5 A Software Architecture Process

5.1 Process Outline

The role of an architect is much more than simply carrying out a software design activity. The architect must typically:

- **Work with the requirements team**: The requirements team will be focused on eliciting the functional requirements from the application stakeholders. The architect plays an important role in requirements gathering by understanding the overall systems needs and ensuring that the appropriate quality attributes are explicit and understood.

- **Work with various application stakeholders**: Architects play a pivotal liaison role by making sure all the application’s stakeholder needs are understood and incorporated into the design. For example, in addition to the business user requirements for an application, system administrators will require that the application can be easily installed, monitored, managed and upgraded.

- **Lead the technical design team**: Defining the application architecture is a design activity. The architect leads a design team, comprising system designers (or on large projects, other architects) and technical leads in order to produce the architecture blueprint.

- **Work with the project management**: The architect works closely with project management, helping with project planning, estimation and task allocation and scheduling.

In order to guide an architect through the definition of the application architecture, it’s useful to follow a defined software engineering process. Fig. 36 shows a simple, three-step iterative architecture process that can be used to guide activities during the design. Briefly, the three steps are:
1. **Define architecture requirements**: This involves creating a statement or model of the requirements that will drive the architecture design.

2. **Architecture design**: This involves defining the structure and responsibilities of the components that will comprise the architecture.

3. **Validation**: This involves "testing" the architecture, typically by walking through the design, against existing requirements and any known or possible future requirements.

![Diagram](image)

**Fig. 36.** A three step architecture design process

This architecture process is inherently iterative. Once a design is proposed, validating it may show that the design needs modification, or that certain requirements need to be further defined and understood. Both these lead to enhancements to the design, subsequent validation, and so on, until the design team is satisfied that the requirements are met.

The rest of this chapter explains each of these steps in more detail.

### 5.1.1 Determine Architectural Requirements

Before an architectural solution can be designed, it’s necessary to have a pretty good idea of the requirements for the application architecture. Architecture requirements, sometimes also called architecturally significant requirements or architecture use cases, are essentially the quality and non-functional requirements for the application.

### 5.1.2 Identifying Architecture Requirements

As Fig. 37 shows, the main sources of architecture requirements are the functional requirements document, and other documents that capture various stakeholder needs. The output of this step is a document that states the architecture requirements for the application.

![Diagram](image)

**Fig. 37.** Inputs and outputs for determining architecture requirements

Let’s look at some examples. A typical architecture requirement concerning reliability of communications is:

"Communications between components must be guaranteed to succeed with no message loss"

Some architecture requirements are really constraints, for example:

"The system must use the existing IIS-based web server and use Active Server Page to process web requests"

Constraints impose restrictions on the architecture and are (almost always) non-negotiable. They limit the range of design choices an architect can make. Sometimes this makes an architect’s life easier, and sometimes it doesn’t. Table 2 lists some example architecture requirements along with the quality attribute they address.
Table 2. Some example architecture requirements

<table>
<thead>
<tr>
<th>Quality Attribute</th>
<th>Architecture Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>Application performance must provide sub-four second response times for 99% of requests.</td>
</tr>
<tr>
<td>Security</td>
<td>All communications must be authenticated and encrypted using certificates.</td>
</tr>
<tr>
<td>Resource Management</td>
<td>The server component must run on a low end office-based server with 512MB memory.</td>
</tr>
<tr>
<td>Usability</td>
<td>The user interface component must run in an Internet browser to support remote users.</td>
</tr>
<tr>
<td>Availability</td>
<td>The system must run 24x7x365, with overall availability of 0.99.</td>
</tr>
<tr>
<td>Reliability</td>
<td>No message loss is allowed, and all message delivery outcomes must be known within 30 seconds.</td>
</tr>
<tr>
<td>Scalability</td>
<td>The application must be able to handle a peak load of 500 concurrent users during the enrollment period.</td>
</tr>
<tr>
<td>Modifiability</td>
<td>The architecture must support a phased migration from the current Fourth Generation Language (4GL) version to a .NET systems technology solution.</td>
</tr>
</tbody>
</table>

Table 3 gives some typical examples of constraints, along with the source of each constraint.

