1 Understanding Software Architecture

1.1 What is Software Architecture?

The last decade has seen a tremendous rise in the prominence of a software engineering sub-discipline known as software architecture. Technical Architects and Chief Architects are job titles that now abound in the software industry. There's a Worldwide Institute of Software Architects\(^1\), and even a certain well-known wealthiest person on earth has architect in his job title. It can't be a bad gig, then?

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

This book is about software architecture. Its aim is to concisely describe the essential elements of knowledge and key skills that are required to be a software architect in the software and information technology (IT) industry. Conciseness is a key objective. For this reason, by no means everything an architect needs to know will be covered. If you want or need to know more, each chapter will point you to additional worthy and useful resources that can lead to far greater illumination.

So, without further ado, let's try and figure out what, at least I think, software architecture really is. The remainder of this chapter will address this question, as well as briefly introducing the major tasks of an architect, and the relationship between architecture and technology in IT applications.

\(^1\) http://www.wswa.org/
1.2 Definitions of Software Architecture

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

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

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

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

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

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

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


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

http://www.sei.cmu.edu/architecture/definitions.html

“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.”


It’s interesting to look at these, as there is much commonality. I include the third mainly as it’s again explicit about certain issues, such as scalability and distribution, which are implicit in the first two. Regardless, analyzing these a little makes it possible to draw out some of the fundamental characteristics of software architectures. These, along with some key approaches, are described below.

1.2.1 Architecture Defines Structure

Much of an architect’s time is concerned with how to sensibly partition an application into a set of inter-related components, modules, objects or whatever unit of software partitioning works for you 5. Different application requirements and constraints will define the precise meaning of “sensibly” in the previous sentence – an architecture must be designed to meet the specific requirements and constraints of the application it is intended for.

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

In partitioning an application, the architect assigns responsibilities to each constituent component. These responsibilities define the tasks a component can be relied upon to perform within the application. In this man-

5 Component here and in the remainder of this book is used very loosely to mean a recognizable “chunk” of software, and not in the sense of the more strict definition in C. K. C. (1998) Component Software: Beyond Object-Oriented Programming, Addison-Wesley
ner, each component plays a specific role in the application, and the overall component ensemble that comprises the architecture collaborates to provide the required functionality.

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

![Diagram of component dependencies](image)

**Fig. 1. Two examples of component dependencies**

A key structural issue for nearly all applications is minimizing dependencies between components, creating a loosely coupled architecture from a set of highly cohesive components. A dependency exists between components when a change in one potentially forces a change in others. By eliminating unnecessary dependencies, changes are localized and do not propagate throughout an architecture (see Fig. 1).

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

### 1.2.2 Architecture Specifies Component Communication

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

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

Each pattern has well-known characteristics that make it appropriate to use to satisfy particular types of requirements. For example, the client-server pattern has several useful characteristics, such as synchronous request-reply communications from client to server, and servers supporting one or more clients through a published interface. Optionally, clients may establish sessions with servers, which may maintain state about their connected clients. Client-server architectures must also provide a mechanism for clients to locate servers, handle errors, and optionally provide security on server access. All these issues are addressed in the client-server architecture pattern.

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

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

---

*Patterns and styles are essentially the same thing, but as a leading software architecture author told me recently, "the patterns people won't This book will therefore use patterns instead of styles!"*
1.2.3 Architecture Addresses Non-functional Requirements

Non-functional requirements are the ones that don’t appear in use cases. Rather than define what the application does, they are concerned with how the application provides the required functionality.

There are three distinct areas of non-functional requirements:

- **Technical constraints**: These will be familiar to everyone. They constrain design options by specifying certain technologies that the application must use. “We only have Java developers, so we must develop in Java.” “The existing database runs on Windows XP only.” These are usually non-negotiable.

- **Business constraints**: These too constraint design options, but for business, not technical reasons. For example, “In order to widen our potential customer base, we must interface with XYZ tools.” Another example is “The supplier of our middleware has raised prices prohibitively, so we’re moving to an open source version.” Most of the time, these too are non-negotiable.

