Software Architecture Process And Management

Computer Science 4th Year, Created By Monkeys

Contents

1 Introduction

1.1 Learning Outcomes of the course

- Integrate knowledge of software architecture to capture **quality attribute** requirements for a system, **evaluate proposed architectures** and create options for **improvement**.
- Analyse and justify complex **trade-off decisions** between **competing software architectures**.
- Evaluate the **strengths** and **weaknesses** of software architecture in support of particular approaches to **design**, **process** and **management** for a particular system and make **recommendations** on the **choice of process** for that system.
- Working in a group to **critically reflect** on aspects of the **software architecture literature** and practice to create a resource that support their learning in software architecture.

1.2 What is Success for a Large Project?

A large project will be considered successful if:

- The software is delivered on schedule
- Development costs are within budget
- The software meets the needs of users

1.3 Software Architecture

Software Architecture Definitions

1. The software architecture of a system is the **set of structures** needed to reason about the system, which comprise software elements, relations among them and properties of both. (Bass, Clements, Kazman 2013)
2. A software system's architecture is the set of **principal design decisions** about the system (R.N Talyor et al)

Architecture is a collection of structures
We observe three frequently types of structure:

1. **Modular Structure**: static structure that focuses on how the functionality is divided up, structured, and assigned to development and implementation teams.
2. **Component and Connector structure**: runtime structures that focus on how components interact.

3. **Allocation structures**: mapping to organizational, development, installation, execution environments.

Architecture is an Abstraction

Architecture is used to **suppress** detail that is unimportant for the reasoning we are doing. In particular it **abstracts away** from the private details of **implementation details** of specific methods.

** All systems have architectures (even if people have forgotten them) **

Complicated systems are embedded in organisation and we can often see architecture through practice:

- *How is the system developed?* -> This will often provide **clues to structures**.
- *How is the maintenance, evolution, issue reporting dealt with?* -> This will often help with **modularity**.
- *What are the failure characteristics of the system in operation?* -> This will often suggest **component and connector structure**.

1.4 Case Study: General Practice Extraction System

"The General Practice Extraction Service (GPES)" is an IT system designed to allow NHS organizations to extract data from **all** GP practice computers. *This is because different GP's have different contracts and therefore use different software to save patient data.*

Basic idea is to create an API to query every system from all different software's created for different GP's. This will allow generic extraction of **patient data** regardless of their GP.

Customers

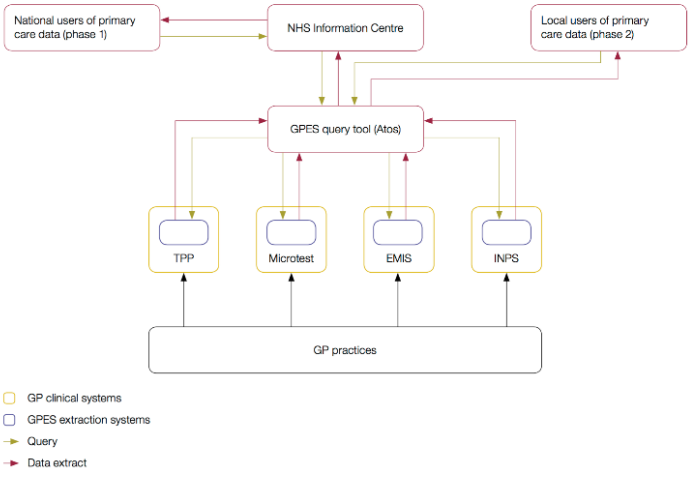
Figure 1
Proposed GPES customers and potential benefits

Organisation	Main use	Benefits of GPES
NHS England	Commission and pay for GP services	Wider and more flexible data indicators, to monitor and reward high-quality care.
National Institute for Health and Care Excellence	Research to help produce clinical guidelines	More records to give greater confidence in data, especially for less common conditions.
UK Biobank	Medical research, on health of 500,000 project participants	Extract detailed data for participants, despite geographical spread and different GP practices.
Healthcare Quality Improvement Partnership	Clinical audits – assessing care quality	Wider range of clinical audits, especially where little data previously available, such as care for those with learning disabilities.
Medicines and Healthcare Products Regulatory Agency	Monitor side effects of medicines	Tapping into data on side effects in GP computer systems to pass on information more efficiently.
Clinical Practice Research Datalink	Support observational and public health research	More records to give greater confidence in data, especially for less common conditions.

Source: National Audit Office interviews with HSCIC staff and proposed GPES customers

Structure

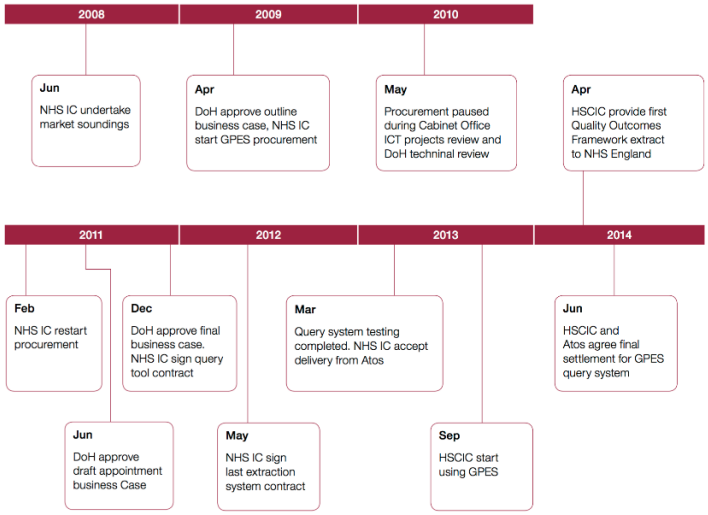
Figure 2
Design of the General Practice Extraction Service



Source: National Audit Office, based on information in the NHS IC GPES business cases

Timeline

Figure 3
Timeline for the GPES procurement



Source: National Audit Office analysis of NHS IC business cases and interviews with HSCIC staff

National Audit office Conclusion of the GPES Project

- The project has been significantly delayed and many customers have yet to receive data.
- Mistakes in the original procurement and contract management contributed to the losses of public funds. This occurred through asset write-off's and settlements with suppliers
- Only **one** customer, *NHS England* has so far received data from GPES.

Originally the business plan for GPES said the service would start in **2009-2010**. It actually took until **2014** for the first extraction to take place. The total expected loss for the GPES project rose from **£14 million** to **£40 million** during the *planning* and *procurement stage*.

Data Extract Issues

- First GP system suppliers were asked to fulfil a common query language for the extraction process (this was not in their interest as it would cost them alot to makes these changes to their current systems and thus pretty much refused to do so).
- This requirement then changed to each GP system supplier creating their own logical 'business rules' which would be used to extract the data. (Different for each supplier, one API to query each supplier to extract data)
- NHS IC's using a non-competitive procurement approach, in-addition to

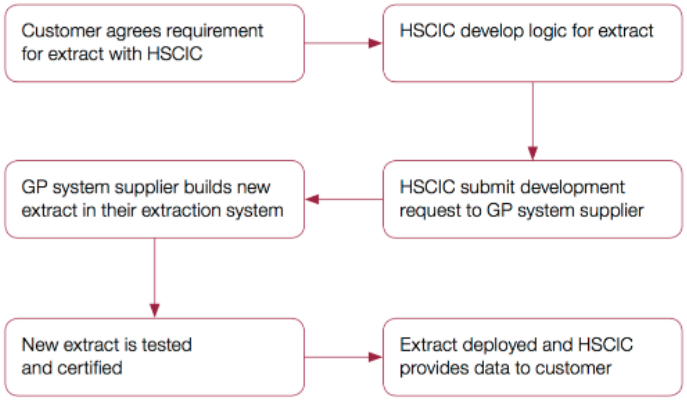
the changes in design both contributed to the restrictive process for *extracts.*

- HSCIC (the successor of the NHS IC) has continued to use the GPSOC framework to require data sharing between NHS systems. *The new framework (2014) states that the principal clinical system suppliers must provide an interface method for third-party system uses.*
- **HSCIC cannot do wide range nor scale of data extracts. Due to the design of the GPES system and the restrictions in the supplier contracts. (Over 100 different extracts have been requested) HSCIC estimate that they will only be able to design 24 new extracts in 2015-16**

Data Extract Issue ctd

Figure 6

Process to develop a new GPES data extract



Source: National Audit Office

Data Extract Issue concluded

- What's the difference in the two approaches to the HSCIC and the Software vendors?
- Are there still issues with the final process?
- **4.10** GPES will continue to operate in the short term, as its data is critical for determining payments to GPs. Its coverage of all practices in England cannot currently be replicated by other primary care data extraction systems.
- **4.11** However, limited capacity and the dif culty of developing new extracts deters wider use. The HSCIC has acknowledged there is unlikely to be a long-term future for all or part of the GPES. However, they intend to reuse parts for a replacement system if possible. The HSCIC estimate that they will achieve less than two more years of use from the GPES in its current form, in contrast to the five-year minimum lifetime assumed for new IT systems.

2 Basic Concepts of Architectures

2.1 What is good architecture?

The architecture is appropriate for the **context of use**. E.g. 3-tier e-commerce architecture is not appropriate for a avionics project.

Guidance on 'good architecture' focuses on:

- **Process**
- **Structure**

Software architecture should capture the **principal design** decisions about the system. The **Blueprint** for software architecture focuses on:

- Structure
- Component behaviour
- Component interaction and how that influences **Quality Attributes** of the *systems*.

2.2 Process

Architect teams are often small and **maintains the integrity** of the architecture. The architecture is *justified* in relation to a **prioritized list of quality attributes** that need to be managed. **Stakeholders interests** are documented and are used to build the type of architecture that will reflect them.

Architecture is often evaluated in terms of *how well it delivers the quality attributes*. Software architectures are often chosen to allow **incremental implementation**. (I.e Low coupling, high cohesion)

- Definitions for coupling and cohesion!

2.3 Structure

The structure of architecture will differ depending on the requirements of the software, often the following are utilised:

- **Modularity** → Hides information, separates concerns, allows good robust interfaces that are unlikely to change
- Well known **patterns and tactics** are often implemented
- Architecture built to NOT depend on **particular versions of tools**, or **special features** *unless its essential!*
- Modules *producing* data should be **separate** from those *consuming* data

- Usually a complex mapping between **modules** *(static structure)* and **components** *(dynamic structure)*
- MINIMISE the number of ways of **interaction between components**
- The architecture should clearly **identify resource contention issues** and deal with them. (E.g. network capacity, minimise network throughput using different techniques [EXC])

Prescriptive vs Descriptive Structures
Prescriptive structure is what we use to model the system before it is built. It is the aim the architect has while generating the blueprint *(UMLAsBlueprint, forward engineering)*, however it is often to *tidy* and unrealistic to be able to model the architecture of a system.

Descriptive structure is usually made after the system has been created. It is used to describe the entire system, how the **components** interact, the responsibilities of each **module** *(usually extremely messy)* etc ...

2.4 The Importance of Architecture

Software Architecture has several uses:

1. Enables us to manage the **key attributes** of a system
2. Allows reasoning about and managing **change**
3. Allows predictions of **key quality attributes**
4. Allows **improved communication** between stakeholders
5. Defines **constraints** on the software's implementation
6. Provides the basis for **evolutionary prototyping**
7. Is the key artefact for reasoning about **cost** and **scheduling**
8. Focuses on the assembly of **components** rather than the **creation/implementation** of components

Other uses are:

- *Reflects the structure of an organisation*
- *Can be used as the transferable, reusable model at the heard of a product line*
- *Restricts design alternative and channels developer effort in a coordinated way*
- *Provides the basis for training new team members*

2.5 Managing Attributes and Change

It is a fact that the majority of software projects will undergo requirements change. This may also change **key quality attributes** of the system. The idea is to use architecture that will minimise the change to the *architecture* and allow the system to be **modifiable** utilising the same abstract **architectural** ideas.