Table 3. Some example constraints

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Architecture Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business</td>
<td>The technology must run as a plug-in for MS BizTalk, as we want to sell this to Microsoft.</td>
</tr>
<tr>
<td>Development</td>
<td>The system must be written in Java so that we can use existing development staff.</td>
</tr>
<tr>
<td>Schedule</td>
<td>The first version of this product must be delivered within six months.</td>
</tr>
<tr>
<td>Business</td>
<td>We want to work closely with and get more development funding from MegaHugeTech Corp, so we need to use their technology in our application.</td>
</tr>
</tbody>
</table>

5.1.3 Prioritizing Architecture Requirements

It’s a rare thing when all architecture requirements for an application are equal. Often the list of architecture requirements contains items that are of low priority, or “this would be good to have, but not necessary” type features. It’s consequently important to explicitly identify these, and rank the architecture requirements using priorities. Initially, it’s usually sufficient to allocate each requirement to one of three categories, namely:

1. **High**: the application must support this requirement. These requirements drive the architecture design;
2. **Medium**: this requirement will need to be supported at some stage, but not necessarily in the first/next release;
3. **Low**: this is part of the requirements wish list. Solutions that can accommodate these requirements are desired, but they are not the drivers of the design;

Prioritization gets trickier in the face of conflicting requirements. Common examples are:

- Reusability of components in the solution versus rapid time-to-market. Making components generalized and reusable always takes more time and effort.
- Minimal expenditure on COTS products versus reduced development effort/cost. COTS products mean you have to develop less code, but they cost money.

There’s no simple solution to these conflicts. It’s part of the architect’s job to discuss these with the relevant stakeholders, and come up with possible solution scenarios to enable the issues to be thoroughly understood. Conflicting requirements may even end up as the same priority. It is then the responsibility of the solution to consider appropriate trade-offs, and to try to find that “fine line” that adequately satisfies both requirements without upsetting anyone or having major undesirable consequences on the application. Remember, good architects know how to say “no”.

In a project with many stakeholders, it’s usually a good idea to get each set of stakeholders to sign off on this prioritization. This is especially true in the face of conflicting requirements. Once this is agreed, the architecture design can commence.

5.2 Architecture Design

While all the tasks an architect performs are important, it’s the quality of the architecture design that really matters. Wonderful requirement documents and attentive networking with stakeholders mean nothing if a poor design is produced.
Not surprisingly, design is typically the most difficult task an architect undertakes. Good architects draw on several years of software engineering and design experience. There’s no substitute for this experience, so all this chapter can do is try to help readers gain some of the necessary knowledge as quickly as possible.

As Fig. 38 shows, the inputs to the design step are the architecture requirements. The design stage itself has two steps, which are iterative in nature. The first involves choosing an overall strategy for the architecture, based around proven architecture patterns. The second involves specifying the individual components that make up the application, showing how they fit into the overall framework and allocating them responsibilities. The output is a set of architecture views that capture the architecture design, and a design document that explains the design, the key reasons for some of the major design decisions, and identifies the risks involved in taking the design forward.

5.2.1 Choosing the Architecture Framework

Most of the IT applications I’ve worked on in the last ten years are based around a small number of well understood, proven architectures. There’s a good reason for this – they work. Leveraging known solutions minimizes the risks that an application will fail due to an inappropriate architecture.

So the initial design step involves selecting an architecture framework that seems likely to satisfy the key requirements. For small applications, a single architecture pattern like n-tier client-server may suffice. For more complex applications, the design will incorporate one or more known patterns, with the architect specifying how these patterns integrate to form the overall architecture.

There’s no magic formula for designing the architecture framework. A pre-requisite, however, is to understand how each of the main architecture patterns addresses certain quality attributes. The following sub-sections briefly cover some of the major patterns used, and describe how they address common quality requirements.

**Fig. 39. N-tier client-server example**

5.2.1.1 N-Tier Client Server

In Fig. 39 the anatomy of this pattern for a web application is illustrated. The key properties of this pattern are:
- **Separation of concerns:** Presentation, business and data handling logic are clearly partitioned in different tiers.
- **Synchronous communications:** Communications between tiers is synchronous request-reply. Requests emanate in a single direction from the client tier, through the web and business logic tiers to the EIS tier. Each tier waits for a response from the other tier before proceeding.
- **Flexible deployment:** There are no restrictions on how a multi-tier application is deployed. All tiers could run on the same machine, or at the other extreme, each tier may be deployed on its own machine. In web applications, the client tier is usually a browser running on a user’s desktop, communicating remotely over the Internet with a web tier components.