- **Quality attributes**: These define an application’s requirements in terms of scalability, availability, ease of change, portability, usability, performance, and so on. Quality attributes address issues of concern to application users, as well as other stakeholders like the project team itself or the project sponsor. Chapter 3 discusses quality attributes in some detail.

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

1.2.4 Architecture is an Abstraction

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

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

One of the most powerful mechanisms for describing an architecture is hierarchical decomposition. Components that appear in one level of description are decomposed in more detail in accompanying design documentation. As an example, Fig. 2 depicts a very simple two level hierarchy using an informal notation, with two of the components in the top-level diagram decomposed further.

Different levels of description in the hierarchy tend to be of interest to different developers in a project. In Fig. 2 it’s likely that the three components in the top level description will be designed and built by different teams working on the application. The architecture clearly partitions the responsibilities of each team, defining the dependencies between them.

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

The simple architecture in Fig. 2 doesn’t decompose the Client component. This is, again presumably, because the internal structure and behavior of the client is not significant in achieving the application’s overall non-functional requirements. How the Client gets the information that is sent to the Broker is not an issue that concerns the architect, and consequently the detailed design is left open to the component’s development team. Of course, the Client component could possibly be the most complex in the application. It might have an internal architecture defined by its design team, which meets specific quality goals for the Client component. These are, however, localized concerns. It’s not necessary for the architect to complicate the application architecture with such issues, as they can be safely left to the Client design team to resolve.
1.2.5 Architecture Views

A software architecture represents a complex design artifact. Not surprisingly then, like most complex artifacts, there are a number of ways of looking at and understanding an architecture. The term “architecture views” rose to prominence in Philippe Kruchten’s 1995 paper on the 4+1 View Model. This presented a way of describing and understanding an architecture based on the following four views:

- **Logical view**: This describes the architecturally significant elements of the architecture and the relationships between them. The logical view essentially captures the structure of the application using class diagrams or equivalents.

- **Process view**: This focuses on describing the concurrency and communications elements of an architecture. In IT applications, the main concerns are describing multi-threaded or replicated components, and the synchronous or asynchronous communication mechanisms used.

- **Physical view**: This depicts how the major processes and components are mapped to the application’s hardware. It might show, for example, how the database and web servers for an application are distributed across a number of server machines.

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

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

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

- **Module**: This is a structural view of the architecture, comprising the code modules such as classes, packages and subsystems in the design. It also captures module decomposition, inheritance, associations and aggregations.

- **Component and Connector**: This view describes the behavioral aspects of the architecture. Components are typically objects, threads or processes, and the connectors describe how the components interact. Common connectors are sockets, middleware like CORBA or shared memory.

- **Allocation**: This view shows how the processes in the architecture are mapped to hardware, and how they communicate using networks and/or databases. It also captures a view of the source code in the configuration management systems, and who in the development group has responsibility for each module.

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

1.3 What Does a Software Architect Do?

The environment that a software architect works in tends to define their exact roles and responsibilities. A good general description of the architect's role is maintained by the SEI on their web site. Instead of summarizing this, I'll briefly describe, in no particular order, four essential skills for a software architect, regardless of their professional environment.

- **Liaison**: Architects play many liaison roles. They liaise between the customers or clients of the application and the technical team, often in conjunction with the business and requirements analysts. They liaise between the various engineering teams on a project, as the architecture is central to each of these. They liaise with management, justifying designs, decisions and costs. They liaise with the sales force, to help promote a system to potential purchasers or investors. Much of the time, this liaison takes the form of simply translating and explaining different terminology between different stakeholders.

- **Software Engineering**: Excellent design skills are what get a software engineer to the position of architect. They are an essential pre-requisite for the role. More broadly though, architects must promote good software engineering practices. Their designs must be adequately documented and communicated and their plans must be explicit and justified. They must understand the downstream impact of their decisions, working appropriately with the application testing, documentation and release teams.

