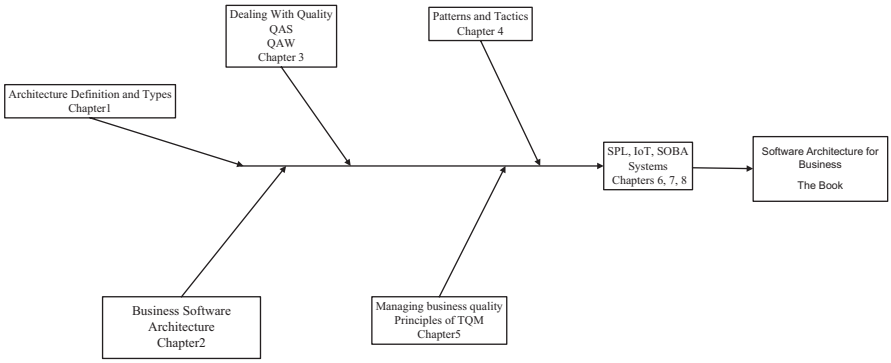
Many thanks to the Springer team for guiding me through this process.

Many thanks go to the light of my life, Leen, for her courage and support and for always giving me a push. Thank you, Leen.

I hope all my efforts yield a well-guiding book for students all over the world.

Lina

Contents


Dealing With Quality
QAS
QAW
Chapter 3
Patterns and Tactics
Chapter 4
Architecture Definition and Types
Chapter1
SPL, IoT, SOBA
Systems
Chapters 6, 7, 8
Software Architecture for
Business
The Book
Business Software
Architecture
Chapter2
Managing business quality
Principles of TQM
Chapter5

A quick tour

Chapter 1
Introduction

Abstract The importance of the architecture concepts is highlighted through the applications in the market place and through the aim of producing high qualities from it. This chapter is the introduction to the set of definitions of the types of architecture, system architecture, software architecture, enterprise architecture, and business architecture, but it focuses mainly on software architecture. There are many contexts that affect in building the architecture of the system such as technical, business, and background of the architect effects; all of them will be affected by the architecture after build. Marketecture is a concept that describes and gives a structural view of the main components when a quick review of the architecture is needed. Finally, the life cycle of architecture with the methods used for each stage in the cycle is described. Briefly, this chapter gives a good introduction for the basic types of architecture and the most important concepts of software architecture, but what makes it differ from other basic chapters in other books on architecture is that it highlights the modern app architecture which the enterprises need when building their architecture. Modern software architecture features will also be defined.

At the end of this chapter, you will learn:

- Definitions of the basic type of the architecture: software architecture, system architecture, enterprise architecture, and business architecture
- What the modern app architecture for the enterprise is
- The life cycle of the architecture
- The influence of architecture on systems

1.1 Architecture Definition

Software systems are built to achieve the business goals of organizations. The architecture of the system paves the way to achieve these goals. The path seems to be complex, but the life cycle of the architecture attenuates this complexity. Architecture is the basic part of any system. It is embodied in its components, their relationships with each other, and the environment.

L. Khalid, *Software Architecture for Business*,
https://doi.org/10.1007/978-3-030-13632-1_1

It is obvious that effective software engineering needs the software architecture for many reasons. First, building the right architecture is very important to the success of the system—the wrong one can lead to catastrophic results. Second, it is very important to build common paradigms so that the big picture of the system can be understood and the new systems, distinct from old systems, can be built. Third, detailed understanding of software architecture allows the software engineer to choose among design alternatives.

1.2 Basic Types of Architecture

Every system has its own architecture, which represents the abstract view of the system, and this is called software architecture. Many other types of architecture are related to software architecture, but they are broader than software architecture. These include system architecture, enterprise architecture, and business architecture.

1.2.1 Software Architecture

There is no standard definition for software architecture. You can realize that easily through searching. For example:

Philippe Kruchten, Grady Booch, Kurt Bittner, and Rich Reitman derived and refined a definition of software architecture as the following:

> Software architecture encompasses the set of significant decisions about the organization of a software system including the selection of the structural elements and their interfaces by which the system is composed; behavior as specified in collaboration among those elements; composition of these structural and behavioral elements into larger subsystems; and an architectural style that guides this organization. Software architecture also involves functionality, usability, resilience, performance, reuse, comprehensibility, economic and technology constraints, tradeoffs and aesthetic concerns.

Mary Shaw and David Garlan, from their early works, defined software architecture as:

> 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

Martin Fowler, in his book *Patterns of Enterprise Application Architecture*, outlines some common recurring themes when explaining architecture. He identifies these themes as “The highest-level breakdown of a system into its parts; the decisions that are hard to change; there are multiple architectures in a system; what is

architecturally significant can change over a system's lifetime; and, in the end, architecture boils down to whatever the important stuff is."

The definition that is going to be used in this book is the one that is defined by Bass, Clements, and Kazman in their book *Software Architecture in Practice (3rd edition):*

> The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

The above makes the thought of software architecture simpler: it consists of elements and relations between these elements. The main characteristics of this definition are:

1. *Architecture Defines Structure*

Structure is a set of elements held together by relations. Elements and relations might be runtime related such as a "sends data to" relation between processes or tasks. Elements and relations might also be non-runtime related such as an "inherits from" relation between classes.

Software systems are composed of many structures, and no single structure can hold the architecture. The most important thing for a structure is minimizing dependencies between elements and components, creating a loosely coupled architecture from a set of highly cohesive components. Every system has documentation for software architecture in case people that uses it are long gone and to prevent the source code from being lost.

> Loosely coupled architecture is an approach to interconnecting the components in a system or network so that those components depend on each other to the least extent practicable.

2. *Architecture Is an Abstraction*

Abstraction means that architecture deals with certain information of the elements without going into their details. Through abstraction, dealing with the complexity of the system becomes very easy.

> *Complexity* of the system means that the system encompasses several properties of pieces of software, all of which affect internal interactions.
>
> *"Complex"* describes the interaction between many entities. Increasing the number of entities will increase the interaction between these entities which increases the complexity which will in turn increase the risk.

3. *Architecture Addresses Functional and Nonfunctional Requirements*

Reasoning of the system should be an attribute that is important to some stakeholders. This includes functional and nonfunctional requirements. Functional requirements appear in use cases, while nonfunctional requirements do not. Nonfunctional requirements are related with *how* the system presents the required functionality.

Finally, a good architecture is the one that allows the system to meet its functional, quality attribute, and life cycle requirements. Also, the most important thing in software architecture is that it highlights on the early design decisions of the work, so architectural design decisions are the result of a software design process. This will be further discussed in the life cycle of architecture.

1.2.1.1 Modern Software Architecture

Modern software architecture has the following features:

- Changeable: Requirements always change. An architect needs to be flexible with these changing requirements.
- Integration with lots of system applications (many of them are open source) and messages.
- Highly scalable.
- Loosely coupled with SOA and Event-Driven Architecture (EDA) for two main reasons: one is to reduce the complexity and two is to increase the modifiability feature of architecture which enables the design to change, this is crucial for software architecture. Implementing the main principles of abstraction, separation of concerns, and information hiding all leads to loose coupling architecture.

> *Event-Driven Architecture* is a framework that arranges behavior about the production, detection, and consumption of events as well as the response they evoke. EDA compliments SOA because these services can be activated by triggers fired on incoming events.

Those were some of the important features of modern software architecture, but it is important to know that architecture itself is:

- A big picture of the system.
- A set of quality attributes: the most important ones are usability, stability, and scalability.
- Costly to change. This affects the architecture decisions.

1.2.2 System Architecture

Architecture is the skeleton of the system; it outlines the elements and their interactions in a system including its hardware and software elements. It includes software architecture in its definition and provides a suitable environment for the software architecture, that is why a system architect should understand not only the individual components but also the interrelationships among them. System architecture is on the top level in system building, strategic decisions, engineering tradeoff, and their associated rationales regarding how the system will meet the allocated requirements. System architecture may dominate functional behavior and emergent behavior. It also provides guidance and structure to later phases of system. In conclusion, system architecture *is a way to understand, design, and manage complex systems.* The complexity of the system comes from two main aspects:

- Integration of components: With integration, a large numbers of components are interrelated.
- Heterogeneity of components: Many fields need to design complex systems which make it very difficult to have a heterogeneous vision.

One way to understand complex systems is by structuring the system as a hierarchy, layers, etc. It is done through two main types of system quality attributes: *modularity and integrity.*

> *Modularity* is a technique to divide a software system into multiple discrete, independent modules, which are capable of carrying out tasks independently.
>
> *Integrity* is a property of data that is resistant to unauthorized modification. This must be in relation to a security policy that defines which data should be modified and by whom.

1.2.3 Enterprise Architecture

The meaning of enterprise architecture differs upon the person you ask. "Enterprise" is any organization or collaborative collections of sub-organizations with the same goals. Previously defined, "architecture" is the structure and behavior of the system.

"Enterprise architecture" is managing enterprise analysis, design, plan, and implementation for successful development and execution. Enterprise applies the principle and practice of architecture to guide the changes for the organization.

The primary reason for developing enterprise architecture is to support the enterprise by providing the fundamental technology and process structure for the strategy of IT.

Enterprise architecture makes sure that the IT landscape has three main qualities: robust, flexible, and efficient.

Enterprise software architecture is closely matched with the enterprise's internal organization, business model, and processes. To improve the speed and functionality of the enterprise, enterprise software architecture should have the following characteristics:

- Simplicity: It should be simple enough to facilitate effective communication among key team members. A lot of people with different viewpoints, skills, and roles regarding the software are engaged for deciding the structure and specifying the enterprise software.
- Flexibility and maintainability: Each enterprise system should continuously adapt to the new needs of the evolving markets, business organizations, and legal changes. So, the architect must create a highly maintainable and flexible system. The architecture should have unique components that could be reorganized. The reorganization should be carried out in a flexible way so that the local modifications done in the system do not influence the system overall.
- Reusability: This can be achieved by developing blocks and constantly reusing them. This can be achieved by providing standard functionality in code libraries, which are used across various projects.
- Adaptation and modification: The final architecture must adapt not only to the changes that occur in technologies but also to the real life cycles of the implemented technologies.

EA is *not* system information, service, or solution architecture. It is the output of the stakeholder. So architects need to know about stakeholders and their objectives and views (Fig. 1.1).

The organization may develop the architecture using industry mechanisms which include IT industry technique and other methods that are used to develop enterprise architecture.

1.2.3.1 Business Architecture

Business architecture defines the structure of an enterprise in terms of enterprise structure, business process, and business information. In defining the structure of the enterprise, business architecture takes into account some important elements such as customers, finances and the market considerations according to products and services, and other things related to the enterprise itself. It is important to know that the business is not limited by the enterprise; therefore the business architecture must be able to represent parts of the business that are outside of the enterprise and stakeholder interests.

This book will define the role of software architecture in business applications; it will show how software architecture can build high-quality applications, and the last three chapters will explain that in detail.

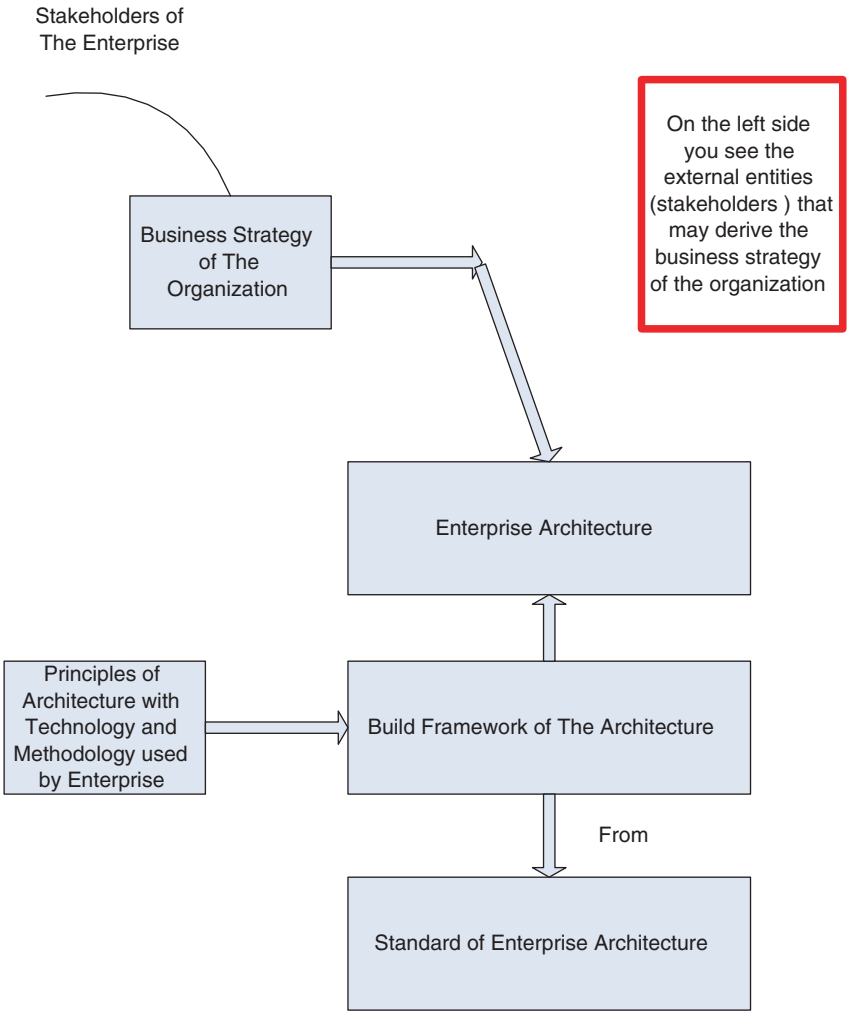


Fig. 1.1 Role of stakeholder in the enterprise system

1.2.4 Modern App Architecture for the Enterprise

Software is the critical part that defines the company regardless of the product. Software is how you connect the customers, reach new customers, understand their data, promote your products, and process their order.

To do this, software is going to be composed of small pieces (*microservices*) that are designed to do a specific job. Each service from the microservices is built with all the important components, which makes it able to "run" a specific job. These services are loosely coupled together so they can be changed at anytime.

Microservice is an approach to application development in which a large application is built as a group of modular services. Each module supports a specific business goal and uses a simple, well-defined interface to communicate with other sets of services.

Organizations are offering Docker Containers-as-a-Service (CaaS) environment which are standard units of software that look the same on the outside but are different when it comes to code and dependencies of that software. This enables developers and IT teams to move them across different environments without requiring any modifications.

Docker containers provide agility for development teams, control for operations teams, and portability of apps across any infrastructure. With this agility, developers are able to use any language and any tool because they are all packaged in a container, and that makes it standard for all the heterogeneity (Fig. 1.2).

The high level of the containers contains:

- An operating system
- A software that you build such as PHP or ASP.net application
- Dependencies to run the software (e.g., your software requires MYSQL to be the pre-request software to the software application you build)
- Environmental variables

Each container has a name for it.

Keywords of Docker are *Build, Ship, and Run* anywhere, and these keywords mean that the developer easily builds applications and packages them along with their dependencies into a container, and then these containers can be easily shipped anywhere.

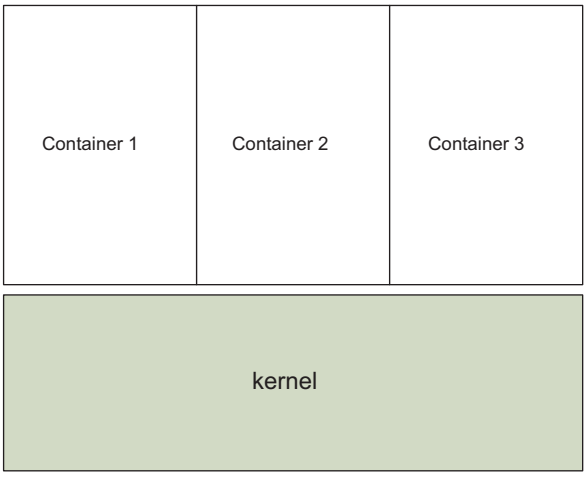


Fig. 1.2 Docker containers

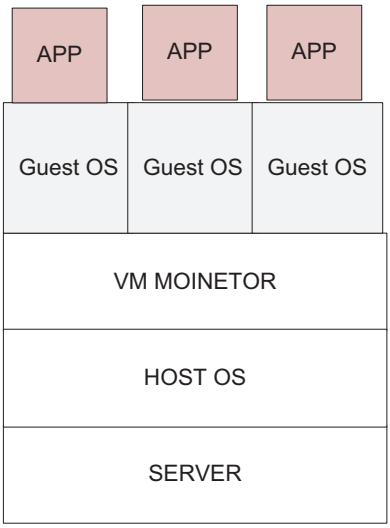


Fig. 1.3 VM architecture

To concludethe features of Docker's are:

- Containers make it easier for teams across heterogeneous environments.
- Docker containers can be deployed anywhere, on any physical and virtual machine and even on cloud.
- Scaling is very easy to apply in Docker containers.

Docker Architecture

To understand the Docker architecture, you need to compare it with the previous architecture (virtual machine architecture).

Virtual machine consists of:

- The server which is the physical server that is used to host multiple virtual machines.
- The host OS is the base machine such as Linux or Windows.
- The hypervisor which means virtual machine monitor (VMM) is a computer software, firmware, or hardware that creates and runs virtual machines.

You would then install multiple operating systems such as virtual machines on top of the existing hypervisor as guest OS and then host your applications on top of each guest OS (Fig. 1.3).

Dockers are the modern VM architecture and it contains the following:

- The server is the physical server that is used to host multiple virtual machines.
- The host OS is the base machine such as Linux or Windows.
- The Dockers engine. Dockers run on Docker engine. This engine used to run the operating system

All of the apps now run as Dockers container (Fig. 1.4).

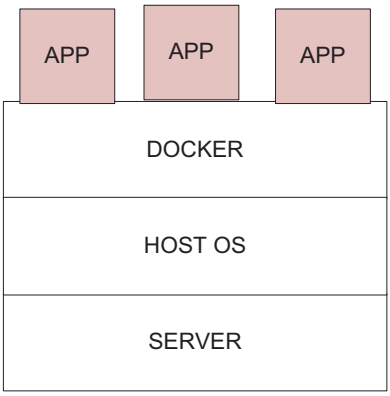


Fig. 1.4 Docker architecture

The clear advantage in this architecture is that you do not need guest OS. Everything works as Dockers containers.

To make apps smarter, a new technology appears to enhance enterprise applications. This technology is called serverless. Serverless is a software development approach. It first described applications that are significantly or fully dependent on services to manage server logic side which is known as backend as a service or "BaaS" that make serverless use third party in its architecture. Another meaning of serverless which overlaps with the previous definition is that serverless can also mean applications where some amount of server side is written briefly. This will be called "Function as a service or FaaS." The most popular implementation of FaaS is AWS lambda. With lambda you can run any type of code; you only upload the code and lambda takes care of everything and scales the code with high availability. AWS lambda is the newer definition of BaaS.

To conclude:

- Docker is a software container platform; it packages all the tools into one isolated container.
- AWS lambda is the implementation of FaaS; it lets you code without servers. Serverless cloud such as lambda is built on container, but the advantage is that it does not need to be managed.

1.3 Architecture Life Cycle

It is important to review the software architecture life cycle because the development of architecture would be carried out within. The architecture development life cycle can be seen as a general model which contains all the activities that are needed to develop software architecture.

1.3.1 Architecture and Requirements

Requirements are descriptions of the services that a software system must provide and the constraints under which it must operate. Requirements can range from high-level abstract statements of services or system constraints to detailed mathematical functional specifications. All requirements fall under the following classification:

- Functional requirements: These are a set of services the system should provide and the reaction of the system to an input. Sometimes functional requirements may also state what the system should not do. The reaction of the architecture to this type of requirement is the basis of design decisions according to the responsibilities assigned to architectural elements
- Quality attribute requirements: These are the qualification of functional requirements. Qualification is how fast the functional requirement must be performed (e.g., how fast the function is executed). The various structures that are designed into architecture satisfy this type of requirements.
- Constraints: A constraint is a design decision that has already been made such as using a specific programming language or reusing a certain existing module; all these choices are made from the view of an architect. Accepting the design decisions and integrating them with others are the reactions of architecture to this type of requirements.

1.3.2 The Life Cycle of Architecture

The life cycle of software architecture is composed of a set of stages: architectural requirements and analysis, architectural design, architectural documentation, and architectural evaluation. Each one of these stages is supported by a set of activities and methods to ensure predictability, repeatability, and high-quality results (Fig. 1.5).

The first stage (architectural requirement) needs a set of activities starting from eliciting requirements, analyzing them, and then prioritizing the most important one. The most important method or technique used to extract these requirements is QAW (Quality Attribute Workshop) . Chapter 3 explains QAW in detail. Architectures are mostly driven by architecturally significant requirements (ASR) which is as set of requirements that influence the architecture. These requirements are captured from requirement documents, by interviewing stakeholders or by QAW.

Next stage is architectural design. This stage also needs set of activities to be completed. These activities start with ASR and derive the set of architectural design decisions which require evaluation at the end. Because of the complexity of the design stage, they are used repeatedly until the architecture is complete and validated. The Attribute-Driven Design (ADD) is an iterative method used in this stage to help the architect to:

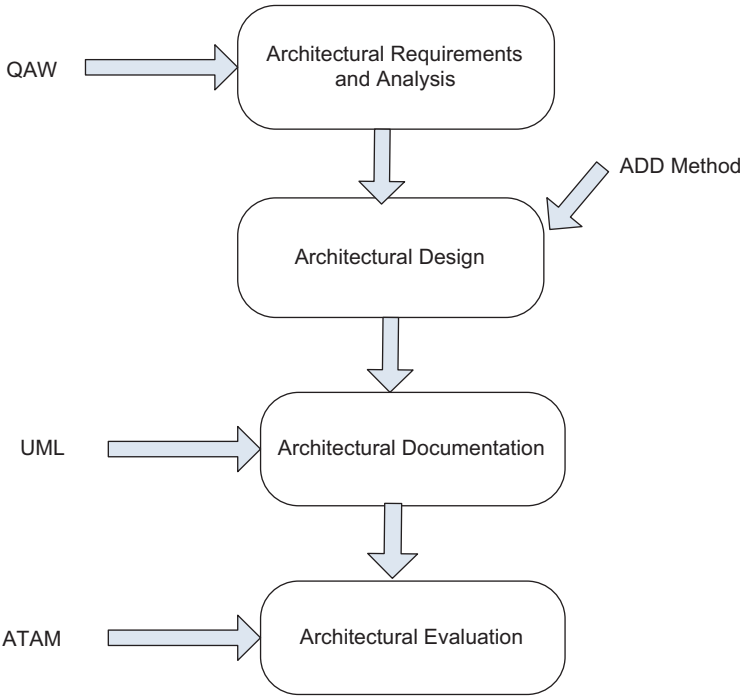


Fig. 1.5 Architecture life cycle with methods

- Choose an element to design
- Assemble the ASR to that element
- Design the chosen element

ADD helps produce a workable architecture early and quickly. ADD is explained in detail in Chap. 4.

> Architecture design decisions are those decisions that address architecturally significant requirements. These decisions are hard to make and/or costly to change.

The next stage is documenting the architecture. A document has to be created to explore the different structures that make up the architecture.

> Architectural view is a representation of the structure written and read by stakeholders.

The last stage is the evaluation of the software architecture. This stage focuses on evaluating the software architecture and whether it satisfies the requirements which it is built on or not. The most important method used to evaluate the final architecture is the one that comes from SEI which is called ATAM (Architecture Tradeoff Analysis Method). The main purpose of this technique is to evaluate the consequences of architectural decisions according to the business goals and quality attribute requirements.

According to SEI, ATAM is a method for evaluating software architecture according to quality attribute goals.

1.3.3 Documenting Architecture

Documenting the architectural involves writing the documents that describe different structures that build the architecture for the purpose of communicating it efficiently to the different system stakeholders. An important output of this process is a set of architectural views, which represent the system's structures, their composing elements, and the relationships among them. Documenting the architecture involves creating a set of related views which can be classified into different types: module views which show the structures where the elements are implementation units, component-and-connector views which show how the elements in the structures behave at runtime, and allocation views which show how the elements in the structures are allocated to the physical resources. Quality view is another type of view which is tailored for specific stakeholders or to deal with a specific concern. For example, security view shows all the responsibility measures to handle the security quality; it will show the components that have some security role and any data repository for security information. UML (Unified Modeling Language) is a way for documenting architecture views.

Documentation is very important to the architects, whether they still work on the system or not. It is used for:

- Construction: Documentation tells the developers what to build, how the elements should behave, and how they connect together.
- Analysis: Analysts will use the documentation to direct the architect for his job specifically to provide the required behavior and quality attributes.
- Education: Architecture documentation can be used to introduce people and new team members to the system.
- Communication: Architecture documentation serves as a primary vehicle for communication among stakeholders.
- Clarifying business goals, requirements, and activities: With a proper documentation, you can share the business goals and requirement with your managers and team so that they have a clear vision and goals.

Finally, the question is what to document?

When a quick system is needed to be produced, the term "marketecture" appears. Marketecture describes the main components and perhaps provides a structural view of these components.

The next thing is the needs of the various project stakeholders. Because architecture documentation serves an important communication role between different members of the project team, in a small team a minimal documentation is fair enough because of the interconnection between the team members. However, in large teams the architecture documentation becomes essential. This is why there is no straightforward answer. Documentation takes time to be developed, and it also costs money. It is therefore important to think carefully about what documentation is going to be most useful within the project's context.

In the market, documentation is important for three main purposes:

- To motivate the user about the product and encourage them for becoming more involved with it
- To inform users exactly what the product does, so that they receive a product according to their expectations
- To compare the product with other alternatives

1.4 Architecture and Technology

Architects make the design decisions of the system early in the life cycle of the project. Many of these decisions are difficult to validate until the final system is built. This is why sometimes building a prototype of a system is useful in the design approach, but it remains difficult to be certain of the success of a specific design choice in the context of the application. Thus, pattern, an abstract representation of architecture, is used. Patterns will be explained in Chap. 4

1.4.1 Influence of Architecture on Systems

Software architecture is influenced by different types of contexts such as technical, businesses, and social contexts. All these influences will affect the architecture in the future. This is why architects need to understand the nature of these influences early in the process.

These influences will be discussed by the following questions:

- Technical: What technical role does the software architecture play in the system of which it is a part?
- Project life cycle: How is software architecture related to the phases of a software development life cycle?

- Business: How does the existence of software architecture affect an organization's business environments?
- Professional: What is the role of software architect in an organization or a development project?

The important thing to know about architecture in the technical context is that *if you care about specific quality attribute, you have to be concerned with specific decisions*; all of these decisions *depend on the nature of the architecture*. For example, if you care about performance, then you should concentrate on managing the time between elements and their use of shared resources. On the other hand, if you care about interoperability, you need to pay attention on the elements that are responsible for external interactions between these elements and so on.

Architects need to understand the goals of the organization in the business context. Many of these goals deeply influence the architecture, while other business goals have no effect on the architecture at all. Business goals are one of the basic drivers for building software architecture.

The architect's background and experience on building the products is very important for architecture in the professional context; he needs many professional skills to build high-quality products. For example, he needs to know how to deal with customers and to have the ability to communicate ideas clearly.

As a result, architecture has influences that influence its construction, and once the architecture is constructed, it then influences a set of influences which lead to its construction. The cycle of influences is called the Architecture Influence Cycle (AIC) (Fig. 1.6).

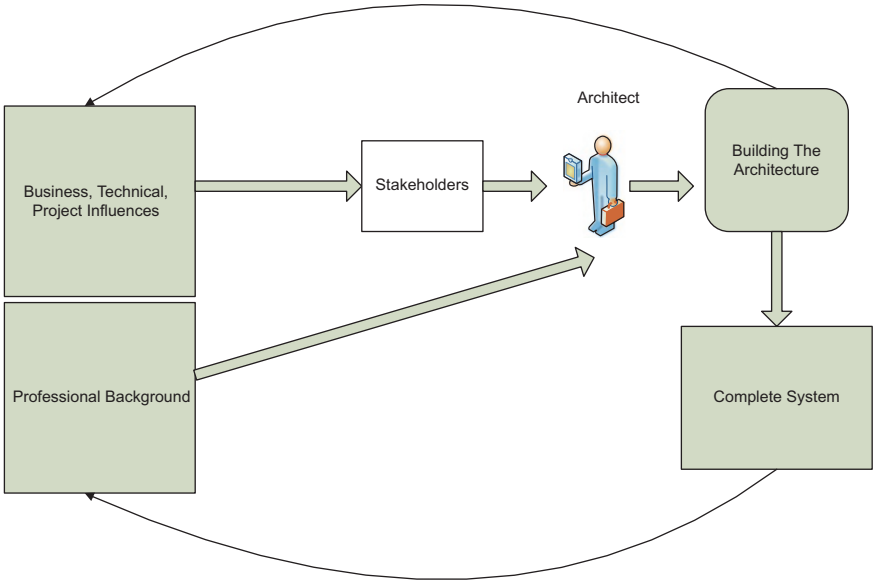


Fig. 1.6 Architecture influence cycle

1.5 Architecture's Role in Business

The role of software architecture in business is very important to improve the understanding of the business clearly and then to help solve the problems of that business and to support the business and its activities.

Software architecture in business increases the productivity and efficiency of the business and all that is done through following:

- Decision support: This needs understanding of ADD (Attribute-Driven Design) method of the software architecture and also the types of pattern that is used. According to SEI, ADD method is an approach to define software architecture in which the design process is based on the software quality attribute requirements.
- Availability of using a new way of technologies, for example, the web services that is used in the business to support interaction with suppliers of goods sold by the web shop application.
- The process improvement of the business, which is supported by the software architect, improves the functionality of the business.

For a business manager, the thing he focuses on the most is being able to change the product quickly, and that makes the software architect constantly thinking about changeability quality. The important thing to know is that the software architect needs practical knowledge of the business to make good judgment or design. That is why the software architect must understand business issues well. Understanding the business helps define the design solutions to solve its problems to reach its goals and then choose a good structure for the business.

The architect has five roles in developing a good business:

- Business strategy: The role of this is for business rather than IT. It works on top management which supports decisions around commercial issues of business.
- Business architect: The role of this is for anyone who looks and observes the way of work in business. Also, the business architect has to identify the implementation of the improvements according to the nature of the organization. The software architects use this as a basis for the design of business software solutions.
- Solution architect: The role of this is sometimes synonymous to application architect. This type is one of software architect types when moving from business to software. The difference between a business architect and a software architect is that the business architect architects the business while the software architect architects the software that supports the business.
- Architect of technical infrastructure: The role of this is not directly involved in the software development. This role is needed to deploy a set of solutions that come from the solution architect. This means these two roles must work together to ensure the productivity of the system operators.
- Enterprise architect: The role of this is to collect business, solution, and technical infrastructure roles, depending on how you define the enterprise architecture. This role focuses on the big picture rather than details; for example, the

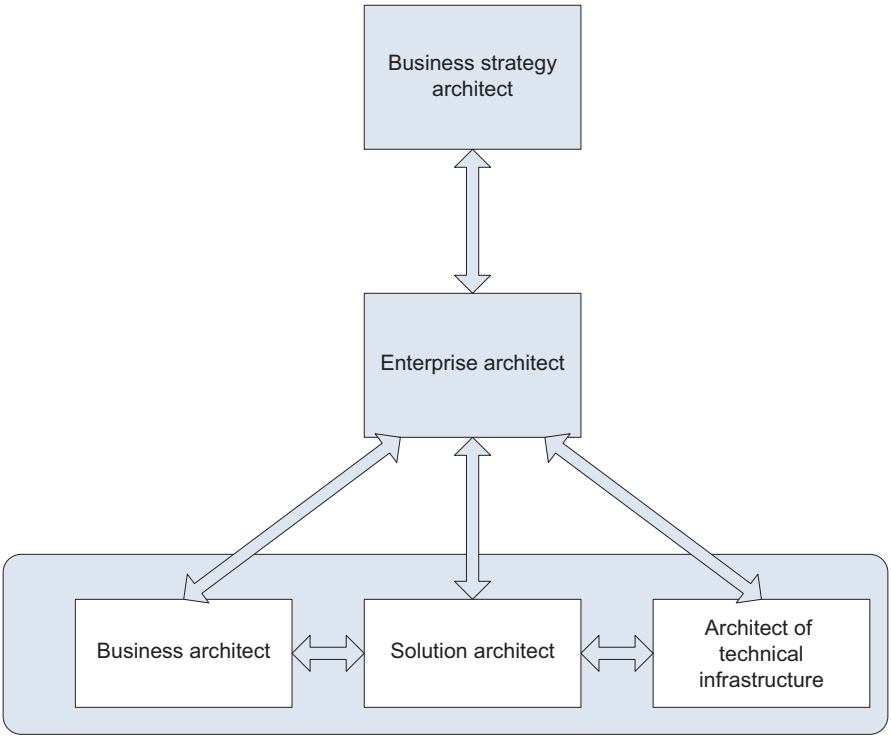


Fig. 1.7 Architect's role in business

enterprise architect should know there is a purchasing business process and what this process is used for, but he does not need to know the activities of that process.

Note Each rectangle in Fig. 1.7 represents a role of person that can play, rather than a specific person. Some roles are played by a single person, while others are played by multiple persons.

1.5.1 What Makes Good Architecture in Business?

Nothing can define exactly what good or bad architecture is, but in general a good architecture is the one that allows a system to meet its functionality and quality attributes to reach its goal.

As with business architecture or other types of architectures, there must be set of rules that must be taken into consideration when designing any type of architecture, and some of them are:

- The architecture should be documented and then evaluated to ensure delivering the system with the specific qualities that we build the architecture for.
- The architect should set and prioritize the most important qualities, and this will give the facts of a trade-off that always occurs.
- Also, good business architecture makes sure that the parts of the project are not be coupled together and that each part has a clear interface.

I agree with the authors who decide that the goodness of software architecture for any type of system does not depend only on the architecture drivers and decisions but also on how business conditions derive these architecture decisions such as the business cost, time to market and the goal of the business itself, and so on.

1.6 Architectural Pattern

Architectural pattern can be defined as the composition of elements to solve recurring problems. The compositions of these elements should be found after a period of time, that is why it will be documented and then disseminated between domains. Patterns can be characterized according to the type of architectural elements they use. For example:

Layered pattern is a common module type pattern when the *uses* relation among software elements is strictly unidirectional. A layer is a coherent set of related functionality. Layers are always designed as abstractions that hide implementation from the layer below.

Component-and-connector patterns are:

- Shared-data (or repository) pattern: This pattern comprises components and connectors that create, store, and access persistent data. The repository takes the form of a database. The connectors are protocols for managing this data.
- Client-server pattern: The components are the clients and the servers. The connectors are protocols and messages they share among each other to carry out the system's work.