Managing change can be reasoned about on three levels:

1. Inside an element *[cheapest]*
2. Between elements maintaining the architecture *[can be costly]*
3. Requiring architecture change (we wish to avoid this as much as possible) *[most expensive change]*

2.6 Prediction of Attributes

We can attempt to predict the **key quality attributes** of the system based on *requirements* and possible (logical) *system extensions* in the future. Planning for these changes will minimise need for architectural change, which in turn will **reduce the cost** in future work.

**** Models should be able to be built based on the predictions of the attributes and requirements ****

2.7 Communication Between Stakeholders

A well documented architecture allows **improved communication** between stakeholders. Some examples of how the documented architecture can help with communication are the following:

- User has particular requirements in terms of user experience
- Customer needs to know about schedule, budget and meeting regulations in their market
- Project manager needs to know the dependences in terms of the modules and components

These might be accommodated by different views of the system that are consistent

2.8 Early Design and Constraints

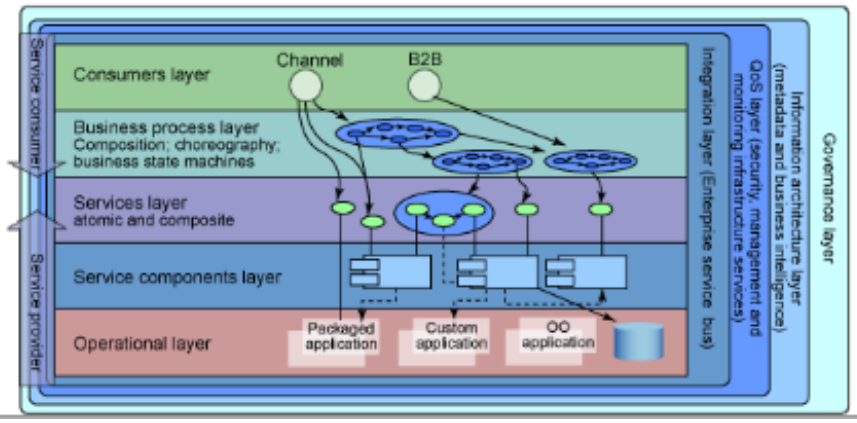
Early design carries the *most fundamental* design decisions, e.g:

- What the **key quality attributes** are

- The **architecture form/type** that will give the best control over these attributes
- The characterisation of the behaviour of the architecture elements

Constraints

- Defines constraints on implementation:
  - Architecture specifies the elements and their interaction
  - For example, layered architecture usually constrain access to be between adjacent layers



2.9 Evolutionary Prototyping

Evolutionary Prototyping allows a system to be constantly tested under real conditions as it is being developed. As *bugs* are detected they are fixed and tested in the next prototype. Examples of systems that used **evolutionary prototyping** are:

- Plug and Play - early experience of the BASE functionality + extensibility.
- Real time architectures - early experience with scheduling. *(Worse case execution times guide design and deployment)*

2.10 Cost and Scheduling

Reasoning about the **following topics** allows for effect cost and scheduling in a software project:

- Capturing dependencies
- Estimation of required efforts for different sections
- Allocating effort to elements
- Understanding of how elements influence each other
- Use architecture to interpret bottom-up estimates from teams working on elements

2.11 Product Line (Model)

The **product line** model is a *transferable and reusable* model. **Elements** are assets that compose to give new *functionality*. The architecture provides the means to **compose the elements**. A planned approach allows the reuse of architectural elements *(think object inheritance)*.

2.12 Component Level & Channelling Development

At the component level we focus on the **assembly** of components rather than the **creation** of them! With well designed elements and architecture we can combine elements from different **producers** *(provided they conform to a standardized interface)*. This provides the following **benefits**:

- Decrease time to market
- More reliability
- Lower cost
- Flexibility *(e.g. using multiple or alternate suppliers for a component)*

Channelling Development restricts alternatives and channels developer effort in a coordinate way. This provides a defined **context** for the developer. Well defined **interfaces** and clear ideas of the **functionality & quality attributes** are required!

**** The overall goal is to provide clarity on what is an architectural decision and what is a development decision. ****

3 Context Design

Software architects and architecture have arisen as systems have grown in: *scale*, *economic importance* and *criticality*. Architecture plays different roles in different contexts. The **main contexts** are:

- Technical Context
- Project Life-cycle Context
- Business Context
- Professional Context

3.1 Technical Context

The **technical context** is whereby the architecture supports technical activity. For example this could be in **measuring** a statistic, the **verification & validation** process, **compliance** ...

The architecture provides a means for controlling **quality attributes** of the system. In the **context of design** activities we try and choose architectures that **enable the attributes** we care most about. We may find through analysing already *existing systems* that specific architectures inhibit (prevent) particular quality attributes.

** Architecture does not often have much to say about the functionality of a system, because they provide containers for functionality. **

3.1.1 Controlling Quality Attributes

Usually we care about multiple quality attributes at once. Selecting a type of architecture will allow specific quality attributes to be ensured for when it is deployed to the end user. Examples of **quality attributes** we might care about for a particular system are:

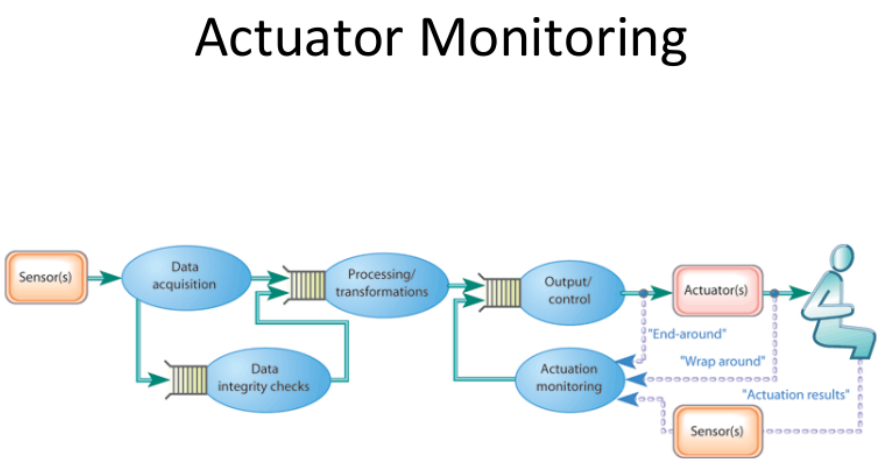
QA	Description
Safety	The safety of a system is whereby we worry about ensuring that the **system only behaves as is intended** and has no additional behaviour that is unspecified.
Testability	The testability of a system ensures that **elements are clearly isolated**. That we know the **expected behaviour of components**. We know the **relations of modules** to track down faulty code and finally we know how the **components are intended to integrate together** to give overall behaviour.
Availability	The availability of a system is whereby we worry about ensuring there is a **system to take over**, in the case the original system fails.

Other examples of quality attributes include **performance**, **usability**, **interoperability** ...

These examples of quality attributes related to the **actuator monitoring** system that was described in lectures. As actuators are physical devices they will suffer from *'wear and tear'* and eventually break. In safety critical system (for example cars, aeroplanes) these actuators require monitoring in order to prevent worst-case scenario's when they do break, and have them repaired beforehand.

The architecture for the actuator monitoring system will be required to hold at least those three quality attributes:

1. Availability - To ensure it is always monitoring the actuators
2. Safety - To ensure the monitoring system does not deviate from intended behaviour (no false positives or false negative)
3. Testability - To provide certainty that of the safety and availability is should provide.



3.2 Project Life-cycle Context

The **project life cycle context** describe how the project will develop over time. The architecture is then created to adopt the life-cycle that is best for a particular project. When creating a project life-cycle the following must be complete *(these are all done best by talking about the architecture)*:

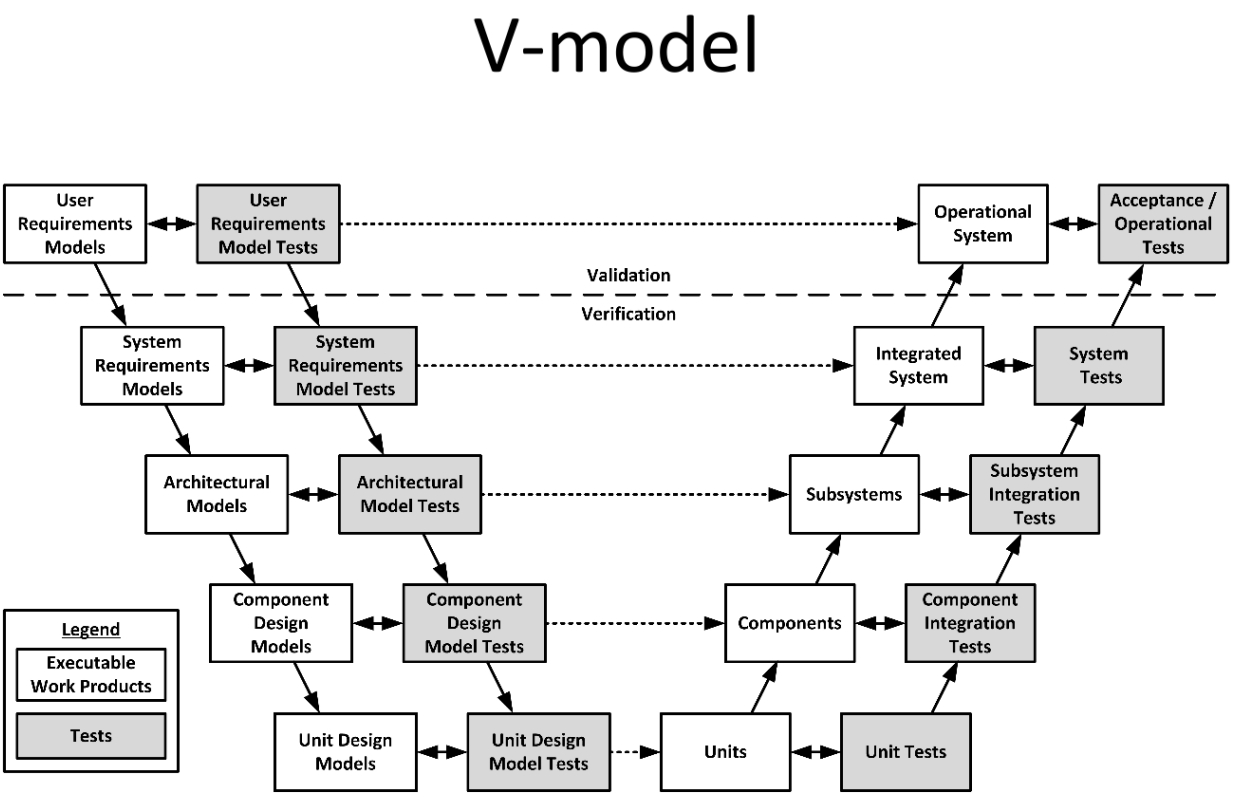
- Making a business case for the system
- Understanding the requirements that concern quality attributes

- Deciding on architecture
- Documenting architecture
- Analysing and evaluating architecture
- Implementing and testing the system based on architecture
- Ensuring the implementation conforms to the architecture

3.2.1 V-Model

In software development, the **V-model represents a development process** that may be considered an extension of the waterfall model, and is an example of the more general V-model. Instead of moving down in a linear way, the **process steps are bent upwards after the coding phase**, to form the typical V shape. The V-Model demonstrates the **relationships between each phase of the development life cycle and its associated phase of testing**. The horizontal and vertical axes represents time or project completeness (left-to-right) and level of abstraction (coarsest-grain abstraction uppermost), respectively. [Wikipedia]

The **V-Model** is a development of *waterfall* and explicitly includes architectural design as a stage. It highly focuses on **requirements based testing** all the way down to the unit level!