- **Technology Knowledge**: Architects have a deep understanding of the technology domains that are relevant to the types of applications they work on. They are influential in evaluating and choosing third party components and technologies. They track technology developments, and understand how new standards, features and products might be usefully exploited in their projects. Just as importantly, good architects know what they don't know.

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

- **Risk Management**: Good architects tend to be cautious. They are constantly enumerating and evaluating the risks associated with the design and technology choices they make. They document and manage these risks in conjunction with project sponsors and management. They develop and instigate risk mitigation strategies, and communicate these to the relevant engineering teams. They try to make sure no unexpected disasters occur.

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

1.4 Architectures and Technologies

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

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

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

Unfortunately, abstract descriptions of architectures don’t yet execute on computers, either directly or through rigorous transformation. Until they do, abstract architectures must be reified by software engineers as concrete software implementations.

Fortunately, software products vendors have come to the rescue. Widely utilized architectural patterns are supported in a variety of commercial off-the-shelf (COTS) technologies. If a design calls for publish-subscribe mes-
saging, or a message broker, or a three-tier architecture, then the choices of available technology are many and varied indeed. This is an example of software technologies providing reusable, application-independent software infrastructures that implement proven architectural approaches.

**Abstract**

![Diagram of Architectural Patterns/Styles](image)

**Concrete COTS technologies**

![Diagram of Concrete COTS technologies](image)

Fig. 3. Mapping between logical architectural patterns and concrete technologies.

As Fig. 3 depicts, several classes of COTS technologies are used in practice to provide packaged implementations of architectural patterns for use in IT systems. Within each class, competing commercial and open source products exist. Although these products are superficially similar, they will have differing feature sets, be implemented differently and have varying constraints on their use.

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

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

Chapter 4 provides a detailed description and analysis of these infrastructural technologies.

### 1.5 Summary

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

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

The remainder of this book will describe methods and techniques that can help you to shed at least some light on architectural design decisions. Much of this light comes from understanding and leveraging design principles and supporting technologies that have proven to work in the past. Armed with this knowledge, you’ll be able to tackle complex architecture problems with more confidence, and after a while, perhaps even a little panache.

### 1.6 Further Reading

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

#### 1.6.1 General Architecture

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

1.6.2 Architecture Requirements

The original book describing use-cases is:


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


1.6.3 Architecture Patterns

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


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


2 Introducing the Case Study

2.1 Requirements Overview

This chapter introduces a case study that will be used in subsequent chapters to illustrate some of the design principles in this book. The Information Capture and Dissemination Environment (ICDE) is part of a suite of software systems for providing intelligent assistance to professionals such as financial analysts, scientific researchers and intelligence analysts. To this end, ICDE automatically captures and stores data that records a range of actions performed by a user when operating a workstation. For example, when a user performs a Google search, the ICDE system will transparently store in a database:

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

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

A use case diagram for the ICDE system is shown in Fig. 4. The three major use cases incorporate the capture of user actions, the querying of

---

7 The case study project is based on an actual system that is operational at the time of writing. Some creative license has been exploited to simplify the functional requirements, so that these don't overwhelm the reader with unnecessary detail. Also, the events, technical details and context described do not always conform to reality, as reality can be far too messy for illustration purposes.
data from the data store, and the interaction of the third party tools with the user.

![ICDE System Use Cases](image)

**ICDE System Use Cases**

![ICDE Version 1.0 Application Architecture](image)

**ICDE Version 1.0 Application Architecture**

### 2.2 Project Context

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

An initial production version (v1.0) of ICDE was implemented by a small development team. Their main aim was to implement the Capture User Actions use case. This allowed several low-level technical issues to be solved, and forced the design and implementation of the data store to be carried out. This was important as the data store was an integral part of the rest of the system's functionality, and its design had to be suitable to support a fairly high transaction rate.