Allocation patterns include:

- Multitier pattern: This pattern describes how to distribute and allocate the components of a system in distinct subsets of hardware and software, connected by some communication medium.
- Competence center and platform pattern: This pattern is allocated to sites depending on the technical or domain expertise located at a site. While in platform, one site is tasked with developing reusable core assets of a software product line, and other sites develop applications that use the core assets.

1.7 Summary

Architecture in any system is important for many reasons:

- It helps in communication among different types of stakeholders.
- It gives a result of the earliest design decisions through building the system.
- It gives the abstract view of the system.

According to SEI, "*software architecture* is a set of structures which comprise software elements, relations among them, and properties of both." This definition encompasses many characteristics in it such as abstraction, addressing functional and nonfunctional qualities, etc. The architect's role is to select suitable structures to build a system with a high quality. In general, a good architecture is the one that achieves its goal with attention to cost and schedule. Software architecture is influenced by different types of contexts such as technical, businesses, and social contexts. All these influences will affect the architecture in the future.

System architecture is the skeleton of the system; it is the way to describing the elements and their interactions in a complete system including its hardware and software elements, so it includes software architecture in its definition, and it provides the environment of software architecture, that is why a system architect not only knows about the individual components but also understands the interrelationships among the components.

Enterprise architecture is managing enterprise analysis, design, planning, and implementation for successful development and execution. Enterprise applies the principle and practice of architecture to guide the changes of the organization.

The primary reason for developing enterprise architecture is to support the enterprise by providing the fundamental technology and process structure for the strategy of IT.

Enterprise architecture makes sure that IT landscape has three main qualities: robust, flexible, as well as efficiency.

Business architecture defines the structure of the enterprise in terms of enterprise structure, business process, and business information. In defining the structure of the enterprise, business architecture takes into account some important elements such as customers, finances, and the market considerations according to products and services and other things related to enterprise itself.

Modern app architecture for the enterprise is a new concept added to the basic traditional introduction chapters in software architecture books; it shows you that today, software is going to be small pieces that are designed for a specific job, and this is called microservices.

Marketecture also appears in this chapter in architecture documentation, especially when a quick system is needed to be produced. Marketecture describes the main components and perhaps provides a structural view of these components.

References

There are many good books, reports, papers, and videos available in the software architecture world. Below are some I recommend to expand information
In terms of defining the landscape of software architecture in general, I recommend the following:
L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, 3rd edn. (Addison-Wesley, 2013) USA

Further Reading

P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford, *Documenting Software Architectures: Views and Beyond*, 2nd edn. (Addison-Wesley, 2010) USA, Boston US
P. Clements, R. Kazman, M. Klein, *Evaluating Software Architectures: Methods and Case Studies* (Addison-Wesley, 2002) USA, Boston US
I. Gorton, *Essential Software Architecture*, 2nd edn. (Springer, 2011) Berlin, Heidelberg
L. Homan, *Beyond Software Architecture: Creating and Sustaining Winning Solutions* (Addison Wesley, 2003) Canada
N. Rozanski, E. Woods, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives* (Safari book online, 2009) USA
Sten and Per Sundblad, "Business Improvement through Better Software Architecture", Microsoft developer network. https://msdn.microsoft.com/en-us/library/bb266336.aspx
Also you can enter SEI (Software Engineering Institute) library and search on software architecture and any other type of related architecture; you can then find a lot of webinars, videos, and articles, as, for example, SEI, "what makes a good software architect", 2016
Ph. Kruchten, What do software architects really do. J. Syst. Softw. (2008). www.elsevier.com/locate/jss
J. McGovern, S. Ambler, J. Linn, V. Sharan, E. Jo, *Practical Guide to Enterprise Architecture*, vol 1 (Prentice Hall, 2001)
For the part of Modern app architecture, I prefer to read:
https://www.docker.com/ The official site for Docker This site has all information and documentation about the Docker software. It also has the download links for various operating systems.
https://martinfowler.com/articles/serverless/

Chapter 2
Business Software Architecture (BSA)

Abstract Enterprises deploy business architecture capabilities. There is some confusion regarding the definition of business software architecture because it merges two terms: business and software architecture. In this chapter, there will be a brief definition of BSA with all related concepts. The basic roles of a business manager and a software architect are explained. Software architects take up an exceptional role in the world of IT. They are expected to know the technologies and software platforms on which their organizations run as well as the businesses that they serve. The relationship between business architecture and IT is described; the relationship between business architecture and IT is intertwined so that anyone talking about business architecture must be able to address how it is related to IT. Business architecture ecosystem will be defined in this chapter to represent the essence of the business. Also, pragmatic architecture shows how the architect adopts the best approach to the architecture.

At the end of the chapter you will learn:

- The concept of business software architecture
- The ecosystem for business architecture
- Software architect and business managers responsibilities in business software architecture
- The requirements for business architecture
- The concept of pragmatic architecture

2.1 Business Software Architecture

According to John Reynolds, business software architecture is "the structure of any software or set of programs used by business users and their customers to perform various business functions." It focuses on the long-term maintenance and evolution of the business functionality, rather than simply on the functionality being delivered today. Also he defines a "business software architect" as a person who designs the structure of software that is used by business users and their customers to perform

L. Khalid, *Software Architecture for Business*,
https://doi.org/10.1007/978-3-030-13632-1_2

various business functions, focusing on the long-term maintenance and evolution of the business functionality.

Picture the enterprise being a tree and business being its branches. *Business Software Architecture, or BSA, is:*

> The structure of a business in any enterprise or organization used by business users as well as customers to perform the functions of the business. Software architecture is an important part of this definition because it helps the business reach its goals with a high quality.

This definition shows how software architecture supports building a business, and this is the main focus of this book. In the business world, the architecture of a software system is very important. It has a deep impact on the long-term success of the business using it. Such an impact can change related things such as:

- Costs of maintenance and development
- Availability of skills and expertise
- Competition with other businesses

A good architecture of the software enables a business to respond to the changes with a little additional cost.

2.1.1 Software Architects Need Business Education

In a business enterprise context, the objective of the business organization should be to light the way to an architect to make a decision. Such a relationship between the architect and the business needs the software architect to have some business

education. For example, the business always plans for the desired return on investment (ROI) before the initiation of software development, and that is why the architect needs to understand the desired ROI to avoid making wrong decisions.

Return on investment (ROI) is a performance measure used to evaluate the efficiency number of different investments.

One of the important things that make the business "drive" is that it provides enough information about the effort of software development to make good decisions for that system.

2.1.2 Roles of Software Architects and Business Managers in Business Software Architecture

Two important roles through building any business software architecture, especially when acquiring a quality from the software architectures, are the roles of the software architect and the business manager.

The responsibilities of the software architect are shown when building any software (including the business software). These responsibilities include:

- Dealing with the complexity of a system by representing it as an abstract view and dividing the system into a manageable model that shows the essence of the system by exposing important details and significant constraints.
- Maintaining control over the architecture life cycle in parallel with the project's software development life cycle. Although an architect's role may be obvious during the requirements and design stages of a project life cycle, he also monitors the loyalty of the implementation to the chosen architecture during all iterations.
- Making critical decisions that lead to design specific direction for the system in terms of implementation, operations, and maintenance. The critical decisions must be made by understanding and evaluating alternative options, and these decisions must be well documented.
- Working closely with managers to explain the software architecture solutions. This may be done by participating in business process activities, by using cost benefit analysis method, or by measuring the level of component/architecture reuse between projects with the help from the software process improvement team.

Cost benefit analysis method (CBA) is an approach to estimate the strength and weakness of alternatives such as transactions and activities. It is used to specify an approach to achieve benefits while protecting savings.

On the other hand, the responsibilities of the business manager are:

- Managing and controlling a company's activities and employees. In a big company, managers typically manage an individual department, such as marketing, sales, or production. In a smaller company, the business manager might manage operations in all departments.
- Overseeing the activities of workers, hiring, training, and evaluating new employees and making sure that a company meets its goals.
- Developing and implementing budgets, preparing reports for senior management, and ensuring the department complies with the company policies. Managers also make sure their employees have the resources to complete their work.

2.2 Defining Requirements for Business Architecture

The important thing for business architects to derive the architecture of the system is what is called the architecture requirements. There are many studies that show that the effectiveness of costs stands on using requirements effectively and caring about the architectural decisions that lead to define goals and constraints. All of that will avoid mistakes in implementation. This is why defining requirements for architecture is very important. The first step is to identify all relevant business processes. Part of identifying the processes is identifying the stakeholders. Business scenarios and stakeholders' perspectives must be supported by the underlying structure of the business architecture.

A high-level view of the processes is enough to define the candidate architecture. The next step is defining the boundaries of the system.

Now, the results are the candidate architecture, a set of identified business processes, and a mapping of those processes onto that architecture—defining the responsibilities of the system. This procedure will continue until meeting the requirements of the business architecture.

In conclusion, developing architecture requirements will provide clarity of the effective "traditional" requirements process for the business.

Another important thing is that the nature of business architecture requires a common approach to represent the essence of the business. Business architecture ecosystem can be adopted for that reason. OMG (Object Management Group), a special group for business architecture, explains business architecture ecosystem by their authors as a set of business artifact domain listed in Fig. 2.1.

> Business ecosystem is "one or more legal entities, in whole or in part, that exist as an integrated community of individuals and assets, or aggregations thereof, interacting as a cohesive whole toward a common mission or purpose."

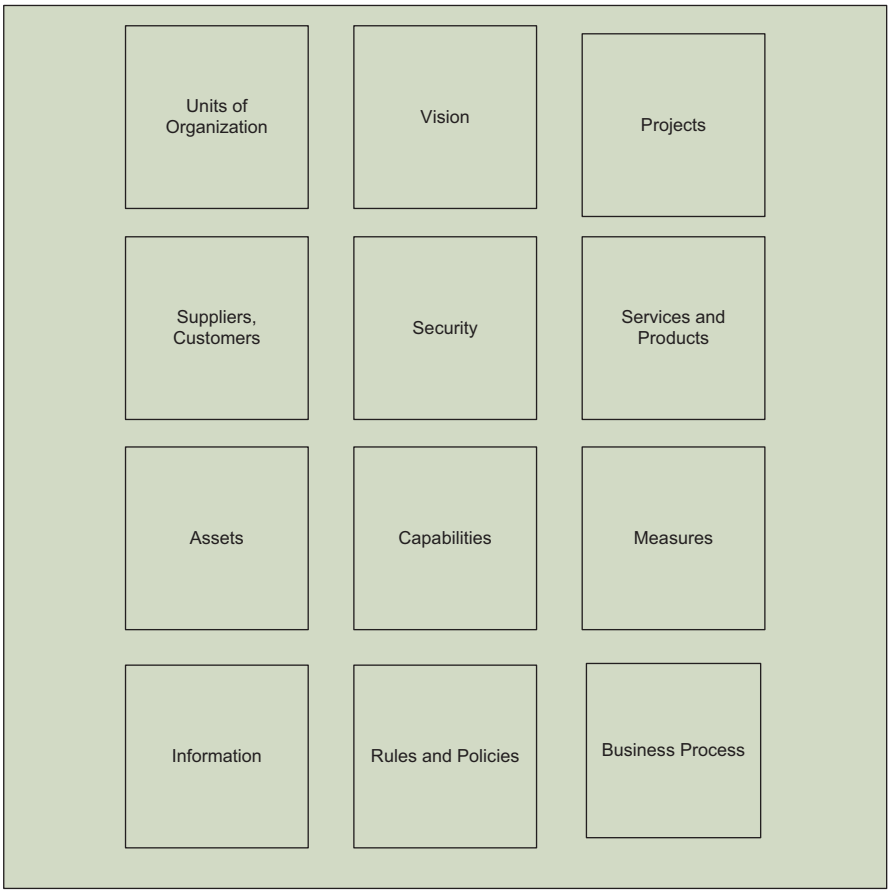


Fig. 2.1 Abstract business views

For example, the concepts of capability, business process, semantics, rules, and decision rules that appear in Fig. 2.1 define what a business does and how it does it. Metrics and measures provide a way to assess the progress toward goals and track performance. The customer and the supplier are both responsible for the inside and outside of the enterprise. Products, services, and assets define what an enterprise produces, what is delivered to customers, what is received from suppliers, and what are the resources used internally to complete specific tasks and so on for other concepts. Every concept stands for specific task.

The relationship between business architecture and IT (Fig. 2.2) is intertwined so that anyone talking about business architecture must be able to address how these business artifacts are related to IT architecture. For example, application architecture artifacts include all of the business systems and services under the control of the IT organization that are used to deliver value to the business. Therefore, the business

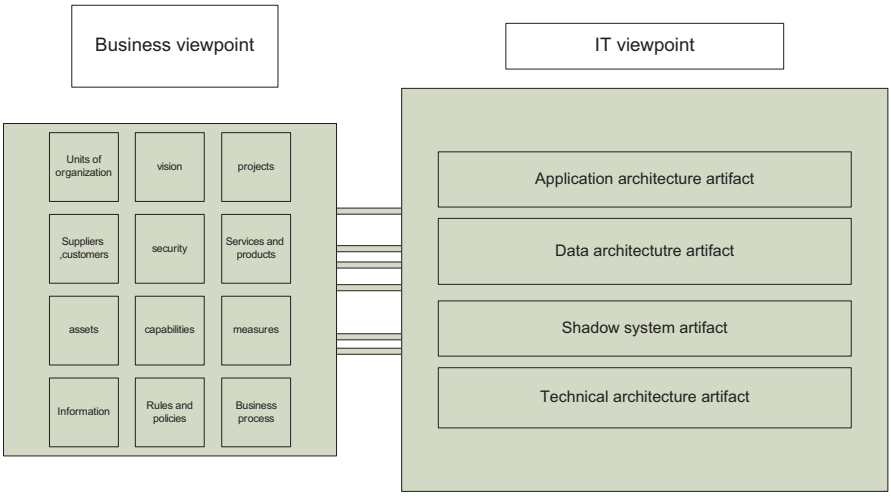


Fig. 2.2 The intertwinement between business and IT views

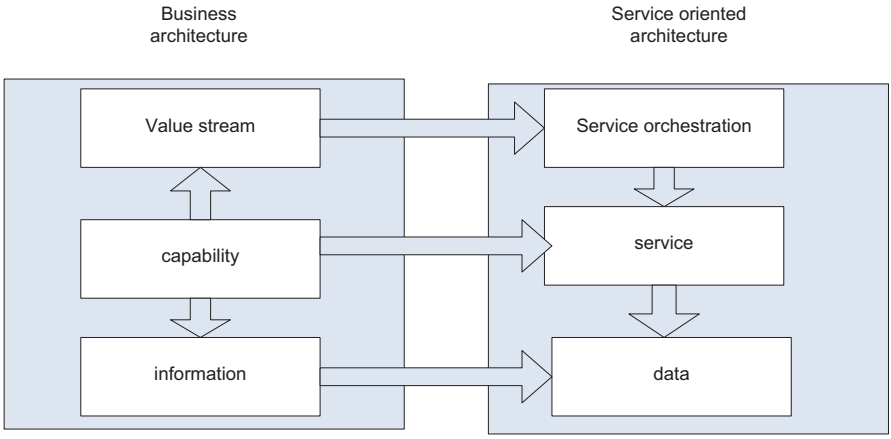


Fig. 2.3 Business architecture and service-oriented architecture

architecture must provide ways in which the business and IT artifacts can be related as necessary to support a given set of objectives for a customer.

At the end, business and IT architecture can be interconnected to assist analysis, planning, and evolution of the ecosystems through visualization and simulation.

Figure 2.2 shows the relationship between business and IT.

Figure 2.3 shows how business architecture is depicted in service-oriented architecture.

2.3 Pragmatic Architecture Today

Architecture definition starts early in the life cycle of the project, where requirements and the scope of the project are still not clear. This is why, when understanding the problem of the project, the architecture stage tends to be a more fluid activity than other stages in a life cycle.

Pragmatic architecture is about adopting the approach that works best for the architect, and what is best for one architect is not necessarily best for another architect. Also, the architectural activities of the adopting approach should be pragmatic because it must take into account the important issues such as lack of time or money, shortage of specific technical skills, and unclear or changing requirements.

The pragmatic architect focuses on essential concrete tasks and prioritizes the work according to the value it brings to the project to achieve the basic goals for functional and nonfunctional requirements. Essentially the pragmatic architect thinks of decisions and the effect of these decisions on the design and then considers these decisions as part of the solution to the problem.

I like the set of goals that is written in one IBM blog. They set the basic goals of *p.r.a.g.m.a.t.i.c* architectural activities as follows:

P: Promote collaborative work that involves all team members.
R: Reduce risks and uncertainties.
A: Adhere to minimalism and simplicity principles.
G: Gather key elements to outline the architecture during your initial timed-boxed iteration.
M: Modify the design throughout the development life cycle to adapt to emergent needs.
A: Address both functional and nonfunctional requirements.
T: Try out theoretical concepts and leverage past empirical experience.
I: Invest in known requirement instead of hypothetical future needs.
C: Concentrate on tasks that support the development team.

2.4 Business Architecture's Roles in Management

Business can be defined as an activity that provides services or goods to meet society needs and aims to get profits. Business is classified according to size into types: micro, small, medium, and large. Note that the size depends on the number of employees working in the organization.

Every business in any organization has three main elements in common:

- The business is for the organization and this organization has its own goals.
- Each business has people working together.
- Each business has its own structure.

Helpful hints from Henry Ford about business:

- Coming together is the beginning.
- Keeping together is the progress.
- Working together is the success.

Important business objectives are survival, stability, growth, profitability, and efficiency.

Business architecture is part of enterprise architecture. The formal definition of business architecture is the one that comes from the Object Management Group. It is defined as "a blueprint of the enterprise that provides a common understanding of the organization and is used to align strategic objectives and tactical demands."

In order to describe the role of business architecture in the management, some basic concepts need to be explained.

Concept 1: Business architecture and its relation to the business process model

A business process model is the activity of representing processes of an enterprise; it is the most important part of business architecture.

The business process is composed of subprocesses which are composed of sets of activities. The output of the business process is the set of business goals, and from business goals the role of software architect starts.

The software architect derives the architecture from the business goals by choosing the best design decisions according to it (Fig. 2.4).

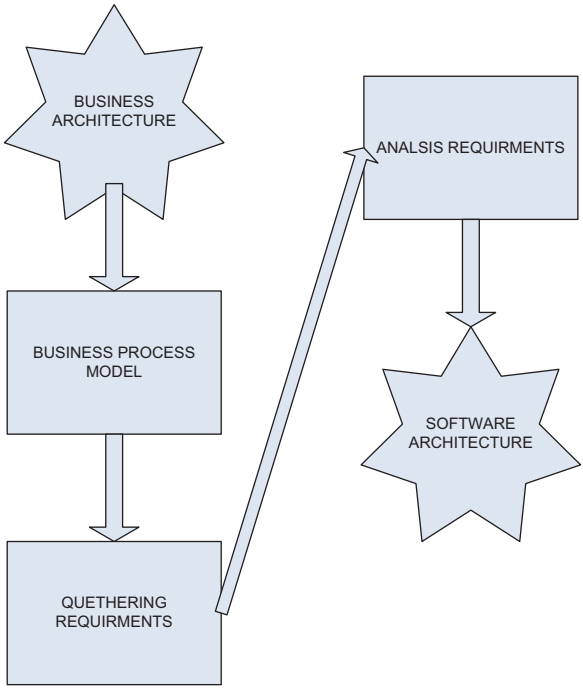


Fig. 2.4 The relationship between business architecture and software architecture

The business process is the best way to achieve software business alignment; it is a way to begin building the architecture of software according to business goals. And if you can do that, the result from business processes is that the architecture will be strongly connected to business strategy.

So the important role of business architecture in management is building a solid architecture and supporting it with software architecture.

Concept 2: The role of the business architect

Business architects focus on business and IT alignment. Their role extends from addressing efficiency of business and IT issues to being involved with more strategic priorities. Also some of them focus on efficiency goals.

Business architects act as a link to the technology groups to enhance the development of business and have skills in business evaluation of technology. According to the business model, the business architect owns and develops business models which will clarify their technology strategy and participate in any business planning activity which makes use of these models.

At this point, it is clear that the business architect plays an important role in business architecture especially through working on process models and prioritizing the strategy.

Successful business architects capabilities should focus on:

- Business design
- Business management and strategy
- Technology strategy
- Portfolio, program, and project management
- Financial methods

Business architects are targeting business effectiveness through:

- Improved EA-business alignment
- Improved IT efficiency
- Improved business efficiency
- Improved business-IT alignment

Table 2.1 Examples on the success of BA result measures

Category	Metric examples
Business architecture values	1. Number of redundant components that reduced 2. Cost saved 3. Cost avoided 4. Number of identified risks 5. Customer satisfaction score
Business architecture progress	1. Business architecture stakeholders satisfaction score 2. Number of business architects 3. Number/percentage of business architects trained 4. Number/percentage of required teams integrated with 5. Number of a required domains staffed with a business architect

- Improved business effectiveness
- Widespread business transformation

Concept 3: Measuring the success of BA
The most important value that results from business architecture is to increase the organizational adoption and responsiveness of business architecture by the business, and this will expand the usage of the business architecture.

Also measuring the progress and effectiveness of business architecture can be useful information for business architecture leaders and practitioners to show the progression to their stakeholders.

There are two ways to measure the results of business architecture: the first is by measuring the effectiveness and the progress of a business, and this will be relatively simple and straight. The second is by measuring the value provided by the business architecture which is more challenging for some reasons, for example, some of business architecture results are intangible, some business results are hard to isolate, etc. Table 2.1 shows some examples on both types of measures..

In conclusion, these three concepts show the importance of business architecture in management; they begin by showing the relation of business goals and building a good software architecture and end with the measurement of a good business architecture going through the role of business architect in a project.

2.5 Summary

Business software architecture is a term that combines two basic concepts: business and software architecture. It is very important for enterprises that need to develop their business continuously to satisfy their customers. It can be defined as the structure of a business in any enterprise or organization used by business users as well as customers to perform the functions of the business. Software architecture is a dominator of this definition because it helps businesses reach their goals with high quality.

This definition has a deep impact on the long-term success of the business using it. Such an impact can change related things such as:

- Costs of maintenance and development
- Availability of skills and expertise
- Competition with the business

A good architecture of software enables a business to respond to the changes with little additional cost.

Business architecture ecosystem, defined in this chapter, represents the essence of the business. It is one or more legal entities that exist as an integrated community of individuals, or aggregations thereof, interacting as a cohesive whole toward common purposes.

Pragmatic architecture is about adopting the approach that works best for the architect. The architectural activities of the adopting approach should be pragmatic also. It must be pragmatic, because it must take into account the important issues such as lack of time or money, shortage of specific technical skills, and unclear or changing requirements.

Two basic roles of BSA are taken into consideration: they are software architects and business managers.

References

A. Randell, E. Spellman, W. Ulrich, J. Walk, Leveraging Business Architecture to Improve Business Requirements Analysis, A Business Architecture Guild Whitepaper (2014)

W. Ulirich, N. Mchorter, Defining requirements for a business architecture standerd, A white paper (2010)

Further Reading

97 Things Every Software Architect Should Know, Collective Wisdom from the Experts (OReilly, 2009)

A guide to the Business Architecture Body of Knowledge http://c.ymcdn.com/sities/www.businessarchitectutreguild.org/resource/resmgr/bizbokv6/bizbokv6glossary.pdf

L. Bass, *Designing Software Architecture to Achieve Business Goals* (Software Engineering Institute, Carnegie Mellon University, 2010)

R. Kaman, L. Bass, Categorizing Business Goals Software Architectures, Technical report CMU/SEI (2006)

R. Kazman, L. Bass, Toward Deriving Software Architectures From Quality Attributes, Technical report CMU/SEI (1994)

Measuring business architecture success, whunde kuehn References published on cutter consortium (http://www.cutter.com, July 2016) Source URL: https://www.cutter.com/article/path-successful-business-architecture-practice-491846

K. Sandkuhl, *Aligning Software Architecture and Business Strategy with Continuous Business Engineering*, Springer International Publishing, LNBIP 286 (2017), pp. 14–26. https://doi.org/10.1007/978-3-319-60048-2_2

The Role of an Architect, www.architecturejournal.net, microsoft. 2008

W. Ulirch, Business architecture's role in re architecturing technology solution author by (http://www.cutter.com) (2017)

Chapter 3
Understanding and Dealing with Qualities

Abstract Quality attributes are the important factors that are used to establish the architecture. There is a constant need to redesign the systems because they are being difficult to maintain or to scale, they are too slow to continue their operations, or they are easy to hack. These are reasons constrain the quality attributes to be the architect's goal through building the architecture of the product. Quality attribute is a measurable part to indicate that the system is satisfying the stakeholder's requirement. The requirements that the architect needs are the ones that directly influence the architecture itself. They play an important role in determining the architecture of the system. Those requirements are called architecturally significant requirements (ASR); such requirements require special attention. Not all requirements have equal significance with regard to the architecture.

This chapter also focuses on the relation between architecture and quality. It shows that the degree of achieving the qualities of the system depends on its architecture; that is why architecture provides the foundation for building high-quality products. The main thing to discuss in this chapter is the meaning of trade-off between qualities and the models that are used to capture them.

Quality Attribute Scenario and Quality Attribute Workshop are two important tools that are used to gather information from stakeholders to reach the important qualities of the system. In this chapter, all the previous concepts will be described in detail.

At the end of this chapter you will learn:

- The meaning of architecturally significant requirement (ASR) and the method that is used to gather this type of requirement
- The meaning of trade-off and the model that is used to capture it
- Techniques that are used to gather information on quality attributes: Quality Attribute Scenario (QAS) and Quality Attribute Workshop (QAW).

L. Khalid, *Software Architecture for Business*,
https://doi.org/10.1007/978-3-030-13632-1_3

3.1 Definition of Quality

A quality attribute is a measurable property of the "goodness" of the product in terms of the stakeholders' requirements (the types of stakeholders will be defined later in Chap. 5).

General software quality attributes include scalability, security, performance, and reliability. These are often informally called an application's "-ilities."

The qualities that will be mentioned are the ones that are relevant for the types of systems in the last three chapters of this book.

Quality attributes are closely related to the functionality of the product. Functionality is *what* a product does, while quality is *how* it does it. Sometimes you see the term quality being used synonymously to the nonfunctional requirements in some books or articles. Specialists in this field do not really like this synonym; they say that a quality in its origin is functional and that was a solid reason not to refer to the quality as nonfunctional requirement.

Quality attributes are usually divided in two main groups based on the quality they are requesting. The first group is those that describe the system at runtime (sometimes called operational qualities) such as performance and usability qualities. The second group is those that describe some properties of the development system (sometimes called development quality) such as maintainability, understandability, and flexibility.

Software quality is becoming an important part in architecture, helping the architect to deal with the complexity of large systems. It is the role of the architect to understand the importance and the priority of each quality requirement. He needs to make the appropriate trade-offs in order to meet the quality levels that are expected by the stakeholders.

3.2 Software Qualities for the Product

According to ISO/IEC 12207:1995, software product is defined as a set of computer programs and associated documentations. This is a very general definition; the product has to be defined from the business point of view as follows:

Software product is the application that is built to accommodate the customers' needs, and from here you can set the specific qualities for that application.

Going back to qualities, software quality for the product (SQP) can be defined as a specific quality for that product, and its specification comes from the stakeholder's point of view to achieve business goals of the product to enhance its productivity.

To evaluate qualities in the product, quality model is used. Architects always build models in order to understand and measure these qualities. The long-term goal of this measurement is to judge these qualities. Some qualities like maintainability and understandability are very difficult to measure because they are affected by some factors such as user experience and education. This is why their internal

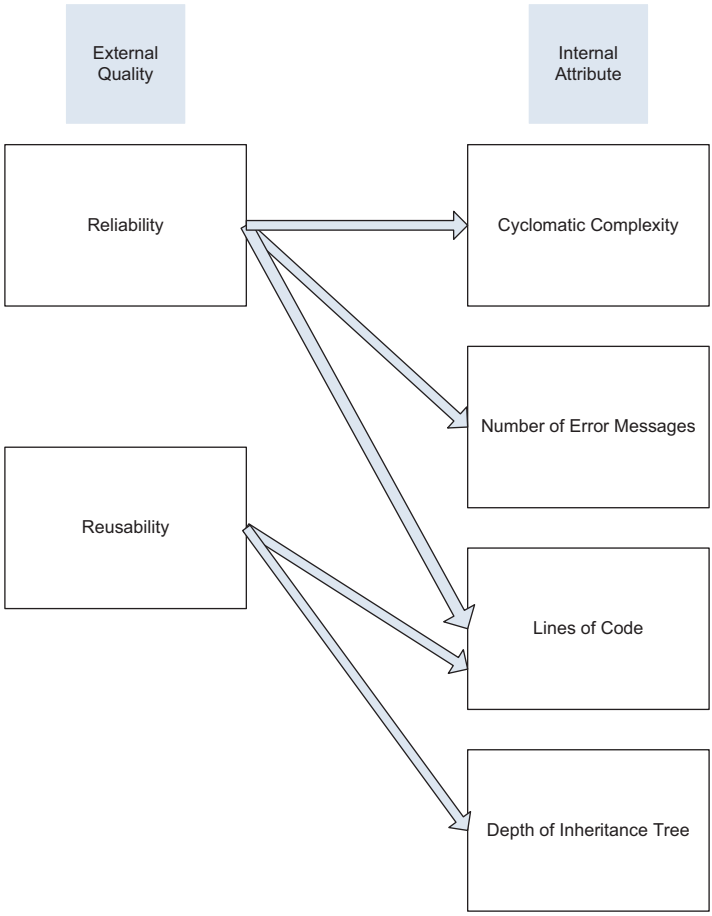


Fig. 3.1 The external software quality attributes and their related internal quality attributes

attributes such as size and complexity have to be measured first. Figure 3.1 represents the external qualities and their related internal attributes.

> *Cyclomatic complexity* is a software metric used to show the complexity of a program. It is a quantitative measure of the number of linearly independent paths through a program's source code. It is computed using the control flow graph of the program. Cyclomatic complexity may also be applied to individual parts in a program such as functions, modules, methods, or classes.

Figure 3.1 shows the relationship between internal and external quality attributes, but it does not show how these attributes are related. Three main conditions should be taken into consideration:

- The internal attribute must be measured accurately.
- A relationship between the attributes' values must exist.
- The relationship between internal and external attributes must be understood, validated, and expressed in terms of model or formula.

There is little information about the current use of systematic software measurement in industry due to many reasons, and some of which are:

- Difficulty in quantifying the return on investment (ROI) of introducing an organizational metrics program.
- Lack of standards for software metrics or standardized processes for measurement.
- In many companies, software processes are not standard and are poorly defined.

There are many other quality models used to measure the qualities; you can find these in suggested references.

No matter the type of business, every industry's goal is to increase its return on investment (ROI). Software qualities are very important goals for the business for the following reasons:

Predictability: software qualities drive predictability, and architects need to build the system in a predictable way. This means do it once and do it right with less rework and less variation in productivity. The architect's goal is to always build the product on time with productivity.

Reputation: this reason is a powerful driver. Some companies have a reputation in building their products with high quality, and of course customers always look for this type of companies. A good and solid reputation is hard to establish and easy to lose. Once the company has this feature, it will be a good driver for business.

Customer satisfaction: this is the most important reason from my point of view. The customer's trust is very important and is strongly driven by the quality of software that the company produces and the services it provides.

3.2.1 Architecture Quality Attribute and Business Quality Attribute

The general definition of business qualities is those qualities that are applied to a specific product to meet the customer's expectations, and I think there is no big difference with the other types of qualities. Business quality goals are to add values for the business. For example, any car can take you from point A to B; what makes the difference are the features that are qualities other than transport, such as comfort

> *Business goal* is the achievement of the company in a specific time.
> *Business quality* is a quality which satisfies the customer's need.

and the usability of its components. Another benefit for qualities in business is that they reflect the nature of the process in the product; high-quality products always keep the company from wasting time and refactoring and rewriting software.

You should know the difference between business goals and business qualities. Business goals describe what a company is supposed to achieve over a specific period of time. Businesses usually summarize their goals in their business plans. While a business quality is a quality which satisfies the customer and aims for a specific purpose, the quality is meaningless if it is not related to a specific function.

3.3 Architecture and Quality

Software architecture base on quality attribute have a strong relationship in any system. This type of relation is specified through deriving software architectures from a concern of the qualities of that system. These qualities (portability, performance, modifiability, and scalability) are thought to be presented in any system, and this is shown when using the architecture.

The conclusion of this relation will be as follows:

- The achievement of the qualities of a system is connected with the software architecture for that system.
- These qualities can be achieved through the appropriate application of a set of operations.
- Specific architectures can be derived from an understanding of the operations, and the qualities need to be achieved by that architecture. Such operations are abstraction, compression, separation, and other process that are used through building architecture.

Fig. 3.2 shows how the architecture is constructed from requirements. The method transforms the requirement into allocations, each one isolated from other and then forming the architecture. The difficult part in defining qualities is that they cannot be described in abstract; for example, it is not right to say portability is the most important feature in the system. Two decisions must be taken into account through building the architecture: the first is determining the required qualities of the system and the second is prioritizing them according to scenarios that are written by the architect.

The important thing to know is that *no* quality is completely dependent on the design, nor is it entirely dependent on implementation or deployment. Satisfaction is the goal of the architecture, in addition to the correct implementation. For example, performance entails both architectural and nonarchitectural dependencies. It depends partially on how much communication is necessary among components (architectural dependency), partially on what functionality has been allocated to each component (architectural dependency), partially on how shared resources are

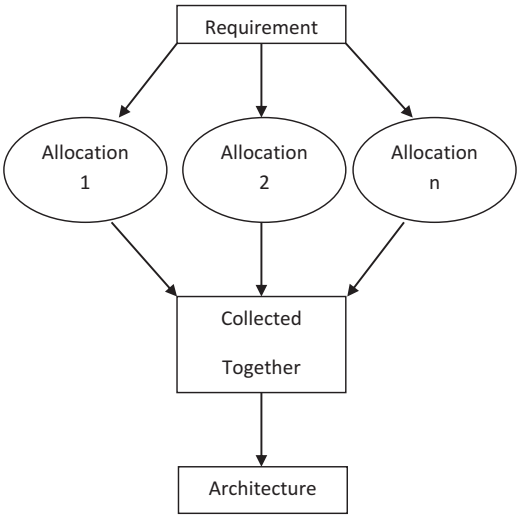


Fig. 3.2 Software architecture base on quality attribute

allocated (architectural dependency), and on implementing selected functionality (nonarchitectural dependency).

In conclusion, the degree of achieving the qualities of the system depends on its architecture; this is why architecture provides the foundation for building high-quality products, but this foundation will be with no benefit if there is no attention to the details. Also any change to the architecture to achieve one quality will impact (either positively or negatively) other quality attributes.