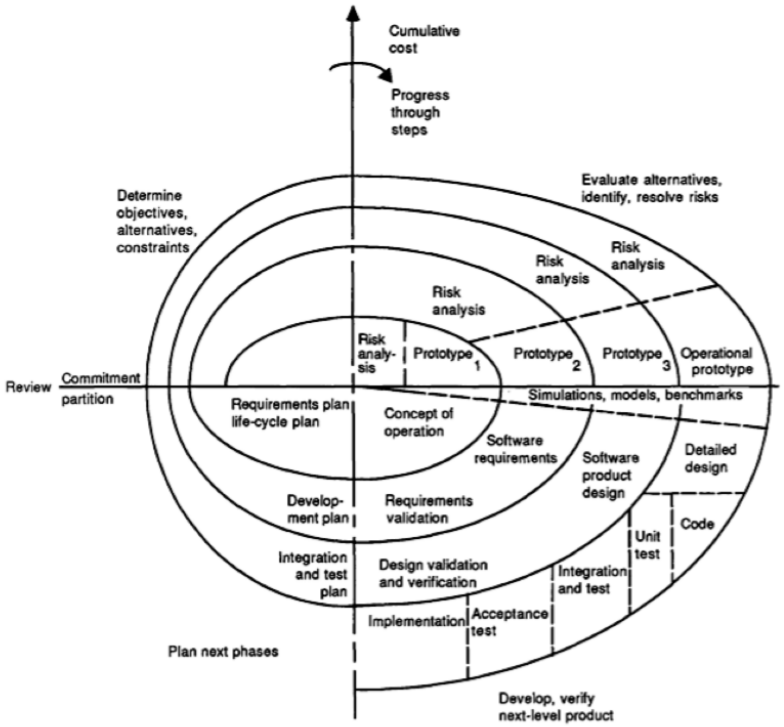


3.2.2 Spiral Model

The **spiral model is a risk-driven process** model generator for software projects. Based on the unique risk patterns of a given project, the **spiral model guides a team to adopt elements of one or more process models**, such as incremental, waterfall, or evolutionary prototyping. [Wikipedia]

The **(Boehm's spiral model** is a type of *iterative model*. It focuses on project risk management by constantly creating prototypes to be tested all the way through the development life-cycle.

Spiral Model

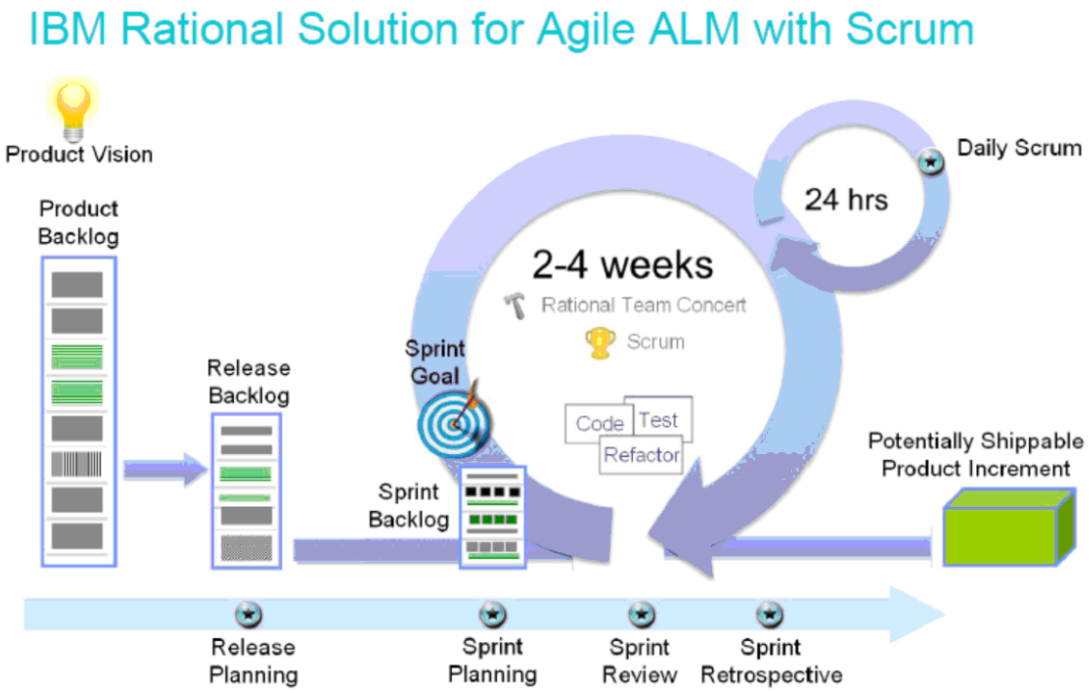


3.2.3 Agile Development

Agile software development describes a set of principles for software development under which **requirements and solutions evolve through the collaborative effort of self-organizing cross-functional teams**. It advocates **adaptive planning**, **evolutionary development**, **early delivery**, and **continuous improvemen**t, and it encourages **rapid and flexible response to change**. These principles support the definition and continuing evolution of many software development methods. [Wikipedia]

The **Agile** development life-cycle is an iterative and incremental method of managing the design and building of a software product. The image below show two different forms of **agile** development. One with and one without Devops.

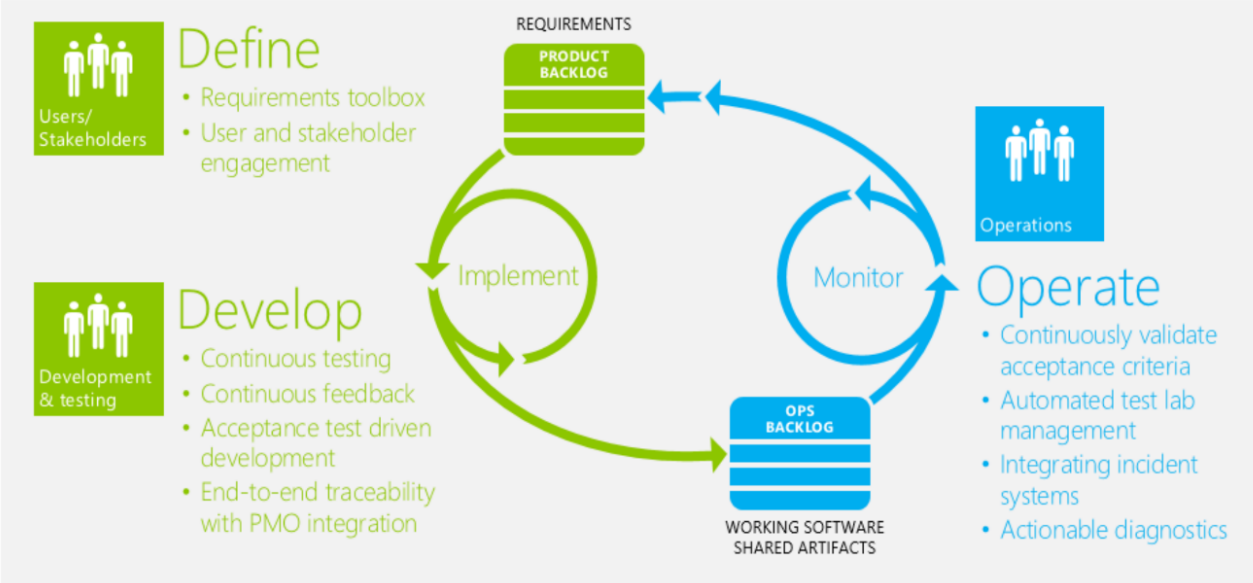
Agile



3.2.4 Agile + Devops

DevOps *(a clipped compound of "software **DEV**elopment" and "information technology **OP**eration**S**")* is a term used to refer to a set of practices that emphasize the collaboration and communication of both **software developers** and **information technology (IT) professionals** while automating the process of software delivery and infrastructure changes. It aims at establishing a culture and environment where **building, testing, and releasing software can happen rapidly, frequently, and more reliably**. [Wikipedia]

Agile + Devops



3.3 Business Context

The **business context** is discussed in later lectures. Two aspects we cover are:

1. How the organisation structure of stakeholders can drive architectural decisions and shapes decisions taking around architecture.
2. How architectural expertise drives the structure of development organisation in terms of their functional units and interrelationships.

3.4 Professional Context

The architectural perspective gives you as a professional:

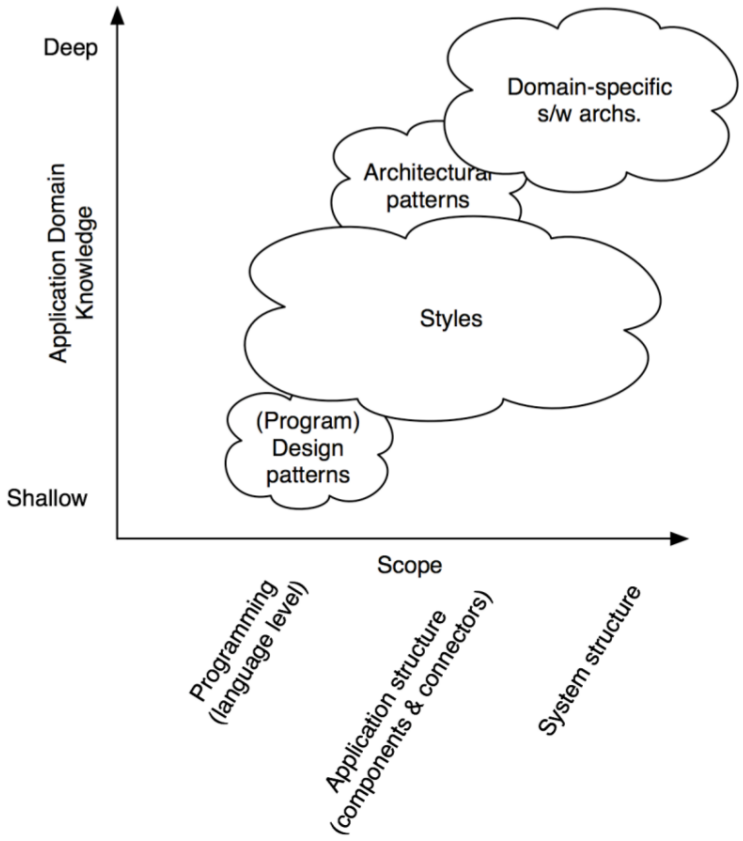
- A way of describing your expertise
- Your skills as an architect will be recognised within organisations you work within
- You can use architecture as a way of describing your past experience

- You can specialise in particular classes of architecture (e.g. financial architecture)

3.5 Domain-Specific Software Architecture

Design in the Technical Context

Design is a mixture of **creativity** and the use of **knowledge** that is institutionalised in the context. This takes the form of **reusable structures**. These reusable structures also influence other aspects of context, helping to shape **processes, organisations** and **professions**. We can plot different sorts of **architectural structures** depending on the degree to which it is **specific to a domain** and the extent to which it **influences the system**.



Domain Specific Software Architectures

DSSA is a collection of (pre-decided) **design decisions**. They capture important aspects of a particular task **(domain)** They are **common** across a range of systems in the domain and typically will have some predefined structures depending on the attributes we want to control.

These are **not** general purpose because they incorporate many specific characteristics of the **domain**. The main benefit is the extent to which **design knowledge is captured**. There are however problems, over time basic information can be forgotten.

** Bridge example given, where key information was forgotten regarding the architecture of suspension bridges (from the 19th century). This results in a bridge collapsing because of wind. **

3.6 Architectural Patterns

An architectural pattern is a set of **architectural design decisions** that are applicable to a **recurring design problem**, and **parametrized** to action for different **software development contexts** in which that problem appears.

They are similar to **DSSA** but capture less of the behaviour and attributes of the system. They are **more general** because they are intended to abstract a common **pattern over several domains**.