ICDE v1.0 was only deployed in a small user trial involving a few users. This deployment successfully tested the software functionality and demonstrated the concepts of data capture and storage. The design of v1.0 was based upon a simple 2-tier architecture, with all components executing on the user's workstation. This design is shown as a UML v1.0 X component diagram in Fig. 5. The collection and analysis components were written in Java and access the data store directly using the JDBC API. The complete ICDE application executed on Microsoft Windows XP.

![Data Collection](image)

**Data Collection**

The role of each component is as follows:

- **Data Collection**: The collection component comprises a number of loosely coupled processes that transparently track the user's relevant activities and store them in the Data Store. The captured events relate to Internet accesses, documents that are opened and browsed, edits made to documents, and some basic windowing information about when the user opens and closes applications on the desktop.

- **Data Store**: This component comprises a commercial-off-the-shelf (COTS) relational database. The relational database stores information in various tables regarding the user activities, with timestamps added so that the order of events can be reconstructed. Large objects such as images on web pages and binary documents are stored as Binary Large Object Fields (BLOBs) using the native database facilities.

---

1. Java Database Connectivity
• **Data Analysis**: A graphical user interface (GUI) based tool supports a set of queries on the data store. This was useful for testing purposes, and to give the third party tool creators an initial look at the data that was being captured, and was hence available to them for analysis.

### 2.3 Business Goals

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

- Encourage third party tool developers to write applications for the ICDE system.
- Promote the ICDE concept and tools to potential customers, in order to enhance their analytical working environment.

Clearly, both these objectives are focused on fostering a growing business around the ICDE technology, by creating an attractive market for third party tools and an advanced advisory environment for users. Achieving these goals requires detailed technical and business plans to be drawn up and followed through. From a purely technical perspective, leaving out such activities as sales and marketing, the following major objectives were identified – see Table 1:

<table>
<thead>
<tr>
<th>Business Goal</th>
<th>Supporting Technical Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encourage third party tool developers</td>
<td>Simple and reliable programmatic access to data store for third party tools</td>
</tr>
<tr>
<td></td>
<td>Heterogeneous (i.e. non-Windows) platform support for running third party tools</td>
</tr>
<tr>
<td></td>
<td>Allow third party tools to communicate with ICDE users from a remote machine</td>
</tr>
<tr>
<td>Promote the ICDE concept to users</td>
<td>Scale the data collection and data store components to support up to 150 users at a single site</td>
</tr>
<tr>
<td></td>
<td>Low-cost deployment for each ICDE user workstation</td>
</tr>
</tbody>
</table>

Table 1. ICDE v2.0 business goals

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

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

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

### 2.4 Constraints

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

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

### 2.5 Summary

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

In the next chapter, we will explore from a general perspective the spectrum of architectural requirements that arise in projects like ICDE. These requirements are fundamental in driving the design of application architec-
3 Software Quality Attributes

3.1 Quality Attributes

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

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

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

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

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

"It must be possible to scale the deployment from an initial 100 geographically dispersed user desktops to 10,000 without an increase in effort/cost for installation and configuration."
This is precise and meaningful. As an architect, this points me down a path to a set of solutions and concrete technologies that facilitate zero-effort installation and deployment.

Note however, that many quality attributes are actually somewhat difficult to validate and test. In this example, it'd be unlikely that in testing for the initial release, a test case would install and configure the application on 10,000 desktops. I just can't see a project manager signing off on that test somehow.

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

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

3.2 Performance

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

A performance quality requirement defines a metric that states the amount of work an application must perform in a given time, and/or deadlines that must be met for correct operation. Few IT applications have hard real-time constraints like those found in military or robotics systems,

³ Ward Cunningham at his finest!

where if some output is produced a millisecond or three too late, really nasty and undesirable things can happen (I'll let the reader use their imagination here). But applications needing to process hundreds, sometimes thousands and tens of thousands of transactions every second are found in many large organizations, especially in the worlds of finance, telecommunications and government.

Performance usually manifests itself in the following measures.