3.3.1 Architecturally Significant Requirement (ASR)

The architect builds the system in order to satisfy the system's requirement. The most important thing is that the architect always looks for the specific type of requirements that affect the architecture: architecturally significant requirements (ASRs).

Architecturally significant requirements are those that play an important role in determining the architecture of the system. Such requirements require special notice. Not all requirements have equal significance with regard to the architecture. ASR often, but not always, takes the form of quality attribute.

Methods that are used to gather ASR are:

3.3.1.1 Interviewing Stakeholders

The first thing the architect does is talking with the stakeholders. The architect starts his work by gathering information that are needed to produce the architecture.

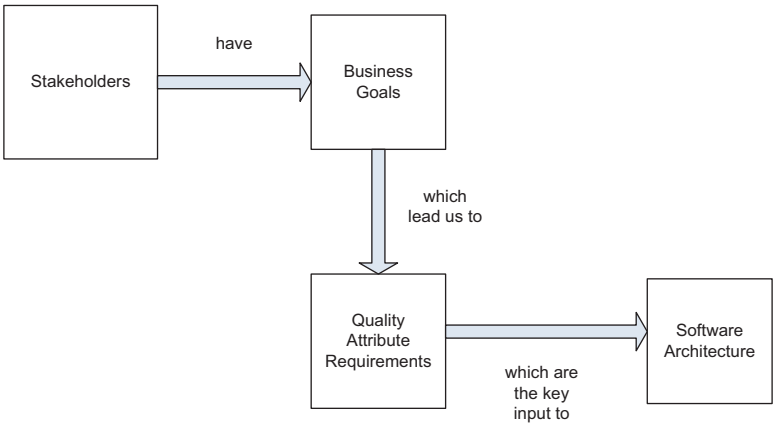


Fig. 3.3 Stakeholder and software architecture

Interviewing stakeholders is the main way to learn what they need, to produce the architecture. The result of this interviewing is a set of architectural drivers and set of Quality Attribute Scenario (QAS) that the stakeholder prioritizes. Two important techniques are used to gather information from stakeholders: QAS and QAW.

Figure 3.3 shows that every stakeholder has business goals (sometimes called missions) and objectives that lead to quality attribute requirements and that will be a key input to software architecture.

3.3.1.2 Gathering by Business Goals

This type of method arises because business goals often lead to ASR. You must know that the business goals are the most important reasons for building the system.

Sometimes business goals are related to quality attributes that lead to building the architecture, and sometimes the architecture is built without any relation to quality attribute.

Figure 3.4 shows three output arrows from business goals:

- *The first arrow* shows that business goals often lead to quality attribute requirements, and this will differentiate the output product from other competitor products and let the developing organization capture market share.
- *The second arrow* shows that business goals directly affect architecture without needing the participation of quality attribute requirement at all.
- *The third arrow* shows no influence at all, for example, when needing to reduce cost.

In conclusion, the important arrows to the architect are the solid ones.

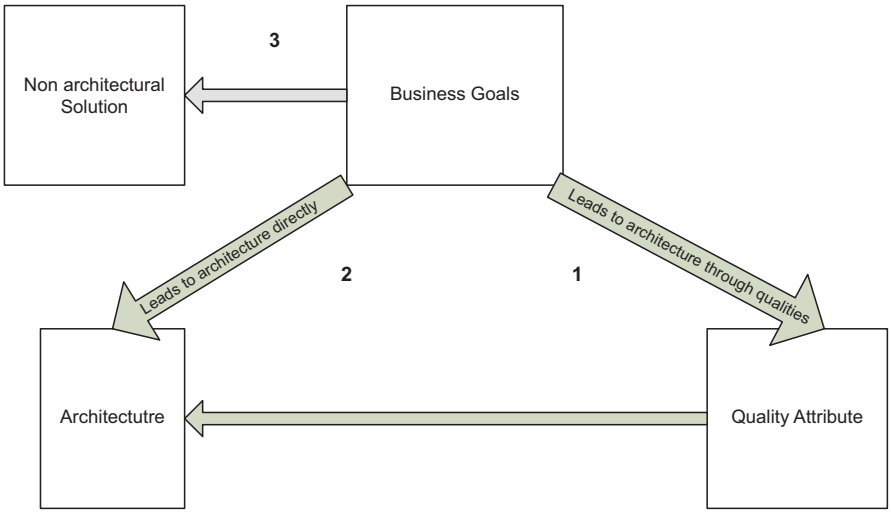


Fig. 3.4 The ways of business goals lead to architecture

3.3.1.3 Gathering Requirements Through Utility Tree

Utility tree is constructed during ATAM evaluation to elicit and prioritize quality attributes. The word utility represents the goodness of the system. Under each quality there is a specific refinement related to that quality; for example, availability is related to hardware failure and backup/recovery refinements. The refinement you choose should relate to the system you build, under each refinement there is ASR connected with it.

Note that ASR may exist in more than one place in the tree which means that ASR may relate to more than one quality. There is a scale used to evaluate the importance of the requirements, such as high, low, medium scales.

> *Utility tree* is a way to categorize the quality attributes. Utility is an expression of the overall "goodness" of the system

3.3.1.4 Gathering ASR Through Requirement Document

Requirement documentation is a documentation that contains all the requirements of a certain system (or product). It is the place that records all candidates ASR. Not all requirements specified in the documentation are important to the architect; the architect needs the requirements that affect the design decisions to the architecture.

At the end, the ASR can be extracted from multiple techniques, two of which are focused on in this book: QAW and QAS. Other types of techniques are used through other stages: requirement documentation through requirement phase and utility tree through evaluating architecture phase.

3.3.2 Qualities and Trade-Offs

According to the business dictionary, a *trade-off* is a technique of reducing one or more required results in exchange for increasing other desirable results in order to maximize the effectiveness in a given situation.

Every software engineering project that is built or developed for specific customers involves several stakeholders of which the corresponding quality attributes are extracted. It is possible to facilitate existing knowledge and solutions for those quality attributes. Existing knowledge might contain suitable solutions or known conflicts with other quality attributes. The complexity and number of possible conflicts grows with the number of identified quality attributes and the number of ideologically different stakeholders. In the general sense, trade-off means that *you get something here, you give something there*. The job of the architect plays an essential role between the business requirements and the realization, the implementation needs a person who aims to balance the trade-offs.As with architecture there are common accepted relations between software quality attributes; however, these relations have more or less impact based on what architectural pattern is used for the final solution. The stakeholders and the trade-offs are accepted with conditions, implying that the minimum value of quality attributes they need must be satisfied. If not, then the solution is unacceptable. For example, if usability is the most prioritized quality for the system, it is probable that efficiency may be affected. However, this situation is acceptable, as long as efficiency is still satisfied to the suitable level.

Many models are used to capture the architectural trade-off, for example, NFR (nonfunctional requirements) framework. This type improves the application by structuring goals (nonfunctional requirements) and solutions. In the NFR framework, the terminology “operationalizations” is used. The solutions can affect the goals depending on the characteristics of the solution and the requirements that are shown through the goals. Figure 3.5 represents the performance goal with two subgoals in the level below it. The solutions in the third level are associated with subgoals and how each one is affected.

Determining what solutions affect other requested quality attributes is an essential part of selecting the final architecture to implement.

There are also model- or scenario-based architecture evaluation techniques used to achieve the prioritized quality attributes. Two of these models are SAAM (Software Architecture Analysis Method) and ATAM (Architecture Trade-off Analysis Method).

The best solution is the one that affects the quality attributes the least.

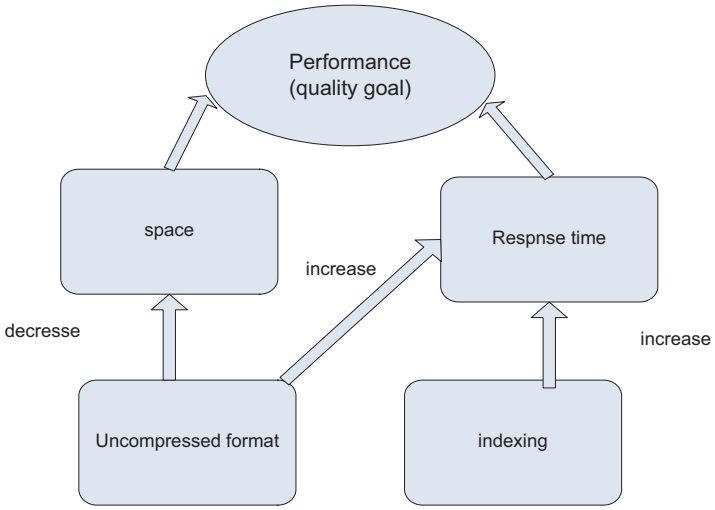


Fig. 3.5 Operationalizations technique

3.4 Gathering Quality Attribute Information

As what is mentioned before, quality attribute requirements are very important to design software architecture. These quality requirements and design constraints are the main things for structuring the software architecture in the beginning. The two main techniques are Quality Attribute Scenario (QAS) and Quality Attribute Workshop (QAW).

3.4.1 Quality Attribute Scenario (QAS)

Quality Attribute Scenario (QAS) was first introduced as a technique to gather information for the qualities in an operational form in order to build an efficient architecture of the system. QAS appears to solve the untestable and overlapping concerns. Before going into more details in the concept of QAS, I want to describe what untestable and overlapping problems mean. The system should not be considered robust if it is only robust in one aspect. The quality attribute should be tested in all the circumstances. As for the concern of overlapping, the discussion must focus on which quality a specific concern belongs to. For example, is a system failure due to a denial-of-service attack an aspect of availability, an aspect of usability, or an aspect of performance? All these qualities would claim ownership of a system failure through a denial-of-service attack.

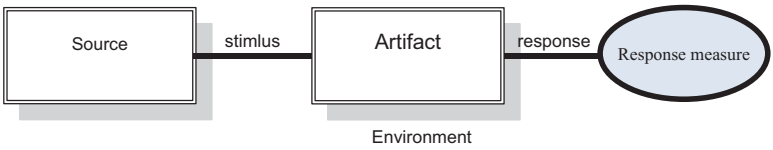


Fig. 3.6 Basic parts of QAS

Another problem may appear when each community has developed its own vocabulary.

The aim of a QAS is to capture the explicit and testable quality requirements in the same way use case scenarios do for functional requirements.

QAS consists of six parts (Fig. 3.6):

1. Source—an entity that produces a stimulus such as humans or a computer system.
2. Stimulus—a condition that affects the system.
3. Artifact—the part of the system that was stimulated by the stimulus; it may be a collection of systems, the whole system, or some parts of it.
4. Environment—the set of circumstances in which the scenario takes place.
5. Response—the activity that results because of the stimulus. The response consists of the responsibilities that the system should perform in response to stimulus.
6. Response measure—the measure by which the system's response will be evaluated.

In the environment, the source throws the stimulus and hits the system in the artifact.

The response measure makes the scenario operational. It is not enough to say the system is fast without knowing what the word fast really stands for; for some people this may be in 100 milliseconds, while according to others, it means 2 seconds. The response measure indicates what is acceptable. The important thing to know is that the response measure does not have enough information to evaluate the architecture. For example, is the response time applied within two clicks from the user?

There are two types of QAS: general and concrete. The former are the scenarios that are general and do not belong to any system. The latter are scenarios that belong to a particular system under specific conditions. General scenarios allow the stakeholder to communicate effectively and help him develop a concrete scenario for the system. In general, QAS can be used as tool by stakeholders to help specify and prioritize the requirements of the architecture.

To conclude, Quality Attribute Scenario is about how a system is required to respond to some stimulus. General concepts of QAS for the availability quality are shown in Table 3.1.

When a stakeholder comes to the architect and says that they want an available system, the architect writes a general scenario for this type of quality. After that, the

Table 3.1 General QAS for availability quality

Source	The source can be internal or external, for example, people, hardware, software, physical infrastructure, etc.
Stimulus	Fault
Artifacts	Processors, communication channels, persistent storage, process
Environment	Normal operation, shutdown, repair mode, overloaded operation
Response	Prevent fault from becoming failure Detect the fault Recovery from the fault
Response measure	Availability percentage, for example (99.999%), time to detect fault, time to repair fault, time or time interval in which a system can be in a degraded mode, etc.

Table 3.2 Extracted table

Source	The source can be *internal* or external. For example, people, *hardware*, software, physical infrastructure, etc.
Stimulus	Fault: incorrect response, incorrect timing, *crash*
Artifacts	*Processors*, communication channels, persistent storage, process
Environment	*Normal operation*, shutdown, repair mode, overloaded operation
Response	Prevent fault from becoming failure *Detect the fault* *Recovery from the fault*
Response measure	Availability percentage, for example (99.999%), time to detect fault, time to repair fault, *time or time interval in which a system can be in a degraded mode, etc.*

Table 3.3 Concrete table

Source	Internal hardware
Stimulus	Crash
Artifacts	Processors
Environment	Normal operation
Response	Detect the fault Recovery from the fault
Response measure	System can be in a degraded mode no more than 15 minutes

stakeholder chooses the parts that he wants to be in his system. For example, the chosen parts from Table 3.1 are put in Table 3.2, and that will lead to the concrete scenario (Table 3.3).

Appendix A shows some examples on general scenarios for qualities and the concrete scenarios that can be derived from them.

Table 3.4 Concrete performance QAS

Source	A *bus subsystem* is the source of this scenario
Stimulus	The event here is *sending a message* with its location and speed every 15 seconds
Artifact	Artifact will be to the system or part of it here it will be to the *central system*
Environment	It will be during the *normal operation*
Response	The system will react with these actions so it will produce an *estimated arrival* time for all relevant display and then send it out
The response measure	The response measure here will be within *30 seconds* under normal conditions, and this will be after receiving the message from the bus

3.4.1.1 Example on QAS

Smart city transport system is a simple example of QAS. In this system each bus will have a subsystem that sends its speed and actual location to a central server. The central server then calculates the estimated arrival times for each bus; the performance is the quality that we need in this system. The concrete QAS of the performance quality to a smart city transport system is shown in Table 3.4.

The important thing to know is that a response time of 30 seconds may only be required under normal conditions, and that leads to create a second QAS under peak load conditions where we allow a response of 30 seconds with a jitter of 10 seconds.

> *A jitter* is the delay that varies over time. In the context of computer networks, jitter is the variation in latency as measured in the variability over time of the packet latency across a network. This means that a network with a constant latency has no jitter.

3.4.2 Quality Attribute Workshop (QAW)

Quality Attribute Workshop is a facilitated method and a few-day workshop that is developed by SEI. It connects stakeholders in the early part of the life cycle in order to find quality attributes for the existing system. The system must exist in the real world in order to go through this technique of elicitation.

The important thing to know about QAW is that:

- It is focused on the stakeholders.
- It is scenario based.
- It is used before the software architecture begins.
- It is focused on the system level concerns and on the role of software in the system. Here the term “system level” is used because the development of the soft-

ware begins with the description of the system's operation, a high-level functional requirement, and any constraint on the system if the system is new or legacy.

QAW is one way to find, document, and prioritize the important qualities because quality attributes are often missing in the documentation of the requirement or even if exists they are weakly defined. The purpose of this workshop is to understand what quality attribute requirements important to stakeholders are and then use them as input into the design of architecture.

3.4.2.1 QAW Steps

The contribution of each stakeholder is very important during QAW; all participants are expected to engage or attend the workshop. The QAW involves the following steps:

Step 1: QAW presentation and introductions

Facilitators explain the goal of QAW and explain each step in the method by using standard slide presentation. Next, they introduce themselves and stakeholders do the same. They briefly state their background, their roles in the company, and their roles in the system.

Step 2: Business/mission presentation

The stakeholder represents the business concerns of the system. The quality attribute of the system will be derived from the business mission needs. The manager usually does the representation and it includes the following:

1. The business/mission context
2. The high-level functional requirements, constraints, and quality attribute requirements

Step 3: Architectural plan and presentation

The technical stakeholder will present the system architectural plans according to the early document. It will contain the following information:

1. The strategies on how to satisfy the business/mission requirements
2. Key technical requirements and constraints that will derive architectural decisions
3. The high-level system diagram and context diagram

Step 4: Identification of architectural drivers

The facilitators in this step will share their list of architectural drivers that are assembled in the previous step. The idea is to reach the architectural driver that includes all the requirements, business drivers, constraints, and quality attributes. This will convince the stakeholders that these concerns are according to the scenario that is collected.

Step 5: Scenario brainstorming

After defining the architectural drivers, the facilitator initiates the brainstorming process in which stakeholders generate scenarios. Facilitators ensure that there is at least one scenario for every driver. The facilitators also reviewed the parts of scenario (stimulus, event, response, etc.) and ensure they will be formed well because the scenario is a key step in QAW and must be carried out with care.

There are some suggested points that come from SEI report (by Mario R. Barbacci et al.) to help the facilitator during this step:

- Facilitators should help the stakeholders create well-formed scenarios. For example, the stakeholders write the requirements such as "the system shall produce reports for the users." This requirement is important but the role of the facilitator is to show the quality attribute of it.
- The vocabulary used to describe qualities varies widely. It is not important what we call "quality," as long as there is a scenario that explains what is meant by it.
- The facilitator should always refer to the list of architectural drivers generated in step 4 to ensure the scenarios exist for each one.

The facilitator usually plays the role of the team leader who types everything during the meeting

Step 6: Scenario consolidation

This is an important step because facilitators usually ask the stakeholder to identify the scenarios that are similar in contents. Scenarios with similar contents are merged, as long as the people who suggested them agree that their scenarios will not affect the process. The consolidation process helps prevent votes from multiple scenarios which have the same concerns. Actually, very few scenarios are merged.

Step 7: Scenario prioritization

This step is accomplished by allocating each stakeholder a number of votes equal to 30 percent of the total number of scenarios that are created after consolidation. For example, if 30 scenarios were created, each stakeholder gets 30×0.3 or 9 votes. The votes are counted and the scenarios are prioritized accordingly.

Step 8: Scenario refinement

After prioritizing, the top four or five scenarios are refined in more detail. Facilitators intricate each one and document the following:

- A clear description of the parts of QAS (stimulus, event, response, response measure, artifact, and environment)
- The business/mission goals that are affected by the scenario
- The related qualities associated with scenario
- The questions and issues of the stakeholders

3.4.2.2 Advantages of QAW

QAW has many benefits; the most important ones are:

- Increasing stakeholder communications because it is the first meeting between the participants of the system
- Clarifying quality attribute requirements
- Creating a base for architectural decisions
- Improving architectural documentation
- Supporting analysis and testing throughout the life of the system

The results of QAW include:

- A list of architectural drivers
- A list of scenarios and the prioritizing the most important one
- Refined scenarios

And these results lead to:

- Updating the organizational architectural vision
- Understanding the system's architectural drivers
- Refining requirements
- Guiding the development of prototypes
- Exercise simulation

All the benefits of this information will help the architect design the architecture of the system. The scenario can also be used to evaluate the architecture if the ATAM is used for evaluation; the scenarios that are generated from QAW will be incorporated with ATAM.

In addition to the evaluation, the documentation has its benefits from scenarios; for example, refined scenario can be documented as a sequence diagram or a collaboration diagram. All stakeholder concerns and any other information should be recorded individually in a form of packet overview documentation.

Also, the benefits of scenarios can extend to the implementation; it can be used to derive test cases development during implementation testing.

3.5 Summary

A quality attribute is a measurable property of the "goodness" of the product in terms of the stakeholders' requirements. Quality attributes are usually divided in two main groups based on the qualities they are requesting: the first is those that describe the system at runtime (operational qualities), and the second is those that describe some property of the development of the system (development qualities).

Software qualities are becoming an important part in architecture, helping the architect to deal with the complexity of large systems. It is the role of the architect to understand the importance and the priority of each quality requirement. He needs to make the appropriate trade-offs in order to meet the quality levels that are expected by the stakeholders.

Software qualities are very important goals for the business for the following reasons:

- Predictability
- Reputation
- Customer satisfaction

A *trade-off* is a technique of reducing one or more required results in exchange for increasing other desirable results in order to maximize the effectiveness in a given situation. NFR is a type of method that is used to improve the application by structuring goals (nonfunctional requirements) and solutions. In the NFR, the term "operationalizations" is used.

Quality Attribute Scenario is a technique to gather information for the qualities to build an efficient architecture of the system. A Quality Attribute Scenario consists of six parts:

- Source
- Stimulus
- Artifact
- Environment
- Response
- Response measure

On the other hand, Quality Attribute Workshop is a facilitated method and few-day workshop that is developed by SEI which connects stakeholders in the early part of the life cycle in order to find quality attributes for the existing system; it consists of eight steps:

Step 1: QAW presentation and introductions
Step 2: Business /mission presentation
Step 3 architectural plan and presentation
Step 4: Identification of architectural drivers
Step 5: Scenario brainstorming
Step 6: Scenario consolidation
Step 7: Scenario prioritization
Step 8: Scenario refinement

References

L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, 3rd edn. (Addison-Wesley, 2013)
I. Sommerville, *Software Engineering*, 9th edn. (Addison-Wesley, 2011)
M. Barbacci et al., Quality AttributeWorkshops (QAWs), 3rd edn. Tecnical Report CMU/ SEI, 2003
On line training from SEI: Software Architecture: Principles and Practices, http://ww.sei.cmu.edu/education-outreach/courses/online-training

Further Reading

Visiting the on line SEI library and searching on keywords of this chapter you will find a list of references as for example
Achieving Product Qualities through Software Architecture Practices by Linda Northop
Toward Deriving Software Architectures From Quality Attributes by Rick Kazman
Architecting high quality software: the role of software architecture in system development and evaluation by Linda Northop
A. Chandrasekar, S. Rajesh, P. Rajesh, A research study on software quality attributes. Int. J. Sci. Res. Publ. **4**(1), ISSN 2250-3153 (2014)
M. Barbacci, Software quality attributes and trade-offs, SEI, CMU, (2003)
G. Bessin, *The Business Value of Business Quality*, (IBM Cooperation, 2004)
J. Bergey, M. Barbacci, W. Wood, Using Quality Attribute Workshops to Evaluate Architectural Design Approaches in a Major System Acquisition: A Case Study, Technical Report, CMU/ SEI, July 2000

Chapter 4
Achieving Quality Attribute

Abstract Patterns help you build the architecture on the shared experience of skilled software engineers. They capture experience in software development and help to promote good design practice. Every pattern deals with a specific, recurring problem in the design or implementation of a software system. Architectural styles and patterns define the way to manage the components of the system so that one can build a system and also achieve the requirements of the stakeholders. Several architectural styles and patterns exist in the world of the software architecture, so one needs to understand which particular architecture style/pattern will be suitable through building the system. Architectural tactic is a design decision that influences the achievement of quality attribute response. Every pattern deals with many qualities, while tactic deals with a specific type of quality. The important thing to know is that most patterns consist of several different types of tactics that serve a common purpose. The relation between pattern and tactic is described in this chapter.

At the end of the chapter you will learn:

- Patterns and their roles in building architecture
- Relation between tactics and patterns and how to achieve the quality through these tactics
- Meaning of business patterns and the importance of these patterns in business
- The SEI attribute-driven design (ADD) method and how this method works and what it will use for

4.1 Introduction

Architects are the main ones responsible to solve recurring problems through building the architecture. Architects are responsible for making design decisions about achieving things that can only be achieved through architectural structure. So it is often the architect's first step to start choosing architectural patterns; this will help structure the architecture to solve different problems and then achieve a variety of qualities. This chapter defines what pattern is, the relation between pattern and tactics, and how one can use it to achieve high-quality products.

L. Khalid, *Software Architecture for Business*,
https://doi.org/10.1007/978-3-030-13632-1_4

4.2 Architectural Pattern

Architectural patterns are compositions of elements in a way that can be used to solve specific problems. These patterns will be useful over time and over many different domains, and that is the reason why they are documented and then distributed. Each pattern describes recurring problems that arise in specific design contexts and presents a solution for the problem. The definition of pattern can be summarized in one statement: it is a solution to a problem in a context.

The solution part describes the architectural structures that solve the problem. The actual solution represents the major elements of the pattern and the roles, responsibilities, and proprieties of these elements; it also describes the relation between these elements. Through writing patterns, it becomes easier to reuse the solution. An architect for a software system designs its architectural structures to solve a variety of design problems. The architectural structures designed by an architect are often based on one or more patterns.

Figure 4.1 shows the three types of structures:

- *Module structure:* it is a static way for considering the system; it shows how the system is structured as a set of codes or units. It is a good way to reason about the modifiability quality.
- *Component-and-connector structure:* it structures the system as a set of components and connectors. Here, components are runtime behavior, and that is why this type of structure is a good reason for performance, security, availability qualities, and even more.
- *Allocation structure:* in this structure, the decisions about how the system will relate to hardware and how teams will relate to their environment.

Appendix B shows a table about these structures; it lists the elements and type of relations and what they are used for.

Each pattern consists of a context, problem, and solution. This is shown in the following template (Table 4.1).

It is up to the architect to decide how patterns are initiated.

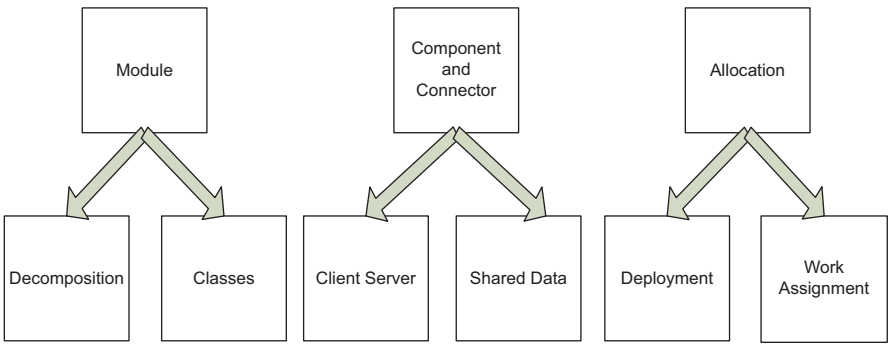


Fig. 4.1 Examples on software architecture structures

Table 4.1 A template for pattern document

Context	The situation that gives rise to a problem
Problem	Recurring problem that arises in the given context. The description of the problem often includes the quality attributes that must be met
Solution	A successful architectural resolution to the problem. The solution describes what quality attributes are provided by the static and runtime elements

4.2.1 Patterns and Their Roles in Building Architecture

To show the role of pattern in building the architecture of a system, an overview of pattern catalogue is needed. The list of the catalogues of this pattern used in this book are the ones that are more useful and used, for example, patterns of runtime such as broker or client-server and patterns of design time such as layer pattern. Definitions of patterns in catalogues are all strict, but in practice, the architect may choose to violate them when a trade-off is needed. For example, the layer pattern prevents software in lower layers from using software in upper layers, but there may be some cases (e.g., when a performance is needed) where an architect allows a small amount of specific exceptions.

Patterns can be categorized by the control type of elements: module patterns show modules, component-and-connector patterns show component and connectors, and allocation patterns show the combination between software elements (module, component, and connector) and non-software elements.

Let us begin with *module patterns*.

A common module *pattern* is the *layered pattern*: a layer is a coherent set of related functionality. In this pattern, the system is decomposed into a number of higher and lower layers in a hierarchy, and each layer has its own task in the system. The pattern layer is described as follows:

Context	Complex systems need to develop parts of the system independently. Modules of this system may be developed and maintained independently
Problem	The software needs to be segmented in such a way that modules can be developed and evolved separately supporting *modifiability, portability, and reuse*
Solution	To solve such problem, the layered pattern divides software into units called layers, and each of these layers consists of modules which can offer a set of services. Layers must be unidirectional, and this will be a constraint on the relationship

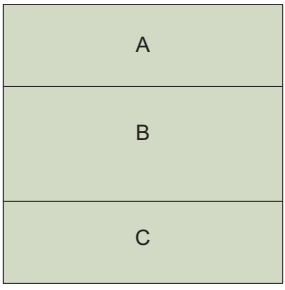
Layered pattern solution

Overview	The layer pattern defines layers (a grouping of modules) and unidirectionality among these layers. These layers are always drawn as a stack of boxes
Elements	Layered, a kind of module
Relations	Allow to use relation, which is a special kind and depends on relation
Constraints	Lower layer cannot use the above layer

Weakness	Layers contribute a performance penalty. While it is true that some layered architectures can perform well, the pattern does not lend itself to high-performance applications because it is not efficient to go through multiple layers of the architecture to fulfill a business request

Layers make the standardization easier because the abstraction of each level enables the development of standardized tasks and interfaces. The most stable abstractions are in the lower layer: a change of the behavior of a layer has no effect on the layers below it, while a change of the behavior of a lower layer has an effect on the layers above it, so that it should be avoided.

Networking protocols such as TCP/IP and web application are examples of this type of pattern.



Notation for layered design (Where A, B, C are layers)

From the notation of layer design, you can see three layers: A, B, and C. You can see that layers A and B are adjacent (each layer is a group of modules that presents a set of services). Layer A can use any public interface of layer B, but sometimes modules need modules in nonadjacent layers, and so a bridge layer is needed. If many bridging layers occur, portability and modifiability goals will not be achieved, and this will restrict this pattern. Upward usages are not allowed in this pattern

Component-and-Connector Patterns

- ***Broker Pattern***

The *broker pattern* is used to organize distributed systems with decoupled components, which interact by remote service invocations.

The broker pattern

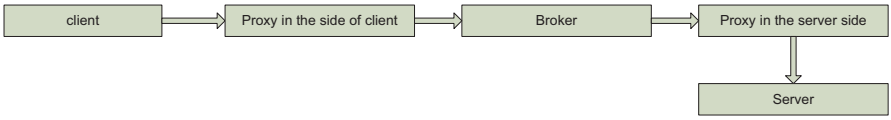
Context	When systems are constructed from many services that are allocated across many servers, then implementing such type of systems will be very complex
Problem	*Structuring distributed software so that service users do not need to know the nature and location of service providers*
Solution	To sol In such problem, *the broker pattern separates users of services* (*clients*) *from providers of services* (*servers*) *by inserting an intermediary, called a broker*

Broker pattern solution

Overview	The pattern defines runtime components that mediate between clients and servers, called brokers
Elements	*Client* which requests services *Server* which provides services *Broker*, an intermediate component between the client and the server *Client side proxy*, an intermediary that manages the communication with the broker *Server side proxy*, an intermediary that manages the communication with the broker
Relations	Attachment relation is used to communicate clients and servers with brokers
Constraints	The client can only attach to a broker through client proxy, and server can only attach to a broker through server proxy
Weakness	Many weaknesses can happen to a broker such as: It may be a target for *a security attack* It may be a *single point of failure* It may be *difficult to test*

The pattern provides all *modifiability* and *availability* benefits because it is very easy to replace a failed server with another. Performance is also one of the quality features of this pattern because the broker pattern easily assigns works to the least active server. The service-oriented architecture depends on this type of pattern, most commonly in the form of an enterprise service.

An example of the broker pattern is CORBA, for cooperation among heterogeneous systems and web services.



Notation for broker design

- ***Model-View-Controller Pattern***

In the *model-view-controller pattern*, or MVC pattern, an interactive application consists of three parts: the model contains the core functionality and data, the view displays the information to the user, and the controller is responsible of the input from the user. The MVC pattern is particularly suitable for multiple graphical user interfaces (GUIs).

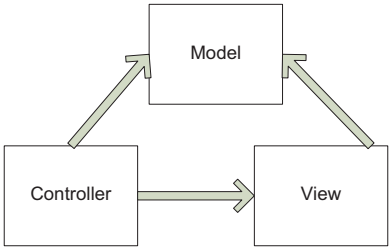
The MVC pattern

Context	User interface software is the most modified part in the user interface that is why it is important to keep this part separate from other systems
Problem	The idea is how to keep the functionality of this part (interface) from the rest of the application. Also how multiple views can appear to the user when the underlying data change

Solution	This pattern separates the functionality of the application into three main parts: *Model* which is the brain of the parts that contains the business rules and the application data *View* which is the user interface that is displayed on the screen *Control* which communicates between the user and the model

The MVC solution pattern

Overview	This pattern splits the functionality of the system into three parts: model, view, and control
Elements	*The model* is a representation of the application data as well as the application logic The view is the user interface part in the system The controller manages the interaction between the previous elements
Relations	The *notifies* relation connects between instances of three parts
Constraints	There must be at least one instance for each element
Weakness	MVC is not appropriate for every situation; it may be costly and that is not appropriate for simple user interface. Also the MVC parts may not fit to the user interface toolkit



Notation for MVC design

Because these components are loosely coupled, it is very easy to develop and test them in parallel, and any change to one component has the minimal effect on the others.

Usability, changeability, and testability are important qualities in this type of pattern.

Learning unit on patterns for enterprise application architecture is an example to this pattern.

- ***Pipe and Filter Pattern***

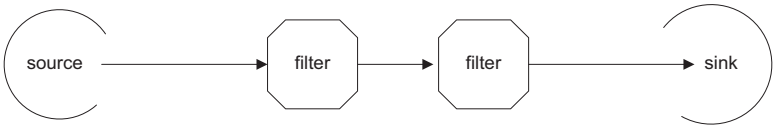
The *pipe-filter architectural pattern* produces a structure for systems that generate a stream of data. The flow of data is driven by data, and the whole system is decomposed into components of data source, filters, and pipes (the arrows between adjacent filters and data sinks). The connections between modules are streams of data which is first-in/first-out buffer that can be streams of bytes, characters, or any other type.

Pipe and filter pattern

Context	Many systems are required to transform streams of discrete data items, from input to output. Many types of transformations happen continually in practice, and so it is desirable to produce these parts as independent and reusable parts
Problem	These systems have to be divided into reusable and loosely coupled components with simple interaction between them. The components, being generic and loosely coupled, are easily reused. The components, being independent, can execute in parallel
Solution	The pipe-and-filter pattern is characterized by successive transformations of streams of data. Data arrives at a filter's input port(s) is transformed and then is passed via its output port(s) through a pipe to the next filter. A single filter can consume data from, or produce data to, one or more ports

Pipe and filter solution pattern

Overview	Data is transformed through a series of transformations performed by its filters connected by pipes
Elements	*Filter*, *which is independently executing components* *Pipe*, *which is a connector between filters. A pipe preserves the sequence of data items*, *and does not alter the data passing through*
Relations	The *attachment* relation associates the output of filters with the input of pipes and vice versa
Constraints	Connected filters must agree on the type of data being passed along the connecting pipe
Weakness	It is not a good choice for interactive systems It is not appropriate for long-running computations without the addition of some check point/restore functionality



Notation for pipe and filter design

The main objective of this approach is to achieve the qualities of *reuse and modifiability.*

An important example of the pipe-filter pattern is compiler, where a lexical analyzer analyzes the source file and then sends the resulting tokens to a parser, which sends the resulting parse tree to a semantic analyzer, which produces an augmented syntax tree, that is used by the code generator to produce code, which may be optimized, and ultimately translated into machine code.