Three common architectural patterns that are used are listed below:

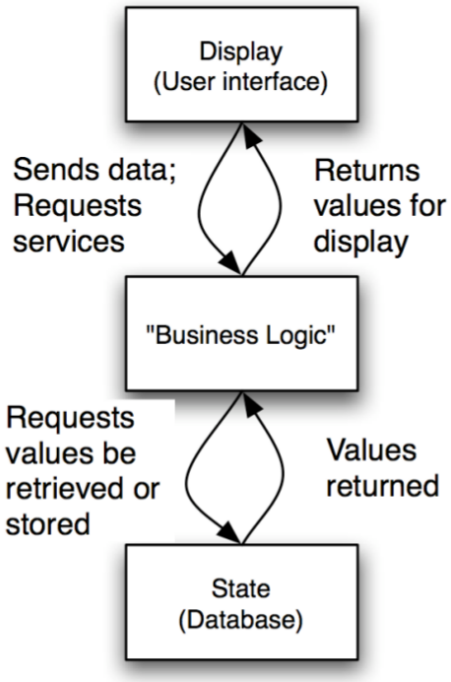
1. State Logic Display: Three-Tiered Pattern
2. Model View Controller Pattern
3. Sense Compute Control Pattern

Contexts shape design. The **technical context** identifies features we want to control and **packages** a range of other properties. Standard architectures (*patterns and domain specific architectures DSSA*) **package these**. The other context we consider also help to shape the choice of architecture.

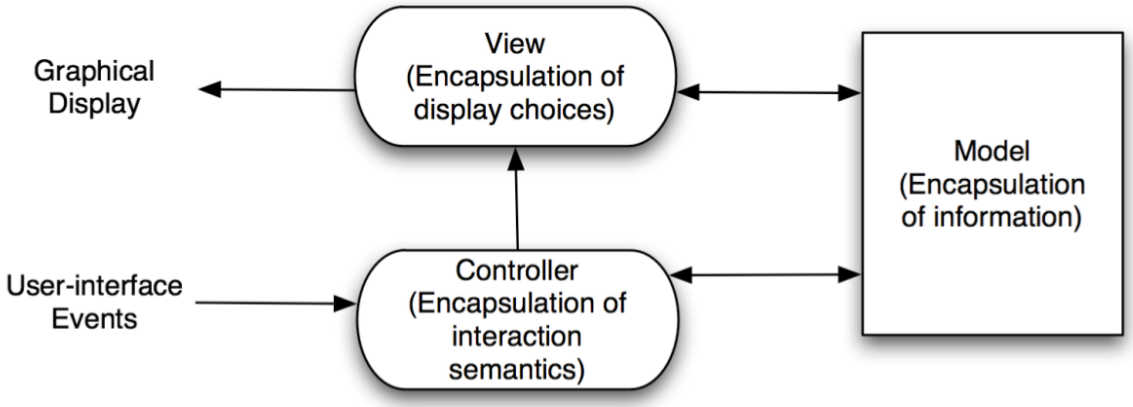
**** In design we use pre-decided strcutures and then alter/extend them as and when we need too. ****

State-Logic-Display: Three-Tiered Pattern

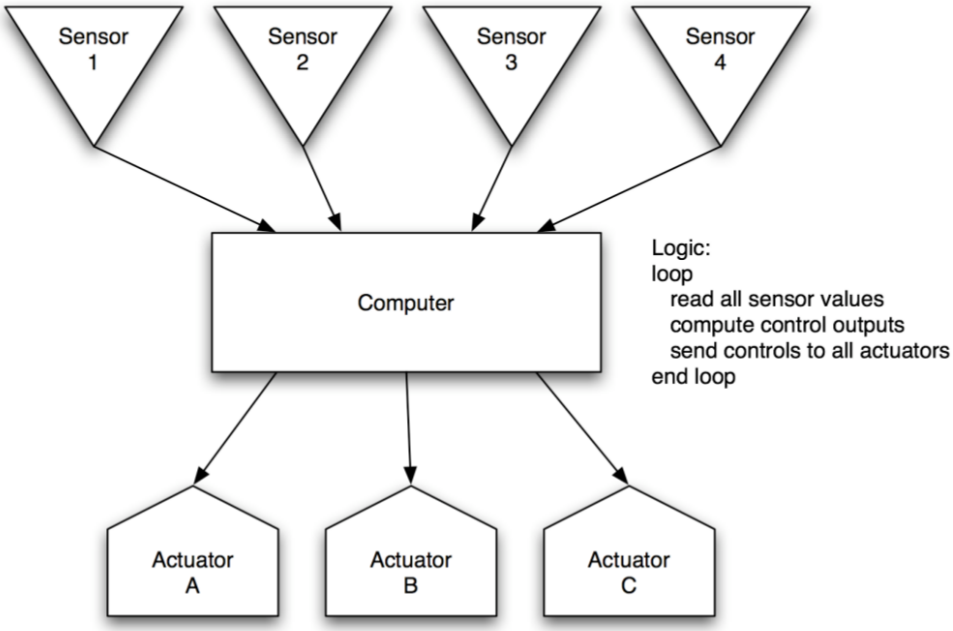
- Application Examples
  - Business applications
  - Multi-player games
  - Web-based applications



Model-View-Controller



Sense-Compute-Control



Objective: Structuring embedded control applications

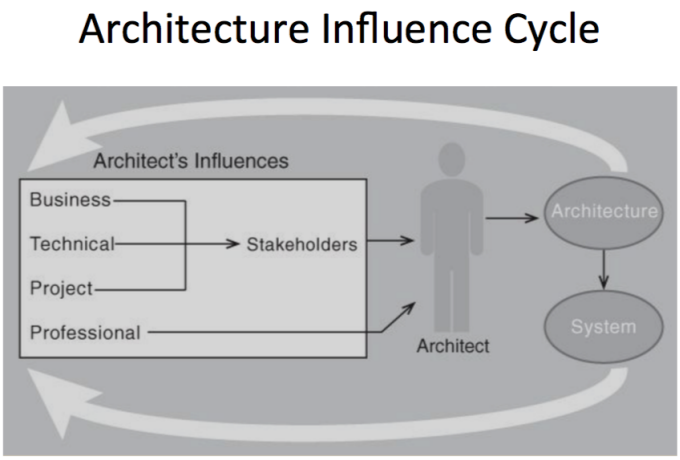
4 Quality Attributes

Quality Attributes specify, usually quantitatively, the **requirements** on particular parts of **functionality** or on the **whole system**. Software Quality Attributes are the **benchmarks that describe systems intended behaviour** within the environment for which it was built. The quality attributes provide the means for measuring the fitness and suitability of a product.

**** Quality Attributes are non-functional requirements ****

4.1 Stakeholders

Stakeholders represent different (typically conflicting) **perspectives on the system** and attempt to influence the architect. The **architect needs to trade-off the different influences and resolve the conflict.** e.g. - Marketing might want to have a big market but maintenance would prefer system to be simple.



4.2 Functional Requirements

These specify what the system does, architecture is important to them as they structure the containers that hold functionality.

**** Functional requirements are things the system does. ****

4.3 Constraints

These are decisions that have already been taken e.g. we will use the Java programming language (because we have a Java development team available) or the system will only support certain web browsers

4.4 Problems with Quality Attributes

- Often **quality attributes** are **not testable**. *(e.g. what does it man to say something is modifiable or usable or dependable or resilient?)*
- It can be difficult to **map from a concern about the system** to a quality attribute. *(For example, a high failure rate in some transaction could be a performance issue or it could be an availability issue)*
- Communities around a **particular quality attribute have developed their own terminology** *(e.g. security has attacks, performance has events etc)*

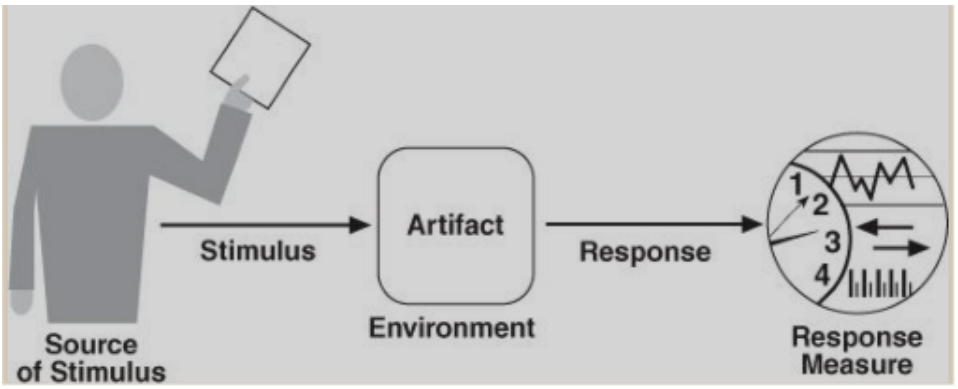
One Solution: explicitly state the use case of a quality attribute scenario, this aids the avoidance of some issues.

4.5 Quality Attribute Scenarios

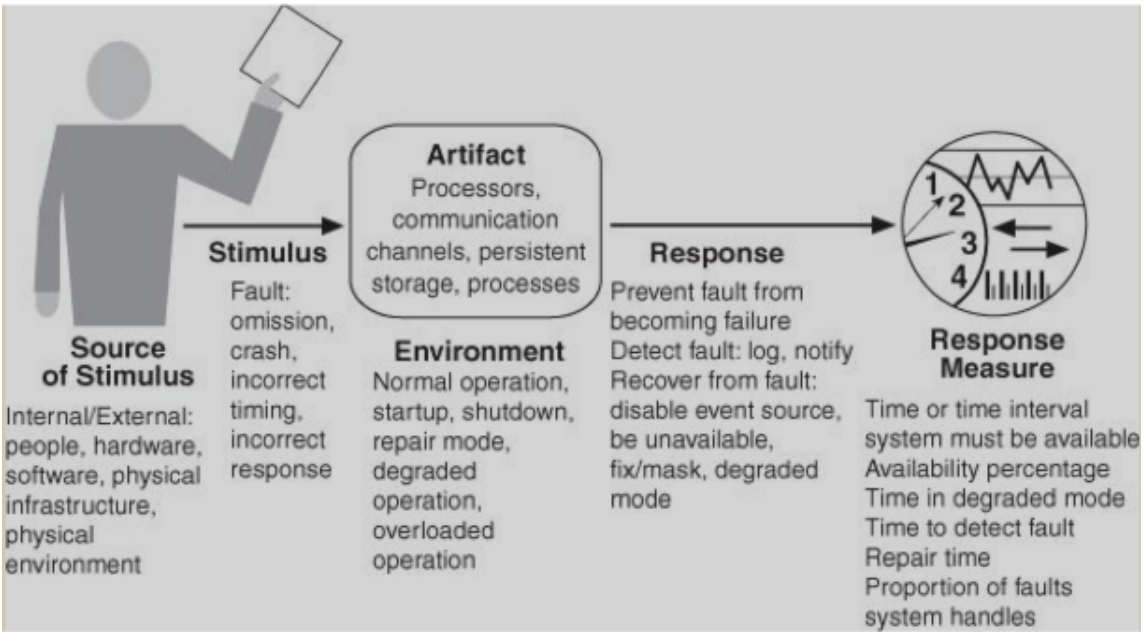
Quality attribute scenario's have the following components:

1. **Source of Stimulus**: Person or another System
2. **Stimulus**: An action the system responds to.

   (e.g. using the wrong configuration specification for the system)
3. **Environment**: Captures wider aspects of the system
4. **Artifact**: Part of the system that is stimulated
5. **Response**: Activity resulting from stimulus
6. **Response measure**: Measure of the response so that the scenario is testable *(e.g time taken to detect wrong config)*



Each quality attribute has a **general scenario** that it **tries to capture with its components**. This acts like a guide for the architect.