3.2.1 Throughput

Throughput is a measure of the amount of work an application must perform in unit time. Work is typically measured in transactions per second (tps), or messages processed per second (mps). For example, an on-line banking application might have to guarantee it can execute 1000 transactions per second from Internet banking customers. An inventory management system for a large warehouse might need to process 50 messages per second from trading partners.

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

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

3.2.2 Response Time

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

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

### 3.2.3 Deadlines

Everyone has probably heard of the weather forecasting system that took 36 hours to produce the forecast for the next day! I’m not sure if this is apocryphal, but it’s an excellent example of the requirement to meet a performance deadline. Deadlines in the IT world are commonly associated with batch systems. A social security payment system must complete in time to deposit claimant’s payments in their accounts on a given day. If it finishes late, claimants don’t get paid when they expect, and this can cause severe disruptions and pain, and not just for claimants. In general, any application that has a limited window of time to complete will have a performance deadline requirement.

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

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

### 3.2.4 Performance for the ICDE System

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

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

Once an event is trapped, the ICDE client must call the server to store the event in the data store. It’s vital therefore that this operation does not contribute any delay that the user might experience. For this reason, when an event is detected, it is written to an in-memory queue in the ICDE client. Once the event is in the queue, the event detection thread returns and waits to capture the next event. Another thread pulls events from the queue and calls the ICDE server.

This solution within the ICDE client decouples event capture and storage. A delayed write to the server cannot delay the GUI code. From the ICDE server’s perspective, this is crucial. The server must be designed to store events in the data store as quickly as possible. But the design can be guaranteed that there will only ever be one client request per user workstation outstanding at any instant.

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

### 3.3 Scalability

Let’s start with a representative definition of scalability\(^{10}\):

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

---

\(^{10}\) From www.hyperdictionary.com
This is useful in an architectural context. It tells us that scalability is about how a design can cope with some aspect of the application's requirements increasing in size. To become a concrete quality attribute requirement, we need to understand exactly what is expected to get bigger. Here are some examples:

3.3.1 Request Load

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

In the perfect world and without additional hardware capacity, as the load increases, application throughput should remain constant (i.e. 100 tps), and response time per request should increase only linearly (i.e. 10 seconds). A scalable solution will then permit additional processing capacity to be deployed to increase throughput and decrease response time. This additional capacity may be deployed in two different ways, one adding more CPUs\(^\text{11}\) (and likely memory) to the machine the applications runs on (scale up), the other from distributing the application on multiple machines (scale out). This is illustrated in Fig. 6.

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

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

Importantly, for either approach, scalability should be achieved without modifications to the underlying architecture (apart from inevitable configuration changes if multiple servers are used). In reality, as load increases, applications will exhibit a decrease in throughput and a subsequent exponential increase in response time. This happens for two reasons. First, the increased load causes increased contention for resources such as CPU and memory by the processes and threads in the server architecture.

Second, each request consumes some additional resource (buffer space, locks, and so on) in the application, and eventually this resource becomes exhausted and limits scalability.

![Fig. 6. Scale out versus scale up](image)

As an illustration, Fig. 7 shows how six different versions of the same application implemented using different J2EE application servers perform as their load increases from 100 to 1000 clients.\(^\text{12}\)

3.3.2 Simultaneous Connections

An architecture may be designed to support 1000 concurrent users. How does the architecture respond if this number grows significantly? If a con-

\(^{11}\) Adding faster CPUs is never a bad idea either. This is especially true if an application has components or calculations that are inherently single-threaded.

\(^{12}\) The full context for these figures is described in: J.Gorton, A.Liu, Performance Evaluation of Alternative Component Architectures for Enterprise JavaBean Applications, in IEEE Internet Computing, vol.7, no. 3, pages 18-23, 2003. Bear in mind, these results are a snapshot in time and are meant for illustrative purposes. Absolutely no conclusions about the performance of the current versions of these technologies can or should be drawn.
nected user consumes some resources, then there will likely be a limit to the number of connections that can be effectively supported.