### Table 4. Quality attributes for the N-Tier Client Server pattern

<table>
<thead>
<tr>
<th>Quality Attribute</th>
<th>Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Availability</strong></td>
<td>Servers in each tier can be replicated, so that if one fails, others remain available. Overall the application will provide a lower quality of service until the failed server is restored.</td>
</tr>
<tr>
<td><strong>Failure handling</strong></td>
<td>If a client is communicating with a server that fails, most web and application servers implement transparent failover. This means a client request is, without its knowledge, redirected to a live replica server that can satisfy the request.</td>
</tr>
<tr>
<td><strong>Modifiability</strong></td>
<td>Separation of concerns enhances modifiability, as the presentation, business and data management logic are all clearly encapsulated. Each can have its internal logic modified in many cases without changes rippling into other tiers.</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>This architecture has proven high performance. Key issues to consider are the amount of concurrent threads supported in each server, the speed of connections between tiers and the amount of data that is transferred. As always with distributed systems, it makes sense to minimize the calls needed between tiers to fulfill each request.</td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td>As servers in each tier can be replicated, and multiple server instances run on the same or different servers, the architecture scales out and up well. In practice, the data management tier often becomes a bottleneck on the capacity of a system.</td>
</tr>
</tbody>
</table>

Table 4 shows how common quality attributes can be addressed with this pattern. Precisely how each quality attribute is addressed depends on the actual web and application server technology used to implement the application. .NET, each implementation of J2EE, and other proprietary application servers all have different concrete features. These need to be understood during architecture design so that no unpleasant surprises are encountered much later in the project, when fixes are much more expensive to perform.

The N-Tier Client-Server pattern is commonly used and the direct support from application server technologies for this pattern makes it relatively easy to implement applications using the pattern. It’s generally appropriate when an application must support a potentially large number of clients and concurrent requests, and each request takes a relatively short interval (a few milliseconds to a few seconds) to process.

#### 5.2.1.2 Messaging

In Fig. 40 the basic components of the messaging pattern are shown. The key properties of this pattern are:

- **Asynchronous communications:** Clients send requests to the queue, where the message is stored until an application removes it. After the client has written the message to the queue, it continues without waiting for the message to be removed.
- **Configurable QoS:** The queue can be configured for high-speed, non-reliable or slower, reliable delivery. Queue operations can be coordinated with database transactions.
- **Loose coupling:** There is no direct binding between clients and servers. The client is oblivious to which server receives the message. The server is oblivious as to which client the message came from.

![Fig. 40. Anatomy of the messaging pattern](image)

Table 5 shows how common quality attributes are addressed by messaging. Again, bear in mind, exact support for these quality attributes is messaging product dependent.
Messaging is especially appropriate when the client does not need an immediate response directly after sending a request. For example, a client may format an email, and place it on a queue in a message for processing. The server will at some stage in the future remove the message and send the email using a mail server. The client really doesn’t need to know when the server processes the message.

<table>
<thead>
<tr>
<th>Quality Attribute</th>
<th>Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>Physical queues with the same logical name can be replicated across different messaging server instances. When one fails, clients can send messages to replica queues.</td>
</tr>
<tr>
<td>Failure handling</td>
<td>If a client is communicating with a queue that fails, it can find a replica queue and post the message there.</td>
</tr>
<tr>
<td>Modifiability</td>
<td>Messaging is inherently loosely coupled, and this promotes high modifiability as clients and servers are not directly bound through an interface. Changes to the format of messages sent by clients may cause changes to the server implementations. Self-describing, discoverable message formats can help reduce this dependency on message formats.</td>
</tr>
<tr>
<td>Performance</td>
<td>Message queuing technology can deliver thousands of messages per second. Non-reliable messaging is faster than reliable, with the difference dependent on the quality of the messaging technology used.</td>
</tr>
<tr>
<td>Scalability</td>
<td>Queues can be hosted on the communicating endpoints, or be replicated across clusters of messaging servers hosted on a single or multiple server machines. This makes messaging a highly scalable solution.</td>
</tr>
</tbody>
</table>

Applications that can divide processing of a request into a number of discrete steps, connected by queues, are a basic extension of the simple messaging pattern. This is identical to the “Pipe and Filter” pattern (see Buschmann).