4.6 Architectural Tactics

Architectural tactics are a way of documenting routes to achieve a particular quality attribute (non-functional) requirement. They are a **design decision** that influence the achievement of a quality attributes **response**. *(They are more **primitive** than design patterns!).*

Tactics are based on a **single** quality attribute and do NOT take into account **trade-off's**. Usually they are more generic and need to be **specialised** to a specific context. It allows the architect to **enumerate possible design decisions**.

** If our architecture fails the scenario, because we can't detect the error arising from the fault in one of our tasks, we look at the tactics. **

4.7 Categories of Architectural Design Decisions

There are seven broad categories of **design decisions**:

1. Allocation of Responsibilities
2. Coordination Model
3. Data Model
4. Management of Resources
5. Mapping Among Architectural Elements
6. Binding Time Decisions

7. Choice of Technology

4.7.1 Allocation of responsibilities

Identify the most important responsibilities and determine how to allocate these to runtime and static elements. These are often **specific** to a quality attribute.

For example in the case of availability fault detection is an important responsibility that will be further decomposed and distributed in the architecture.

4.7.2 Coordination Model

Components in the architecture **interact with one another via a collection of mechanisms**. This is called the coordination model. Things it takes into account are:

- What **elements in the system** need to coordinate with one another.
- What **properties** *(e.g timing, security)* does the coordination need to have
- The **mechanisms and their properties** *(e.g. state fullness, synchrony, delivery guaranties)*

4.7.3 Data Model

How data is created and destroyed?

Data **access methods, operations** on the data and the **properties** of the data are all included in the Data Model.

We also need to decide on how the data would be **organised, stored, backed up** and **recovered** in case of data loss. It also includes maintaining *meta-data* that controls the interpretation of the data.

4.7.4 Management of resources

We can have both **hardware** *(e.g. CPU, memory, battery)* and **software** *(buffers, processes)* resources which need to be managed. Management includes:

- **Identification** of the resources need to be managed.
- The **system element** should manage a resource.
- Work out **sharing strategies** and how to arbitrate (resolve) in **contention situations**
- Consider the **consequences of running out of a resource** (e.g. Memory).

4.7.5 Mapping Among Architectural Elements

We have 2 types of mapping:

- **Mapping between different types of elements in the architecture**
  e.g. between static development structures and threads or processes
- **Mapping between software elements and environment elements**
  e.g. from process to specific processors.

Some important mappings:

- Code → runtime structures
- Runtime elements → environment
- Data model elements → data stores

4.7.6 Binding Time Decisions

Binding time decisions **introduce allowable ranges of variation**. This variation can be bound at **different times in the software life cycle** by different entities from design time by a developer to runtime by an end user.

A binding time decision establishes the scope, the point in the life cycle, and the mechanism for achieving the variation.

The decisions in the other six categories have an associated binding time decision. **Examples of such binding time decisions include the following**:

- For **allocation of responsibilities**, you can have build-time selection of modules via a parameterized makefile.
- For **choice of coordination model**, you can design runtime negotiation of protocols.
- For **resource management**, you can design a system to accept new peripheral devices plugged in at runtime, after which the system recognizes them and downloads and installs the right drivers automatically.
- For **choice of technology**, you can build an app store for a smartphone that automatically downloads the version of the app appropriate for the phone of the customer buying the app.

When making **binding time decisions**, you should consider the **costs to implement the decision** and the **costs to make a modification** after you have implemented the decision.

*For **example**, if you are considering changing platforms at some time after code time, you can insulate yourself from the effects caused by porting your system to another platform at some cost. Making this decision depends on the costs incurred by having to modify an early binding compared to the costs incurred by implementing the mechanisms involved in the late binding.*

4.7.7 Choice of technology

Every architecture decision must eventually be realized using a specific technology. This is completed by completing the following points below:

** Additional Information ** *Sometimes the technology selection is made by others, before the intentional architecture design process begins. In this case, the chosen technology becomes a constraint on decisions in each of our seven categories. In other cases, the architect must choose a suitable technology to realize a decision in every one of the categories.*

- Deciding which **technologies are available** to realize the decisions made in the other categories
- Determining whether the **available tools to support this technology choice** are adequate for development to proceed.

  (IDEs, simulators, testing tools, etc.)
- Determining the extent of **internal familiarity as well as the degree of external support available** for the technology and deciding whether this is adequate to proceed.

  Such as courses, tutorials, examples, and availability of contractors who can provide expertise in a crunch
- Determining the side effects **(consequences)** of choosing a technology

  Such as a required coordination model or constrained resource management opportunities.
- Determining whether a **new technology is compatible** with the existing technology stack.

  For example, can the new technology run on top of or alongside the existing technology stack? Can it communicate with the existing technology stack? Can the new technology be monitored and managed?

5 QA: Availability

Some systems we need to be there whenever they need to be used. These are usually called **high availability systems**. **(Examples)** There can be different reasons for high availability:

- 999 telephone system
- Interplanetary spacecraft systems
- Electricity supply grid
- Large Computer System Power Supply

From Hardware, there are two key **measures** of availability:

- MTBF: Mean Time Between Failures
- MTTR: Mean Time To Repair

Availability is the probability of the system working, when you ask it to work.

$$availability = \frac{MTBF}{MTBF + MTTR}$$

To maximise the probability, either make the **time between failures larger**, or a **shorter repair time**.

5.1 Fault, Error, and Failure

A **fault** is something in the system.
(e.g. a broken wire, failed component, wrong bit of code ...)
(Example) A fault in a sorting routine means that under some circumstances it fails to sort an array.

The system moves into an **error** state when the fault is activated.
Under these conditions, the system might be assuming an array is sorted but it isn't. In this state there is an error in the system because things are not as they should be.

Failure is the externally observable deviation from intended operation, this can be caused by an error.
If the system uses binary search to look for things in the array, sometimes an item will be in the array but will not be found – this might cause a visible failure of the system.

** Most **high availability systems** try to tolerate or mask faults by detecting erroneous conditions before they move into failure conditions. **

5.2 General Scenario

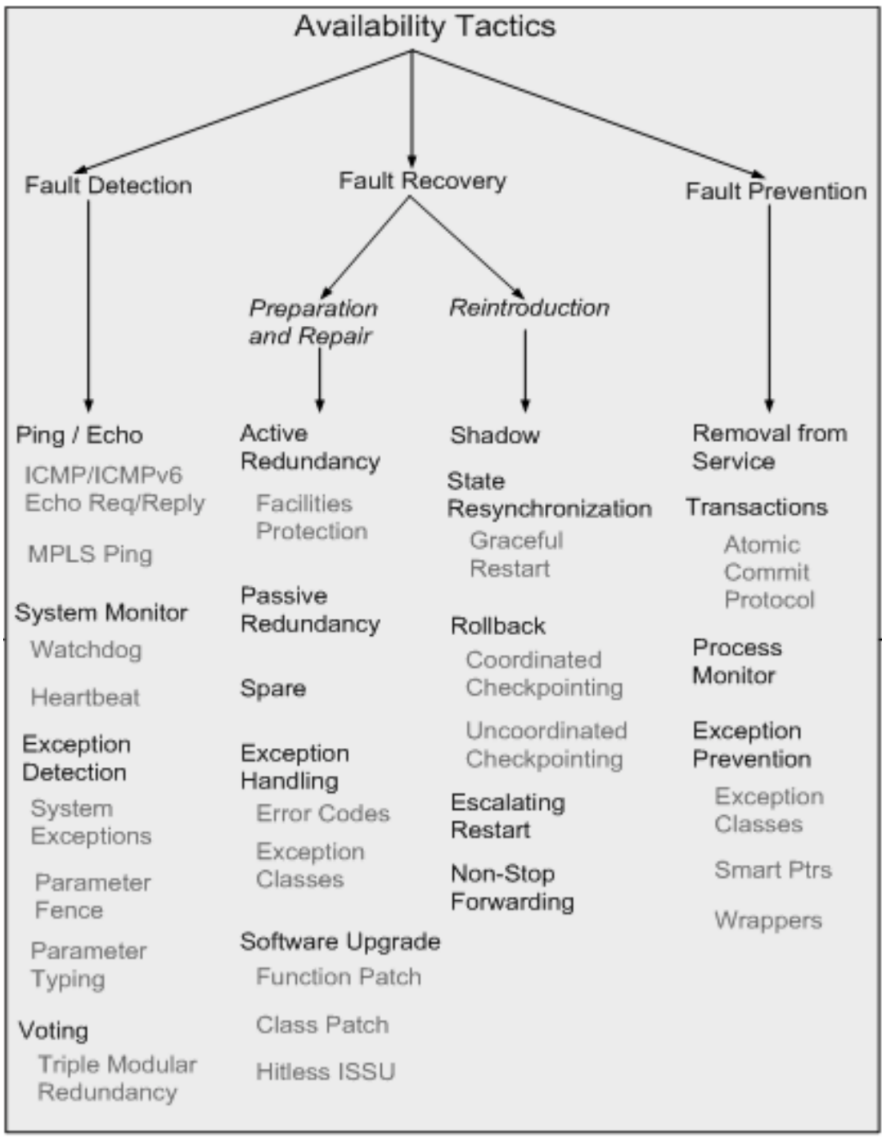
- **Source**: Internal or external sources important to differentiate because different measures are possible.
- **Stimulus**: Fault causes errors: omission (no result), crash (repeated omissions), timing (late, early), response (incorrect value).
- **Artifact**: Specifies what has to be available: process, channel, store, ...
- **Environment**: what the mode of operation is: normal, degraded, startup, shutdown, ...
- **Response**: how to respond to the stimulus
- **Response measure**: this will be some measure related to the availability or the "liveness" of the artifact

5.3 Concrete Scenario

In **mission critical systems** there is typically a schedule that activates a sequence of tasks in turn. These take longer or shorter times to complete and the whole set is carried out cyclically. What happens if there is a bug in a task and it never completes?

- Cycles through each of the tasks.
- Passes control to one of the tasks
- Waits for control to pass back.
- If one of the task fails, the architecture fails the scenario.

5.4 Availability Tactics



5.4.1 Fault Detection

- **Ping/echo.** One component issues a ping and expects to receive back an echo, within a predefined time, from the component under scrutiny. This can be used within a group of components mutually responsible for one task. It can also used be used by clients to ensure that a server object and the communication path to the server are operating within the expected performance bounds. "Ping/echo" fault detectors can be organized in a hierarchy, in which a lowest-level detector pings the software processes with which it shares a processor, and the higher-level fault detectors ping

lower-level ones. This uses less communications bandwidth than a remote fault detector that pings all processes.

- **Heartbeat (dead man timer).** In this case one component emits a heartbeat message periodically and another component listens for it. If the heartbeat fails, the originating component is assumed to have failed and a fault correction component is notified. The heartbeat can also carry data. For example, an automated teller machine can periodically send the log of the last transaction to a server. This message not only acts as a heartbeat but also carries data to be processed.
- **Exceptions.** One method for recognizing faults is to encounter an exception, which is raised when one of the fault classes we discussed in Chapter 4 is recognized. The exception handler typically executes in the same process that introduced the exception.