- ***Client-Server***

A client-server pattern consists of two main components: server component, which supplies services to many components of the clients, and a client component, which requests services from the server component. Servers are active, listening for clients.

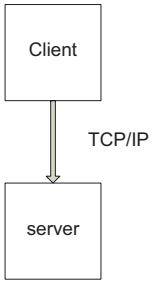
Client-server pattern

Context	Shared resources and services are accessed by a large number of distributed clients to access the quality of services
Problem	Managing a set of shared resources and services, *modifiability* and *reuse* can be supported, and this is done by factoring out common services and having to modify them in a single location or a small number of locations. *Scalability* and *availability* need to be improved by centralizing the control of these resources and services, while these resources are distributed across multiple physical servers
Solution	Clients and servers are interacting by requesting services from clients of servers, while the latter provide a set of services. Some components may act as both clients and servers. There may be one central server or multiple distributed ones

Client-server solution pattern

Overview	Clients initiate interactions with servers by sending requests to these servers and waiting for the results of these requests
Elements	*Client*: a component that invokes services of a server *Server*: a component the provides services to the client
Relations	The *attachment* relation connects between the client and server
Constraints	Clients are connected to servers through connectors Servers can be clients to other servers Components of clients and servers may be formed as layers; clients form the higher level, and servers form the lower level
Weakness	The server can be the performance bottleneck The server can be a single point of failure It is costly to change decisions after a system has been built

Modifiability and reusability are qualities that are supported by this type of pattern. Scalability and availability qualities can be improved by centralizing the control of these resources and services.



Notation for client-server design

- ***Service-Oriented Architecture (SOA) Pattern***

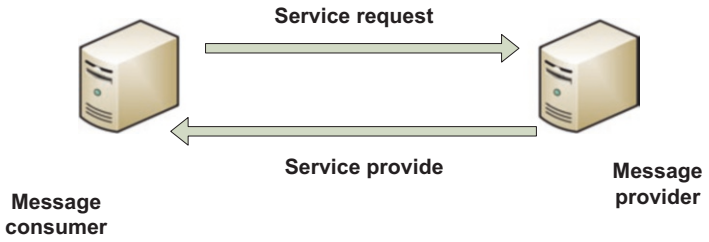
The service-oriented architecture is an architectural design which includes a collection of services in a network which communicate with each other. The service is a kind of operation which is well defined and self-contained and provides separate functionality such as checking customer account details and does not depend on the state of other services.

Service-oriented architecture pattern

Context	A number of services are provided and consumed by the service provider and the service consumer, respectively. The service consumer needs to understand and use these services without any detailed knowledge of their implementation
Problem	The main idea here is how to support interoperability between components running on different platforms and written in different programming languages, provided by different organizations, and distributed across the internet
Solution	The service-oriented architecture (SOA) pattern describes a collection of distributed components that provide and/or consume services. These components can use different programming languages and different platforms, and they are usually deployed independently

SOA pattern solution

Overview	Computation is achieved by a set of collaborating components that provide and/or consume services over a network
Elements	*Service provider:* This provides one or more services *Service consumer:* This invokes services directly or through an intermediary *Enterprise service bus:* An intermediary element that transports messages between service provider and service components *Registry of services:* This element is used by the service provider to register the services and by service consumers to discover services at runtime *Orchestration server:* Coordinates the interactions between service consumers and providers in SOA system *Connectors:* Such as SOAP (the standard protocol for communication in the web service technology), REST (representational state transfer), and asynchronous messaging
Relations	The *attachment* relation connects between different kinds of components
Constraints	The intermediary components (ESB, registry, orchestration) may be used through connections between service consumers and service providers
Weakness	It is complex to build The evolution of independent services is not controlled Because of the shared services between servers and clients, the middleware may act as a performance bottleneck



Notation for service-oriented design

Microservices are a modern interpretation of SOA used to build distributed software systems. Services in microservice architecture are processes that communicate with each other over the network in order to reach a goal.

Patterns in the allocation structure are:

- ***Multi-tier Pattern***

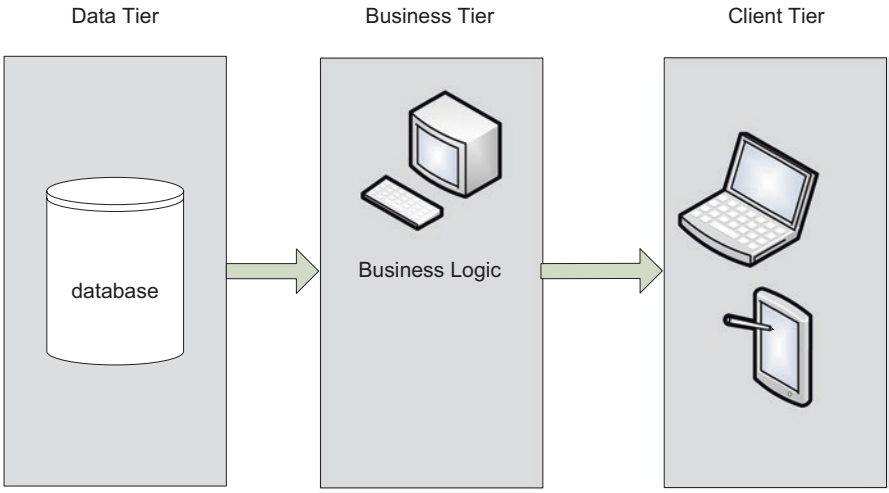
In a Multi-tier pattern, tiers can be created to group components of similar functionality, in which case it is a C&C pattern. The reason that makes it an allocation pattern is that the client tier in an enterprise system maps software elements to computing elements because they are not running on the computer that hosts the database.

Multi-tier pattern

Context	The infrastructure of the system needs to be distributed into distinct subsets; for example, different parts of infrastructure may belong to different organizations
Problem	How the system can be split into groups of software and hardware that communicate by some communications media
Solution	The system is organized as a set of logical grouping of components, and this is done on criteria such as sharing the same execution environment

Multi-tier pattern solution

Overview	The system is organized as a set of logical grouping of components termed tiers, and this is done on criteria such as sharing the same execution environment
Elements	*Tier*, which is a logical grouping of software components
Relations	*Is part of*, to group components into tiers *Communicates with*, to show how tiers and components they contain interact with each other *Allocated to*, in the case that tiers map to computing platforms
Constraints	A software component belongs to exactly one tier
Weakness	Cost and complexity



Multi-tier notation

Two important things must be known on tiers:

- Tiers themselves are not components but are logical groupings of components.
- Tiers are not layers; layering is a pattern of modules, while tiers apply to runtime only.

- ***Publish-subscribe Pattern***

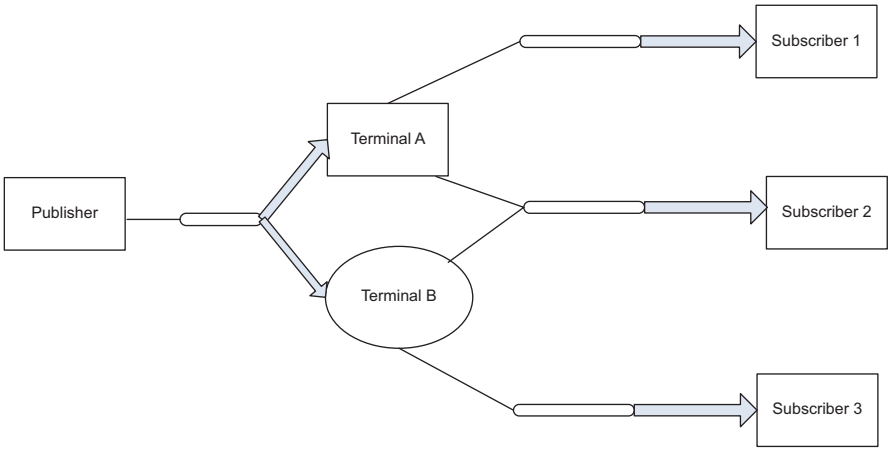
In this pattern, a number of independent producers and consumers of data interact, and because of that the publisher "produces" the messages without knowing the consumer identities (here called subscribers), and similarly, the subscribers receive messages that are of their interest without knowing the publisher identity.

The pattern here is:

Context	A number of independent producers and consumers of data interact. The accurate number of the data producers and consumers is not fixed
Problem	How can we create a mechanism for communicating the producers and consumers of data?
Solution	In this pattern, components interact through messages or events. Components may subscribe to a set of events. Publisher components put events on the bus by announcing them; the connector then delivers those events to the subscriber components that have recorded their interests in those events. Any component may be both a publisher and a subscriber

The solution pattern of the publish-subscribe

Overview	The components are produce (publish) and consume (subscribe) to events; once the events are announced by the components, then the connector infrastructure dispatches the event to all registered subscribers
Elements	*Publisher* (*sender*)*:* Application that tags each message with the name of topic *Subscriber* (*receiver*)*:* Application that chooses the interesting topic *The publish-subscribe connector*, which will have *announce* and *listen* roles for components that wish to publish and subscribe to events
Relations	*Attachment relation* that connects components (publisher/subscriber) with *the publish-subscribe connector*
Constraints	All components are connected to an event distributor that may be viewed as either a connector or a component
Weakness	Scalability is affected negatively through increasing latency This pattern suffers in controlling over ordering messages, and delivery of messages is not guaranteed. This is why this pattern is not suitable for complex interactions



Publish-subscribe notation

Patterns address multiple quality attributes. Patterns give pluses to some qualities and minuses to others. A good example is MVC application; the view components are notified when the state of model of the object changed. This pattern is highly *modifiable*.

The benefits of patterns in general are:

- Patterns are always built to address a recurring problem that arises in a specific design situation.
- Patterns provide common vocabulary and understanding to design principles such as the name of the pattern.

- Patterns are means of documenting software architecture.
- Patterns help build and manage complex and heterogeneous software architecture.

> *Pattern systems* bind individual patterns together. It describes how the basic patterns are connected with other patterns in the system, how these patterns can be implemented, and how software development with patterns is supported. A pattern system is a dominant mean for constructing software architectures.

In general, patterns differ in scale and abstraction because various ranges of scale are applied at different levels of abstraction; this is why they are classified as:

- Architectural pattern: a set of architectural elements, their responsibilities, rules, and guidelines for organizing the relationships between the elements.
- Design pattern: a recurring structure of communicating elements that solves a general design problem within a particular context.
- Idioms: are patterns specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them.

The three main architectural concepts that are used in this section are architectural structure, architectural pattern, and architectural style. Although these terms are used interchangeably in most books and papers, there is a little difference between them:

- Architectural structure is a set of elements and the relations among them. Three categories of structures are represented here: module, component and connector, and allocation structures.
- Architectural pattern is described in three main dimensions: context, problem, and solution.
- Architectural style is the representation of the architecture in terms of the pattern of organizational structure. It establishes the meaning of components and the relation that is used in the pattern, the constraints on the communication of the parts, and the weaknesses of using such type of style.

In Clements et al. (2011), you can find the difference between an architectural pattern and an architectural style. It argues that a pattern is a context-problem-solution triple and a style is simply a condensation that focuses most heavily on the solution part.

4.3 Tactics and Quality Attributes

Quality attributes are the most important part that the architect focuses on through building the system to get the stakeholder's confidence. On the other hand, stakeholders always focus on the quality through working with the system. All that will

make the quality the major part of the architect's work. This section will present the technique that an architect can use to achieve the required quality attributes; it is called architectural tactic. I will define the meaning of architectural tactic and then describe how this technique can help to achieve the specific quality. The relation between pattern and tactics will be described as well.

4.3.1 Achieving Quality Through Tactics

Architectural tactic is a design decision that influences the achievement of quality attribute response, and this makes the architectural tactic a means of satisfying quality attribute measure.

When designing the system that consists of a set of design decisions, we must know that some of these decisions achieve the functionality part of the system and other decisions help control the quality control responses. The relationship between stimulus, tactics, and response is represented in Fig. 4.2.

The important thing to know is that the focus of the tactic is on a single quality attribute response; unlike patterns, they address many quality attributes. The other important thing is that within tactic, there is no consideration to the trade-offs which makes it different from patterns (Fig. 4.3).

Now, given the usability quality as an example, you will see the following tactics related to it (Fig. 4.4):

Usability tactics 1:

Support user initiative: usability is giving the user feedback for what the system is doing once the system is executing to allow the user to give the appropriate response. The tactics for that are:

- Cancel: the system must listen for the cancel request; the command being canceled must be terminated; resources used must be free; and collaborating components must be informed.
- Pause/resume: effectively pausing a long-running operation will temporarily free resources so that they may be reallocated to other tasks.
- Undo: maintain enough information about system state so that an earlier state may be restored, at the user's request.

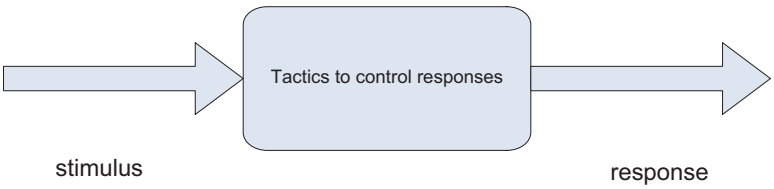


Fig. 4.2 Relationship between tactic, stimulus, and response

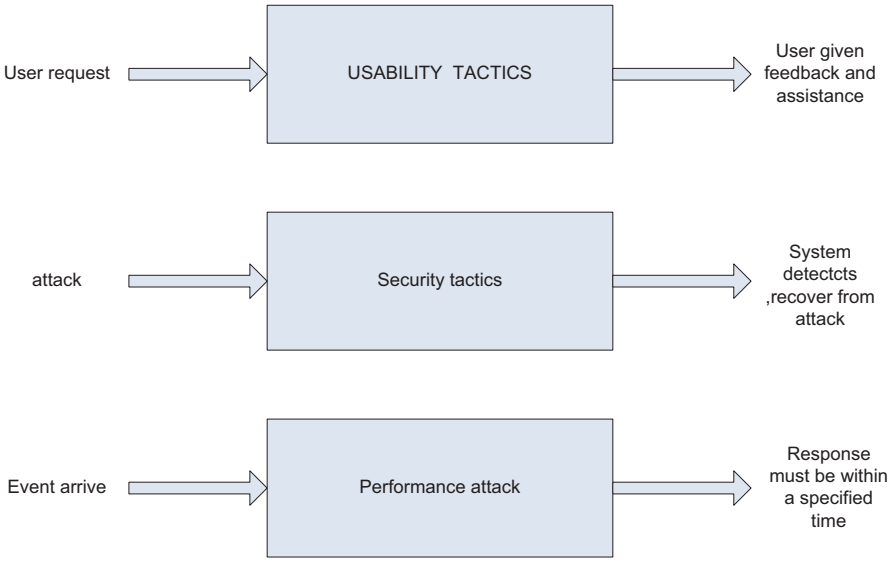


Fig. 4.3 Tactics classified by quality attributes

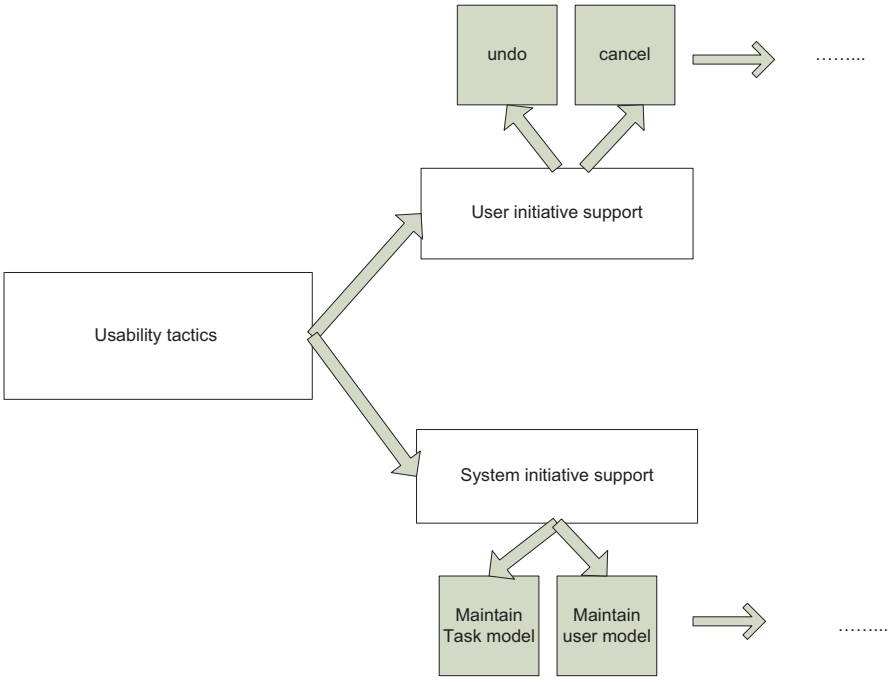


Fig. 4.4 Tactics for usability

- Aggregate: ability to aggregate lower-level objects into a group, so that a user operation may be applied to the group, and this is done through performing repetitive operations or operations that affect huge amounts of objects in the same way.

Usability tactics 2:

Support system initiative: the task must be either undertaken by the user or the system state itself.

- Maintain task model: the system can have some idea of what the user is attempting and provide assistance. For example, capital letter in the middle of the sentence must convert to small letter.
- Maintain user model: the user's behavior, in terms of expected response time, represents the user's knowledge of the system, for example, user interface customization for individual users.
- Maintain system model: it is used to determine the expected system behavior, so that suitable feedback can be given to the user, for example, system progress bar.

Note *The goal of the usability tactic is that after request it by user, the suitable feedback and assistance must be provided.*

Tactics can be the way to satisfy quality attribute response measure; this is done through many ways. The first way is done by manipulating some aspect of a quality attribute model through architectural design decisions. Some quality attributes like performance and availability have well understood such type of models. In such types of qualities, the ideal tactic processes are:

1. Begin with the analytical model for that quality.
2. Identify the parameters of that model.
3. Identify architectural techniques to manipulate the parameters of the model.

> *Analytic model* is a set of equations describing the performance of a computer system. Practically, it describes a collection of measured and calculated behaviors of different elements over a finite period of time.

For example, the queuing model for the performance is represented in Fig. 4.5.

The request arrives, it is put in the queue and then served by the server, so latency (time to compute the result) can change one of the parameters of this model.

The parameters that can be seen in the model are:

- Arrival rate
- The queue
- Scheduling algorithm
- Service time
- Topology

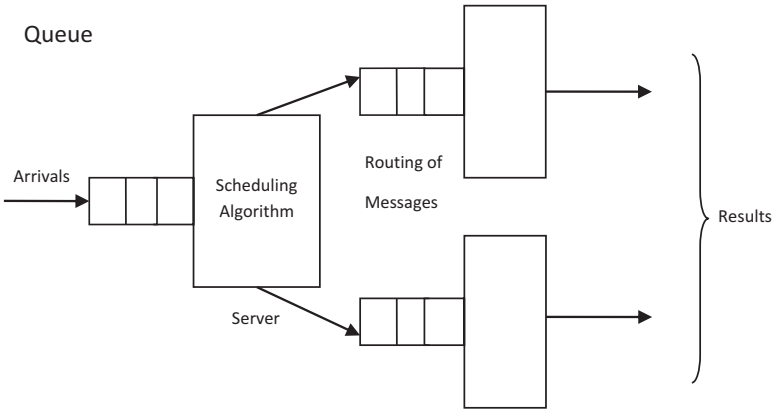


Fig. 4.5 Queuing model for performance quality

- Network bandwidth

The queuing discipline can be affected by first come first service (FCFS). The important thing to know is that not all qualities have analytic models. Checklist and thought experiments exist to guide the architect when making the design decisions. Quality attribute checklists can come from government organizations or from private organization or it can come from industry consortia. For example, there is a security checklist for the financial industry.

On the other hand a thought experiment is the kind of discussions that occur between the developers and architects daily in the offices or any kind of meeting. The purpose of this kind of meeting is to find confirmation to the nonexistent problems in the quality attribute requirement.

Models and checklists focus on one quality attribute, while thought experiments consider many quality attributes simultaneously and focus on the most important ones.

Depending on the project's state of development, different forms of analysis are possible, each type of analysis associated with its own cost and level of confidence.

You will see the tactics of other qualities in Chaps. 7 and 8.

4.3.2 The Relationship Between Tactics and Patterns

As said before, every pattern deals with many qualities, while tactic deals with a specific type of qualities such as modifiability, availability, etc. The important thing to know is that most patterns consist of several different types of tactics that serve a common purpose. For example, choosing a particular tactic to make an availability pattern more secure or applying a performance tactic on a modifiability pattern.

For example, the layer pattern can be seen as a mixture of several tactics as increased semantic cohesion, restrict dependencies, and encapsulation. As with increased semantic cohesion, the goal of ensuring that layer's responsibilities all work together is achieved only through choosing responsibilities that have semantic cohesion; for example, responsibilities that deal with hardware must be allocated to a hardware layer not in an application layer.

4.4 Business Pattern

A business pattern is a generalized solution to a common business problem. So to present the architecture of the business you work on, you need pattern in your description because it reflects common solutions to common problems. This definition makes sense that the concept of pattern is same everywhere.

Another definition of business pattern comes from Microsoft: "a reusable approach to the solution of particular business problems, the offered solutions are according on previous success in defining the same solutions. A business pattern can represent a template for a business solution." The term "reusable" in this definition explains the importance of pattern in the business context.

It takes considerable effort to change from one business pattern to another. Companies should know the various business patterns and how they can benefit from applying them. By recognizing the various available business patterns, it is possible to solve existing business problems within an enterprise or enable the identification of opportunities to create a significant competitive advantage. A business pattern is characterized by being structural, reusable, proven and lastly as making business sense.

4.4.1 Pattern for Enterprises

Layering technique is one of the most common things that software designers use to break a complicated software system into parts. The class book titled *Patterns of Enterprise Application Architecture* by Martin Fowler describes set of patterns grouped by layers that form such systems.

Enterprise application is about using a layer structure to present, manipulate, and store large amount of data with a business process that support these data. Such types of applications are financial system and reservation system.

This section is about describing layers of enterprise systems (in an abstract way). Three principle layers are presentation, domain, and data source layers.

- *Presentation layer*: is responsible for handling the communication between user and software.
- *Domain logic* also called (business logic): this is the work that the application needs to do for the domain you are working with. This layer is the real point of the system.

- *Data source layer*: for most enterprise applications, the biggest piece of data source logic is a database that is primarily responsible for storing persistent data.

In the domain logic, three patterns are described *transaction script*, *domain model*, *and table module:*

- A transaction script is a procedure that takes the input from the presentation, processes it, stores data in the database, and invokes any operations from other systems. It then replies with more data to the presentation. Book a hotel room and booking for a certain day are two examples on transaction script. The main advantage of a transaction script is that it is suitable for a small and simple application. On the other hand, the disadvantage arises when a system gets larger and the business logic gets more complex, and then code duplication between scripts increases.
- Domain model deals with more complex systems when dealing with objects. Rather than one routine having all the logic for a user action, each object takes a part of the logic that is relevant to it. A simple domain model looks very much like the database design with mostly one domain object for each database table.
- Table module: here, a single instance will handle the business logic for all rows in a database table or view. A table module organizes domain logic with one class per table in the database, and a single instance of a class contains the various procedures that will act on the data. The primary distinction with domain model is that, if you have many orders, a domain model will have one object per order, while a table module will have one object to handle all orders.

One benefit of using layering technique is that you can understand the layer itself as a stand-alone coherent layer without having to know about other layers. Also layers are a good way to standardization.

Layers also have disadvantages. For example, in the encapsulation features of the layer (encapsulation hides some but not everything), any new field in the presentation part must be added into database part and any other layers in between them. Another disadvantage of layers is that by adding many layers, this will harm the performance. The hardest part is to know what layer to have and also deciding the pattern in the layer according to the complexity of the system.

4.5 Importance of Patterns in Business

What is the benefit of pattern in business? Or why does business follow patterns?

- At the most abstract level, all businesses (or “enterprises”) have the same pattern.
- Businesses in the same industry share common external influences, internal influences, and goals.

- At the view of improving the design of business, patterns support the reuse feature. This feature has the important benefit of reducing implementation and will lower the costs of maintenance.
- The reuse feature at the more abstract level can enable interoperability between systems that follow patterns.

4.6 The SEI Attribute-Driven Design (ADD) Method

ADD (attribute-driven design) is a method that was developed at SEI to define and design software architecture in which the design process is based on the software's quality attribute requirements. ADD essentially follows a "Plan, Do, Check, and Act" cycle:

- *In the Plan phase*: Quality attributes and design constraints are considered to select which types of elements will be used in the architecture.
- *In the Do phase*: Elements are instantiated to satisfy quality attribute requirements as well as functional requirements.
- *In the Check phase:* The resulting design is analyzed to determine if the requirements are met.
- *In the Act phase:* you should act upon the feedback received from the check phase process.

This process is repeated until all architecturally significant requirements are met.

The input for the ADD process is the set of requirements for the project, which contains functional requirements, quality attribute requirements, and design constraints. Also you should know the purpose of the design, for example, to produce a design for early estimation or to design and generate a prototype, etc. The requirements should be described in detail to check if they can be met with the architecture.

ADD version 3 is created to address the weakness of the previous versions. The steps of ADD are:

Step 1: review inputs
First, the purpose of the design must be clearly defined, and all the drivers of the architecture must be described. At this point, the primary functionality and quality attribute scenarios have been prioritized, ideally done by the most important project stakeholders. Prioritization can be done by techniques described previously. The architect must own all these drivers.

These drivers drive the design, so getting them right and prioritizing them are very important.

> Architecture drivers are the questions that arise through building the architecture and these questions categorized as what and why questions. Architectural drivers include design purpose, quality attributes, primary functions and constraints, and architectural concerns (additional concerns to the architecture).

Step 2: establish the iteration goal by selecting drivers
A design round is performed in a series of design iterations; each iteration focuses on achieving a specific goal. Such a goal typically involves deigning to satisfy a subset of the drivers, for example, creating a structure from elements to support a specific scenario for performance.

Step 3: choose one or more elements to refine
Refinement can mean decomposing the elements of the architectural structure either top down or bottom up or it can mean improvement of the previously identified elements. In order to satisfy the drivers of the architecture, you do need to refine the elements of the structure that are identified in the previous iteration.

The elements that you will select are the ones that are engaged in satisfying of the specific drivers. That is why you should understand of such elements.

Step 4: choose one or more design concepts that satisfy the selected drivers
Choosing the design concepts is the most difficult decision in the design process. This is because you need to identify alternatives among design concepts that can be used to achieve the goal of the iteration and to make the selection of these alternatives. The term design concepts is used through the design process; they are building blocks that are used to build the structure of the architecture.

Step 5: instantiate architectural elements, allocate responsibilities, and define interfaces
After defining the design concept in the previous step, you need to make another design decision, which requires instantiating elements of the design concepts that you selected. For example, if you select the layer pattern as a design concept, then you must decide how many layers will be used, then allocate the responsibility of each layer, and then define the interfaces.

Another example is a typical web-based enterprise system: at least three important layers are presented: the presentation layer, the business layer, and the data layer. Each of these layers has a different responsibility; the responsibility of the

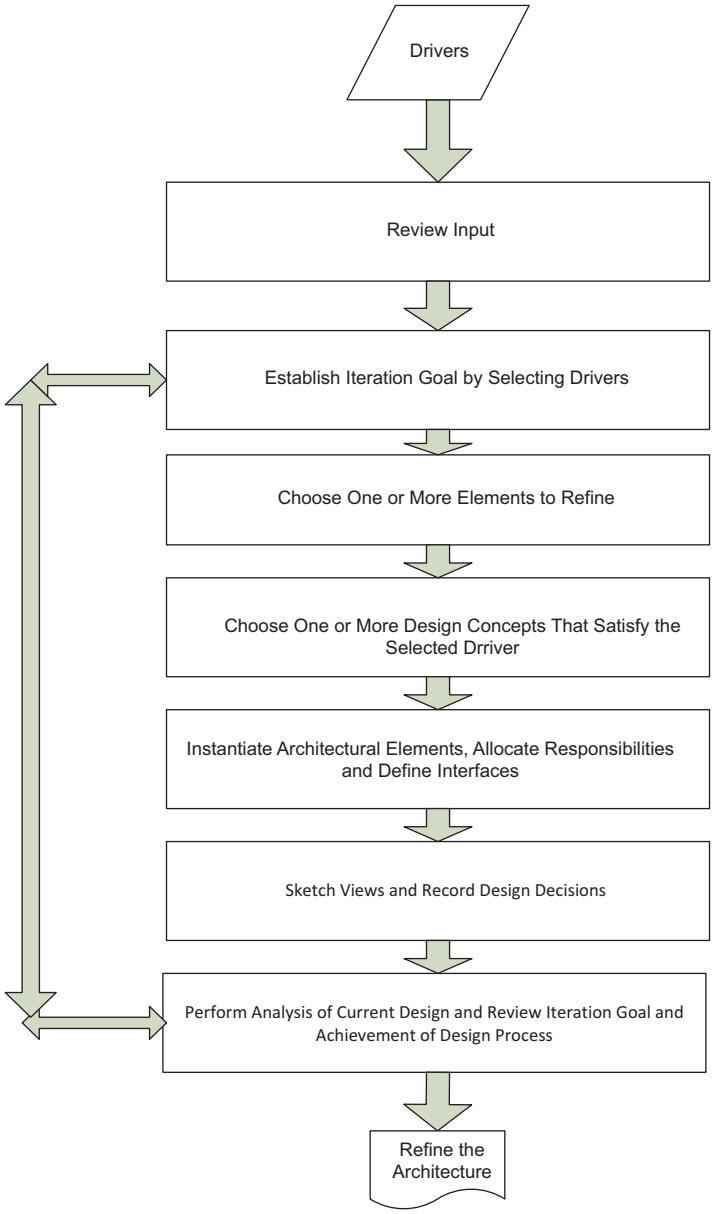


Fig. 4.6 Attribute-driven design steps

presentation layer includes managing the interaction of users, while the responsibility of the data layer includes managing the persistent data.

Step 6: sketch views and record design decisions
After finishing performing the design activities for the iteration, this step is about to sketch the views (the representations of the structure that are created previously), but the views that are created are most of the time incomplete. So all the diagrams that are created have be to refined later in subsequent iteration. You must know that these created sketches are the beginning of the documentation. In addition to sketching the views, recording significant decisions that are made in the specified iteration is also important.

Step 7: perform an analysis of the current design and review iteration goals and achievement of design purpose
In this step, a specific design that addresses the goal must be created for the iteration to avoid an unhappy stakeholder, so once the design is formed in the iteration, it needs to be analyzed by another person or team because having someone with a different view is important to find bugs in both code and architecture. After that, you should review the state of the architecture you built in order to see it if it establishes the purpose of the design.

Step 8: iterate if necessary
You should perform additional iterations for every driver that considers being part of the input. The highest driver priorities should be addressed. Also critical drivers should be satisfied or at least the created design should satisfy them.

The drivers in Fig. 4.6 are design purpose, functional requirement, quality attributes scenario, constraints, and architecture concerns.

4.7 Summary

Architectural pattern introduces a relationship between *context*, *problem*, *and solution*, and this will form the template for documenting that pattern. Complex systems exhibit multiple patterns at once.

Pattern can be categorized by the dominate type of elements as:

- Module pattern
- Component-and-connector pattern
- Allocation pattern

Patterns employed in architecture help us expect which properties will be displayed by architecture. Tactics are fundamental design decisions that influence the responses of the quality attribute; they are the “building blocks” of design from which patterns are created. Most patterns consist of several different tactics.

The three main architectural concepts that are used in this section are architectural structure, architectural pattern, and architectural style. Although these terms are used interchangeably in most books and papers, there is a little difference between them:

- Architectural structure is a set of elements and the relations among them. Three categories of structures are represented here: module, component and connector, and allocation structures.
- Architectural pattern is described in three main dimensions: context, problem, and solution.
- Architectural style is the representation of the architecture in terms of the pattern of organizational structure. It establishes the meaning of components and the relation that is used in the pattern, the constraints on the communication of the parts, and the weaknesses of using such type of style.

In Clements et al. (2011), you can find the difference between an architectural pattern and an architectural style: a pattern is a context-problem-solution triple, while a style is simply a condensation that focuses mainly on the solution part.

Moving the definition of pattern to the business, the concept of business pattern arises. It can be defined as a solution to common business problems in any enterprise. The importance of business pattern can be described in the following:

- At the most abstract level, all businesses (or "enterprises") have the same pattern.
- Businesses in the same industry share common external influences, internal influences, and goals.
- At the view of improving the design of business, patterns support the reuse feature. This feature has the important benefit of reducing implementation and will lower the costs of maintenance.
- The reuse feature at the more abstract level can enable interoperability between systems that follow patterns.

ADD (attribute-driven design) is a method that was developed at the SEI decomposition process to define and design a software architecture in which the design process is based on the software's quality attribute requirements.

References

L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, 3rd edn. (2013) United States

H. Cervants, R. Kazman, *Designing Software Architectures: A Practical Approach* (Addison-Wesley, 2016)

M. Fowler, *Pattern of Enterprise Application Architecture* (Addison-Wesely, 2003)

P. Teale, *Business Patterns for Software Engineering Use* (Microsoft Corporation, Robert Jarvis SA Ltd, 2004)

Further Reading

F. Bachmann, L. Bass, R. Nord, Modifiability Tactic, Technical report CMU/SEI (John Wiley & Sons Ltd, 2007) Chichester, West Sussex, England

J. Bosch, *Software Architecture: The Next Step*, University of Groningen, Department of Computing. LNCS 3047 (Springer, Berlin Heidelberg, 2004), pp. 194–199

F. Buchman, K. Hennery, D. Schmidt, *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing* (Wiley, 2007)

I. Gorton, *Essential Software Architecture*, 2nd edn. (Springer-Verlag, 2011) Berlin, Heidelberg

T. Graves, *Everyday Enterprise Architecture: Sensemaking, Strategy, Structures and Solutions* (Tetradian Books, 2010). http://www.tetradianbooks.com

N. Harrison, P. Avgeriou, How do architecture patterns and tactics interact? A model and annotation. J. Syst. Softw. **83**, 1735–1758 (2010). http://www.elsevier.com/locate/jss

S. Kim, Dae-Kyoo Kim, et al., Quality-driven architecture development using architectural tactics. J. Syst. Softw. **82**, 1211–1231 (2009)

A. Moore, L. Bass, M. Klein, F. Bachmann, Security and Survivability Reasoning Frameworks and Architectural Design Tactics, Technical note (2004)

M. Richards, *Software Architecture Patterns: Understanding Common Architecture Patterns and When to Use Them* (O'Reilly Media, Inc, 2015) Sebastopol, CA

A. Sharmaa, M. Kumarb, S. Agarwal, *A Complete Survey on Software Architectural Styles and Patterns.4th International Conference on Eco-friendly Computing and Communication Systems, ICECCS, 2015* (Elsevier, 2015)

Chapter 5
Managing Business Qualities

Abstract Business quality is a non-software system quality that influences other types of qualities in the system. This type of quality is very important because it is taken from the market's point of view. This chapter is going through this type of quality and its goals (business goals). Business goals must be specific, measurable, attainable, realistic, and time bound.