Messaging also provides a resilient solution for applications in which connectivity to a server application is transient, either due to network or server unreliability. In such cases, the messages are held in the queue until the server connects and removes messages. Finally, as Chapter 4 explains, messaging can be used to implement synchronous request-response using a request-reply queue pair.

### 5.2.1.3 Publish-Subscribe

The essential elements of the Publish-Subscribe pattern are depicted in Fig. 41. The key properties of this pattern are:

- **Many-to-Many messaging**: Published messages are sent to all subscribers who are registered with the topic. Many publishers can publish on the same topic, and many subscribers can listen to the same topic.
- **Configurable QoS**: In addition to non-reliable and reliable messaging, the underlying communication mechanism may be point-to-point or broadcast/multicast. The former sends a distinct message for every subscriber on a topic, the latter sends one message which every subscriber receives.
- **Loose Coupling**: As with messaging, there is no direct binding between publishers and subscribers. Publishers do not know who receives their message, and subscribers do not know which publisher sent the message.

![Fig. 41. The Publish-Subscribe pattern](image)

Table 6 explains how publish-subscribe supports common quality attributes.

Architectures based on publish-subscribe are highly flexible and suited to applications which require asynchronous one-many, many-one or many-to-many messaging amongst components. Like messaging, two-way communications is possible using request-reply topic pairs.
### Table 6. Quality attributes for the Publish-Subscribe pattern

<table>
<thead>
<tr>
<th>Quality Attribute</th>
<th>Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>Topics with the same logical name can be replicated across different server instances managed as a cluster. When one fails, publishers send messages to replica queues.</td>
</tr>
<tr>
<td>Failure handling</td>
<td>If a publisher is communicating with a topic hosted by a server that fails, it can find a live replica server and send the message there.</td>
</tr>
<tr>
<td>Modifiability</td>
<td>Publish-subscribe is inherently loosely coupled, and this promotes high modifiability. New publishers and subscribers can be added to the system without change to the architecture or configuration. Changes to the format of messages published may cause changes to the subscriber implementations.</td>
</tr>
<tr>
<td>Performance</td>
<td>Publish-subscribe can deliver thousands of messages per second, with non-reliable messaging faster than reliable. If a publish-subscribe broker supports multicast/broadcast, it will deliver multiple messages in a more uniform time to each subscriber.</td>
</tr>
<tr>
<td>Scalability</td>
<td>Topics can be replicated across clusters of servers hosted on a single or multiple server machines. Clusters of server can scale to provide very high message volume throughput. Also, multicast/broadcast solutions scale better than their point-to-point counterparts.</td>
</tr>
</tbody>
</table>

#### 5.2.1.4 Broker

The major elements of the Broker pattern are shown in Fig. 42. The properties of a broker-based solution are:

- **Hub-and-spoke architecture:** The broker acts as a messaging hub, and senders and receivers connect as spokes. Connections to the broker are via ports that are associated with a specific message format.
- **Performs message routing:** The broker embeds processing logic to deliver a message received on an input port to an output port. The delivery path can be hard coded or depend on values in the input message.
- **Performs message transformation:** The broker logic transforms the source message type received on the input port to the destination message type required on the output port.

![Fig. 42. Elements of the Broker pattern](image)

Table 7 shows the pattern's support for common quality attributes.

### Table 7. Quality attributes for the Broker pattern

<table>
<thead>
<tr>
<th>Quality Attribute</th>
<th>Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>To build high availability architectures, brokers must be replicated. This is typically supported using similar mechanisms to messaging and publish-subscribe server clustering.</td>
</tr>
<tr>
<td>Failure handling</td>
<td>As brokers have typed input ports, they validate and discard any messages that are sent in the wrong format. With replicated brokers, senders can fail over to a live broker should one of the replicas fail.</td>
</tr>
<tr>
<td>Modifiability</td>
<td>Brokers separate the transformation and message routing logic from the senders and receivers. This enhances modifiability, as changes to transformation and routing logic can be made without affecting senders or receivers.</td>
</tr>
<tr>
<td>Performance</td>
<td>Brokers can potentially become a bottleneck, especially if they must service high message volumes and execute complex transformation logic. Their throughput is typically lower than simple messaging with reliable delivery.</td>
</tr>
<tr>
<td>Scalability</td>
<td>Clustering broker instances makes it possible to construct systems scale to handle high request loads.</td>
</tr>
</tbody>
</table>

Brokers are suited to applications in which components exchange messages that require extensive transformation between source and destination...
formats. The broker decouples the sender and receiver, allowing them to produce or consume their native message format, and centralizes the definition of the transformation logic in the broker for ease of understanding and modification.