5.4.2 Fault Recovery

- **Voting.** Processes running on redundant processors each take equivalent input and compute a simple output value that is sent to a voter. If the voter detects deviant behavior from a single processor, it fails it. The voting algorithm can be "majority rules" or "preferred component" or some other algorithm. This method is used to correct faulty operation of algorithms or failure of a processor and is often used in control systems. If all of the processors utilize the same algorithms, the redundancy detects only a processor fault and not an algorithm fault. Thus, if the consequence of a failure is extreme, such as potential loss of life, the redundant components can be diverse.
- **Active redundancy (hot restart).** All redundant components respond to events in parallel. Consequently, they are all in the same state. The response from only one component is used (usually the first to respond), and the rest are discarded. When a fault occurs, the downtime of systems using this tactic is usually milliseconds since the backup is current and the only time to recover is the switching time. Active redundancy is often used in a client/server configuration, such as database management systems, where quick responses are necessary even when a fault occurs. In a highly available distributed system, the redundancy may be in the communication paths. For example, it may be desirable to use a LAN with a number of parallel paths and place each redundant component in a separate path. In this case, a single bridge or path failure will not make all of the system's components unavailable.
- **Passive redundancy (warm restart/dual redundancy/triple redundancy).** One component (the primary) responds to events and informs the other components (the standbys) of state updates they must make. When a fault occurs, the system must first ensure that the backup state is sufficiently fresh before resuming services. This approach is also used in control systems, often when the inputs come over communication channels or from sensors and have to be switched from the primary to the backup on failure. Chapter 6, describing an air traffic control example,

shows a system using it. In the air traffic control system, the secondary decides when to take over from the primary, but in other systems this decision can be done in other components. This tactic depends on the standby components taking over reliably. Forcing switchovers periodically-for example, once a day or once a week-increases the availability of the system. Some database systems force a switch with storage of every new data item. The new data item is stored in a shadow page and the old page becomes a backup for recovery. In this case, the downtime can usually be limited to seconds.

- **Spare.** A standby spare computing platform is configured to replace many different failed components. It must be rebooted to the appropriate software configuration and have its state initialized when a failure occurs. Making a checkpoint of the system state to a persistent device periodically and logging all state changes to a persistent device allows for the spare to be set to the appropriate state. This is often used as the standby client workstation, where the user can move when a failure occurs. The downtime for this tactic is usually minutes.
- **Shadow operation.** A previously failed component may be run in "shadow mode" for a short time to make sure that it mimics the behaviour of the working components before restoring it to service.
- **State resynchronization.** The passive and active redundancy tactics require the component being restored to have its state upgraded before its return to service. The updating approach will depend on the downtime that can be sustained, the size of the update, and the number of messages required for the update. A single message containing the state is preferable, if possible. Incremental state upgrades, with periods of service between increments, lead to complicated software.
- **Checkpoint/roll-back.** A checkpoint is a recording of a consistent state created either periodically or in response to specific events. Sometimes a system fails in an unusual manner, with a detectably inconsistent state. In this case, the system should be restored using a previous checkpoint of a consistent state and a log of the transactions that occurred since the snapshot was taken.

5.4.3 Fault Prevention

- **Removal from service.** This tactic removes a component of the system from operation to undergo some activities to prevent anticipated failures. One example is rebooting a component to prevent memory leaks from causing a failure. If this removal from service is automatic, an architectural strategy can be designed to support it. If it is manual, the system must be designed to support it.
- **Transactions.** A transaction is the bundling of several sequential steps such that the entire bundle can be undone at once. Transactions are used to prevent any data from being affected if one step in a process fails and

also to prevent collisions among several simultaneous threads accessing the same data.

- **Process monitor.** Once a fault in a process has been detected, a monitoring process can delete the nonperforming process and create a new instance of it, initialized to some appropriate state as in the spare tactic.

5.5 WatchDog (Tactic Example)

A **watchdog timer** *(sometimes called a computer operating properly or COP timer, or simply a watchdog)* is an electronic timer that is used to detect and **recover from computer malfunctions**. During normal operation, the computer regularly resets the watchdog timer to prevent it from elapsing, or "timing out". If, due to a hardware fault or program error, the computer fails to reset the watchdog, the timer will elapse and generate a timeout signal. The timeout signal is used to initiate corrective action or actions. The corrective actions typically include placing the computer system in a safe state and restoring normal system operation.

Use-Cases: Embedded Systems, Space Probes, or any system where humans cannot easily access the equipment or would be unable to react to faults in a timely manner.

5.6 Architectural Design Decisions (for availability)

- Allocation of responsibilities
- Coordination model
- Data Model
- Management of resources
- Mapping among architectural elements
- Binding time decisions
- Choice of technology

5.6.1 Allocation of Responsibilities

- Determine what needs to be high availability (maybe not all functions).
- Responsibility for detecting error (and possible cause).
- Responsibility to log errors
- Responsible respond to a detected error
- Manage sources of events
- Decide on mode of operation

- Decide on how to repair faults
- **In the previous case, assign a watchdog and allocate its responsibility of responding to error**

5.6.2 Coordination Model

- Are the error detection capabilities of the coordination model adequate to detect errors?
- Is the coordination model sufficient to ensure communication and coordination between error detection, log and response.?
- Will coordination work in the presence of error, degraded modes?
- If repair involve replacement of elements will the coordination model allow this?
- **In our example the wakeup between watchdog and controller might be an addition to the coordination mechanism.**

5.6.3 Data Model

- How do error conditions affect the data model?
- Does this mean we have to deal with some forms of corrupt data or incomplete operations?
- Perhaps the data model needs to be extended to include new operations to recover from failed earlier operations.
- **For example, extending the model with checkpoint and rollback operations may be enough in some situations.**

5.6.4 Management of Resources

- See what resources are essential to maintain operation in the presence of errors.
- Identify what resources are necessary for meaningful degraded modes.
- Work out if different scheduling changes the demand on critical resources.
- **In our example if task 1 is in error because of a bad processor and task 4 is OK but not necessary for some degraded mode it may be best to switch task 1 and 4 and never schedule task 4 again to provide a degraded mode of operation.**

5.6.5 Mapping among architectural elements

- Determining what resources might be in error or might be affected by errors.

- Checking that remapping of elements is possible dynamically.
- How fast can elements be restarted or reinitialised, can a process be moved to a new processor ...
- **In our example it may be necessary to identify the watchdog as a new element and that a failing task may need to be mapped to a different processor.**

5.6.6 Binding Time Decisions

- Look at binding time and see where this will allow flexibility.
- For example, if we can tolerate a 0.5s delay on a response but are currently using 0.1s as the time to signal an error then we might want to rebind and operate in a degraded mode.
- **In our example, if the the taks code is burned into PROMS on the processors there is no chance to rebind task/processor.**

5.6.7 Choice of Technology

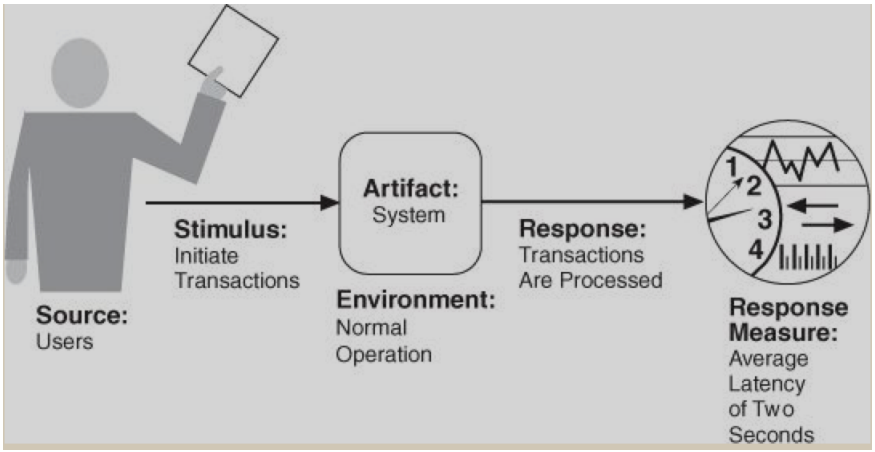
- Explore technologies that package useful functionality for availability.
- Use an established element if it is available.
- Use already established data on the availability characteristics of components.

6 QA: Performance

6.1 General Scenario

Portion Of Scenario	Possible Values
No Dropout, Baseline	
Source	Internal or External to the System
Stimulus	Arrival of a periodic, sporadic or a stochastic event
Artifact	System or one or more components in the system
Environment	Operational Mode: Normal, Emergency, Peak, Overload
Response	Process Events, Changed level of service
Response Measure	Latency, deadline, throughput, jitter, miss rate

6.2 Concrete Scenario



We need to say something about the distribution of the arrival of the stimuli.

- E.g. The inter-arrival time is always greater than 1.0 secs
- How is this different from the arrival rate is less than 1 per second?

  Any stimulus needs to be processed within 2 seconds of arriving. The responses should appear in the same order as the stimuli.

6.3 A Possible Architecture

Queue → Process → Output

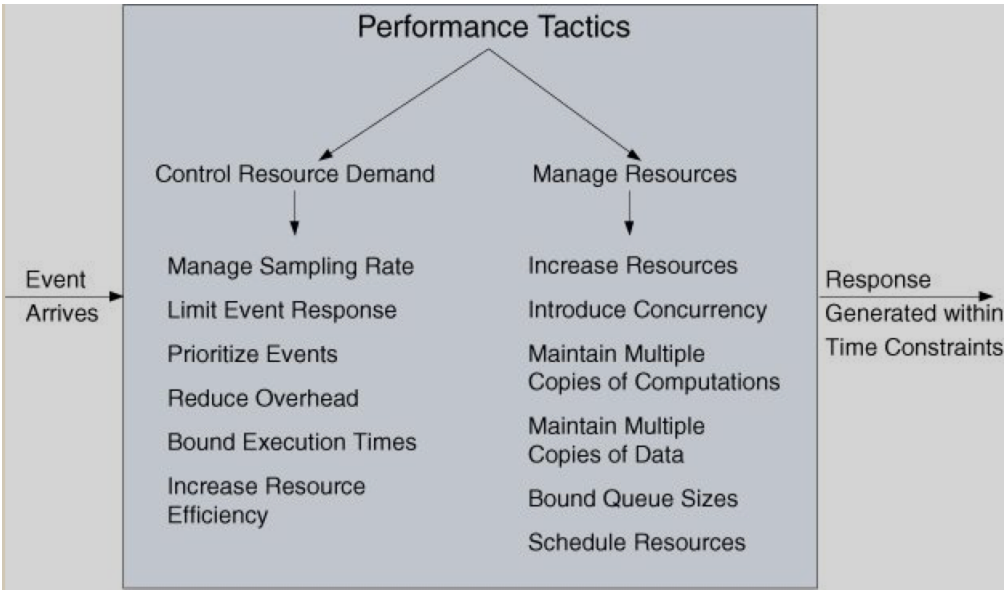
We need to say something about the capacity of the processor:

- The worst case processing time for a stimulus is 1.5 seconds best case time is 1.0 secs

- The processor can only process one stimulus at a time. We need to say that the queue capacity is 7 stimuli (or some other). This architecture fails the scenario

  Because the arrival rate is greater than the processing rate therefore a queue $>$ 7 is inevitable

6.4 Performance Tactics



6.4.1 Control Resource Demand

- **Increase computational efficiency.** One step in the processing of an event or a message is applying some algorithm. Improving the algorithms used in critical areas will decrease latency. Sometimes one resource can be traded for another. For example, intermediate data may be kept in a repository or it may be regenerated depending on time and space resource availability. This tactic is usually applied to the processor but is also effective when applied to other resources such as a disk.
- **Reduce computational overhead.** If there is no request for a resource, processing needs are reduced. In Chapter 17, we will see an example of using Java classes rather than Remote Method Invocation (RMI) because the former reduces communication requirements. The use of intermediaries (so important for modifiability) increases the resources consumed in processing an event stream, and so removing them improves latency. This is a classic modifiability/performance tradeoff.
- **Manage event rate.** If it is possible to reduce the sampling frequency at which environmental variables are monitored, demand can be reduced. Sometimes this is possible if the system was over engineered. Other times

an unnecessarily high sampling rate is used to establish harmonic periods between multiple streams. That is, some stream or streams of events are oversampled so that they can be synchronized.

- **Control frequency of sampling.** If there is no control over the arrival of externally generated events, queued requests can be sampled at a lower frequency, possibly resulting in the loss of requests.
- **Bound execution times.** Place a limit on how much execution time is used to respond to an event. Sometimes this makes sense and sometimes it does not. For iterative, data-dependent algorithms, limiting the number of iterations is a method for bounding execution times.
- **Bound queue sizes.** This controls the maximum number of queued arrivals and consequently the resources used to process the arrivals.

