Chapter 3
Software Quality Attributes

3.1 Quality Attributes

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

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

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

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

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

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

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

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

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

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

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

3.2 Performance

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

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

Performance usually manifests itself in the following measures.

3.2.1 Throughput

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

---

[1]Ward Cunningham at his finest!

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

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

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

3.2.2   Response Time

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

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

3.2.3   Deadlines

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

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

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

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

3.2.4   *Performance for the ICDE System*

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

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

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

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

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

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

3.3 Scalability

Let's start with a representative definition of scalability[2]:

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

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

3.3.1 Request Load

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

In the perfect world and without additional hardware capacity, as the load increases, application throughput should remain constant (i.e., 100 tps), and response time per request should increase only linearly (i.e., 10 s). A scalable solution will then permit additional processing capacity to be deployed to increase throughput and decrease response time. This additional capacity may be deployed in two different ways, one by adding more CPUs[3] (and likely memory) to the machine the applications runs on (scale up), the other from distributing the application on multiple machines (scale out). This is illustrated in Fig. 3.1.

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

---

[2]From http://www.hyperdictionary.com

[3]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.
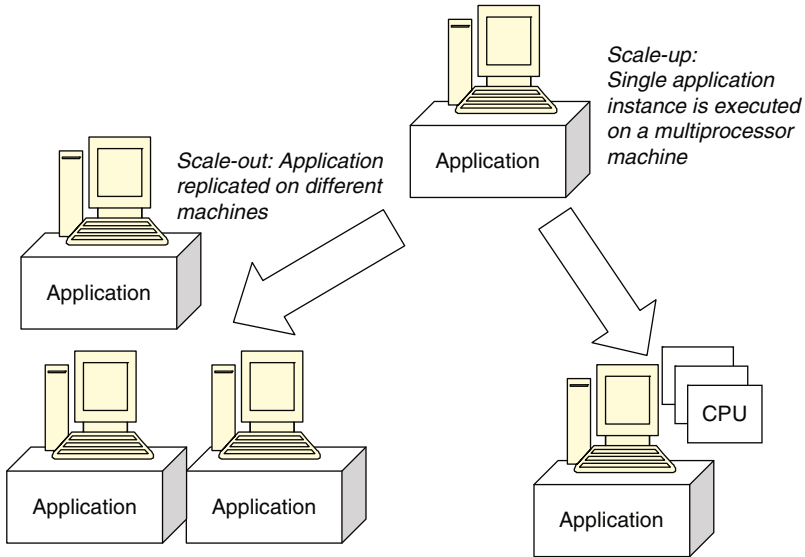
Fig. 3.1 Scale out versus scale up

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

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

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

---

[4]The full context for these figures is described in: I.Gorton, A Liu, *Performance Evaluation of Alternative Component Architectures for Enterprise JavaBean Applications, in IEEE Internet Computing, vol.7, no. 3, pages 18-23, 2003*. Bear in mind, these results are a snapshot in time and are meant for illustrative purposes. Absolutely no conclusions about the performance of the current versions of these technologies can or should be drawn.
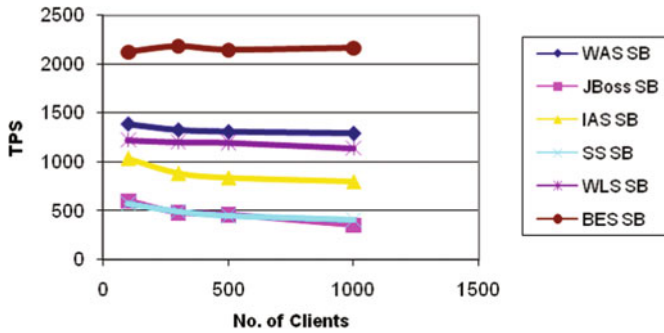
Fig. 3.2 Effects of increasing client request load on JEE platforms

3.3.2 Simultaneous Connections

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

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

3.3.3 Data Size

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

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

3.3.4 Deployment

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

3.3.5 Some Thoughts on Scalability

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

3.3.6 Scalability for the ICDE Application

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

3.4 Modifiability

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

---

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

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

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

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

Illustrative change scenarios are:

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

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

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

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

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

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

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

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

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

---

[6]http://martinfowler.com/articles/injection.html

3.4.1   Modifiability for the ICDE Application

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

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

3.5   Security

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

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

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

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

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

3.5.1   Security for the ICDE Application

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

3.6   Availability

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

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

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

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

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

3.6.1   Availability for the ICDE Application

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

3.7   Integration

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

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

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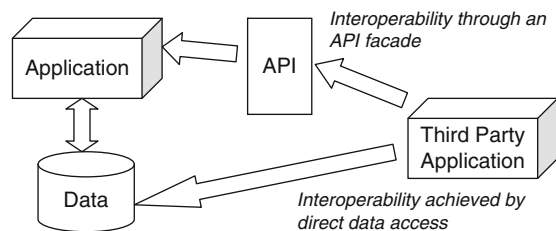

Fig. 3.3 Integration options

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

3.7.1   Integration for the ICDE Application

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

3.8   Other Quality Attributes

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

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

---

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

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

3.9   Design Trade-Offs

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

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

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

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

3.10   Summary

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

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

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

3.11   Further Reading

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

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

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

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

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

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