5.2.1.5 Process Coordinator
The Process Coordinator pattern is illustrated in Fig. 43. The essential elements of this pattern are:

- **Process encapsulation**: The process coordinator encapsulates the sequence of steps needed to fulfill the business process. The sequence can be arbitrarily complex. The coordinator is a single point of definition for the business process, making it easier to understand and modify. It receives a process initiation request, calls the servers in the order defined by the process, and emits the results.

- **Loose coupling**: The server components are unaware of their role in the overall business process, and of the order of the steps in the process. The servers simply define a set of services which they can perform, and the coordinator calls them as necessary as part of the business process.

- **Flexible communications**: Communications between the coordinator and servers can be synchronous or asynchronous. For synchronous communications, the coordinator waits until the server responds. For asynchronous communications, the coordinator provides a callback or reply queue/topic, and waits until the server responds using the defined mechanism.

![Fig. 43. Components of the Process Coordinator pattern](image)

Table 8 shows how this pattern addresses quality requirements.

<table>
<thead>
<tr>
<th>Quality Attribute</th>
<th>Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>The coordinator is a single point of failure. Hence it needs to be replicated to create a high availability solution.</td>
</tr>
<tr>
<td>Failure handling</td>
<td>Failure handling is complex, as it can occur at any stage in the business process coordination. Failure of a later step in the process may require earlier steps to be undone using compensating transactions. Handling failures needs careful design to ensure the data maintained by the servers remains consistent.</td>
</tr>
<tr>
<td>Modifiability</td>
<td>Process modifiability is enhanced because the process definition is encapsulated in the coordinator process. Servers can change their implementation without affecting the coordinator or other servers, as long as their external service definition doesn’t change.</td>
</tr>
<tr>
<td>Performance</td>
<td>To achieve high performance, the coordinator must be able to handle multiple concurrent requests and manage the state of each as they progress through the process. Also, the performance of any process will be limited by the slowest step, namely the slowest server in the process.</td>
</tr>
<tr>
<td>Scalability</td>
<td>The coordinator can be replicated to scale the application both up and out.</td>
</tr>
</tbody>
</table>

The Process Coordinator pattern is commonly used to implement complex business processes that must issue requests to several different server components. By encapsulating the process logic in one place, it is easier to change, manage and monitor process performance. Message broker and Business Process Orchestrator technologies are designed specifically to support this pattern, the former for short lived requests, the latter for processes that may take several minutes or hours or days to complete. In less complex applications, the pattern is also relatively simple to implement without sophisticated technology support, although failure handling is an issue that requires careful attention.
5.2.2 Allocate Components

Once an overall architecture framework has been selected, based on one or more architecture patterns, the next task is to define the major components that will comprise the design. The framework defines the overall communication patterns for the components. This must be augmented by the following:

- Identifying the major application components, and how they plug into the framework.
- Identifying the interface or services that each component supports.
- Identifying the responsibilities of the component, stating what it can be relied upon to do when it receives a request.
- Identifying dependencies between components.
- Identifying partitions in the architecture that are candidates for distribution over servers in a network.

The components in the architecture are the major abstractions that will exist in the application. Hence, it’s probably no surprise that component design has much in common with widely used object-oriented design techniques. In fact, class and package diagrams are often used to depict components in an architecture.

Some guidelines for component design are:

- Minimize dependencies between components. Strive for a loosely coupled solution in which changes to one component do not ripple through the architecture, propagating across many components. Remember, every time you change something, you have to re-test it.
- Design components that encapsulate a highly “cohesive” set of responsibilities. Cohesion is a measure of how well the parts of a component fit together. Highly cohesive components tend to have a small set of well-defined responsibilities that implement a single logical function. For example, an EnrollmentReports component encapsulates all the functions required to produce reports on student enrollments in courses. If changes to report format or type are needed, then it’s likely the changes will be made in this component. Hence, strong cohesion limits many types of changes to a single component, minimizing maintenance and testing efforts.
- Isolate dependencies on middleware and any COTS infrastructure technologies. The fewer components that are dependent on specific middleware and COTS components API calls, the easier it is to change or update the middleware or other infrastructure services. Of course this takes more effort to build, and introduces a performance penalty.
- Use decomposition to structure components hierarchically. The outer-most level component defines the publicly available interface to the composite component. Internally, calls to this interface are delegated to the locally defined components, whose interfaces are not visible externally.
- Minimize calls between components, as these can prove costly if the components are distributed. Try to aggregate sequences of calls between components into a single call that can perform the necessary processing in a single request. This creates coarser grain methods or services in interfaces that do more work per request.