Stakeholders of the system and their relations to business goals will be described in this chapter. Also the approach that shapes the relation between the organization and the stakeholder will be shown.

To enhance the quality of the software, many companies use an approach called software process improvement. This approach can reduce cost or increase their development process through it. It is a way to understand the existing process and then changing it to increase the quality of the product and/or reduce the time of development and cost. This approach will be described in this chapter.

In brief, at the end of this chapter, you will learn:

- The meaning of business qualities and business goals and the main qualities in the business
- The meaning of stakeholders, their types, and their roles in achieving the goals
- The meaning of TQM and its main principles
- The process improvement and its life cycle

5.1 Business Quality Definition

This chapter will focus on a new type of quality, the business quality, which guides the architect to build high-quality architecture. This type of qualities leads to the business goals. The important thing to know is that ASR, previously defined, is related mainly to business goals, and when an architect understands the desired qualities before building the system, then the right architecture will be created.

L. Khalid, *Software Architecture for Business*,
https://doi.org/10.1007/978-3-030-13632-1_5

Business quality can be defined as a non-software system quality that influences other types of qualities. Business qualities are important qualities because they are taken from the market's point of view. Goals of such types of qualities are centered on the following:

1. *Cost and time to market:* the budget of the developing system is a very important consideration which must be taken into account through building any system, and of course, the cost is different from system to system. The system that is built to be highly flexible will be more costly than rigid systems. The cost of the system must be with respect to the time to market, and the latter can be achieved by using pre-built elements such as commercial off the shelf (COTS) or the elements that are reused from previous projects.
2. *Marketability:* building the system with respect to the market competition. Portability and functionality are two important features that share mass market. When talking about mass market, product line approach will be considered, and this will be explained in detail in Chap. 6.

> *Mass markets are products and services that are needed by almost every member of society.*

Also when talking about marketability, the most useful quality business is that it will be fit to its purpose and meet the business user's requirements system. The quality assurance team will then validate the processes, systems, and services.

3. *Standardization:* manufactures use standards and process improvement methodologies to improve the process and then the product, so here the quality is compliance to the best known standards.
4. *The schedule time of building the system:* building a system according to the scheduled time is a very important quality in the business point of view, because adding any extra features will usually compromise time to market.

5.2 Business Goals

Business goals are the basic part for building any system. Every architect is interested in business goals. Such goals start with the vision and mission of the company to structure the system. It must be SMART (specific, measurable, attainable, realistic, and time bound). Profit is a common example for the business goals despite the fact that profit is not always the main concern to some organization because they have other concerns in mind. Another example of the business goal is the customer service goals; one of the important goals here is how to reduce the response time in the customer service and improve satisfactions to the customer.

Table 5.1 Categories of business goal

1	Contributing to the expansion and continuity of the organization
2	Meeting financial goals
3	Meeting personal objectives
4	Meeting responsibility to employees
5	Meeting responsibility to society
6	Meeting responsibility to shareholders
7	Meeting responsibility to states
8	Managing market position
9	Improving business process
10	Managing the quality and reputation of products
11	Managing change in environmental factors

Table 5.1 represents the main categories for business goals; they help brain storming and elicitation techniques.

1. *Contributing the expansion and continuity of the organization*
   This category discusses how the system being developed contributes to the growth and continuity of the organization. Product lines come up in this category.
2. *Meeting financial goals*
   Profits from systems, costs of developing and deploying the system, and any financial objectives of individuals, for example, managers hoping for a raise.
3. *Meeting personal objectives*
   This category is interested with the goals of individuals that are associated with the construction of the system.
4. *Meeting responsibility to employee*
   Responsibilities for employees involved in development may include a certain type of employees, for example, learning new skills. Another example of responsibilities is when employees operate the system; here, safety, workload, or other things that are related to the system's work will be considered.
5. *Meeting responsibility to society*
   Some organization serve society; the system under development helps them meet these responsibilities and also ethics; security are under this type of category.
6. *Meeting responsibility to state*
   Government systems are planned to meet the responsibilities to a state or country. Supporting government plans and dealing with the export control are under this type of category.
7. *Meeting responsibility to shareholders*
   There is an overlap between this category and the second one. Shareholders can be defined as any person, company, or institution that owns at least one

share of a company's stock, also called stockholders. Shareholders are always stakeholders in a corporation but not every stakeholder is shareholder.

8. *Managing market position*
   Strategies that are used to hold or increase the market's share or time to market are under this category.
9. *Improving business process*
   Improving business process may overlap with financial objectives, dealing with the new market or improving the way to support customers.
10. *Managing the quality and reputation of the product*
    Branding of the product, qualities of existing products, and testing strategies.
11. *Managing change in environment factors*
    Business context for any system may change this. Encourage stakeholders to consider what the change in the business goals is.

A better way to express business goals is through scenarios. The template of this scenario has seven parts, all related to the system under development. The parts are abbreviated in Table 5.2.

Note If you want to express the business goal scenario, you say:

For systems to be developed, the *goal subject* desires the *goal object* to achieve the *goal* in the context of *environments* and will be satisfied if the *goal measured.*

PALM (Pedigree Attribute eLicitation Method) is a method used to capture business goals and then document them. Pedigree means that a business goal has a clear derivation or background. PALM consists of seven steps between the architect and the stakeholders, done through a workshop which normally take a day and a half. These steps are:

- *PALM overview presentation*
- *Business drivers presentation*
- *Architecture drivers presentation*
- *Business goals elicitation*
- *Identification of potential quality attributes from business goals*
- *Assignment of pedigree to existing quality attribute drivers*
- *Exercise conclusion*

Table 5.2 Business goal scenario

Goal source	The source of the goal are people or written artifacts
Goal subject	Stakeholders own the goal and wish it to be true. Stakeholders might be individuals or organizations
Goal object	The entities to which the goal applies, for example, the goal object is developing the organization
Environment	The context of the goal, such as the social environment and customers
Goal	This is a business goal
Goal measure	A testable measurement to help achieve the goal
Pedigree and value	The degree of confidence of the person who states the goal

PALM can be used to find the missing quality attribute requirements early in the life cycle and to notify the architect of business goals that directly affect the architecture without needing to quickly find new requirements. PALM can also be used to examine difficult requirements, and finally, because of the different overview of the stakeholders, this method provides the environment for capturing goals to be broadcast and resolved.

Finally, capturing business goals is very important because they will be the key to ASRs that emerge in no other context and they also derive the architecture.

Paul Clements, John McGregor, and Len Bass applied PALM to a system being developed by Boeing's air traffic management; they call the system TSUC (the system under consideration). This system provides on-line services to the airline companies to help improving efficiency. Two classes for this system of stakeholders: Boeing and airline companies. This exercise helps chief architect and project managers, who previously applied PALM, see the same vision of the TSUC. Some of the uncovered goals through PALM were:

- The possibility of using TSUC in the future
- Impact of this system on both, users and developers

The result is that the categories of business goals must capture the business goals that are relevant to TSUC.

5.2.1 The Role of the Architect in Achieving the Quality

Every architect knows that its role extends from technical activities to being consultants. The important thing in business and strategy is predicting the right architectural approach to the customer's problem. This is why the architect has to be a good listener to the concerns and goals of the stakeholders.

The main principle that declares that it is the role of the architect to achieve the qualities that the stakeholder needs is that the architect is responsible for designing, documenting, and leading the construction of a system. Four main aspects to this role are:

- Identifying and engaging the stakeholders
- Understanding and capturing the concerns of the stakeholders
- Creating and defining architecture that deals with these concerns
- Taking a leading role in the realization of the architecture into system

A *concern* about architecture is a requirement, an objective, an intention, or an aspiration a stakeholder has for that architecture.

Every problem has a number of possible architectural solutions; the architect must select an architecture that is fit for purpose and then document that architecture in an appropriate way. The architect also acts as a mentor working with developers to tackle any challenges that arise. In fact, the architect is involved through the life cycle of software development.

5.3 Definition of Total Quality Management (TQM)

All the definitions of TQM concentrate on reaching high qualities for the products to satisfy the requirements of the customers. It is hard for a company that aims to compete effectively to ignore the use of the concept of TQM. It is for this reason that businesses should not ignore implementing principles of TQM. Hence, total quality management can be defined as integrated efforts to prevent possibilities of errors and improve quality performance at every level of the organization to reach the optimal customer satisfaction. The customer satisfaction is achieved through efficiency and the effectiveness that prevents defects.

The philosophy of TQM began as a result of dramatic changes in the business environment. The customers are usually the basis of any business, whether the business is in the service industry or the manufacturing industry. Customers would always seek for quality in order to be satisfied, but when it comes to the service industry, customers become important and sensitive part compared to the manufacturing industry because the interaction between customers and the companies occurs in the frontline of the company.

The differences between the most important sectors in the business (manufacturing and service organizations) are as follows: in manufacturing, the product is a tangible; it is touchable and can be directly measured, such as clothes and CDs. Thus, the quality in manufacturing is focused on the features of these tangible products such as reliability, performance, etc. On the other hand, in the service organizations sector, the products are intangible, such as health care and learning at a university (i.e., the final product cannot be seen or touched). The quality of such type of products is focused on consistency (the degree to which all the services are the same at all times), responsiveness to customer needs, etc. Both sectors must satisfy the customer's requirements.

Benefits of TQM for customers can be shown through quality improvement, design, and service of the product and also in increasing the acceptance of the product in the marketplace. The benefits are also shown economically through reduction of operating costs, field service costs, operating losses, etc.

TQM attempts to embed quality in every aspect of the organization. It is concerned with technical aspects of quality as well as the involvement of people in quality.

Table 5.3 Main principles for TQM

Customer focus	The main objective is to identify the customer's needs
Continuous improvement	The main idea is continuous development
Employee empowerment	Employees are always seeking out qualities, identifying them, and correcting their problems
Use of quality tools	Continuously training the employees to use the quality tool
Product design	Designing products to meet the customer's needs
Process management	Qualities should also be designed in the process
Managing supplier quality	The concept of the quality must be included in the company's suppliers

5.3.1 Principles of TQM

TQM embeds quality in each part of the organization; dealing with quality management systems not only allows businesses to react to the business changes but also to impose them by controlling the future. There are many principles related to TQM; the ones mentioned here are the ones in the book of operations management. Managers must understand the following principles to improve the competitiveness of their organizations. The main principles of TQM are shown in Table 5.3.

5.3.1.1 Customer Focus

The goal here is to identify and then meet the customer's needs, and that is because the quality is customer driven. This can be achieved by communicating with the customers in an organized frame, by involving the customer in the developing process for the product he requires, etc. Companies always need to continue gathering information by means of market surveys and customer interviews in order to stay focused on what customers want.

5.3.1.2 Continuous Improvement

This is a very important principle that most experts consider. Continuous improvement, from the Japanese point of view, is a philosophy of a never-ending process. The performance must always be evaluated, and measures should be taken to improve it. Two main approaches are used for that: the Plan-Do-Study-Act (PDSA) cycle and benchmarking.

PDSA cycle is a diagram that describes the activities that need to incorporate continuous improvement into the operation, while benchmarking is studying the business practices of other companies for purposes of comparison.

5.3.1.3 Employee Empowerment

This principle focuses on empowering all employees to find quality problems and then correct them. In TQM, employees are rewarded for uncovering quality problems. TQM also empowers workers to make decisions that are relative to qualities in the production process; their assistance is highly appreciated, and their suggestions are implemented. In order to perform this function, employees are given continuous and extensive training in quality measurement tools.

TQM stresses that quality is an organizational effort. To facilitate solving any problem related to quality, it places great importance on teamwork. Using techniques such as brainstorming, discussion, and quality control tools, teams work regularly to correct problems. The contributions of teams are considered essential to the success of the company. For this reason, companies set aside time in the workday for team meetings.

5.3.1.4 Product Design

The important thing in building quality into a product is to ensure that the product design meets the customer's expectations. This typically needs effort to understand the customer's language because his language often changes. To produce a product that customers want, a useful tool for translating the voice of the customer into specific technical requirements is produced, and this tool is called quality function deployment (QFD). QFD is also useful in enhancing communication between different functions, such as marketing, operations, and engineering. For example, an automobile manufacturer evaluating how changes in materials affect customer safety requirements. This type of analysis can be very beneficial in developing a product design that meets the customer's needs, yet does not create unnecessary technical requirements for production. QFD starts by defining the important customer requirements, which typically come from the marketing department. These requirements are numerically scored based on their importance, and scores are translated into specific product characteristics. Evaluations are then made of how the product compares with its main competitors relative to the identified characteristics.

QFD is a method first developed in Japan. Yoji Akao, the original developer of QFD, described it as a "method to transform qualitative user demands into quantitative parameters, to deploy the functions forming quality, and to deploy methods for achieving the designed quality into subsystems and component parts, and ultimately to specific elements of the manufacturing process."

5.3.1.5 Process Management

According to TQM, a quality product comes from a quality process. This means that quality should be built into the process. The source of quality is the belief that it is far better to uncover problems and correct them than to remove defective items after production. If the source of the problem is not corrected, the problem will continue to occur. When dealing with the source of quality, the difference between old and new concepts should be illustrated. Old concepts focus on inspecting goods after production; if an inspection discovers defects, then the products are either removed or sent back for reproduction (the new concepts). All that will cost money for the company.

5.3.1.6 Managing Supplier Quality

TQM extends the concept of quality to a company's suppliers. When materials arrive from the company, an inspection is performed to check their quality. The philosophy of TQM extends the concept of quality to suppliers and ensures that they engage in the same quality practices. Today, many companies have an agent at their supplier's location, thereby involving the supplier in every stage from product design to final production. In fact, the application of quality management in terms of efficiency is impossible without the existence of these association relationships with suppliers.

5.3.1.7 Use of Quality Tools

TQM employees need to understand how to assess the quality by using a variety of quality control tools, how to interpret findings, and how to correct problems. Table 5.4 represents the basic tools that can be used through work.

Within these principles, you can see that leadership plays a central role in TQM; it makes the difference between the reputations of industries. Leadership ensures the personal commitment of the general manager and the management structure to be involved in the implementation of the integrated approach to TQM. Leaders involve employees in the implementation of quality management, with an important role in the operation of all the principles underlying quality management.

Some of the basic characteristics of successful leaders are:

1. Listening to customers
2. Encouraging workers by providing resources, training, and a work environment to help them do their jobs
3. Emphasizing improvement rather than maintenance
4. Encouraging collaboration rather than competition
5. Training and coaching, not directing and supervising
6. Learning from problems—an opportunity for improvement

Table 5.4 Quality tools

Cause and effect	A chart that identifies potential causes of particular quality problems
Flowchart	A diagram of the sequence of steps involved in an operation or process; it provides a visual tool that is easy to use and understand
Checklists	A list of common defects and the number of observed occurrences of these defects. It is an effective fact-finding tool that allows the worker to collect specific information about the defects that have been observed
Control charts	Charts that are used to evaluate whether a process is operating within a set of expectations relative to some measured value such as weight, width, or volume
Pareto analysis	A technique that is used to identify quality problems according to their importance
Histogram	A chart that shows the frequency distribution of observed values of a variable; it shows what type of distribution a particular variable displays, whether symmetric distribution or normal distribution
Scatter diagrams	Graphs that show the relation between two variables. They are particularly useful in detecting the amount of correlation or the degree of linear relationship between two variables

7. Continuously trying to improve communications
8. Continuously demonstrating commitment to quality
9. Choosing suppliers on the basis of quality, not price

5.4 Stakeholders

The people interested in software system are termed "stakeholders." Stakeholders can be defined as any person who has a stake in the success and progress of the system, for example, customer, developer, project manager, and even the person who markets the system are also termed stakeholders. Understanding the role of the stakeholder is fundamental to understanding the role of the architect in the development of a software product or system. Some stakeholders are more interested in their roles than others, so part of the architect's roles is to engage with those people to show the importance of their involvement and to obtain their commitment to the task.

Despite the fact that stakeholders have shared stakes in the success of that system, they have different concerns that they wish the system guarantees. Such concerns include short time to market or low cost of developing the system. Early engagement with stakeholders allows understanding the constraints of the task, negotiating the priorities, and also making the trade-off. So knowing the stakeholder is one of the priorities of the architect. Table 5.5 shows some stakeholders and their interests in the architecture.

Table 5.5 Stakeholders and their roles

Analyst	Analyzing the architecture to make sure it meets the quality attribute requirements
Architect	Making a trade-off among competing requirements, recording design decisions, and providing evidence that the architecture satisfies the list of requirements. The architect is mainly responsible of the system
Designer	Understanding how the parts of the system are interacting and communicating; applying the architecture to meet specific requirements of the parts for which they are responsible
Customer	Assuring the required functionality is available and estimating the cost and setting expectation of the delivered part. The main responsibility of the customer is paying for the system and ensuring its delivery
User	Might use architecture documentation to see if the specific function is delivered and seeing the major elements of the system

5.4.1 Stakeholders and Business Goals

Stakeholders (explicitly or implicitly) drive the whole shape and direction of the architecture to serve their needs. Of course, without stakeholders, there would be no point in developing the architecture because there would be no need for the system. Basically, a good architecture is one that successfully meets the objectives, goals, and needs of its stakeholders.

Many times stakeholders will directly say what they need, but often they are not so direct. Thus, one of the important roles of architects is to figure out what are the quality attributes that are going to impact the design of the system.

If the stakeholders tell the architect directly, they give them something to work with. If the stakeholders tell the architect indirectly, the architect has to outline what the impact on the architecture is going to be. Stakeholders have business goals and objectives, and this will lead to quality attribute requirements that will be a key input to the design of software architecture. Collecting requirements, designing, and documenting the architecture will start with stakeholders. The roles of stakeholders differ between businesses, depending on the rules and responsibilities laid out at the foundation of the company.

The example in Fig. 5.1 shows that if the concern of the stakeholder is the integration with other systems easily, then the quality requirements will be interoperability, modifiability, and portability. If his concern is to increase market share, then the quality requirements will be modifiability, usability, and so on.

A collaborative approach will be the shape between stakeholder and the organization, and this type of approach is coherent and driven by business goals especially the long-term ones. The manager is not separate from the stakeholder but is part of it. Thus, the idea of "managing" is viewed as being counterproductive for both the corporation and its stakeholders in the long run.

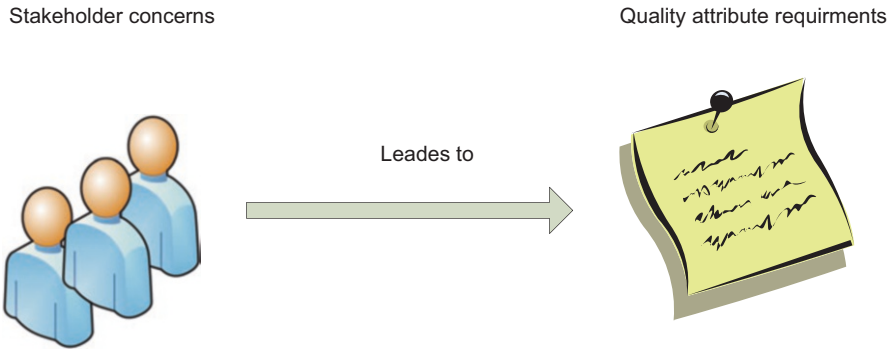


Fig. 5.1 Stakeholder concerns and quality attributes

5.5 Process Improvement

Many software companies have turned to software process improvement as an approach of enhancing the quality of their software, reducing costs, or accelerating their development processes. Process improvement can be defined as understanding the existing processes and changing these processes to increase product quality and/or reduce the time of development and cost. The approaches that are used to process improvement and change differ from one project to another. For example, for large projects, the process approach is used; this approach is focused on improving process and project management and introducing good software engineering practice into an organization. For small and medium projects, the agile approach is used; this approach focuses on iterative development in the software process.

The process maturity approach is rooted in the plan-driven development, while in the agile approach, the focus is on the code being developed.

5.5.1 Process and Product Quality

Process quality and product quality are closely related. The benefits of the process improvement occur because the quality of the product depends mainly on its development process.

A good process is usually required in order to produce a good product. There are four important factors affecting the quality of products, whether they are software product or any other intellectual products such as books or films, these are:

- Development technology
- Process quality
- Product quality
- Cost, schedule, and time

An intellectual product is a product of which its quality depends on its design.

The influence of each factor depends mainly on the size and type of the project. For large systems, the principal factor that affects the quality of the product is the software process, while for small projects, the qualities of the people that work on the product (development team) play an important role in these types of projects. If the team is inexperienced, a good process may limit the damage, but it will not lead by itself to a high-quality product.

Irrespective of people, process, or tool factors, if a project has an insufficient budget or unrealistic schedule, poor quality for the product might save the project. The organization must compete to survive.

5.5.2 The Process Improvement Life Cycle

We all observe the software process in all organizations, these processes differ depending on many things such as the types of products that are being developed and the size of the organization itself (it plays an important role when choosing the software process), and many other things are taken into account when designing the software process. There is no such thing as a standard software process that is applicable in all organizations or for all software products of a particular type. Any change of the processes in the organization must consider the local environment and culture. Each company has to develop its own process depending on its size, the type of software being developed, the skills of its staff, market requirements, and the company's culture.

Improving the process must be under consideration; specifying the goal from this improvement is very important, for example, the goal might be to improve software quality so you may wish to introduce new process activities that change the way software is developed. Also, if the interest is improving some attributes of the process (such as development time), then you decide which process attribute is the most important to the company. It is difficult to gain all process attributes at the same time.

Process improvement is a cyclic process; it consists of three sub-processes, and these are (Fig. 5.2):

- Process measurement: attributes of the current process are measured. These form a baseline for deciding that improvements have been effective.

GQM (goal-question-metric) paradigm is one of the important measurements of the process.

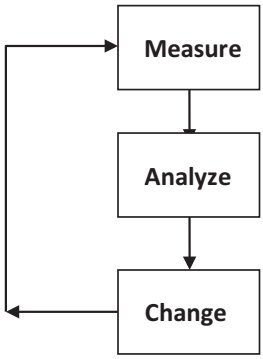


Fig. 5.2 Process improvement activity

- Process analysis: the current process is assessed, and bottlenecks and weaknesses of the process are identified.

Questionnaires and interviews and ethnographic studies are the most commonly used techniques of process analysis.

- Process change: changes to the process that have been identified during the analysis are introduced. The cycle resumes to collect data about the effectiveness of the changes.

Improvement identification and prioritization, process change introduction, process training, and change tuning are the important stages in the process change process.

Without concrete data on a process, it is impossible to evaluate the value of process improvement.

Improving the process is a continuous activity; whenever a new process is introduced, it will change the business environment; and it will have to evolve to take these changes into consideration. The important focus of process improvement is on:

- *Improving quality*
- *Reducing costs*
- *Shortening life cycle time of the project*
- *Decreasing risk*

5.6 Important Qualities in Business

At the end of the chapter, we see that the company wins in their competition with others through building high-quality products. From that, you can conclude how qualities are important in the business or even nonbusiness fields because qualities in business not only help the company meet its customer's expectations but also help to keep costs down. Companies can build their reputation by building high-quality products.

Some of the main important benefits of qualities in business are:

- *Meeting customer expectations*
- This is the main point from using quality in any company; if customers do not find the qualities they expect from you, they will go to others to gain what they need. Quality products make an important contribution to long-term profits.
- *Managing a reputation*
- Quality influences the reputation of the company; a strong reputation for quality can make the difference in a competitive market.
- *Meeting the standards of the company*
- Accreditation to the standards of quality is very essential for dealing with certain customers; it also helps winning new customers and entering new markets.
- *Managing costs*
- Poor quality will increase the cost; if you do not have quality control, you may gain the cost of analyzing nonconforming services to determine the root causes and then retesting the services after rework; and this process costs effort, time, and money.

5.7 Summary

Business qualities can be defined as non-software system qualities that influence other types of qualities. Business qualities are important qualities that are taken from the market's point of view, and the goals of such types of qualities are centered on:

- Cost and time to market
- Marketability
- Standardization
- The scheduled time for building the system

This chapter also focuses on the stakeholders and their relation to the business goals. Stakeholders (customers, developers, project managers, etc.) can be defined as any person who has a stake in the success and progress of the system.

Business goals are the parts that drive the methods of design and are the elements that shape the architecture. The important thing is that all business goals correspond

to quality attributes. Collaborative approach will be the shape between stakeholders and the organization, and this type of approach is coherent and driven by business goals especially the long-term goals.

Business goals are the state of events that users would like to achieve. Basic categories of business goals are:

- Contributing to the growth and continuity of the organization
- Meeting financial objectives
- Meeting personal objectives
- Meeting responsibility to employees
- Meeting responsibility to society
- Meeting responsibility to shareholders
- Meeting responsibility to states
- Managing market position
- Improving business process
- Managing the quality and reputation of products
- Managing change in environmental factors

StakehStakeholders are those people who have their roles and responsibilities in the success of the system and achieve the business goals. Finally, the importance of qualities in the business are concluded in four main pillars:

- Meeting customer's expectations
- Managing a reputation
- Meeting the standards of the company
- Managing costs

References

L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, 3rd edn. (Addison-Wesley, Upper Saddle River, 2013)

P. Clements, D. John, Mc Gregorand, L. Bass, Software Engineering Institute, Carnegie Mellon University, 14th international conference SPL 2010, Springer LNCS 6287 PP:393–403

L. Khaled, Achieving goals through architectural design decision. J. Comput. Sci. **6**(12), 1424–1429 (2010a)., 2010, ISSN 1549-3636 Science Publications

N. Rozanski, E. Woods. *Software system architecture: Working with the stakeholder using viewpoints and perspectives*. Prepared for doug.taylor@jeppesen.com. Douglas Taylor (2009)

N. Sanders, D. Reid, *Operations Management:An Integrated Approach*, 5th edn. (John Wiley & sons, New York, 2012). ISBN:9781118122679
I. Somerville, *Software Engineering*, 9th edn. (Addission Wiesly, Boston, 2011)

Further Reading

SEI on line training from SEI: Software Architecture: Principles and Practices, http://ww.sei.cmu.edu/education-outreach/courses/online-training

Journal Papers

L. Bass, P. Clements, R. Kazman, R. Nord, Architectural, Business Cycle Revisted, Software Engineering Institute, Carengie Mellon (2009)
D. Gross, E. Yu, Evolving System Architecture to Meet Changing Business Goals: An Agent and Goal –Oriented Approach, University of Toronto. IEEE (2001). http://ieeexplore.ieee.org
L. Khaled, Driving architectural design through bussiness goals, software engineering researcher. Int. J. Comput. Sci. Inf. Secur. 8(3), (2010b)
I. Liu, E. Yu, From Requirment to Architectural Design -Using Goals and Scenarios, University of Toronto (2001)
V. Omachouno, J. Ross, *Principle of Total Quality* (CRC Press, 2005)
D. Susnienė, P. Vanaga, Integration of total quality management into stakeholder management policy and harmonization of their interests. Eng. Econ **44**(4), 71–77 (2005). ISSN 1392–2785. Commerce of Engineering Decisions. http://inzeko.ktu.lt/index.php/EE/article/view/11322/6046
R. Van Solingen, E. Berghout, *The Goal Question Metric: A Practical Guide for Quality Improvement of Software Development* (McGraw Hill, Chicago, 1999)
M.K. Verma, *Importance of Leadership In Total Quality Management* (Mizoram University, 2014). http://www.reseachgate.ney/publication/295531927

Chapter 6
Software Product Line (SPL)

Abstract To be competitive, most product engineering organizations must deliver product lines; it becomes an important and widely used approach for the efficient life cycle of the organization's software product.

Software product line can be defined as a set of software intensive systems that share common features to satisfy the specific needs of a particular market. The quality attribute associated with this type of systems is variability which is a kind of modifiability qualities. Variability is the ability of a system to support the production of a set of artifacts that vary from each other in a preplanned way; its goal is to satisfy the variations and commonalities that are identified by the scope of the product line. Developers have to decide what type of variation mechanism is needed to encapsulate the variable parts, and this mechanism must be appropriate with the product strategy.

The SAAM method that is used to evaluate such type of products is described. The evaluation of SPL will focus on the variation points to make sure they are suitable and allow products to be built quickly and to avoid unacceptable runtime performance cost.

At the end of this chapter, you will learn:

- The definition of SPL and its framework
- The definition of variability and its goal
- The method that is used to evaluate SPL system

6.1 SPL Definition

Previous chapters described the architecture and its relation to produce high qualities; the following chapters will show how this will happen in the practice, starting with software product line (SPL). SPL is another way of reusing the architecture across a family of related systems, and from the architecture's point of view, SPL reuses architectural assets. This has many benefits including reducing the cost of construction, reducing time to market, and increasing the quality of building. All these benefits will be the core of the SPL approach to system building.

L. Khalid, *Software Architecture for Business*,
https://doi.org/10.1007/978-3-030-13632-1_6

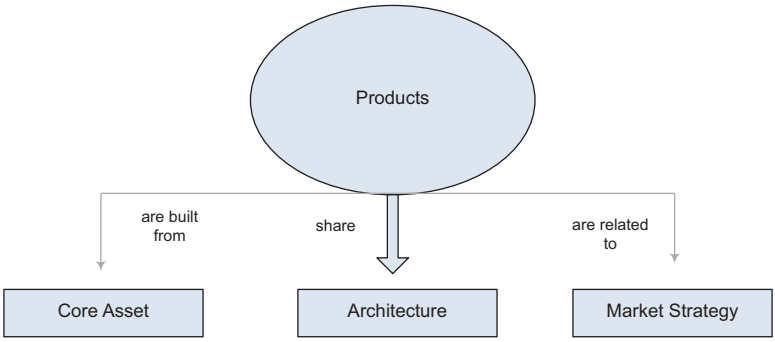


Fig. 6.1 Software product line

The Software Engineering Institute defines software product line as "a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."

Figure 6.1 shows that the products relevant to a specific application domain share common architecture and are built from the same core assets (but there are other core assets as well).

Assets are engineered in a way to be shared across a product line that is used in any product; it works with a variation point (more details later in this chapter).

Any organization that develops software creates multiple software applications that have commonalities. Software product line was designed to manage these software products, and their commonalities are used to increase the benefits of the organization.

Core assets are defined as those reusable artifacts and resources that form the center for the software product line. Core assets often include, but are not limited to, requirement statements, the architecture, reusable software components, domain models, documentation, specifications, schedules, budgets, test plans, test cases, work plans, and process descriptions. The architecture is the key among the core assets.

The commonalties between applications are embodied in the core assets which are the reusable parts of the products. Core assets can be components, tools, framework, etc. Each core asset shares an architecture that all products in the product line will have in common. In addition, a process is attached to each core asset and recommends the optimal method of using the asset to build a product in the product line architecture. Although it is possible to create core assets without any adaptations, in many cases, we need to make some adaptations to these assets in order to use them in the broader context of the product line.

Software product line approaches increase benefits at the organizational level, for example:

- Productivity gains
- Decreased time to market
- Increased product quality
- Decreased product risk
- Increased market agility
- Increased customer satisfaction

These benefits give the organizations a competitive gain, and any organization that launches a product line should have specific and solid business goals that it plans to achieve through product line practice. Moreover, the benefits given above should support carefully with the achievement of these goals, because a software product line requires a start-up investment as well as ongoing costs to maintain the core assets.

6.2 A Framework for Software Product Line Engineering

A framework for software product line is a document that helps the software community in the aims of the software product line. The essential activities of SPL play an important role in the framework before going through developing a product line to achieve the product line development goals.

The essential activities of SPL are:

- Core assets development
- Product development
- Management at organizational and technical levels

In their essence, product lines engage core asset development and product development using the core assets, both under the guidance of technical and organizational management. Core asset development and product development from the core assets can occur in either order: new products are built from core assets, or core assets are extracted from existing products.

Core Asset Development
Core asset development is the creation and maintenance of the artifacts in the product line. These core assets are used to create systems that are equal to the quality standards of the product line. The goal of this activity is to create a capability within the organization to produce a specific type of application and thus will give the similar architecture. The outputs of this activity will then be input to the production development activity. One of the most important outputs is the core assets' base which includes all the core assets that are the basis for the production of products in the product line. These core assets mostly include an architecture that the products in the product line will share, as well as software components that are developed for reuse across the product line. Requirement specifications and domain models are core assets, as is the statement of the product line's scope. Web services and commercial off-the-shelf (COTS) software, if adopted, also form a core asset.

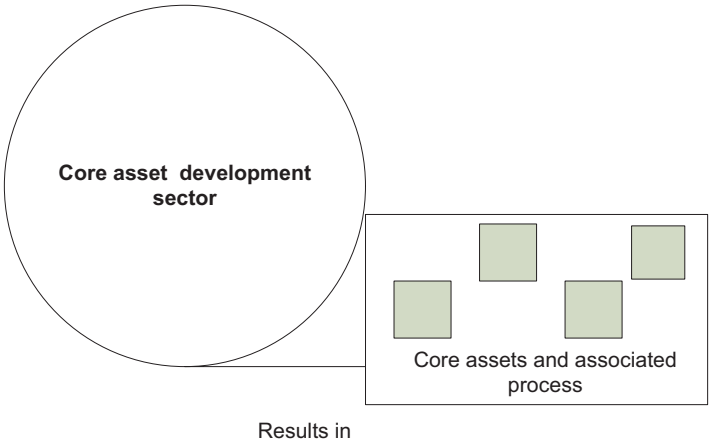


Fig. 6.2 Core asset development sector

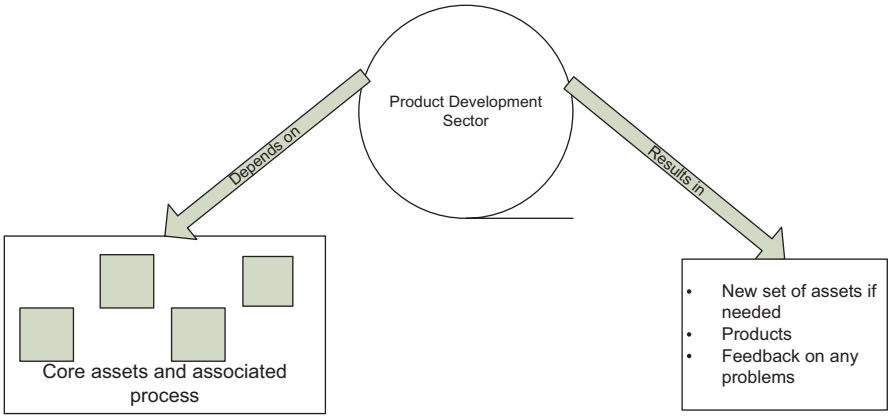


Fig. 6.3 Product development sector

Management artifacts such as schedules, budgets, and plans also constitute core assets. Also, any infrastructure of production such as specific languages, tools, are considered core assets. Each core asset has an associated process that specifies how it will be used in the development of actual products (Fig.6.2).