6.4.2 Manage Resources

- **Introduce concurrency.** If requests can be processed in parallel, the blocked time can be reduced. Concurrency can be introduced by processing different streams of events on different threads or by creating additional threads to process different sets of activities. Once concurrency has been introduced, appropriately allocating the threads to resources (load balancing) is important in order to maximally exploit the concurrency.
- **Maintain multiple copies of either data or computations.** Clients in a client-server pattern are replicas of the computation. The purpose of replicas is to reduce the contention that would occur if all computations took place on a central server. Caching is a tactic in which data is replicated, either on different speed repositories or on separate repositories, to reduce contention. Since the data being cached is usually a copy of existing data, keeping the copies consistent and synchronized becomes a responsibility that the system must assume.
- **Increase available resources.** Faster processors, additional processors, additional memory, and faster networks all have the potential for reducing latency. Cost is usually a consideration in the choice of resources, but increasing the resources is definitely a tactic to reduce latency. This kind of cost/performance trade-off is analysed in Chapter 12.

Resource Arbitration Scheduling policy. → (FIFO, Fixed-priority, Dynamic priority, Static)

6.4.3 Control Resource Demand (old version)

- **Manage the sampling rate** (not always applicable) – ensure you do not have too much to handle.
- **Limit the event response** – if you are receiving too many events, throw some away.
- **Prioritize events** – some need a respons in a certain time – some don't

- **Reduce overhead** – can you take resource out of handling an event?
- **Improve the efficiency of processing** – so you can handle more with the same processing

6.5 Architectural Design Decisions (for performance)

- Allocation of responsibilities
- Coordination model
- Data Model
- Management of resources
- Mapping among architectural elements
- Binding time decisions
- Choice of technology

6.5.1 Manage Resources

- Increase resources
- Introduce concurrency
- Maintain multiple copies of compute and/or data
- Bound queue sizes
- Schedule resource when there is contention (hard scheduling for highest priority events)

6.5.2 Allocation of Responsibilities

- Work out areas responsibility of that require heavy resource use to ensure time-critical events take place.
- Work out processing requirements.
- Take account of:
  - Responsibilities arising from threads crossing boundaries of responsibility
  - Responsibilities for thread management
  - Responsibilities for scheduling shared resources

6.5.3 Coordination Model

- What needs to coordinate.
- Is there concurrency? Ensure it is safe.
- Ensure coordination is appropriate for the style of stimulus.
- Ensure the properties of the coordination model are good for the stimuli and concurrency control?

6.5.4 Data Model

- Determine what parts of the data model will be heavily loaded or have tight time constraints.
  - Would keeping multiple copies help?
  - Would partitioning the data help?
  - Is it possible to reduce processing requirements for the data?
  - Does adding resource help deal with data bottlenecks?

6.5.5 Mapping Among Architecture Elements

- Does collocation of some components reduce latencies?
- Ensure components with high processing needs are allocated to big processors
- Consider introducing concurrency when you map.
- Consider whether some mappings introduce bottlenecks (e.g. allocating non-interfering tasks to the same thread)

6.5.6 Resource Management

- Work out what needs high levels of resource
- Ensure these are monitored and managed under all operating modes.
- For example:
  - Time critical components
  - Thread management
  - Prioritization
  - Locking and scheduling strategies
  - Deploying additional resource to meet elevated load.

6.5.7 Binding Time

- Look at when you bind.
- Consider the cost of binding at different times.
- Try to avoid performance penalties caused by late binding.

6.5.8 Choice of Technology

- Is the technology right to let you meet hard deadlines and resource use (e.g. use a real-time OS with proper scheduling).
- You need:
  - Good scheduling
  - Priorities
  - Policies for demand reduction
  - Allocating processing to tasks
  - Other performance-related measurement and management.

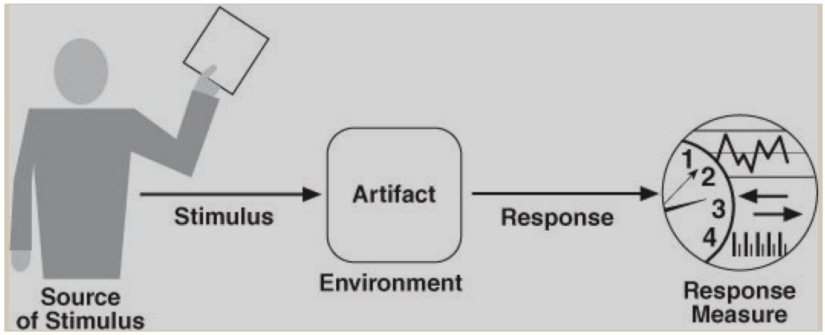
7 QA: Security

We need to consider attacks on **confidentiality**, **integrity** and **availability**. Attacks need to be **monitored** and **detected** to allow them to be **resisted where possible**, otherwise we may need some other form of reaction. In the ideal situation we will eventually be able to **fully recover**.

7.1 Important Quality Attributes (for security)

1. **Confidentiality** - Only those who should have access are given access
2. **Integrity** - Data or services are not subject to unauthorised manipulation.
3. **Availability** - The system is available for legitimate use.
4. **Security Mechanisms** Authentication, Authorisation, non-repudiation

7.2 General Scenario



- **Source** - Humans or systems that may or may not have been identified and can be either inside or outside the organisation
- **Stimulus** - Unauthorized attempt to access, manipulate or disable the artifact.
- **Artifact** - System services, data, components, data produced or consumed by the system.
- **Environment** - online, offline, connected to network, disconnected to network, behind firewall, fully/partially operating, not operational
- **Response** - Transaction carried out such that:
  - Data or services protected from unauthorised access
  - No data manipulation without authorization
  - Parties in a transaction are identified with assurance
  - Parties cannot repudiate their participation
  - Data resources etc ... are available for legitimate use

  - Appropriate people are notified when threat is identified

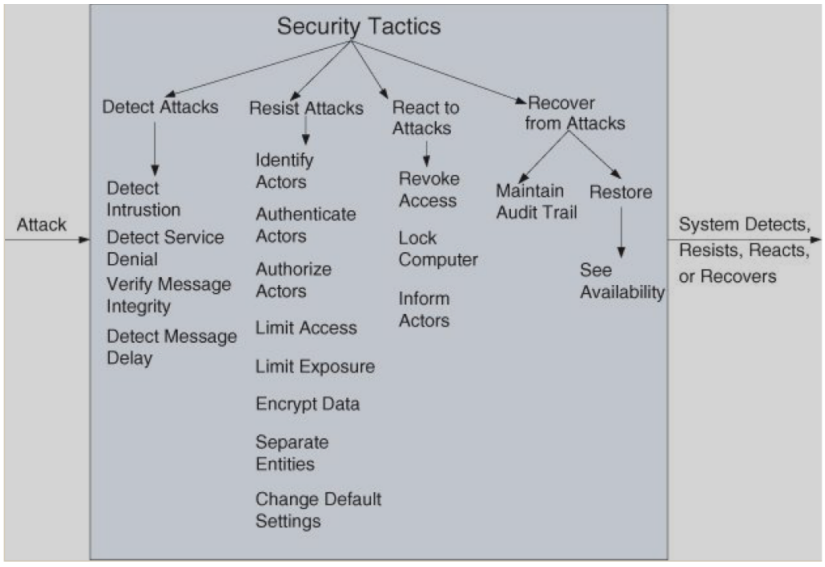
- **Response Measure** - assessment of the degree of compromise, temporal and spatial data on the compromise, how many attacks were resisted, how much data is vulnerable

7.3 Concrete Scenario

Assume a more concrete example of **Distributed Denial of Service** (DDOS). The components could be:

- **Source**- A wide range of systems with different IP.
- **Stimulus**- Access to the service provided
- **Artifact**- The service we are concerned with
- **Environment**- Normal operation
- **Response**- Detect Normal Load
- **Response Measure**- Mode of operation is changed to ensure normal service to trusted IP addresses (block those doing DDOS)

7.4 Security tactics



7.5 Architectural Design Decisions (for security)

- Allocation of responsibilities
- Coordination model
- Data Model

- Management of resources
- Mapping among architectural elements
- Binding time decisions
- Choice of technology

7.5.1 Manage Resource

- Explore the overheads resulting from responsibilities (logging, encryption, recovery etc).
- Analyse how a user can make demands on a critical resource.
- Make sure malicious use of resources is detected and managed.
- Identify and manage the potential for corruption/contamination.
- Explore the potential for resource use to be used as a covert channel to transmit data.
- Limit resources used to manage attempts at unauthorised use.

7.5.2 Allocation of Responsibilities

For security-system responsibilities do the following:

- Ensure all actors have identities (like roles)
- Authenticate identities to appropriate actors.
- Check and Ensure Authorization
- Ensure Data Encryption
- Log attempts, successes, failures and other sensitive operations.

7.5.3 Coordination Model

The following things in the coordination model need to be accounted for:

- Ensure coordination mechanisms use authentication and
- Ensure the coordination model is not vulnerable to tampering, interception, impersonation.
- The model data involved is encrypted
- Monitor level of demand for communication to identify excessive demands

7.5.4 Data Model

- Ensure there is a valid data model that disallows invalid data flows
- Ensure logging of access, modification and attempted access or modification
- Data is protected in flight at rest using appropriate encryption
- Ensure appropriate backup/recovery mechanisms are in place

7.5.5 Mapping Among Architectural Elements

- Explore and be wary how different mappings change the way users can access resources
- Ensure for all the mappings, the responsibilities (authorization, logging,encryption etc ...) are preserved
- Ensure recovery from attack is possible

7.5.6 Binding Time

- Explore the consequences of varying binding times on the ability to trust actor or components.
- place mechanisms to ensure trust given in binding time.
- Explore impact on resource use, capacity/throughput, response time.
- Ensure the time bindings are ensured with all responsibilities (authorization, logging,encryption etc...)
- Explore the potential of variation in binding time as a covert channel

7.5.7 Choices of Technologies

- Ensure limitations of technologies are understood and the potential for future compromise is well identified
- Ensure your chosen technologies support the tactics you want to deploy to protect the system

8 QA: Testability

Testability is concerned with **ensuring the software architecture eases the work of testers**. The way in which the software is designed should make it easy for **precise tests** to be developed to measure how successful the software was made.