![Diagram of Order Processing Example Architecture](image)

**Figure Key**
- Existing Component
- New Component
- Database
- Persistent Queue

Fig. 44. Order processing example architecture
Let’s explore a simple example to illustrate some of these issues. Fig. 44 is an example of a structural view of an order processing application, defined using a simple informal notation. New orders are received (from where is irrelevant) and loaded into a database. Each order must be validated against an existing customer details system to check the customer information and that valid payment options exist. Once validated, the order data is simply stored in the order processing database, and an email is generated to the customer to inform them that their order is being processed.

The general architecture framework is based on straightforward messaging. The customer order details are read from the database, validated, and if valid, they are stored in the order application and written to a queue. Information about each valid order is removed from the queue, formatted as an email and sent to the customer using the mail server. Hence, using a message queue this architecture decouples the order processing from the email formatting and delivery.

Four components are introduced to solve this problem. These are described below, along with their responsibilities:

- **OrderInput**: This is responsible for accessing the new orders database, encapsulating the order processing logic, and writing to the queue.
- **Validate**: This encapsulates the responsibility of interacting with the customer system to carry out validation, and writing to the error logs if an order is invalid.
- **Store**: This has the responsibility of interacting with the order system to store the order data.
- **SendEmail**: This removes a message from the queue, formats an email message and sends it via an email server. It encapsulates all knowledge of the email format and email server access.

So, each component has clear dependencies and a small set of responsibilities, creating a loosely coupled and cohesive architecture. We’ll return to this example and further analyze its properties in the next section, in which the validation of an architecture design is discussed.

### 5.3 Validation

During the architecture process, the aim of the validation phase is to increase the confidence of the design team that the architecture is fit for purpose. The validation has to be achieved within the project constraints of time and budget, as the detailed design and implementation cannot generally fully commence until the architecture is agreed. The trick is to be as rigorous and efficient as possible.

Validating an architecture design poses some tough challenges. Whether it’s the architecture for a new application, or an evolution of an existing system, the proposed design is, well, just that – a design. It can’t be executed or tested to see if it fulfills its requirements. It will also likely consist of new components that have to be built, and black box off-the-shelf components such as middleware and specialized libraries and existing applications. All these have to be integrated and made to work together.

So, what can sensibly be done? There are two main techniques that have proved useful. The first essentially involves manual testing of the architecture using test scenarios. The second involves the construction of a prototype that creates a simple archetype of the desired application, so that its ability to satisfy requirements can be assessed in more detail through prototype testing. The aim of both is to identify potential flaws and weaknesses in the design so that they can be improved before implementation commences. These approaches should be used to explicitly identify potential risk areas for tracking and monitoring during the subsequent build activities.

#### 5.3.1 Using Scenarios

Scenarios are a technique developed at the SEI to tease out issues concerning an architecture through manual evaluation and testing. Scenarios are related to architectural concerns such as quality attributes, and they aim to highlight the consequences of the architectural decisions that are encapsulated in the design.

The SEI ATAM work describes scenarios and their generation in great detail. In essence though, scenarios are relatively simple artifacts. They involve defining some kind of stimulus that will have an impact on the architecture. The scenario then involves working out how the architecture responds to this stimulus. If the response is desirable, then a scenario is deemed to be satisfied by the architecture. If the response is undesirable, or hard to quantify, then a flaw or at least an area of risk in the architecture may have been uncovered.

Scenarios can be conceived to address any quality requirement of interest in a given application. Some general hypothetical examples are shown in Table 9. These scenarios highlight the implications of the architecture design decisions in the context of the stimulus and the effects it elicits. For example, the “availability” scenario shows that messages can be lost if a
server fails before messages have been delivered. The implication here is that messages are not being persisted to disk, most likely for performance reasons. The loss of messages in some application contexts may be acceptable. If it is not, this scenario highlights a problem, which may force the design to adopt persistent messaging to avoid message loss.