Product Development

The product development activity depends on the output of the core asset activity. It involves the creation of products or systems from the core assets of the product line. The core assets must be created if the system requires an asset that is not included in the core assets. If the existing core asset does not match the quality requirements of the product under development, the core asset must then be modified. Product builders have a responsibility to give feedback on any problems encountered with the core assets, so the core asset base remains working (Fig. 6.3).

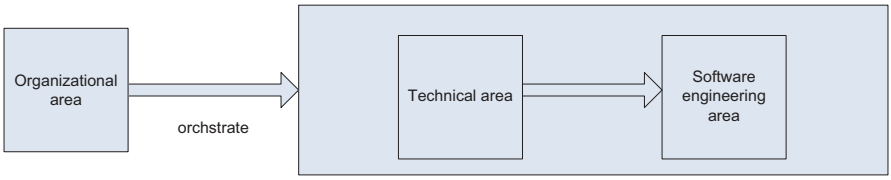


Fig. 6.4 Categories of practice areas

Management
Management (both at the organizational level and the technical level) plays an important role in the success of a product line. Organizational management must create an organizational structure that makes sense for the enterprise and makes sure that the organization receives the right resources. Organizational management also arranges the technical activities and iterations between the essential activities of core asset development and product development. On the other hand, technical management manages the core asset development and product development activities by ensuring that those who build core assets and products are connected in the required activities, follow the processes defined for the product line, and collect data sufficient to follow the progress. Management includes managing specific projects within the product line as well as all product line managers.

To build a software product line, you must carry out the three essential activities described previously, and you also need to know the type of work the organization does in order to successfully carry out the essential work of a product line; this is called a practice area. A practice area is a collection of activities that makes the essential activities more achievable.

The practice area provides starting points from which organizations can make (and measure) progress in adopting a product line approach for software. There are three categories of practice areas (Fig. 6.4):

1. Software engineering practice areas are those necessary for applying the suitable technology to create and evolve both core assets and products.
2. Technical management practice areas are those necessary for managing the creation and evolution of the core assets and the products.
3. Organizational management practice areas are those necessary for managing the complete software product line effort.

Each of these categories requires a different skill set for the people who must carry them out. The categories represent disciplines of the work.

6.3 Architecture and Software Product Line

Every time you make a change to a system, you are reusing its architecture, and you will see that clearly through sets of reusable assets (core assets, defined previously) in the SPL architecture. These sets can be used across many multiple systems. What

make core assets work quickly are the built-in variation points (will be explained later in this chapter) or places where they can be quickly tailored in preplanned ways. Once the core assets are in place, building the system becomes a matter of accessing the appropriate assets in the core asset base, exercising the variation points to configure them as required for the system being built, and then assembling that system.

6.3.1 What Makes a Software Product Line Succeed?

A product line succeeds because the commonalities shared by the software products can be exploited through reuse to achieve economies of production. The products are built from common assets in a predetermined way. Reusing of these commonalities includes:

- Requirements: most of the requirements are common early in the life cycle of the system.
- Architectural design: the quality goals of the system such as reliability and performance are inhibited once the architecture is created. That is why an architecture represents a large investment of time from the organization's most talented engineers.
- Software elements: software elements are applicable and reusable across individual products. It includes initial design work such as designing the element's interface, its documentation, etc. The successful design is captured, and unsuccessful designs are avoided.
- Modeling and analysis: performance models, schedule analysis, distributed system analysis, network load, and so forth can be reused across products in the product line.
- Testing: test plans, test processes, test cases, test data, and so forth can be established once for the product line.
- Project planning: budgeting and scheduling are more predictable because experience is the best sign of future performance. Planning the team's size and composition is easily done from past experience.

Artifact reuse enables reuse of:

- Processes, methods, and tools: configuration control, facilities, documentation plans, tools, system generation, deployment processes, and coding standards are all established once for the whole product line.
- People: the people's experience is applicable across the entire product line.
- Exemplar systems: deployed products serve as demonstration prototypes of the system.
- Defect elimination: product lines enhance quality because each new system takes advantage of the defect elimination for the entire family.

All of the above reuse helps products begin more quickly with high quality and more expected budget and schedule.

The software architecture plays the important role in the set of core assets. Product line architecture must apply to all members of the product line—even if their functions and quality attributes are different. Also, the architecture of the product must define the commonalities and hold the variations between the products in the product line.

The architects of the software product line need to think about the following things:

- Identifying variation points in the products
- Supporting variation points by introducing the variation mechanism
- Evaluating the architecture for product line suitability

These roles will be explained later in this chapter.

6.4 The Quality Attribute of SPL (Variability Quality)

Every system has its own qualities. Variability is the most obvious quality associated with SPL (in spite of other existing qualities such as performance and security). Variability is a special form of modifiability; it is the ability of a system to support the production of a set of artifacts that vary from each other in a preplanned way. Sometimes, it can be defined as the ability of core assets to adapt usages in the different contexts that are within the scope of the product line; its goal is to satisfy the variations and commonalities that are identified by the scope of the product line.

> *Product line scope* can be defined as a statement about what systems an organization is ready to build as part of its line and what systems it is not ready to build. The scope is driven by strategic planners, domain analysts, marketing, and the need to build multiple products that are similar. It is very important to the architect because it defines what is common across all members of the product line and what is different.

Table 6.1 shows the general scenario for variability quality. This is a general table; it can be used as a starting point for any future work.

SPL is described by the following representations:

Variability point

the specific points where the variability takes place. It is the difference between two or more products. Differences can involve:

- Functions
- Quality attributes
- Platforms
- Interfaces

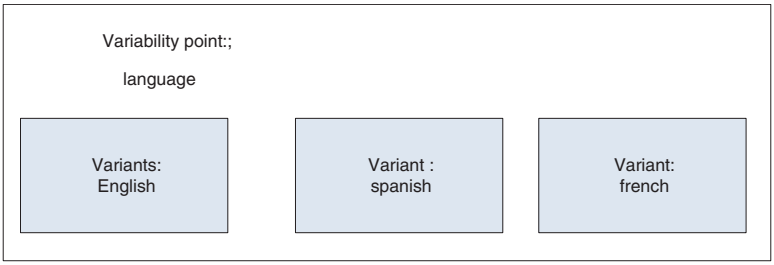
Table 6.1 General scenario for variability

Portion of scenario	Possible values
Source	Requests variability
Stimulus	Requests to support variations in hardware, technologies, feature sets, quality attributes, etc.
Artifacts	Affected assets, such as requirements, architecture, component x, etc.
Environment	Variants are to be created at runtime, build time, and development time
Response	The requested variants can be created
Response measure	A specified cost and/or time to create the core assets and to create the variants using these core assets

Ideally, variation points are identified during requirements, elicitation, and analysis. In reality, identifying variation points must be a continuing activity.

Variants

a set of alternative variability; all variability scenarios specify variations that have to be included in a range of product. For example, the customers of a home automation system can decide on the language of the user interface before the system is installed.



The relation between variability point and variants

6.4.1 The Goal of Variability

The goal of variability in SPL is to increase return on investment (ROI) for building and maintaining products over a specified period of time. The cost of producing products includes the cost of building the core assets on which they are based on and that is why the variation mechanism (described later) will affect both the cost of building the core assets and the cost of building products from core assets.

ROI can also be used in producing the higher quality. Higher quality needs more cost to be built, but this will satisfy the customer's requirements, enhance reputation for quality, and cause fewer defects to fix and other goals that increase the ROI.

6.4.2 *Variation Mechanism*

As a part of the core asset design, the developer has to choose a variation mechanism to encapsulate the variable parts; this mechanism must be appropriate with the product strategy. For example, if implementing a mechanism needs 2 months but the product must be delivered in 1, then choosing that mechanism is not suitable. To help the decision process, a variation mechanism has a set of properties, some of which are:

- The cost to implement the mechanism
- The cost and time to exercise the mechanism
- The group of users that use the mechanism
- The effect of the mechanism on quality

and so on.

The important thing is the information required to make the decision on using the variation mechanism. Organizations that use product line should have a catalogue of variation mechanisms that includes rating and stakeholders related to that organization. Table 6.2 shows some variation mechanisms.

The cost values are relatively value, for example, the cost to exercise parameter values (low) is less expensive than using a new class using inheritance (high).

The architect of a product line should document these mechanisms; it will be under the section of variability guide in the documentation. The variability guide should describe each variation mechanism, how and when to exercise it, and what variations it supports. The documentation also has to explain valid and invalid variation choices if certain combinations of variation are disallowed.

> *Variability guide* is a section in the documentation; it shows how to exercise any variation points that are part of the architecture shown in the view.

Table 6.2 Variation mechanism examples

Variation mechanism	Properties to be built into core asset	Properties to be exercised when building products
Inheritance	Cost: medium Skills: object oriented languages	Cost: medium Tools: compiler Stakeholder: product developers
Templates	Cost: medium Skills: abstractions	Cost: medium Tools: none Stakeholder: the administrator of the system, developers
Aspects	Cost: medium Skills: aspect oriented programming	Cost: medium Tools: aspect oriented language compiler Stakeholder: product developers

The documentation for product line architectures will also include:

- Documentation for all core assets (requirements, design, and so forth)
- Variation points: Where variation occurs in the products and how it is accommodated
- Production plan: Describes how products will be built or instantiated

6.5 Evaluating a Product Line Architecture

One of the most important things in software system development today is the quality of that system. The purpose of evaluation is to analyze the software architecture to identify the possible risks and verify that the quality requirements have addressed the design of the system.

The architecture of a software product line is one of its most important artifacts; it represents an abstraction of the products that can be generated. It is important to evaluate the quality attributes of product line architecture for the following reasons:

- Increasing the productivity of the product line process and the quality of their products.
- Decreasing their time to market.
- Improving the handling of the product line variability.
- Evaluating product line architecture can serve as a basis to analyze the managerial and economical values of a product line for software managers and architects.

The evaluation of SPL will focus on the variation points to make sure they are suitable and allow products to be built quickly and that they avoid unacceptable runtime performance cost. If the evaluation depends on a scenario, then it will be expected to gather scenarios that involve instantiating the architecture to support different projects in the product line.

Software product line has different quality attribute requirements; evaluating the architecture must have the ability to satisfy all the combination of these qualities. SEI developed the following scenario-based methods and for different types of qualities:

- SAAM: Software Architecture Analysis Method
- ATAM: Architecture Trade-off Analysis Method
- ARID: Architecture Reviews for Intermediate Design

The three methods have been applied for years on lot of projects of all sizes in different types of domains, so you can use these methods in SPL architecture. When evaluating the architecture of product line (the core assets specifically), you need to be able to know whether it is appropriate not only for that particular product but for

the continuation of the products to come. SAAM method concentrates on modifiability in its various types (portability, subsetability, and variability) and functionality.

It may be worthwhile to evaluate product line architecture:

- Immediately after the architecture for the entire product family has been created
- When creating the first product "instance" using the architecture
- After major revisions to the architecture
- When market demand or stakeholders change significantly

When a new product that lies outside of the product family is being proposed, the product line architecture should be evaluated to see if it will suffice for the new product.

At last, you must know that two groups of people are involved in architecture evaluation:

The first is the evaluation team: those are the people who will analyze and evaluate the system.

*The secon*d group is the stakeholders: these are the people interested in the architecture of system.

6.6 Summary

The Software Engineering Institute defines software product line as *"a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."*

A framework for software product line is a document that helps the software community in the aims of software product line. The essential activities of SPL play an important role in the framework before developing a product line to achieve the product line development goals.

The essential activities of SPL are:

- Core assets development
- Product development
- Management organizational and technical levels

A product line succeeds because the commonalities shared by the software products can be exploited through reuse to achieve economies of production. The products are built from common assets.

The evaluation of SPL will focus on the variation points to make sure they are suitable and allow products to be built quickly and to avoid unacceptable runtime performance cost. If the evaluation depends on scenario, then it will be expected to gather scenarios that involve instantiating the architecture to support different

projects in the product line. SEI developed the following methods for scenario-based methods and different types of qualities:

- SAAM: Software Architecture Analysis Method
- ATAM: Architecture Trade-off Analysis Method
- ARID: Architecture Reviews for Intermediate Design

SAAM concentrates on modifiability in its various types (portability, subsetability, and variability) and functionality.

References

P. Clements, R. Kazman, M. Klein, *Evaluating Software Architectures: Methods and Case Studies* (Addison Wesley, 2002)

J. Frank et al., *Software Product Line Engineering: Foundations, Principles and Techniques* (Springer Science & Business Media, 2005)

L. Northrop et al. "A framework for software product line practice", version 5.0 (2012)

Further Reading

Sh. Cohen, P. Krut, Managing variation in services in a software product line context. Technical notes, CMU/SEI (2010). http://www.sei.cmu.edu/

D. Reis, P. Valle, *Variability and Software Product Line Architecture* (University of São Paulo, Institute of Mathematics and Computer Science, 2016). https://edisciplinas.usp.br/pluginfile.php/977101/course/section/268869/Seminar5_SPL_SPLA.pdf

L. Sion, G. Jong, *Systematic Quality Trade-off Support in the Software Product-Line Configuration Process* (ACM, 2016). https://doi.org/10.1145/2934466.2934481. ISBN 978-1-4503-4050-2/16/09

Chapter 7
Internet of Things (IoT)

Abstract The origin of the Internet of Things dates back to Kevin Ashton in 1999. Ashton is an innovator and consumer sensor expert who describes the IoT as a network that connects objects in the physical world to the Internet; but this definition is still seen as odd. In the twenty-first century, computers can sense things for themselves, for example, GPS-based location sensing. This chapter describes the architecture of IoT with its basic characteristics. Two qualities of IoT with their QAS's will be defined: interoperability and modifiability.

DYAMAND, a case study for IoT, will be explained in detail to solve the interoperability problem. Its requirements and architecture will be described in detail.

What makes the content of this chapter different from other IoT books is that it takes the software architecture's point of view, and that is the main core for this book.

7.1 IoT Definition

Internet of Things is a new revolution of the Internet. It can be defined as a type of network that connects anything with the Internet, and this will be based on protocols through sensing equipments in order to exchange information and communication to achieve smart recognition. The goal of IoT is to connect things anytime, anyplace with anything using any path or network and any service. The important thing to take into account is that IoT is not a single technology; it is a combination between hardware and software technologies; without these technologies IoT would not be possible.

IoT provides solutions depending on the integration of information technology (that refers to hardware, software that is used to store, retrieve, and process data) and communication technology (which includes electronic systems that are used to communicate between individuals and groups).

L. Khalid, *Software Architecture for Business*,
https://doi.org/10.1007/978-3-030-13632-1_7

How does Kevin Ashton[1] define IoT?

"Internet of Things means sensors connected to the Internet and behaving in an Internet-like way by making open, ad hoc connections, sharing data freely and allowing unexpected applications, so computers can understand the world around them and become humanity's nervous system."

The main vision of IoT is that things are able to talk and their data can be processed to perform the specific tasks.

Finally, some sources mention which technologies and protocols can be used for the IoT, such as the Internet, cloud computing, RFID, IPv6, and much more, but these will be excluded since they are out of the scope for this book.

Cloud computing: Cloud computing is a type of computing that relies on shared computing resources rather than having local servers or personal devices to handle applications.

RFID: Radio frequency identification devices are wireless microchips used for tagging objects for automated identification.

IPv6: Internet Protocol version 6 is the most recent version of the Internet Protocol (IP), the communications protocol that provides an identification and location system for computers on networks and routes traffic across the Internet.

The fundamental characteristics of the IoT are:

- *Things-related services*: The IoT is capable of providing things-related services within the constraints of things, such as privacy protection between physical things and their associated virtual things. To provide this fundamental within the constraints of things, both the technologies in the physical world and the information world will change.
- *Interconnectivity*: In IoT, anything can be interconnected with the global information and communication infrastructure.
- *Heterogeneity*: The devices in the IoT are heterogeneous. They can interact with other devices or service platforms through different networks.
- *Dynamic changes*: The state of devices change dynamically, for example, connected and/or disconnected. The number of devices can also change dynamically.

[1] *Kevin Ashton is an innovator and consumer sensor expert who coined the phrase the Internet of Things to describe the network connecting objects in the physical world to the Internet.*

- *Scale*: The number of devices that need to be managed and that communicate with each other will be larger than the devices connected to the current Internet.
- *Safety*: This includes safety of personal data and safety of physical well-being. Security of endpoints, networks, and data movement.
- *Connectivity*: Connectivity enables network accessibility and compatibility. Accessibility is getting on a network, while compatibility provides the common ability to produce and consume data.

Despite the fact that the architecture layers of IoT are not in the scope of this book, it will be described briefly in order to know how scalability and other qualities can perform through IoT. Some references describe three layers, others describe six, but in general IoT must have layers to obtain information from the physical world layer for transferring data and must also have a layer for using this data. This book divides the architecture of IoT into four layers (Fig. 7.1):

Smart Device/Sensor Layer
This is the lowest layer. It is made up of smart objects that are integrated with sensors. Sensors are grouped according to their purposes, for example, environmental sensors, home appliance sensors, etc. Those sensors have the ability to measure the physical property and then convert it to signals that can be understood by instruments. Most sensors need connectivity to the sensor gateway which can be in the form of LAN (Local Area Network) such as Wi-Fi or PAN (Personal Area Network) such as ZigBee. Some sensors do not require a connection to the aggregators; their connectivity to backend servers/applications can be provided using wide area network (WAN) such as GSM (Global System for Mobile communication). Sensors that use low power and low data rate connectivity typically form networks commonly known as wireless sensor networks (WSNs).

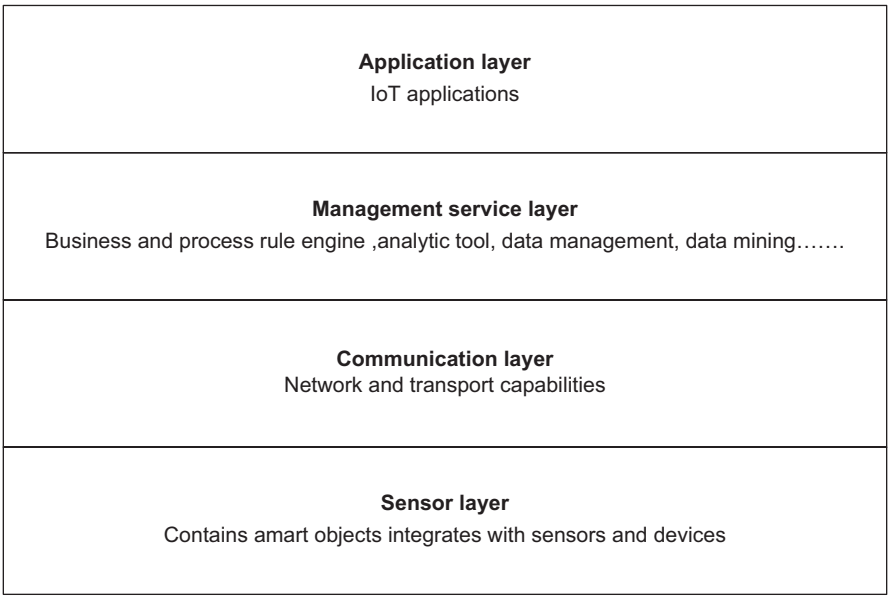


Fig. 7.1 Architecture layers of IoT

Gateways and Networks
Large numbers of data produced by small sensors will require high performance and robust wired or wireless network infrastructure as a transport medium. Multiple networks with different technologies and access protocols are needed to work with each other in a heterogeneous configuration. These networks can be in the type of a private, public, or hybrid models and are built to support the communication requirements for latency, bandwidth, or security.

Management Service Layer
This layer enables the processing of information possible through controlling the security, analytic, process modeling, and management devices. The most important feature of this layer is the business and the process rule engines. IoT brings connection and interconnection of objects and the system together presenting the information in the form of data, events, current location, and traffic data. Some events need filtering, while others need response to the immediate situations. Rule engines support the formulation of decision logics.

Application Layer
This layer covers the smart environmental spaces such as transportation city, factory, and emergency health care.

The thing you must know that the security features must include for all four layers beginning from sensor layer ending with the application layer, this feature prevents systems from hacking by unauthorized persons, and of course this will reduce the possibility of risk.

Three Cs impact of IoT to the business and society:

Communication: IoT communicates information to people and systems and data from sensors that can monitor a person's signs; location is very important for items that move.

Control and Automation: in many cases, a consumer or business is able to control a device remotely, for example, controlling the temperature of the environment by a consumer.

Cost Saving: with new sensor information, IoT can save money by minimizing the failure of equipments and permitting the business to perform planned maintenance. Many other ways can be used by IoT to save money.

7.2 Architecture and IoT

The goal of software architecture is to design a system that meets the quality attributes. Choosing one style for software architecture will show different levels of quality attributes for the same system. This is very important for IoT systems because it can include different kinds of categories, for example:

- Wireless sensor/actuator networks; sensors provide data either to users or actuators. Actuators receive data either from sensors or from users.
- RFID enable tracking.
- Smart homes.
- Connected cars.
- Devices that connect via Bluetooth-enabled mobile phone to the Internet.

The result from that is that there is no single architecture that suits all these areas and the requirements of each area. It always depends on the goal of quality requirements. Also, it is hard to provide reference architecture for the entire IoT system.

The terms reference model and reference architecture will be described in Fig. 7.2

A reference model is a division of functionality into elements and the data flow between them; it is the domain analysis.

Reference architecture is a reference model that is mapped onto the software elements that implement the functionality defined in the model. It is a mirror to the software architecture.

7.3 Basic Qualities of IoT

Previously, I talked about quality attributes in details. In order to build systems that support the required qualities, you need a way to express the quality attributes and understand how these qualities can be achieved. In this section, two important

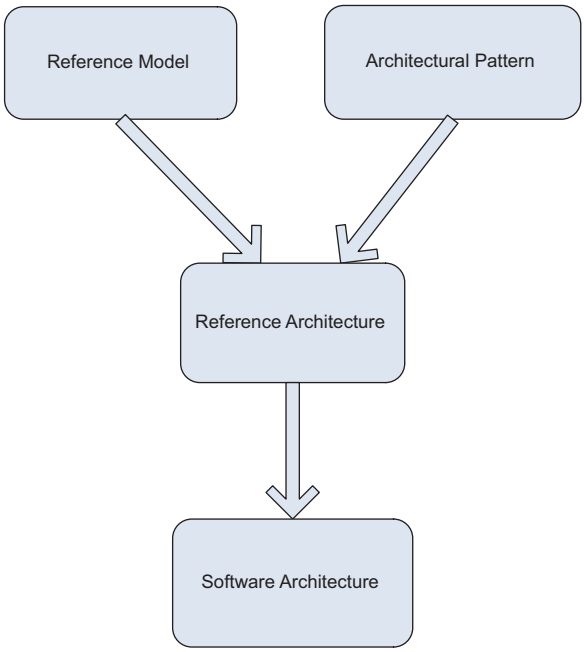


Fig. 7.2 Reference architecture

qualities related to IoT will be discussed: interoperability and modifiability. Also, security is an important quality to IoT; description of general and concrete scenarios for security will be in Appendix A.

7.3.1 Interoperability Quality

Interoperability is the ability of more than two systems or components to exchange meaningful information through interfaces in a specific context. In this definition, there is no limit on exchanging data (syntactic interoperability) but having the capability to interpret data being exchanged (semantic interoperability). Any discussion regarding interoperability needs to identify with whom, with what, and under what circumstances; that is why the context has to be included throughout exchanging.

Interoperability is a runtime quality attribute, the architect always needs to design the system to be interoperable with other systems, and that is done through bounding the system that is built with another system later in the life cycle. However, sometimes systems will connect early with the architect's system. In that case, quality will be produced at the design time.

Regarding interoperability in IoT, Internet of Things provides connectivity between people, processes, and things, but we are far away from speaking the same language. Existing solutions cannot easily talk to each other. Probably, the greatest

difficulty for large-scale adoption of IoT is the lack of interoperability. It prevents companies from delivering services that enable full connection to the digital world.

Interoperability can be on different levels. Systems can simply exchange data and call each other's functions. They can also work together as a single business application with integrated communication channels, state synchronization, and a common interpretation of the data. For example, in the banking sector, the success of electronic and mobile payment systems largely depends on the interoperability with the back office systems and the secure handling of the communication between them.

Some important reasons why the system has to be interoperable are:

- Increasing level of reusability because interoperability enables using component-based software engineering.
- Sometimes systems need to interoperate with the created system even when the architect does not know anything about those systems, for example, Google Map application.
- In IOT, the architect often builds integrated systems where applications use capabilities from existing systems. For example, connecting medical sensors with systems that process the data and secure event servers that send out alerts.

General quality attribute scenario for interoperability is described in Fig. 7.3.

Now let us apply this quality on a smart medicine cabinet: *A medicine cabinet is able to send an alert to a doctor when an at-home patient's supply of drugs runs low. The new instruction is verified by that doctor and forwarded to the pharmacist who sends a new supply of drugs with a courier.*

This needs a huge number of interoperability requirements:

- The drug recipient talks to the cabinet of the system.
- The medicine cabinet interacts with the doctor's patient system.
- The pharmacist communicates with the dispatching service of the messenger company.

To make it even more complex, there are many different ways to notice the supply level of drugs, and there are many existing patient systems used by doctors.

Let's focus on the interoperability scenario between a drug distributor device and the smart medicine cabinet. Pharmaceutical companies have different types of smart

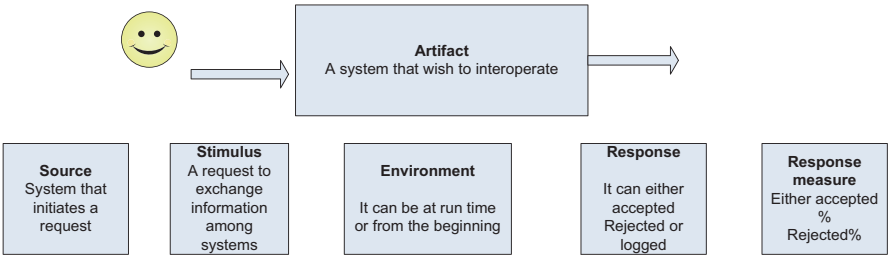


Fig. 7.3 QAS for interoperability

medicine bottles or tablet strips. This is also an area of innovation with new sensors and protocols appearing on the market regularly. Table 7.1 shows the concrete interoperability QAS

There are two categories of interoperability tactics: locate and manage interfaces (Fig. 7.4).

Table 7.1 Concrete scenario for interoperability quality

Source	The drug point to a device, for example, a smart medicine bottle
Stimulus	Sends a message that includes an identification to the drug and the remaining supply level
Artifact	Smart medicine cabinet local control system
Environment	The drug distributor unit discovered at runtime
Response	The smart medicine cabinet if the received data is complete and matches the data in the database
Measure	For the new drug that is discovered before: 95% of the notifications has to be correct and complete A maximum 5% can be rejected with a notification to the smart cabinet support center

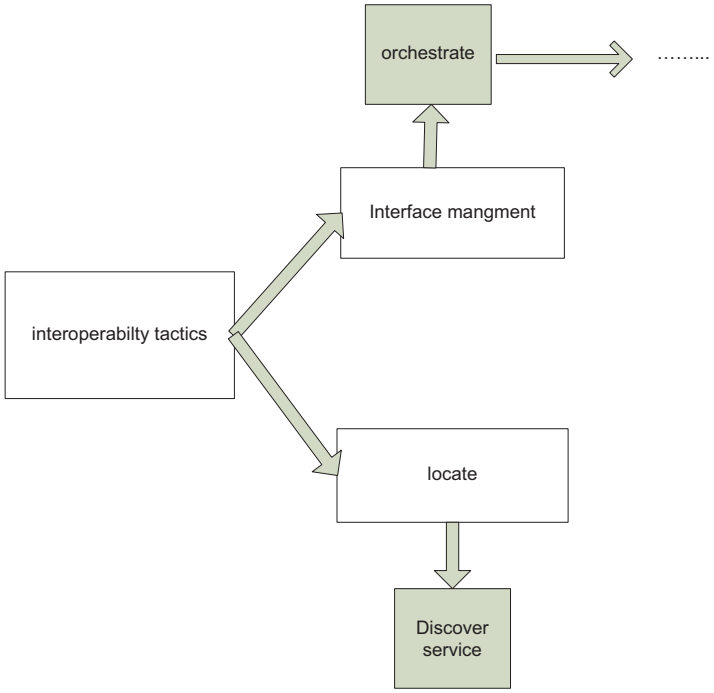


Fig. 7.4 Tactics for interoperability

Note *The basic goal of these tactics is that after requesting to exchange information, the request must be handled correctly.*

Figure 7.4 shows that there is only one tactic for locate and discover service. It is used when systems that interoperate must be discovered at runtime. On the other hand, managing interfaces has two tactics: orchestrate which is used as a control mechanism to invocate specific service and tailor interface which adds or removes capability to an interface.

7.3.2 Modifiability Quality

Modifiability is a design quality attribute that can be supported by mechanisms in all phases of the system life cycle. The goal of modifiability is to make flexible systems that can hold change at a minimum cost.

Modifiability has great effects on business competitiveness. A modifiable architecture also creates opportunities for reusable components and can simplify maintenance because the small components are less complex and, if well designed, have fewer dependencies on other components. To plan for modifiability, the software architect asks four main questions:

Who Makes the Change?
Different stakeholders can implement changes to a system: software engineers, system administrators, field engineers, and users.

What Is the Probability of the Changes?
A system cannot plan for all potential changes; that is too expensive, and it may suffer from quality attributes problems. The architect has to make the tough decisions about which changes are probable and which changes are to be supported.

What Can Change?
A change can occur at any aspect of the system.

How Is the Cost of Change Measured?
Two types of costs must be taken into consideration through dealing with modifiability: the first is the cost of introducing the mechanism(s) to make the system more modifiable and the second is the cost of making the modifications using the mechanism(s). The dynamic environment of IOT systems will lead to adaptive architectures that need to support modifiability. This will be the subject of the DYAMAND case study, which you will discover in the next section. Concrete modifiability scenario will show in Fig. 7.5.

The impact that appears in the response measure will show that modifiability has a big impact on business competitiveness. It determines the effort to produce the next release, and it will shorten the time to market for releasing new features with less cost. Table 7.2 represents the general scenario for modifiability.

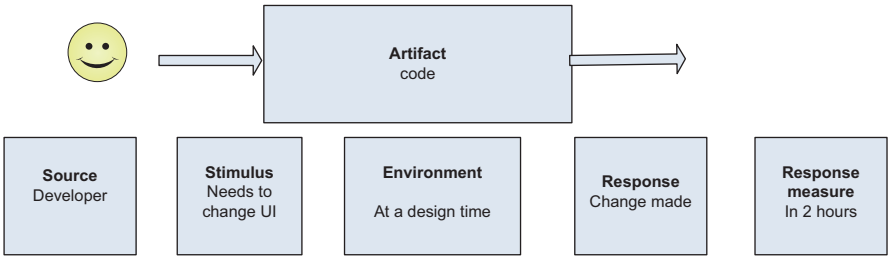


Fig. 7.5 Concrete scenario for modifiability

Table 7.2 General scenario for modifiability

Source	End user, developer, system administrator
Stimulus	An order to add/delete/modify functionality or change a quality attribute, capacity, or technology
Artifacts	Code, data, interfaces, components, resources, etc.
Environment	It can be at any time in the life cycle, it may be at runtime, compile time, build time, etc.
Response	One or more of the following: Make a modification Test a modification Deploy a modification
Response measure	Cost in terms of size, complexity of affected artifacts, effort, calendar time Money, etc.

There are many tactics to control modifiability; their goal is to control the complexity of change (Fig. 7.6):

- Size of the module – splitting the module will reduce the cost of making a modification to the module that is being split. The split is also chosen to reflect the type of change that has to be made.
- Increasing cohesion – increasing semantic coherence; if the responsibilities A and B in a module do not provide the same purpose, one of them should be put in a different module.
- Reducing coupling – the main tactics for reducing the coupling are:

1. Encapsulation: it introduces an explicit interface to a module. The interface designed to increase modifiability should be abstract with respect to the details of the module.
2. Using an intermediary: breaks a dependency, the type of intermediary depends on the type of dependency, for example, in publish-subscribe, data repository is used to separate readers of data from writers of that data.
3. Reactor: is a tactic undertaken where two modules are affected by the same change because they are duplicates of each other.
4. Restrict dependencies: restricts the modules which a given module interacts with or depends on.

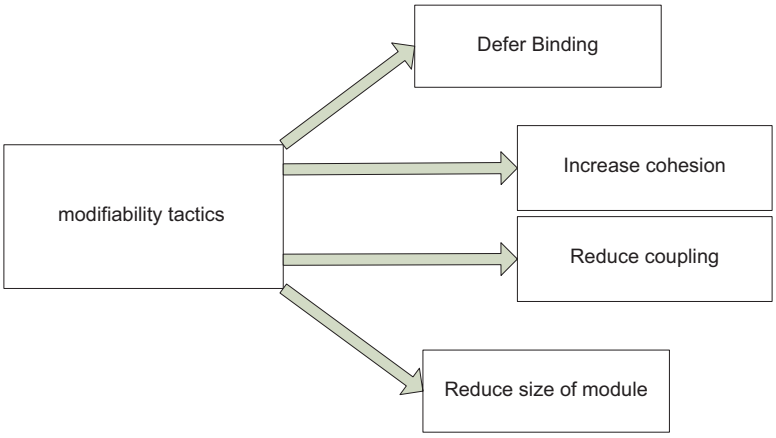


Fig. 7.6 Tactics for modifiability

5. Abstract common services: where two modules provide similar (but not quite the same) services, any modification to such services will be in one place, and this may be cost-effective.
6. Defer binding: in general, an architecture that equips modification late in the life cycle will cost less than doing this earlier in the life cycle.

Note *The main goal of this tactic is that when change is requested, the change must be made and deployed.*

7.4 DYAMAND: Case Study

In the real world, a wide range of technologies that want to state IoT rise, but, as said before, all of these technologies have a basic problem which is lack of interoperability with each other, and this causes problems. DYAMAND (Dynamic Adaptive Management of Network and Devices) solves this problem by combining these technologies in order to be used by application.

DYAMAND is a software platform that enables application developers to be more easily connected in order to develop the application without any efforts.