![Graph showing TPS vs. No. of Clients for different platforms: WAS SB, JBoss SB, IAS SB, SS SB, WLS SB, BES SB](image)

**Fig. 7.** Effects of increasing client request load on J2EE platforms.

I encountered a classic example of this problem while performing an architecture review for an Internet Service Provider (ISP). Every time a user connected to the service, the ISP application spawned a new process on their server that was responsible for distributing targeted advertisements to the user. This worked beautifully, but each process consumed considerable memory and processing resources, even when the user simply connected and did nothing. Testing quickly revealed that the ISP’s server machines could only support about 2000 connections before their virtual memory was exhausted and the machines effectively ground to a halt in a disk thrashing frenzy. This made scaling the ISP’s operations to support 100,000 users a prohibitively expensive proposition, and eventually, despite frantic redesign efforts, this was a root cause of the ISP going out of business.

### 3.3.3 Data Size

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

### 3.3.4 Deployment

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

### 3.3.5 Some Thoughts on Scalability

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

### 3.3.6 Scalability for the ICDE Application

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

### 3.4 Modifiability

All capable software architects know that along with death and taxes, modifications to a software system during its lifetime are simply a fact of life. That’s why taking into account likely changes to the application is a good practice during architecture formulation. The more flexibility that
can be built into a design upfront, then the less painful and expensive subsequent changes will be. That’s the theory anyway.

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

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

Illustrative change scenarios are:

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

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

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

### 3.4.1 Modifiability for the ICDE Application

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

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

### 3.5 Security

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

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

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

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

3.5.1 Security for the ICDE Application

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

3.6 Availability

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

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

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

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

3.6.1 Availability for the ICDE Application

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

3.7 Integration

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

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

With data integration, the ways in which the data is used (or abused) by other applications is pretty much out of control of the original data owner. This is because the data integrity and business rules imposed by the application logic are by-passed. The alternative is for interoperability to be achieved through an API (see Fig. 8). In this case, the raw data the application owns is hidden behind a set of functions that facilitate controlled external access to the data. In this manner, business rules and security can be enforced in the API implementation. The only way to access the data and integrate with the application is by using the supplied API.

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

![Integration options diagram](image)

**3.7.1 Integration for the ICDE Application**

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

**3.8 Other Quality Attributes**

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

- **Portability:** Can an application be easily executed on a different software/hardware platform to the one it has been developed for? Portability depends on the choices of software technology used to implement the application, and the characteristics of the platforms that it needs to execute on. Easily portable code bases will have their platform dependencies isolated and encapsulated in a small set of components that can be replaced without affecting the rest of the application.

- **Testability:** How easy or difficult is an application to test? Early design decisions can greatly affect the amount of test cases that are required. As a rule of thumb, the more complex a design, the more difficult it is to thoroughly test. Simplicity tends to promote ease of testing.\(^{13}\) Likewise, writing less of your own code by incorporating pre-tested components reduces test efforts.

- **Supportability:** This is a measure of how easy an application is to support once it is deployed. Support typically involves diagnosing and fixing problems that occur during application use. Supportable systems tend to provide explicit facilities for diagnosis, such as application error logs that record the causes of failures. They are also built in a modular fashion so that code fixes can be deployed without severely inconveniencing application use.

---

\(^{13}\) "There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.", C.A.R. Hoare
3.9 Design Trade-Offs

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

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

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

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

3.10 Summary

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

Of course, understanding the quality attribute requirements is merely a necessary prerequisite to designing a solution to satisfy them. Conflicting quality attributes are a reality in every application of even mediocre complexity. Creating solutions that choose a point in the design space that adequately satisfies these requirements is remarkably difficult, both technically and socially. The latter involves communications with stakeholders to discuss design tolerances, discovering scenarios when certain quality requirements can be safely relaxed, and clearly communicating design compromises so that the stakeholders understand what they are signing up for.

3.11 Further Reading

The broad topic of non-functional requirements is covered extremely thoroughly in:


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


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