<table>
<thead>
<tr>
<th>Quality Attribute</th>
<th>Stimulus</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>The network connection to the message consumers fails.</td>
<td>Messages are stored on the MOM server until the connection is restored. Messages will only be lost if the server fails before the connection comes back up.</td>
</tr>
<tr>
<td>Modifiability</td>
<td>A new set of data analysis components must be made available in the application.</td>
<td>The application needs to be rebuilt with the new libraries, and the all configuration files must be updated on every desktop to make the new components visible in the GUI toolbox.</td>
</tr>
<tr>
<td>Security</td>
<td>No requests are received on a user session for ten minutes.</td>
<td>The system treats this session as potentially insecure and invalidates the security credentials associated with the session. The user must logon again to connect to the application.</td>
</tr>
<tr>
<td>Modifiability</td>
<td>The supplier of the transformation engine goes out of business.</td>
<td>A new transformation engine must be purchased. The abstract service layer that wraps the transformation engine component must be re-implemented to support the new engine. Client components are unaffected as they only use the abstract service layer.</td>
</tr>
<tr>
<td>Scalability</td>
<td>The concurrent user request load doubles during the 3 week enrollment period.</td>
<td>The application server is scaled out on a two machine cluster to handle the increased request load.</td>
</tr>
</tbody>
</table>

Let’s look at some more specific examples for the order processing example introduced in the previous section. The design in Fig. 44 needs to be validated, and the scenarios in Table 10 probe more deeply into the architecture, looking to expose flaws or areas of risk.

**Table 10. Scenarios for the order processing example**

<table>
<thead>
<tr>
<th>Quality Attribute</th>
<th>Stimulus</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modifiability</td>
<td>The Customer System packaged application is updated to an Oracle database.</td>
<td>The Validate component must be rewritten to interface to the Oracle system.</td>
</tr>
<tr>
<td>Availability</td>
<td>The email server fails.</td>
<td>Messages build up in the OrderQ until the email server re-starts. Messages are then sent by the SendEmail component to remove the backlog. Order processing is not affected.</td>
</tr>
<tr>
<td>Reliability</td>
<td>The Customer or Order systems are unavailable.</td>
<td>If either fails, order processing halts and alerts are sent to system administrators so that the problem can be fixed.</td>
</tr>
</tbody>
</table>

The first two scenarios seem to elicit positive responses from the design. The Validate component bounds the changes needed to accommodate a new customer database, and hence it insulates other components from change. And should the email server be unavailable, then the implication is that emails are merely delayed until the email server returns.

The failure of the Customer or Order applications is more revealing however. The communications with these two systems is synchronous, so if either is not available, order processing must halt until the applications are restored. This may be less than desirable.

Note the design does not discriminate between the interactions with the two applications. It’s pretty obvious, however, that the interaction with the Customer System requires a response saying whether the order data is valid. If it is not, it is written to an error log and the order processing ceases for that order. The Order System though simply stores the order data for subsequent processing. There’s no need for the Store component to require an immediate response.

So, the reliability scenario has highlighted an area where the architecture could be improved. An order can’t be processed until it has been successfully validated, so a response from the Customer System is necessary. If it is unavailable, processing can’t continue.

But the Order System is a different matter. Asynchronous communications is better in this case. Store could just write to a persistent queue, and order processing can continue. Another component could then be introduced to read the order from the queue and add the details to the Order.
System. This solution is more resilient to failure, as the Order System can be unavailable but order processing can continue.

5.3.2 Prototyping

Scenarios are a really useful technique for validating a proposed architecture. But some scenarios aren't so simple to answer based only on a design description. Consider a performance scenario for the order processing system:

"On Friday afternoon, orders must be processed before close-of-business to ensure delivery by Monday. Five thousand orders arrive through various channels (Web/Call centre/business partners) five minutes before close-of-business."

The question here then is simply, can the five thousand orders be processed in five minutes? This is a tough question to answer when some of the components of the solution don't yet exist.

The only way to address such questions with some degree of confidence is to build a prototype. Prototypes are minimal, restricted or cut-down versions of the desired application, created specifically to test some high risk or poorly understood aspects of the design. Prototypes are typically used for two purposes:

1. Proof-of-concept: Can the architecture as designed be built in a way that can satisfy the requirements?
2. Proof-of-technology: Does the technology (middleware, integrated applications, libraries, etc) selected to implement the application behave as expected?