Before going through details, the brief definition of the framework of DYAMAND needs to be explained in order to show how it works. SDPs (Service Discovery Protocols) are a part of this framework; the framework acts as a middleware layer

UPnP (Universal Plug and Play) describes the devices that work with a computer system as soon as they are connected. It uses network protocols to allow a wide range of devices to interconnect with each other.

between the application developer and the controllable devices and permits to translate devices/services as declared by one SDP instance to another, while abstracting the SDP toward the application developer.

SDPs are a protocol that enables dynamic discovery of services in a network; DNS-SD (domain name system service discovery) is an example of this protocol.

SDPs are composed of three basic functions:

1. *Discovery*: the protocol must offer the discovery of devices and/or services.
2. *Control*: SDP allows controlling the services it discovers (after supposing that the interoperability issues of different discovery/service models and different service type representations were solved and services could be discovered uniformly). If SDP does not provide control, an external control protocol must be used.
3. *Eventing*: SDP embeds events in it; for example, the General Event Notification Architecture (GENA) that is embedded in UPnP supports the application to receive events without knowing about the service.

Going back to the DYAMAND, runtime modifiability is the most important quality attribute the framework needs to take into consideration. This is why plug-in

> The term "POJO" initially denoted a Java object which does not follow any of the major Java object models, conventions, or frameworks. Nowadays "POJO" may be used as an acronym for "Plain Old JavaScript Object," in which case the term denotes a JavaScript object of similar pedigree.

architecture was developed, to take into consideration the possible differences between SDPs, devices, and services. Also the framework should deliver API that will be simple to use and easy to understand. Plug-in can be in three types:

- SDP plug-in which is responsible to implement (parts of) SDP
- Translation plug-ins which are responsible for translating SDP-specific objects into generic ones and vice versa
- Application plug-in which is responsible for implementing a specific use case

All three types of plug-in will go through *device/state change translation cycle*. After this cycle, the framework will notify the application plug-in of translated services or state changes.

Sometimes command translation is used in the framework. Here, the operation will go in the opposite direction which means that the application plug-in needs to interact with a service; it retrieves the POJO associated with the service (just acts with java interface). Using POJO improves the usability of the framework.

As a student, all you need to know is a simple background on how the framework of DYAMAND works from the point of view of application developers; even developers do not need to understand how SDP works to develop the application.

7.4.1 DYAMAND Requirement

Software requirement from the DYAMAND point of view starting with the use cases of DYAMAND. From the perspective of DYAMAND, a limited set of use cases exist.

You can conclude from Table 7.3 that the basic actors of DYAMAND are two: devices and applications. Three groups appear in the table. The first group of use cases is to discover the devices; an application must be able to discover the availability of devices to be able to control them (*use cases 1–3*). After that, an application has to be able to execute that device effectively (*use cases 4–6*). Adding, removing, and updating new technologies and functions that relate to the modifiability of the system of software will be in use cases 7–9.

Now going through the quality attribute, you see the performance quality of DYAMAND as in Fig. 7.7

Performance requirement is an essential quality for the application using this framework. Figure 7.7 shows that whenever a device notifies its availability, the system translates the device to its generic representation and notifies the interested application within 200 ms.

Apart from performance requirements, modifiability (that is tightly coupled with interoperability) is essential for the system. Whenever support for a new technology gets added at runtime, additional devices must be supported without any change to the application. For example, if there is an application that uses motion sensors, adding a new technology that supports motion sensors just means the application will be notified of new devices being available. Adding support needs to be done within 500 ms. This framework has no use without solving the interoperability problem, and modifiability would be the unique selling proposition of the framework (Fig. 7.8).

Table 7.3 Basic actors and their use cases

Use cases	Actors
1. Get list of devices	Application
2. Notify availability	Device
3. Notification of new device	Application
4. Execute command on device	Application
5. Send status update	Device
6. Receive status update of device	Application
7. Add support of new technology	Admin application
8. Remove support of technology	Admin application
9. Update support of technology	Admin application

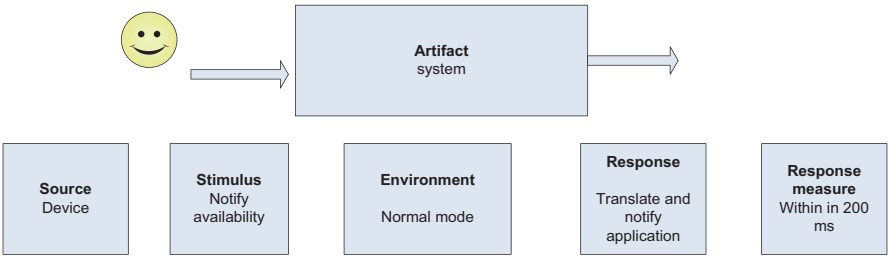


Fig. 7.7 The concrete scenario for performance quality

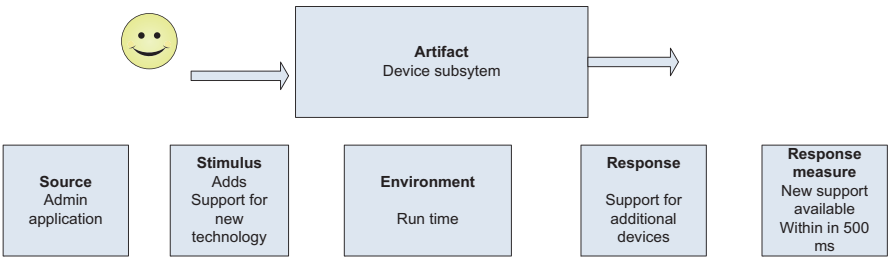


Fig. 7.8 Modifiability/interoperability qualities for DYAMAND

7.4.2 DYAMAND Architecture

Architectural pattern is a basis of the complete architecture. Sometimes combining more than pattern features can help build the architecture. According to the important qualities that associate the DYAMAND (modifiability and interoperability), building the architecture tactics that are associated with each quality will be needed. Candidate tactics for each quality will be as in the following:

The candidate tactics for modifiability are:

- Increase semantic coherence
- Use an intermediary
- Restrict dependencies
- Anticipate expected changes
- Abstract common services
- Restrict communication paths
- Defer binding

The candidate tactics for interoperability are:

- Discover services
- Orchestrate

Relevant tactics are selected to be used for selected patterns, for example, in Table 7.4.

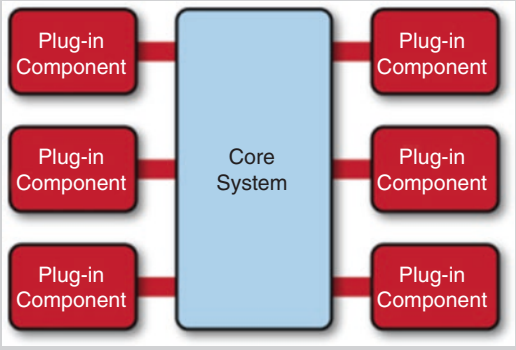
Table 7.4 Some tactics for modifiability and interoperability with candidate patterns

Tactics	Microkernel	Component configuration	Layers
Abstract common service	Yes	Yes	Yes
Runtime registration	No	Yes	No
Anticipate expected changes	Yes	Yes	No
Discover services	Yes	No	No
Orchestrate	Yes	Yes	No

Based on Table 7.4, "anticipate expected changes" is better in both microkernel and component configurator than in the layers pattern. Layers explicitly defines the services needed in each layer, while microkernel *abstracts common services* and enables easy addition through the use of internal and external servers. "Discover services" is inherent when using the microkernel pattern. "Runtime registration" is only available in the component configurator pattern. According to features of Table 7.4, combination between microkernel and component configurator patterns seems to be the best fit for the requirements of this system and that will be shown in Fig. 7.9: a static view of the architecture also the plug in context is shown in Fig. 7.10.

Quick Note

Microkernel pattern (plug-in architecture pattern) is a natural pattern for implementing product-based applications. The microkernel architecture pattern consists of two types of architecture components: a *core system* and *plug-in*.



Component configurator pattern consists of:
Component: a uniform interface that can be used to configure and control the type of application service or functionality provided by component implementation (initializing, suspending, resuming, and terminating)
Concrete components: implement component control interface to provide a specific type of component
Component repository: manages all concrete components that are configured currently into an application

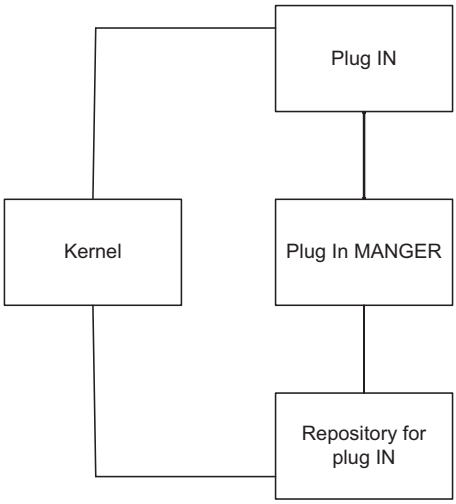


Fig. 7.9 A view of the combination patterns

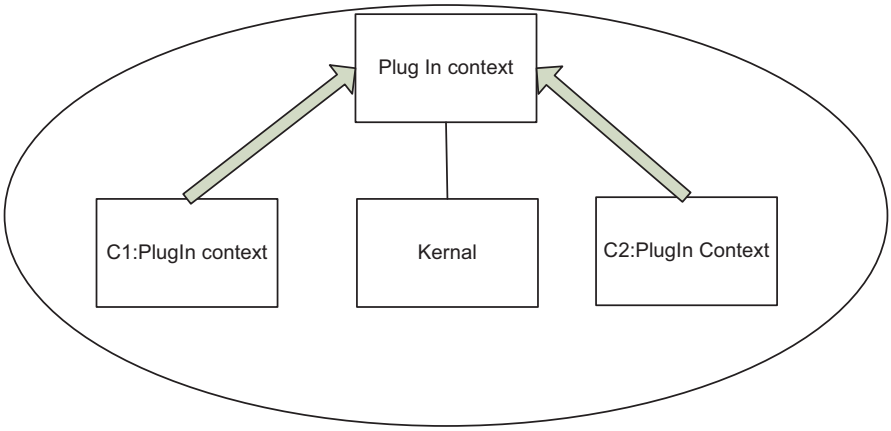


Fig. 7.10 Plug-in context

Figure 7.9 shows that the plug-in manager takes the responsibility of the component configurator component in the component configurator pattern, which is to hold the life cycle of the concrete implementations of plug-in. On the other hand, plug-in repository is the component repository in the component configurator pattern.

Plug-ins should use the available communication mechanisms provided by the kernel in order to communicate with each other. This highly strengthens the modifiability of the system.

Plug-ins that provide generic interfaces (e.g., motion sensors) implement the adapters in the microkernel component. Different than in the microkernel pattern,

adapters will not communicate with external servers, only with the kernel component itself.

Plug-ins that translate from protocol-specific devices to generic ones implement the external server functionality in the microkernel pattern in that it implements policies on how to interpret protocol-specific parameters and translate them to generic parameters.

Application plug-in uses the generic adapters of devices to implement a generic application; these are the client components in the microkernel pattern.

In addition to the modifiability, many other qualities can be satisfied through this view; for example, security can be provided by assigning a plug-in context object to a concrete plug-in while starting. This is the "single point of entry" that acts as a gatekeeper to the kernel (Fig. 7.10).

Plug-in context is a single point entry that acts as a gatekeeper toward kernel services.

7.5 Evaluating IoT Architecture

IoT application integrates multiple software elements that are distributed across several nodes and communicate with each other, through using Internet protocols and standards. IoT is composed of:

- A sensing device, i.e., a node with computing and communication capabilities
- A gateway device responsible for enabling the connectivity between the short-range network of sensing devices, on one side, and the wide area network, on the other
- A user interface (UI) device that complements the sensors and actuators with limited UI capabilities and allows the users to interact with the IoT application and that will usually be through a dedicated app or through a web interface
- A web component responsible for executing the application logic of the application or service on cloud infrastructure and mediating the communication with other sensing and actuating devices

Evaluating terms that are related to IoT can take many forms from different types of view; for example, if you search on the application provider's viewpoint, you go through the framework of the IoT which depends mainly on the framework of BPF (Business Process Framework) by TM (Telecom Operation Map) forum. In this framework the difference is made between the processes dealing with the design and development of the service and the configure operations process, such type of framework will be the basis for evaluation the framework of IoT.

Figure 7.11 represents the IoT evaluation framework that provides to support for:

1. Design and implementation: the design and implementation of application elements to be deployed at the device, gateway, UI device, and web component

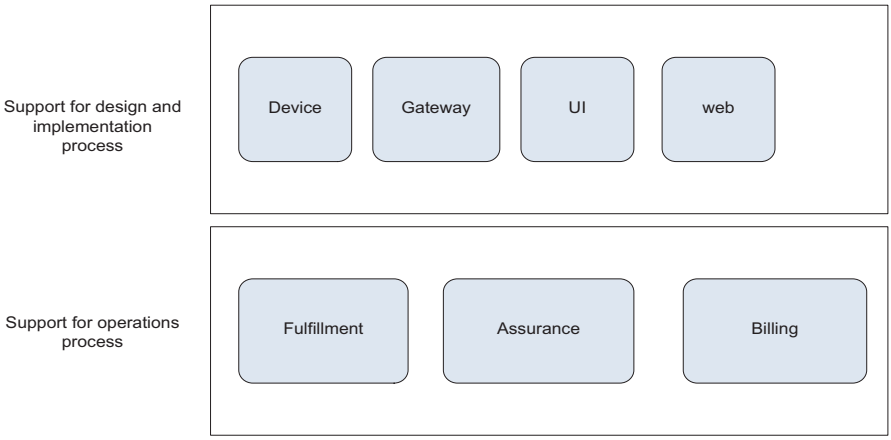


Fig. 7.11 Evaluation framework for IoT application

2. Operations including:

   - Fulfillment: the discovery and purchasing of applications, as well as the delivery, configuration, and activation of application software
   - Assurance: proactive and reactive assurance through monitoring of relevant performance characteristics
   - Billing: the accounting and billing of applications and services

You can find some of the analysis of IoT framework in the (A Framework for Evaluating Internet of Things Platforms: Application Provider Viewpoint reference).

Quality model of IoT Applications, which will be another suggestion to the evaluation, commonly uses the characteristics defined in the quality model of IOT applications. The effectiveness of the quality model for evaluating IOT application through scenario-based is validated.

A quality model for software applications acts as a framework for the evaluation of attributes of the applications that supply the quality model. It is important that every relevant quality characteristic of software applications must be specified and evaluated, whenever possible, using validated metrics. It is necessity to customize a quality model to identify the acceptance criteria and evaluate a particular application domain, IoT applications.

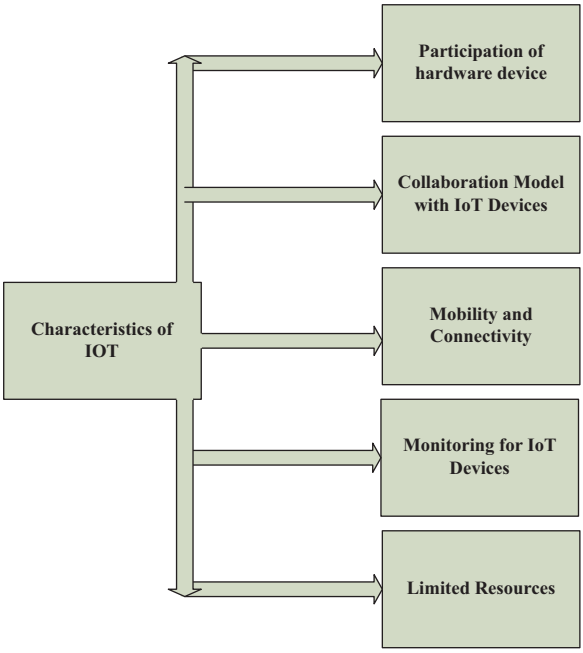
A natural consequence of the trends would be to manage the quality of IoT applications. However, measuring the quality of IoT applications is considerably different from measuring the quality of conventional software systems. This is because the characteristics of IoT devices and their applications are not presented in conventional software systems.

The main characteristics of IoT are:

- *Participation of hardware device*: IoT application consists of two types of elements; software components and hardware devices/components. In the design

and implementation of IoT applications, the presence of hardware devices should be considered.

- *Collaboration model with IoT devices*: The collaborations of IoT application should consider the hardware functionality as well as conventional software collaboration.
- *Mobility and connectivity*: It refers to the capability for device mobility to correct the information processing in different stages of a process.
- Connectivity characteristics refer to the user's quick and efficient connection to information of IoT applications. This characteristic is only used in web applications and mainly in IoT applications.
- *Monitoring for IoT devices*: Remote monitoring for smart devices attached to networks can support multiple functions. Through smart devices, delivered services can provide more real time and precise.
- *Limited resources*: The resource types of IoT devices can be battery, network communication facility, memory, and computation power. IoT devices suffer from some limited resources such as battery lifetime and dominate energy consumption. Energy efficiency can be increased by wisely adjusting transmission power.



These characteristics become the basis for deriving the quality model of IoT applications, that is, defining quality attributes for evaluating IoT applications by considering the impacts of the identified characteristics on the quality of IoT applications, as shown in Fig. 7.12.

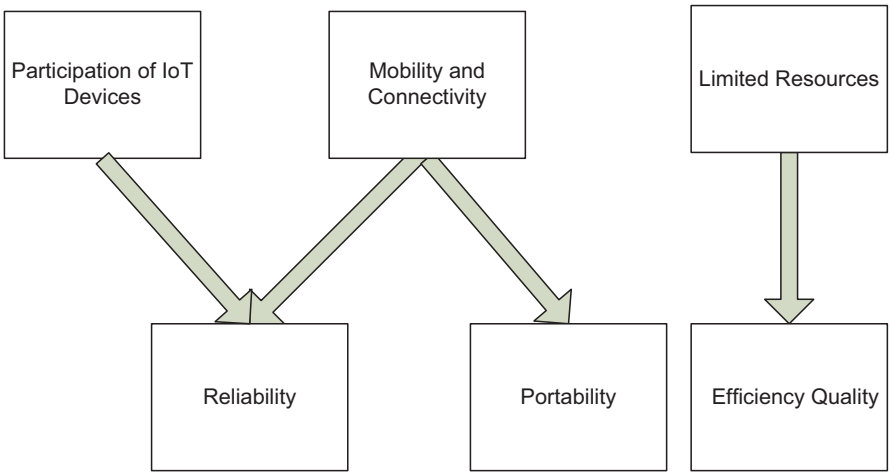


Fig. 7.12 Some IoT characteristics and their relation to quality attribute

Figure 7.12 shows that each characteristic of IoT has a set of associate quality factors, and you must know that each factor has its subfactors, all of that will be measured and the results will be between 0 and 1. Any number near 1 approximately reaches very high quality.

Portability as an example has four main subfactors:

Install ability is the effort for installing the software, *conformance* means the conformance of applications to the standards, *replace ability* is the opportunity to use an application as a replacement for another IoT application, and *sustainability* is the resource types of IoT. In order to calculate metric for conformance, as an example, you have to measure the degree of change to related environments.

***X* = *A*/*B*, where *A* is the number of system response requests which can be customized** and *B* is the supposed user response. The range of *X* is 0–1, and the value 1 is the best which is the response request number for a heterogeneous system; for example, a sensory data through IoT applications is higher for system utilization.

Another example is metric for efficiency, which will measure the appropriate time and resource. Again $X = A/B$ where A is the time of utilization of limited battery and B is the user requirement time and resources. The range will be from 0 to 1, 1 being the best. Using the proposed set of factors and subfactors, the next step is computing the overall quality of IoT application.

In order to apply the quality model (QM) for IoT, all qualities should be computed in this model, and then the summation of all quality factors multiplied by a given weight (where the given weight for the quality attributes is proposed as high for (0.3), medium for (0.2), and low for (0.1) would be represented) will represent the total factor, and of course a value near 1 will be the perfect (high quality).

7.6 Summary

IoT can be defined as a type of network that connects anything with the Internet, and this will be based on protocols through sensing equipments in order to exchange information and communication to achieve smart recognition. The goal of IoT is to connect things anytime, at any place with anything using any path or network and any service.

The fundamental characteristics of the IoT are:

- Things-related services
- Interconnectivity
- Heterogeneity
- Dynamic changes
- Scale
- Safety
- Connectivity

What makes this chapter different from other books on IoT is that it deals with the qualities of IoT: interoperability and modifiability. Security is also an important quality to IoT.

Quality model is the effective way for evaluating IoT application. That will be through characteristics of qualities and subfactors for IoT application for each quality, summing up all the qualities and then giving the total results.

References

K. Patel, S. Patel, Internet of Things-IOT: definition, characteristics, architecture, enabling technologies, and application & future challenges. IJESC (2016). https://doi.org/10.4010/2016.1482

An introduction to the internet of things by Lopez, 2013. http://www.lopezresearch.com/

On line training from SEI

On line training from course area

M. Kim, A quality model for evaluating IoT applications, Department of Computer Science, Soongsil University, Seoul, Korea. IJECE (2016). https://doi.org/10.17706/ijcee.2016.8.1.66-76

Further Reading

Ashton K (2017), 'Making Sense of IoT"

A. Bassi et al., *Evaluating Things to Talk Designing IoT Solutions with IoT Architectural Reference Model* (eBook Springer, Heidelberg New York Dordrecht London, 2013). https://doi.org/10.1007/978-3-642-40403-0. ISBN 978-3-642-40402-3 ISBN 978-3-642-40403-0

J. Tan, S. Koo, Survey of technologies in internet of things, in *IEEE International Conference on Distributed Computing in Sensor Systems*, (2014). https://doi.org/10.1109/DCOSS.2014.45

Chapter 8
Service-Oriented Business Architecture (SOBA)

Abstract Service-oriented architecture (SOA) can be best defined as "services" that provide a platform for different systems to communicate with each other. This chapter explains how SOA can be the basis in the world of business to bring benefits to the organizations that deploy it. The services I talk about are essentially groups of software components that help a company seamlessly carry out important business processes. SOA implementation makes interoperability between heterogeneous applications and technologies possible.

SOA implementation carries several benefits to the organizations:

- Shortened deployment time
- Reduced operating costs
- Reduced risk of failure

All the above (and more) are the reasons for emergence of the concept of Service-Oriented Business Architecture (SOBA). SOBA can be defined as set of steps executed within business processes in the organization, and this will be as services delivered to these processes. These services can then be matched with IT services or others that deliver the necessary functions. The foundation of this concept has been pioneered by IBM to move the position of SOA from IT to the business domain. SOA, as a style of business architecture, adds value to the business architecture by enabling modularity at the business service level, and this will improve manageability. Relation of SOA and quality will be discussed in this chapter.

At the end of this chapter, you will learn:

- What the Service-Oriented Business Architecture means
- The impact of service-oriented architecture on quality attribute and business goals
- The method that is used to evaluate SOBA

L. Khalid, *Software Architecture for Business*,
https://doi.org/10.1007/978-3-030-13632-1_8

8.1 Definition of Service-Oriented Business Architecture (SOBA)

In Chaps. 1 and 2, enterprise and business architecture are defined; you can go through these chapters to review these two terms. EA refers to the connection of design infrastructure with both business organization and the information technology (IT). Organizations' IT infrastructures are becoming increasingly service-oriented.

Before discussing SOBA, an overview definition of SOA is needed. SOA is defined in many different points of view, including:

- "A Service-Oriented Architecture (SOA) is an application framework that takes everyday business applications and breaks them down into individual business functions and processes, called services. An SOA lets you build, deploy and integrate these services independent of applications and the computing platforms on which they run."—*IBM Corporation point of view*
- "Service-Oriented Architecture is an approach to organizing information technology in which data, logic, and infrastructure resources are accessed by routing messages between network interfaces."—*Microsoft view*
- An SOA is "a set of components which can be invoked, and whose interface descriptions can be published and discovered."— *Worldwide Web Consortium [W3C 04] point of view*

From the IT viewpoint, SOA is a component providing technology services which represent an interface that hides the internal implementation technology; but SOA is more than that. If you consider the business point of view, you can see that SOA is a style of designing the business architecture. SOA aims to encapsulate the complexity in business services. The benefits of SOA for business are to provide good business governance with clear accountabilities for business service delivery.

> Business governance is a process, organizational function, set of techniques, and systematic approach for creating and deploying business policies and business rules into day-to-day business activity. (BPM glossary)

SOA solutions have been created to satisfy business goals that include flexible integration with legacy systems, making more effective business processes, reducing costs, and creating values for service for which customers will pay.

Architects of SOA are often in a conflict. On one part, there is business/mission goals and quality attribute requirements that drive the architecture of the system, and on the other part, there is the set of service-oriented principles that influence the architecture and affect its quality attributes. From the intersection of these qualities, the architect needs to make the decisions that are responsible to make a trade-off and achieve the business goals of the system.

Nowadays, information management research is studying how service-orientation can be applied to the business infrastructure of an organization, and that leads to the rise of the concept of Service-Oriented Business Architecture (SOBA). Foundational concepts underlying the idea of Service-Oriented Business Architecture (SOBA) have been founded by IBM's Research Division, moving the service from the IT to the business domain. SOBA explains the individual steps to be executed within the organization's business processes as services delivered to these processes. These business services can then be matched with IT services or others that deliver the necessary functions, and business processes can be reconfigured by changing their mapping onto business services. For many businesses, processes are everything. Processes drive the action of business units and define the parameters of the success of that action. Processes can also help define goals and the steps that a business needs to take toward those goals to reach success. Service-oriented architecture has become a trending topic in the business process management world.

Business architecture focuses on the work of the organization and how the organization does that work. In order to do that and apply SOA, business and business services need to be aligned.

> Business services are operational entities which may have a technical appearance

Three main pillars are needed to align SOA with business technology (Fig. 8.1):

- Strategic pillar: this is about the strategy of the organization, orienting the organization around vision, missions, and goals.
- Operational pillar: this is about the execution of the organization.
- Operational model: this is about focusing on both running and growing the business. The operating model alignment deals with questions such as "How does the organization structure itself?", "How does that structure support sharing?", and "How are business executives, and in effect organizational units, measured for success?" The last question depends on the culture of the organization.

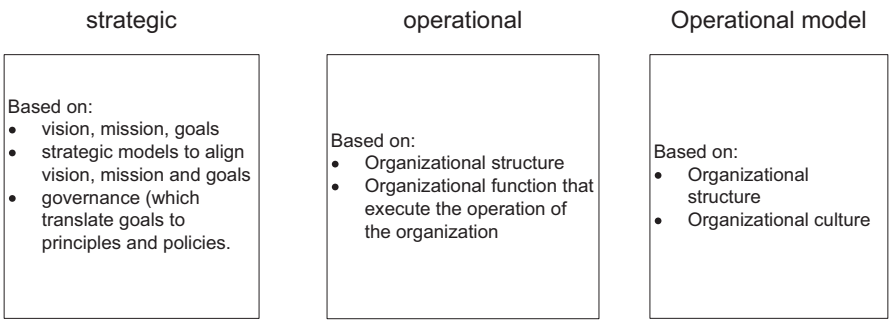


Fig 8.1 Main Pillars of Alignment of Business and SOA

8.2 Basic Qualities in SOBA

Quality attribute requirements drive the software architecture design of the business. That is why designing architecture for such systems has to satisfy both functional and nonfunctional requirements, and this will be the essential part to the success of business. Recently, the use of a SOA as the underlying business architecture has been gaining popularity and well-known use as the architectural approach for various types of business. Services in SOA are the implementation part of business functionality, with a published interface that is discoverable and can be used by service consumers when building different applications and business processes.

Choosing SOA to be an approach for building business architecture depends on several factors; these factors include the architecture's ability to meet functional and quality attribute requirements. Usually, architecture needs to satisfy many quality attribute requirements in order to achieve the business goals of the organization. In almost all cases, trade-offs appear between these requirements. In some cases, satisfying these requirements may be easier using an SOA; in others, it may be more difficult.

In all cases several questions need to be asked if SOA is being considered, for example:

- What is the effect of SOA on the business goal?
- Which quality attribute requirements will impact positively, and which will impact negatively by the use of the SOA?
- What trade-offs need to be made among the quality attributes?

As said previously, quality attribute requirements drive the design of architecture. Usually, interoperability and modifiability qualities are the core of SOA systems, but these qualities are explained in the IoT chapter. Here, two other qualities are taken into consideration through designing business which has SOA in its structure: availability and scalability. Business quality can be defined as a non-software system quality that influences other types of qualities. Business qualities are important qualities because it is taken from the market's point of view. This means that we need to reach some goals like cost and time to market from market's point of view, but that will be affected from the software quality that can be reached by our stakeholders. To conclude, business systems built from software and the quality of this software will affect the goal of the business. This is why you see the same qualities in other software architecture books and why you see variability in SPL systems (you can add other qualities you need in your system according to the type of business you build).

8.2.1 Availability

Availability is the degree to which a system or component is operational and accessible when required for use. In fact, availability is built on reliability by adding the notion of repair.

Availability is closely related to many other qualities. It is related to security; denial of service attack is basically designed to make the system unavailable. Availability is also closely related to performance; it is difficult to know when a system will fail or simply slow to respond. Again availability is related to safety when keeping the system from hazards and recovering the damage when it occurs.

Going back to the availability of services, both from the user's and provider's points of view, you see that this quality is a concern for the success of an SOA. From the service user's perspective, if the system relies on a set of services being available in order to meet its functional requirements and one of those services becomes unavailable, it could influence the system's success. From the service provider's perspective, in order for the services to be used, they must be available when needed. Otherwise, the provider's finances could be impacted. Figure 8.2 represents quality attribute scenario for availability.

Service providers usually agree to provide the service users a set of services and to include each service in a SLA (service-level agreement) . The SLA defines the contract of the terms of the service with details such as who provides the service, the guaranteed availability of the service, the rapid increase process (which is followed if the service is not handled to the service user's satisfaction), and the penalties to the provider if the service level is not met.

When the system no longer delivers a service that is consistent with its specification, it will fail, and this will be observed by the system's actors. A fault has the potential to cause a failure. Recovery or repair is an important aspect of availability. Availability tactics are designed to tolerate the system's fault so that the system still delivers services consistent with its specification.

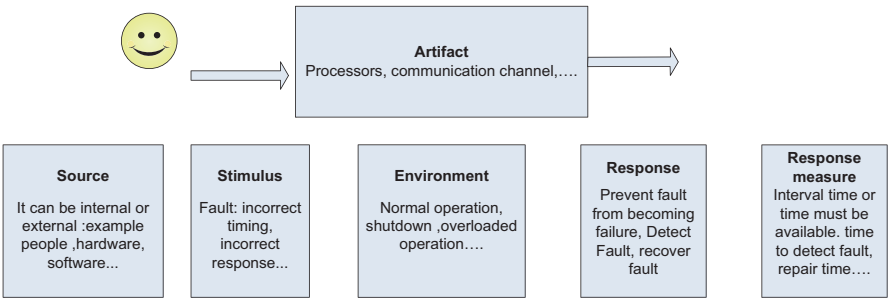


Fig. 8.2 The quality attribute scenario of availability

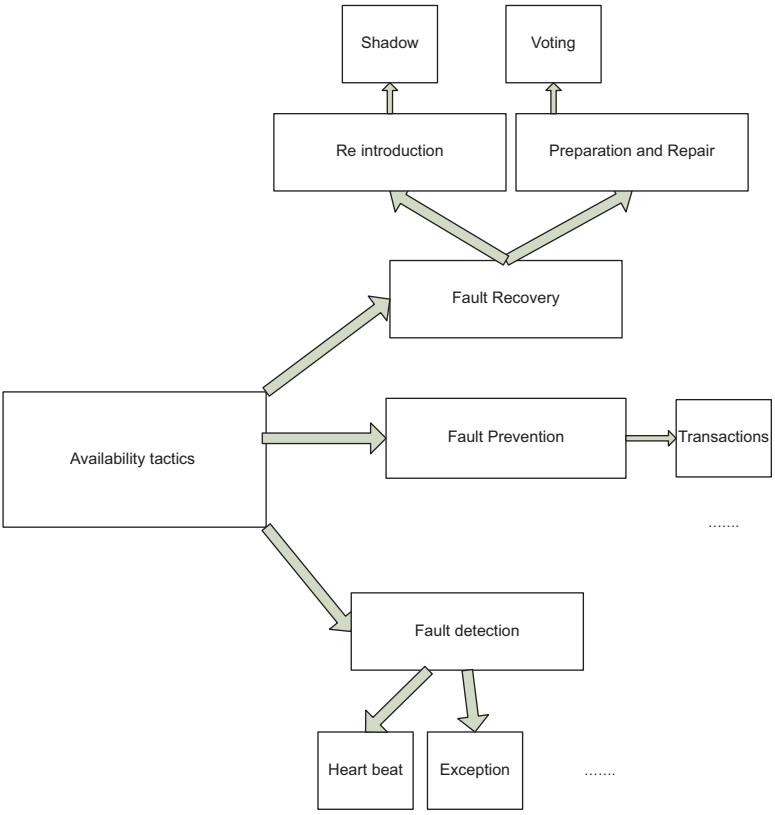


Fig. 8.3 Availability tactics (the square of means that there are other examples on this tactic)

Availability tactics may be categorized as:

- Fault detection
- Fault recovery
- Fault prevention

So your basic job as an architect is to choose and assess the right availability tactics (Fig. 8.3).

Note The main goal of availability tactic is that when the fault appears, it must be repaired or masked.

In conclusion, the impact of SOA on availability is that it is up to the service users to negotiate the SLA that can be used to set an agreed-upon level of availability and include penalties for nonconformity with the agreement. Also, when invoking a service that is not available, here availability could actually be improved as compared with other architectural approaches.

8.2.2 Scalability

Scalability is the ability of SOA to do its purpose when the system is changed in size or in volume in order to meet the users' needs. One of the major issues in scalability is the capacity of the site where the services are located to accommodate an increasing number of service users without affecting the services' performance.

One of the most important aspects of SOA in relation to scalability is the ability for developers to distribute and deploy applications to multiple locations. These locations may be separated physically or logically, or they may be distributed across a platform such as the web. It may also be between different departments in an organization regardless of where or how they are deployed; the concept of SOA allows for flexibility in the design and development of these applications. Services can be used by anyone in any location for whatever needs they may have. This will increase flexibility and the ability to integrate existing scalable services into an application built for heavy loads.

There are two basic kinds of scalability:

- Horizontal scalability: distributing the workload across more computers, for example, adding tier or more service sites. It does not require more expensive hardware, but does require architectural and design decisions to ensure that the application can run perfectly across multiple machines.
- Vertical scalability: upgrading to more powerful hardware for the service site. In the short term, vertical scaling is easier—upgrading machines to handle more is quite easy and allows the developer to expand without reengineering the application. However, this type of scaling does not work in the long term.