In both cases, prototypes can provide concrete evidence about concerns that are otherwise difficult, if not impossible to validate in any other way.

To answer our performance scenario above, what kind of prototype might we build? The general answer is one that incorporates all the performance sensitive operations in the design, and that executes on a platform as similar as possible (ideally identical) to the one the application will be deployed on.

For example, the architect might know that the queue and email systems are easily capable of supporting five thousand messages in five minutes, as these solutions are used in another similar application. There would therefore be no need to build this as part of the prototype. However, the

throughput of interactions between the Customer and Order applications using their APIs are an unknown, and hence these two must be tested to see if they can process five thousand messages in five minutes. The simplest way to do this is:

- Write a test program that calls the Customer System validation APIs five thousand times, and time how long this takes.
- Write a test program that calls the Order System store APIs five thousand times, and time how long this takes.

Once the prototypes have been created and tested, the response of the architecture to the stimulus in the scenario can be answered with a high degree of confidence.

Prototypes should be used judiciously to help reduce the risks inherent in a design. They are the only way that concerns related to performance, scalability, ease of integration and capabilities of off-the-shelf components can be addressed with any degree of certainty.

Despite their usefulness, a word of caution on prototyping is necessary. Prototyping efforts should be carefully scoped and managed. Ideally a prototype should be developed in a day or two, a week or two at most. Most proof-of-technology and proof-of-concept prototypes get thrown away after they've served their purpose. They are a means to an end, so don't let them acquire a life of their own and become an end in themselves.

5.4 Summary and Further Reading

Designing an application architecture is an inherently creative activity. However, by following a simple process that explicitly captures architecturally significant requirements, exploits known architecture patterns and systematically validates the design, some of the mystique of design can be exposed.

The three step process described in this chapter is inherently iterative. The initial design is validated against requirements and scenarios, and the outcome of the validation can cause the requirements or the design to be revisited. The iteration continues until all the stakeholders are happy with the architecture, which then becomes the blueprint from which detailed design commences.

The process is also scalable. For small projects, the architect may be working mostly directly with the customer, or there may in fact be no tangible customer (often the case in new, innovative product development). The architect is also likely to be a major part of the small development
team that will build the project. In such projects, the process can be followed informally, producing minimal documentation. For large projects, the process can be followed more formally, involving the requirements and design teams, gathering inputs from the various stakeholders involved, and producing extensive documentation.

Of course, other architecture processes exist, and probably the most widely used is the Rational Unified Process (RUP). A good reference to RUP is:


The most comprehensive source of information on methods and techniques for architecture evaluation is:


This describes the ATAM process, and provides excellent examples illustrating the approach. Its focus is evaluating large, complex systems, but many of the techniques are appropriate for smaller scale applications.

A group of luminaries in the software architecture area got together in 1999 and produced a report known as the Software Architecture Review and Assessment (SARA) Report. This is a comprehensive source of experience-based guidance that can be employed to carry out architecture reviews. The best way to find this report is to google for it, as at the time of writing, its location seems to be transient!

6 Documenting a Software Architecture

6.1 Introduction

Architecture documentation is often a thorny issue in IT projects. It’s common for there to be little or no documentation covering the architecture in many projects. Sometimes, if there is some, it’s out-of-date, inappropriate and basically not very useful.

At the other extreme there are projects that have masses of architecture related information captured in various documents and design tools. Sometimes this is invaluable, but at times it’s out-of-date, inappropriate and not very useful!

Clearly then, experience tells us that documenting architectures is not a simple job. But there are many good reasons why we want to document our architectures, for example:

- Others can understand and evaluate the design. This includes any of the application stakeholders, but most commonly other members of the design and development team.
- We can understand the design when we return to it after a period of time.
- Others in the project team and development organization can learn from the architecture by digesting the thinking behind the design.
- We can do analysis on the design, perhaps to assess its likely performance, or to generate standard metrics like coupling and cohesion.

Documenting architectures is problematic though, because:

- There’s no universally accepted architecture documentation standard.
- An architecture can be complex, and documenting it in a comprehensible manner is time consuming and non-trivial.
- An architecture has many possible views. Documenting all the potentially useful ones is time consuming and expensive.