Another type of scalability known as scaling by depth describes the ability to add or remove services on the fly. This would allow an application to be constructed dynamically and be customized for individual use cases. Another type of scalability is time scaling utilizing SOA, and just-in-time resource allocation in the cloud allows for runtime creation of applications that only use resources where needed. This on-demand scaling saves costs in many areas.

The performance of the system must be studied, and performance models must capture the degree of the scalability and load tests required. These performance models will need to make some changes based on solution option or a combination of options that is chosen to make sure that the new deployment configuration will meet the intended scalability requirements without affecting the system's performance.

Using service-oriented architecture does not automatically guarantee scalability. Many problems exist between such architecture and a truly scalable application, ranging from simple bottlenecks to bad design decisions, for example, all layers must scale or there is a physical limitation.

In conclusion, the impact of SOA on scalability is that there are ways to deal with increasing the number of service users, and this will need to support more requests for services. However, these solutions need detailed analysis by the service providers to make sure that other quality attributes are not affected negatively.

These two qualities and other sets of qualities are required to build SOBA, trade-off is made between the qualities (and other qualities) through building the business, and all of that will be advantageous to the business. Some other advantages for SOBA are:

- Services (sometimes called functions) will be *reusable*. Since each part of the business will be coded independently, these pieces will be reused in different parts of the business. One of the good examples is a USB cord; it can be used to charge and power many devices.
- Better *productivity* is another advantage for SOBA; reusing the services enhances the efficiency and the productivity of the business.
- *Easy consulting*. You can easily find partners who use SOA who you can consult to find the solution for the business.

8.3 The Impact of Service-Oriented Architecture on Quality Attribute and Business Goals

The important part for the success of any organization is building systems that satisfy its business goals. Software architecture is the bridge between business goals and the final results of the system.

Specific business goals require specific types of quality attributes. For example, the business goals of agility and being first to market require adaptability, interoperability, scalability, and extensibility, and these will be the major drivers for the architecture of that application and development of the services. Agility will require that services can be adapted and reused in new applications, combined in new ways, and scaled to meet the increasing demands for the functionality within the application.

Typical business goals are:

- Agility
- Being first to market with innovative services for its customers
- Streamlining business processes to reduce operating costs
- Enabling easy and flexible integration with legacy systems

Because architects try to satisfy the system's requirements (both functional and nonfunctional) and meet the organization's business goals, they may choose to use an SOA approach in architecting and designing the system.

There are sets of principles for architecting and designing an SOA that impacts agility. These principles include loose coupling and precise specification of services and more such as standardized services, standards compliance, and defining services as coarse-grained.

Reducing the time required to be in the market is done by reusing both internal and external services within the SOA approach, combining services in new and different ways, and adding additional services where needed.

Table 8.1 The impact of SOA approach on some of quality attributes

Quality attribute	Conclusion effect	Grade
Security	The need for authentication, trust, and encryption within an SOA approach needs specified attention within the architecture. Standards to support security are still not finished, and this will impact negatively on security	Zero
Interoperability	SOA provides good interoperability quality, and that will allow services and applications to be built in different languages and to interact through different platforms	2
Performance	In general, SOA has a negative impact on this quality through the delay in network or through requiring a call to a directory of services to locate the desired services. Designing and evaluating the architecture carefully by the user of the service is important to make sure the requirements of the performance are met	Zero
Usability	It is up to service users and providers to build this quality in their system. It may decrease if the service within the application supports human interaction within the system and has some problematic issues in the performance of that service	1
Extensibility	Extending SOA by adding new services or incorporating additional capabilities into existing services is supported in SOA, but you must make sure that the interface is carefully designed to help this quality without any effect on service users	2

Using a service-oriented approach allows organizations to combine existing and newly developed processes, and this combination will meet the new process's needs.

The SOA approach's impact on the quality attributes is shown in Table 8.1. The "grade" column refers to the state level of SOA in each area. Number 2 indicates that there are known solutions for the SOA based on relatively mature standards and technology. Number 1 indicates that some solutions exist but need further research to prove their usefulness in handling the requirements for the quality attribute. Zero indicates that the standards and technology are immature and further significant effort is required to fully support the quality attribute within an SOA.

8.4 Service-Oriented Business Architecture and the Evaluation Method

The first question that arises in mind is: what does it mean that particular software architecture is suitable for its proposed purpose? The answer that comes from SEI is: it is the one that is built based on what is determined by quality attribute requirements. These requirements are important to the stakeholders of the system, and that is the reason the attribute requirements rely on the ATAM method; it depends mainly on eliciting QAS from different groups of system stakeholders.

If software architecture is a key business benefit for an organization, then a key practice for that organization must be architectural analysis. The reason is that such

type of architectures are complex and they hold many design trade-offs, and with a formal analysis process, the organization can make sure that the architectural decisions are made. You must also know that if the architectural design decisions form a system's quality attributes, then it is possible to evaluate these decisions according to their effect on those attributes.

QAS gives exact statements of the usage of the requirements, failure, threats, modification, as well as performance. Once the important quality attributes are identified, the decisions that related to each high priority scenarios are analyzed, and this will be in the form of the following:

- Risks: architectural decisions that might create problems in the future for some quality attribute
- Non-risks: architectural decisions that are suitable in the context of the quality attribute that they affect
- Trade-offs: architectural decisions that have an effect on more than one quality attribute
- Sensitivity points: a property of one or more components and/or component relationships, critical for achieving a particular quality attribute requirement

ATAM (developed by the Carnegie Mellon Software Engineering Institute (SEI)) is the foundation for defining the activities and information that are important for the evaluation of a system that uses an SOA approach. ATAM method consists of the following steps:

1. *Present the ATAM:* The evaluation team presents a quick overview of the ATAM steps, the used techniques, and the outputs from the process.
2. *Present the business drivers:* The system manager presents the business drivers in a brief way and context for the architecture.
3. *Present the architecture:* The architect presents an overview of the architecture.
4. *Identify architectural approaches:* The evaluation team and the architect document the architectural approaches discovered in the previous step.
5. *Generate the quality attribute utility tree:* A small group of technically oriented stakeholders identifies, prioritizes, and refines the most important quality attribute goals in a utility tree format.
6. *Analyze the architectural approaches:* The evaluation team look into the architectural approaches in light of the quality attributes to identify risks, non-risks, and trade-offs.
7. *Brainstorm and prioritize scenarios:* Diverse group of stakeholders creates scenarios that represent their various interests. Then the group votes to prioritize the scenarios based on their relative importance.
8. *Analyze architectural approaches:* The evaluation team continues to identify risks, non-risks, and trade-offs while noting the impact of each scenario on the architectural approaches.
9. *Present results:* The evaluation team reviews the ATAM steps, outputs, and recommendations.

These steps are usually carried out in two phases:

Phase 1 is architect-centric and concentrates on eliciting and analyzing architectural information. This phase will be from steps 1 to 6.

Phase 2 is stakeholder-centric, elicits points of view from a more diverse group of stakeholders, and verifies the results of the first phase. This phase will be from steps 7 to 9.

There is also a phase 0 which consists of planning and preparation. In this phase the evaluation team looks at the existing architecture documentation to identify questions or areas of incompleteness. If the documentation is considered insufficient to express the multiple structures of the architecture, the evaluation does not proceed to phase 1.

Adventure Builder is a simple and brief example on evaluation a system. Basic functionalities of this application are shown in Fig. 8.4.

Table 8.2 shows some quality attribute requirements by using quality attribute scenarios for the Adventure Builder system. It shows sample scenarios that may be related to an SOA-based architecture.

Note OPC (order processing center) acts as a service user when it interacts with airline provider, lodging provider, and activity provider (external services in Fig. 8.5). These interactions are asynchronous, because processing the requests can take a long time and the OPC application should not be blocked waiting for the

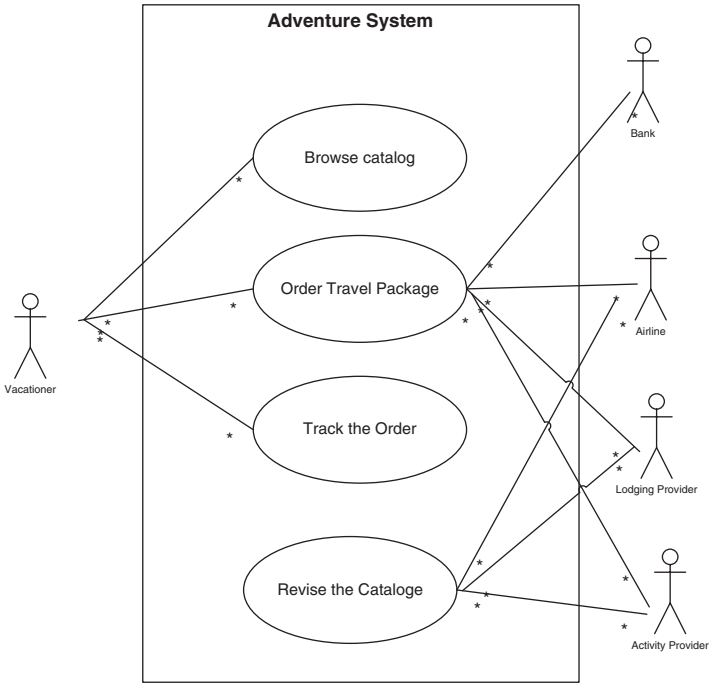


Fig. 8.4 Use case diagram for the Adventure Builder

Table 8.2 Quality attribute scenario for performance and availability quality for the adventure system

Performance	*(Source):* User *(Stimulus):* User submits an order for a package to the consumer web site. *(Artifact):* Adventure Builder system and the bank *(Environment):* Under normal operation *(Response):* The consumer web site informs the user that the order has been successfully submitted and is being processed by the OPC *(Response measure):* The system reacts to the user in less than 5 seconds
Availability	*(Source):* Internal to the system *(Stimulus)*: Fault occurs in the OPC *(Artifact)*: OPC *(Environment)*: Under normal operation *(Response)*: The system administrator is informed of the fault; the system continues taking order requests; another OPC instance is created; and data remains in consistent state *(Response measure)*: The fault is detected, and fail over action is taken within 30 seconds

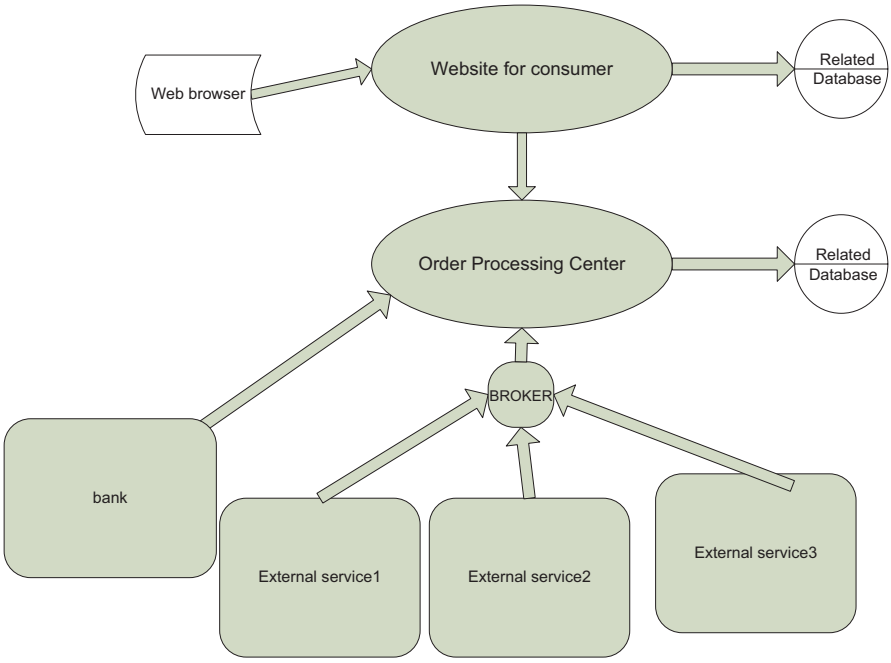


Fig. 8.5 The architecture view of adventure application

results, and that is why broker is needed. Fig. 8.5 shows the role of OPC on the architecture's view of adventure

As defined in previous sections, a service is a distributed component which its details of implementation will be hidden. The distributed nature of a service and the

interaction between a service user and a service provider are noticeable at runtime. Thus, the runtime view best captures a service-oriented design that leads to say an SOA is a style of the component-and-connector view type.

In an ATAM evaluation, the architectural approaches are identified during steps 3 (Present architecture) and 4 (Identify architectural approaches). Hub-and-spoke is the architectural approach of the Adventure Builder application. The use of the "hub" as an active mediator reduces the dependencies between the "spokes" to promote modifiability. Most changes to any single "spoke" should be localized and should only require changes to the "hub," but when you deal with service integration, the evaluation does not cover every important architectural decision. For example, the architectural pattern used in the consumer web site is the Model-View-Controller (MVC) pattern to promote modifiability.

Hub-and-spoke: it is one of traditional architectures. In this type of architecture, there is a single integration server named hub that holds information to be exchanged between many applications or data stores named spokes.

Going through architectural analysis in the ATAM does not mean that it will be accurate and detailed; there is also no numerical value for different qualities. The key is to elicit enough architectural information to identify risks, which results from the correlation between the architectural decisions and their effect on quality attributes. The evaluation is done to capture the architectural approaches and identify their risks, non-risks, sensitivities, and trade-offs. For example, the architectural analysis of performance quality is as shown:

Scenario abstract	User submits an order for a package to the consumer web site. The system responds to the user in a specific time
Business goals	Provides satisfactory user skills
The quality attribute	Performance
Architectural approaches	The web services were designed around the documents handled, such as purchase orders and invoices. The OPC purchase order service interface improves the performance. This interface reduces the overhead of calling a fine-grained service for each step of a business process The OPC interacts with the bank in a synchronous way. After authorizing the charge quickly, the OPC sends requests to transportation, lodging, and activity providers, which will later respond through the web service broker callback endpoint. These requests are sent asynchronously to improve scalability and throughput and also because of the nature of the legacy systems supporting this interface
Risk	The design does not meet the requirement in this scenario, for the reason that it assumes that all external transportation providers implement the same web service interface

Trade-off	The homogenous action of all transportation providers in OPC increases modifiability. On the other hand, intermediaries are needed to cooperate with external providers that offer heterogeneous service interfaces, as in this scenario These intermediaries represent a performance overhead, because they may require wide processing of overhead and routing messages

8.5 Summary

Service-oriented architecture is defined from different points of view. The IT point of view of SOA is a component providing technology services which represents an interface that hides the internal implementation technology. If you take the business point of view, you can see that SOA is a style of designing the business architecture. SOA aims to encapsulate the complexity in business services. SOA benefits for business are providing good business governance with clear accountabilities for business service delivery. Nowadays, information management research is studying how service-orientation can be applied to the business infrastructure of an organization, and that leads to the rise of the concept of Service-Oriented Business Architecture (SOBA). Foundational concepts underlying the idea of Service-Oriented Business Architecture (SOBA) have been founded by IBM's Research Division move service from the IT thinking to the business domain. SOBA explains the individual steps to be executed within the organization's business processes as services delivered to these processes. These business services can then be matched with IT services or others that deliver the necessary functions, and business processes can be reconfigured by changing their mapping onto business services.

Choosing SOA as an approach in the development of architecture depends on several factors including the architecture's ability to meet functional and quality attribute requirements. Usually, the architecture needs to satisfy many quality attribute requirements in order to achieve the business goals of the organization. In almost all cases, trade-offs have to be made between these requirements. Usually, interoperability and modifiability are the cores of SOA systems.

If software architecture is a key business benefit for an organization, then a key practice for that organization must be architectural analysis. This is because these architectures are complex and hold many design trade-offs. And with a formal analysis process, the organization can make sure that the architectural decisions are made. That is why ATAM is the appropriate method to SOBA.

References

L. Bass, *Software architecture in practice* (Addision Wisely, 2013)

P. Bianco, R. Kotermanski, P. Merson, *Evaluating a Service-Oriented Architecture* (2007). Technical Report. CMU/SEI-

Capagmenin Consulting Technology and Outsourcing, Service-oriented enterprise: how to make your business fast, flexible and responsive. http://www.cio.co.uk

E. Lemey Supervised by: Geert Poels, *Investigating service-oriented business architecture, 7221* (Springer-Verlag, Berlin, Heidelberg, 2012), pp. 220–225
L. O'Brien, L. Bass, P. Merson, Quality attributes and service-oriented architectures, 2005, Technical note from CMU- SEI
The Open Group. SOA for Business Technology – SOA and Business Architecture

Further Reading

A. Grigoriu, SOA, BPM, EA, and service oriented enterprise architecture, BPTrends 2007
I. Sommerville, Software engineering, 9th edn (Addison Wesley, 2010) ISBN:013703515297801370351 51

Conclusion Thoughts

Now at last I want to conclude the main thoughts that I want you to know through beginning your future in software architecture especially if you are just beginning your study in this field or just beginning your first day in your job as an architect:

- First: *qualities* are the basic part for any individual, company, and organization that need to start building their products, because this will make the competition between products or even between the people whose responsibility is building the products in the market. It is the main target to building any system.
- Second: knowing *the patterns and tactics*. As beginning with the first thought, the quality is the basic thing that all must focus on, and the second thought that you must know is that gaining the quality can be done through other stages in the life cycle. Here the focus is on software architecture stage because the title of this book is going around this stage. And because I talked especially on software architecture and its relation to building high-quality products, then you need to know the architectural patterns and their tactics to use the most appropriate of them to have the high-quality products and also make a trade-off between qualities in the same product.
- Third: *stakeholders*. Through years ago I read many books and article papers and attend conferences on software architecture; through that I saw the important advice from professionals; and this advice says: know your stakeholders. Knowing the stakeholder is a very important part to any architect because you as an architect can extract the goals that they want from the product that you can go through this road to reach the quality.
- Fourth: *evaluation.* It *is* very important to any architecture because it avoid architectures from disaster. To put it in a different way, if you were building a house, you wouldn't proceed without carefully looking at the blueprints before building began. It is also the right thing to know the appropriate method according to the quality of the product as what said through this book but in general ATAM method is used as an evaluation method. Also you must know that the evaluation will tell you that the architecture is suitable according with one goal (or set of

L. Khalid, *Software Architecture for Business*,
https://doi.org/10.1007/978-3-030-13632-1

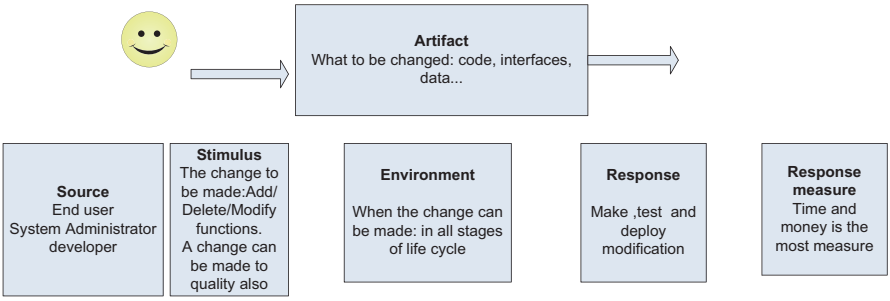
goals) and according to other qualities there is a problem, sometimes one goal is important than another sometimes a conflict occurs between qualities, the manager of the project will have the decision to make if the architecture evaluates good in some areas and not evaluated very well in other area.

- Also you showed that SAAM method focuses on modifiability quality in its different forms (such as portability, subsetability, and variability) and functionality, while ARID method provides a deep understanding about the suitability of part of the architecture to be used by developer to complete their responsibilities. That is why architecture evaluation methods can be choosing according to the qualities that are related to that architecture.
- In short, architecture evaluation produces better architectures.

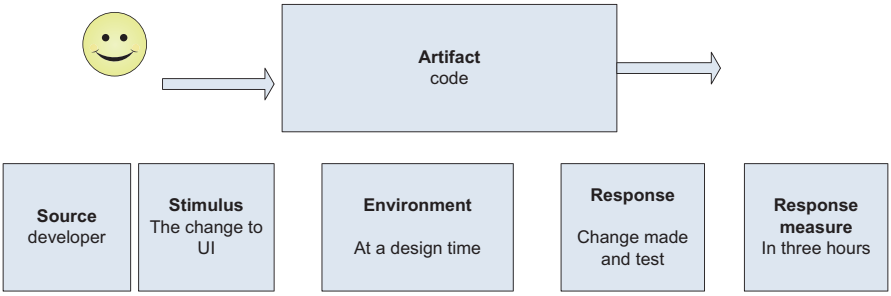
Finally I hope you have the basic stone to start your journey in software architecture and its relation to building high-quality products.

Appendix A

General Scenario for Modifiability

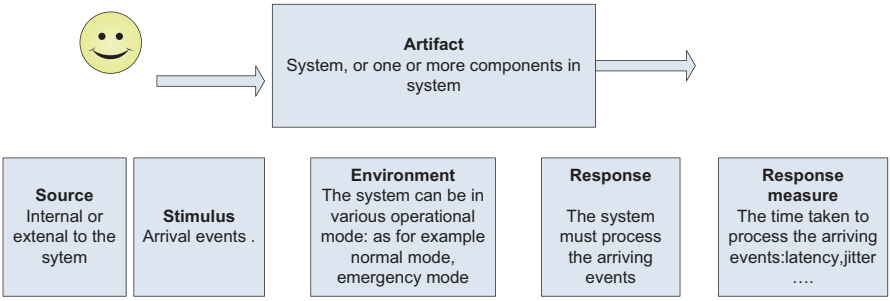


Concrete Scenario for Modifiability

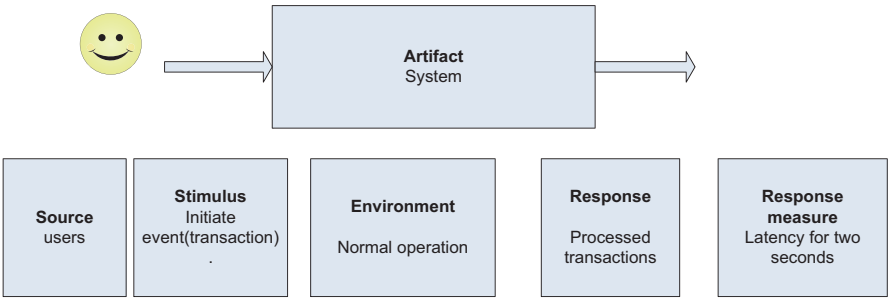


L. Khalid, *Software Architecture for Business*,
https://doi.org/10.1007/978-3-030-13632-1

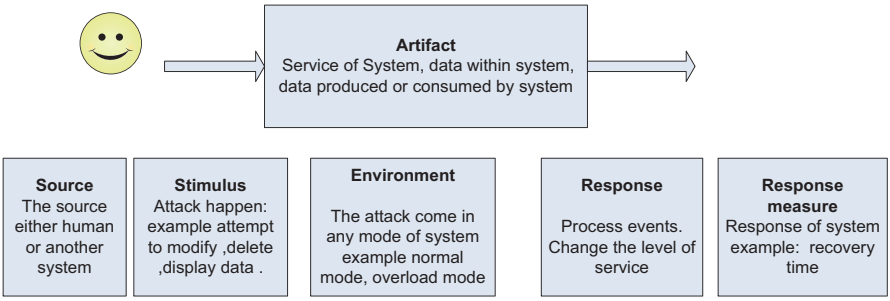
Role of General Scenario for Performance



Concrete Scenario for Performance



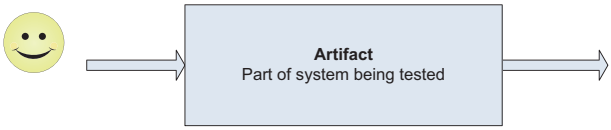
General Scenario for Security



Concrete Scenario for Security

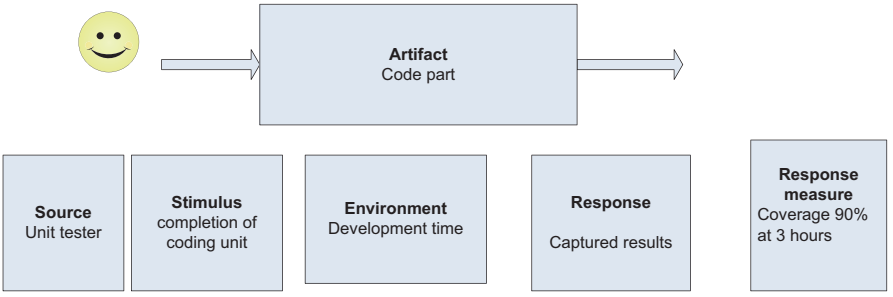
Source	Stimulus	Environment	Response	Response measure
Distinguished employee from remote place	Attempts to modify pay rate .	Normal operation	System maintains audit trail	Resore correct data within a day

General Scenario for Testability

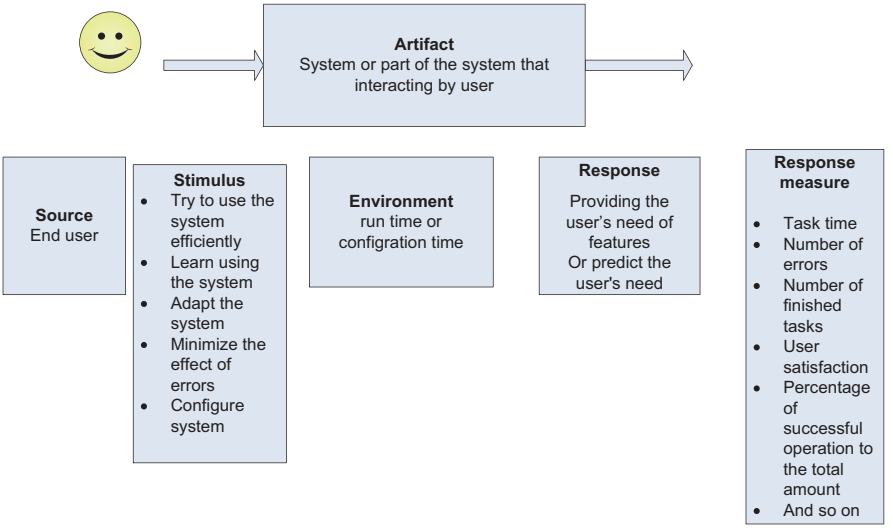


Source	Stimulus	Environment	Response	Response measure
Could be human or automated tester	Set of tests through completion of coding increment example: service	Happen at: development time ,compile time, deployment time ,run time, design time, integration time	Controlled the system to perform test and the result can be observed	• Effort to find a fault. • effort o achieve a coverage %, • Probability of fault being revealed by next stage, • Time to carry out tests, • Reduction in risks • Time to prepare test environment • Effort to detect fault And so on

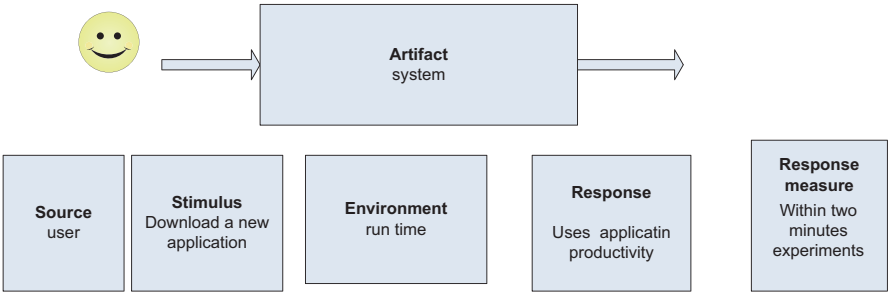
Concrete Scenario for Testability



General Scenario for Usability



Concrete Scenario for Usability



Appendix B

	Software structure	Element types	Relation	Useful for
Module structure	Decomposition	Module	Is a sub module of	Resource allocation and project structuring and planning; information hiding, encapsulation; configuration control
	Uses	Module	Uses	Engineering subsets; engineering extensions
	Layered	Layer	Requires the correct presence of; uses the services of; provides abstraction to	Incremental development; implementing systems on top of "virtual machines"
	Class	Classes, objects	Is an instance of; shares access methods of	In object-oriented design systems
C&C structure	Service	Service, registry, others	Run concurrently with, etc.	Scheduling analysis, performance analysis
	Concurrency	Process, thread	Can run in parallel	Identifying locations where resource contention exists, where threads may fork, join, be created or be killed
Allocation structure	Deployment	Components, hardware elements	Allocated to; migrates to	Performance, availability, security analysis
	Implementation	Modules, file structure	Stored in	Configuration control, integration, test activities
	Work assignment	Modules, organizational unit	Assigned to	Project management, best use of expertise, management of commonality

Architectural structure for the system

L. Khalid, *Software Architecture for Business*,
https://doi.org/10.1007/978-3-030-13632-1

Appendix C

ATAM (Architecture Trade-off Analysis Method)

First of all, ATAM gets its name because it reveals on how well an architecture satisfies the quality goals, and the most important thing is how trade-off between qualities has been used for over a decade to evaluate software architectures and draws its technique from three areas:

- The architectural style concepts
- The quality attributes analysis communities
- SAAM method the predecessor method to the ATAM

ATAM steps can be separated into four main groups.
Presentation: exchange the information through the presentation which includes:

1. Present the ATAM: evaluation leader introduce the method to the participate.
2. Present the business driver: the project manager or customer describes the business goals that motivated the development and then what is the main architectural driver (such as time to market).
3. Present the architecture: here, the role of architect is to describe the architecture and focus his attention on how it addresses the business driver.

Investigation and analysis

4. Identify the architectural approaches: the approaches of architecture are identified by architect but not analyzed.
5. Generate the utility tree: the quality attribute of system is elicited and specified down to the level of scenarios, stimulus, and response and prioritized.
6. Analyze the architectural approach: based on the high priority in previous step, the architectural approaches that address the scenario will be analyzed. Here, risk, sensitivity point, and non-risk trade-off will be identified in this step.

L. Khalid, *Software Architecture for Business*,
https://doi.org/10.1007/978-3-030-13632-1

Testing

7. Brainstorming and prioritize scenario: the set of scenario that elicited earlier will be prioritized through voting process involving all the stakeholders.
8. Analyze the architectural approaches: in this step, reiteration of the activities in step 6 occurs, but high-ranked scenarios from step 7 are used. Those scenarios are supposed to be the test cases to confirm the analysis performed thus far.

Reporting

9. Present the result based on the information collected during the steps, and then the results will be presented to the stakeholders.

These steps can be carried out through the following phases:

Phase	Activity
0	Preparation
1	Evaluation from steps 1 to 6
2	Evaluation from steps 7 to 9
3	Process improvement and delivery

According to quality, ATAM is not oriented to any specific type of quality

SAAM (the Software Architecture Analysis Method)

It is a simple method and good place to start if this is the first time you evaluate and architecture, *especially if you work on modifiability and functionality*

The output tangible results from SAAM evaluation are:

- Mapping onto scenarios that represent possible future changes to the system
- Understanding the functionality of the system

SAAM steps are:

1. *Develop scenario*: these scenarios represent tasks related to different roles of stakeholders; these scenarios are all brainstorming exercise, and also they are collected in two or more elicitations.
2. *Describe architecture*: describing the architectures should be through notations that will be understand by parties and must specify data components with related connections and system's computation, and all these will take in the form of natural language or some other formal specification.
3. *Classify/prioritize scenarios*: in SAAM, scenarios are classified into direct and indirect scenarios. Prioritization is done by choosing the most important scenarios and that is done by stakeholders through voting process.
4. *Individually evaluate indirect scenario*: chosen scenarios are mapped onto the architectural description. According to direct scenarios, the architect shows how

these scenarios are executed by architecture. On the other hand, in indirect scenarios, the architect should describe how the architecture needs to be changed in order to accommodate the scenarios.

5. *Assess scenario interaction*: interaction scenarios are called to the indirect scenarios that require changes to a single component of architecture. These types of scenarios are important for two reasons: first reason is it depicts allocation of functionality to the product's design. Second important reason is that it can expose the architecture not documented to the right level of structural decomposition.
6. *Create overall evaluation*: at the final stage, a weight is assigned to each scenario for the reason of showing the related importance to the success of system. This weight usually attaches to the business goals that support the scenarios.

Notes

- Step 1 and 2 done on interleaved way or on several iterations.
- There are two important concepts in these iterations, and those are direct and indirect scenarios. Scenarios represent tasks relevant to different roles such as developer, maintainer, customer, etc.

Direct scenarios are those scenarios that are specified by the architecture through the execution of the system. Such types of scenarios are increasing the understanding of the architecture to the stakeholders and allow the systematic exploration of their architectural qualities. On the other hand, there is *an indirect scenario* which defines as that requires a modification to the architecture to be satisfied. It is those scenarios that play as a central role to the measurements of the degree to which an architecture can hold evolutionary changes that are important to the stakeholder. The indirect scenarios measure the suitability for continuing use throughout the lifetime of the family.

ARID (Active Reviewers for the Intermediate Design)

It is a method that used to evaluate the architecture partially or in the intermediate designs when all the architecture passes through. This method lies at the intersection between ATAM and ADRs (active design reviewers) methods. It consists of nine steps going through two phases.

Phase 1: Rehearsal

In this phase, a meeting between the lead designer and review facilitator to prepare for exercise.

Step 1: identify the reviewers—the reviewers of the ARID are the design's stakeholders.

Step2: prepare the design briefing—the designers organize a brief explanation of the design. The goal is to present the design in a good so that members can use it in a sufficient way.

Step3: prepare the seed scenarios—here, designer and the review facilitator present a set of seed scenarios; the aim is roughly a dozen scenarios.

Step 4: prepare the materials—this step is the preparation to phase 2 by preparing copies of the presentation, scenarios, and review agenda to the reviewers during phase 2.

Phase 2: Review

Step 5: present ARID—the explanation of the steps of ARID to the participants is done by review facilitator.

Step 6: present the design—the suitability of the design is the goal of this step. The lead designer gives an overview presentation with examples.

Step 7: brainstorm and prioritize scenarios—this session is just for brainstorming and prioritizes scenarios. Voting process is done through process, and the most voted scenarios received are used to test the design for testability.

Step 8: apply the scenarios—when received the most voted scenarios, the facilitator ask reviewers to expertise the code that uses the design services to solve the problem posed in the scenario whenever the group went to the wrong direction they will be stopped to get the group moving again by providing whatever information is supposed it will be necessary.

Step 9: summarize—recounting the list of issues is done by facilitator; ask the participants for their opinions and thank them for their participations.

Note

ARD method relies on actively engaging reviewers by assigning them review tasks that are structured; it is used to evaluate detailed design of coherent units of software for example modules or components.

According to quality, ARID is used for suitability of the design approach.

Index

L. Khalid, *Software Architecture for Business*,
https://doi.org/10.1007/978-3-030-13632-1