- **Not the same as testing itself**
- Software architecture is structured in a way so that desired testing can take place
- Testability is often **determined by the code structure itself** rather than the connector/component or deployment view
- Consider the code modules and the dependencies between them that are used to build up the components

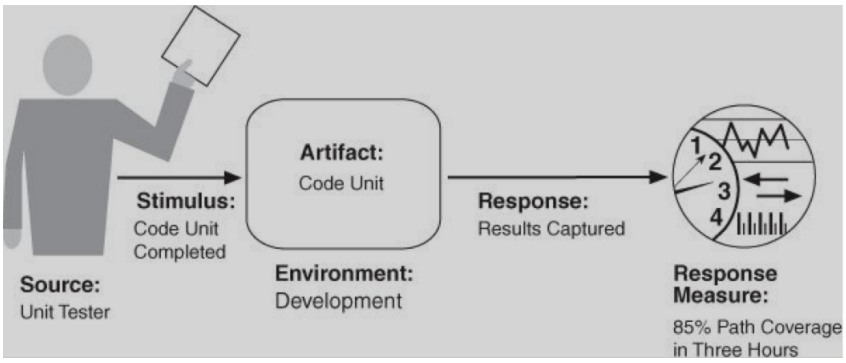
8.1 General Scenario

- **Source:** Can think of it as who is doing the testing:
  - unit testers
  - integration testers
  - system testers
  - acceptance testers
  - end users

  These tests can be run manually or using automated testing tools
- **Stimulus:** Why is the set of tests being executed? Could be due to:
  - completion of a certain part of the code, or of an entire system
  - complete integration of subsystem into larger system
  - completion of entire system
  - time to deliver the system to the customers
- **Environment:** Which environment we are dealing with:
  - Design time
  - Development time
  - Compile time
  - Integration time
  - Deployment time
  - Run time
- **Artifacts:** The portion of the system that is being tested

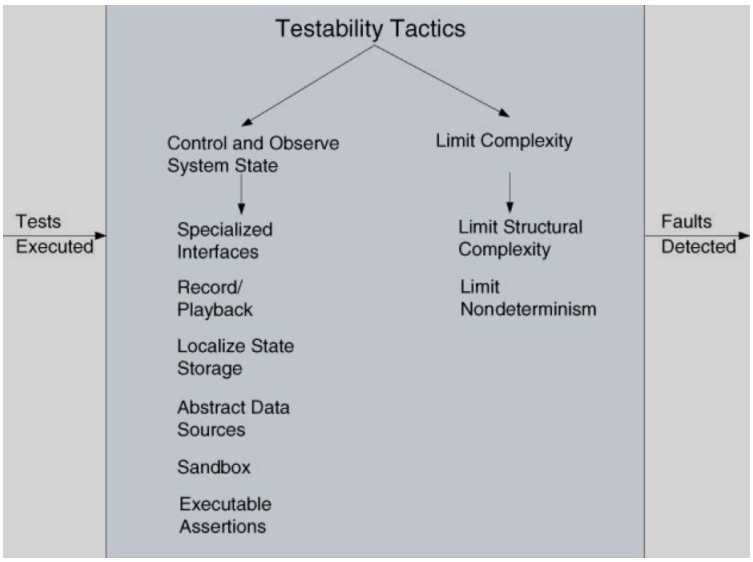
- **Response:** What is the outcome of running the tests:
  - Execute test suite and capture results
  - Capture activity that resulted in the fault
  - Control and monitor the state of the system
- **Response Measure:** How exactly we will measure the results and if we were successful in the testing Could be:
  - Find a fault or type of faults
  - Achieve a certain percentage of code coverage
  - Time limit for running the tests

8.2 Concrete Scenario



- **Source** - Regression tester
- **Stimulus** - Completion of maintenance development to repair a critical bug
- **Artifact** - Modules for the full system
- **Environment** - Maintenance development
- **Response** - Results from path coverage tool
- **Response Measure** - Path coverage is better than 95% of non-looping paths inside modules

8.3 Testability Tactics



The goal of tactics for testability is to allow for easier testing when an increment of software development has completed. Anything the architect can do to reduce the high cost of testing will yield a significant benefit. There are two categories of tactics for testability:

- Adding controllability and observability to the system.
- Limiting complexity in the system's design

8.3.1 Category 1 - Control and Observe System State

- **Specialized Interfaces:** to control or capture variable values for a component either through a test harness or through normal execution.
- **Record/Playback:** capturing information crossing an interface and using it as input for further testing.
- **Abstract Data Sources:** Abstracting the interfaces lets you substitute test data more easily.
- **Sandbox:** isolate the system from the real world to enable experimentation that is unconstrained by the worry about having to undo the consequences of the experiment
- **Executable Assertions:** assertions are (usually) hand coded and placed at desired locations to indicate when and where a program is in a faulty state - basically when you want to check

8.3.2 Category 2 - Limit Complexity

- **Limit Structural Complexity:** Avoiding or resolving cyclic dependencies between components, isolating and encapsulating dependencies on the external environment, and reducing dependencies between components in general.
- **Limit Non-determinism:** finding all the sources of non-determinism, such as unconstrained parallelism, and weeding them out as far as possible.

8.4 Architectural Design Descisions (for testability)

During testing, tester has to control and observe the state of the system.

- Allocation of responsibilities
- Coordination model
- Data Model
- Management of resources
- Mapping among architectural elements
- Binding time decisions
- Choice of technology

8.4.1 Allocation of responsibilities

This means determining which system responsibilities are most critical and thus need the most testing. Make sure that system responsibilities have been created to actually perform the testing and capture the results, capture (log) activity that resulted in fault or unexpected behaviour, control and observe relevant system state for testing.

8.4.2 Coordination Model

Ensure system's coordination and communication mechanisms support the execution of the test suite, support capturing activity that resulted in a fault, support injection and monitoring of state, and do not introduce necessary nondeterminism

8.4.3 Data Model

Determine the major data abstractions that must be tested to ensure the correct operation of the system.

8.4.4 Mapping Among Architectural Elements

Determine how to test possible mappings of architectural elements so that the desired test response is achieved.

8.4.5 Resource Management

Ensure that there are sufficient resources available to execute the test suite and capture the results. Ensure test environment is representative of the environment in which the system will run in.

8.4.6 Binding Time

Ensure that components that are bound later than compile time can be tested in the late-bound context.

8.4.7 Choice of Technology

Determine what technologies are available to help achieve the testability scenarios that apply to your architecture.

8.5 Summary

Testability becomes increasingly important in an environment where development is test driven. The main ways we can improve testability is via changing the observability of phenomena or by limiting complexity.

9 QA: Modifiability

Modifiability is concerned with **how easy it is to make changes to the system** if this is required. *So, for example, having a lot of hard coded values would not make a system very modifiable.*

It's not about the actual modifying but with how easy it would be to do the modifying. **It is very important for a system to be modifiable because changes can be very costly to make.**

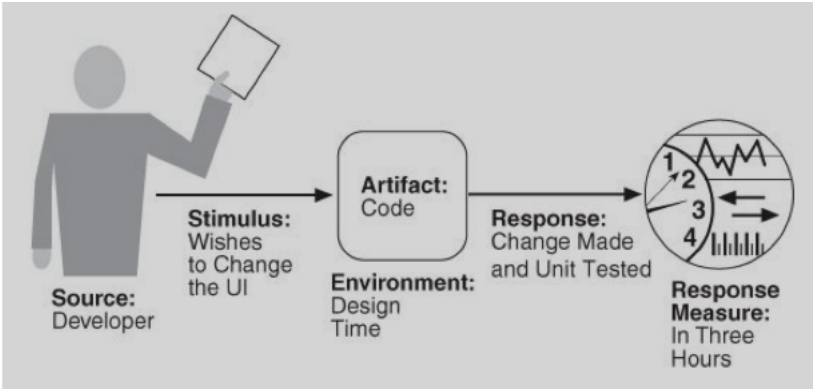
9.1 Four key questions that are asked when making a system:

1) What can change?
2) How likely is something to change?
3) When, where, how, and by whom will the change be made?
4) What is the cost of making these changes?

9.2 General Scenario

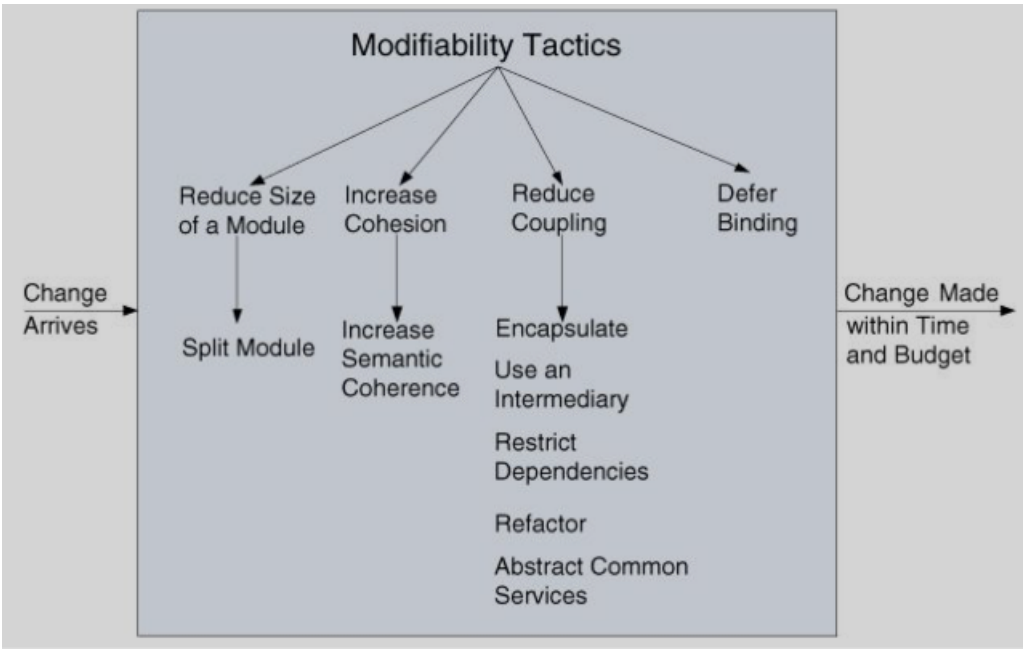
Portion of Scenario	Possible Values
Source	End user, developer, system administrator
Stimulus	A directive to add/delete/modify functionality, or change a quality attribute, capacity, or technology
Artifacts	Code, data, interfaces, components, resources, configurations, . . .
Environment	Runtime, compile time, build time, initiation time, design time
Response	One or more of the following: ▪ Make modification ▪ Test modification ▪ Deploy modification
Response Measure	Cost in terms of the following: ▪ Number, size, complexity of affected artifacts ▪ Effort ▪ Calendar time ▪ Money (direct outlay or opportunity cost) ▪ Extent to which this modification affects other functions or quality attributes ▪ New defects introduced

9.3 Concrete Scenario



- Source: Developer
- Stimulus: Wishes to change the UI
- Artifact: The code itself
- Environment: Design time
- Response: Change made and unit tested
- Response Measure: This should all be done in 3 hours

9.4 Modifiability Tactics



9.4.1 Important Terms

- **Cohesion -** refers to the degree to which the elements inside a module belong together.
- **Semantic coherence** – all the responsibilities in a layer work together without too much reliance on other layers
- **Coupling** – degree of interdependence between software modules
- **Intermediary -** a program or set of programs that in some way evaluates, filters, modifies, or otherwise interjects some function between two end users or end-use programs
- **Binding -** generally refers to a mapping of one thing to another. In the context of software libraries, bindings are wrapper libraries that bridge two programming languages, so that a library written for one language can be used in another language.
- **Deferred Binding -** When a modification is made by the developer, there is usually a testing and distribution process that determines the time lag between the making of the change and the availability of that change to the end user. Binding at runtime means that the system has been prepared for that binding and all of the testing and distribution steps have been completed. Deferring binding time also supports allowing the end user or system administrator to make settings or provide input that affects behaviour.

9.4.2 Tactics of Modifiability

- **Reduce size of a module** – split it up
- **Increase cohesion within a module** – increase semantic coherence
- **Decrease the coupling in the module** – this can be done in the following ways:
  - Encapsulate
  - Use an intermediary
  - Restrict dependencies
  - Refactor
  - Abstract common servies – think of OOP
- **Defer binding** – we can use the "reduce coupling" tactics later in the process so that they are more likely to be done by a computer rather than a human

9.5 Architectural Design Decisions (for modifiability)

- Allocation of responsibilities

- Coordination model
- Data Model
- Management of resources
- Mapping among architectural elements
- Binding time decisions
- Choice of technology

9.5.1 Allocation of Responsibilities

Try to work out how things are likely to change, work out what responsibilities change. Try to modularize so that change does not affect responsibilities that span many modules.

9.5.2 Coordination Model

See how changes are likely to affect coordination and make it so that most probable changes impact coordination across a small number of modules.

9.5.3 Data Model

Data model changes impact as few modules as possible.

9.5.4 Mapping Among Architectural Elements

Look at potential changes and see if some may best be responded to by changing the mapping to elements.

9.5.5 Resource Management

Determine how a change in responsibility will change resource.

9.5.6 Binding Time

Control choice of binding times so there are not too many combinations to consider. Defer binding to later.

9.5.7 Choice of Technology

Choose technologies that make the most likely changes easier – of course, balance with costs.

9.6 Summary

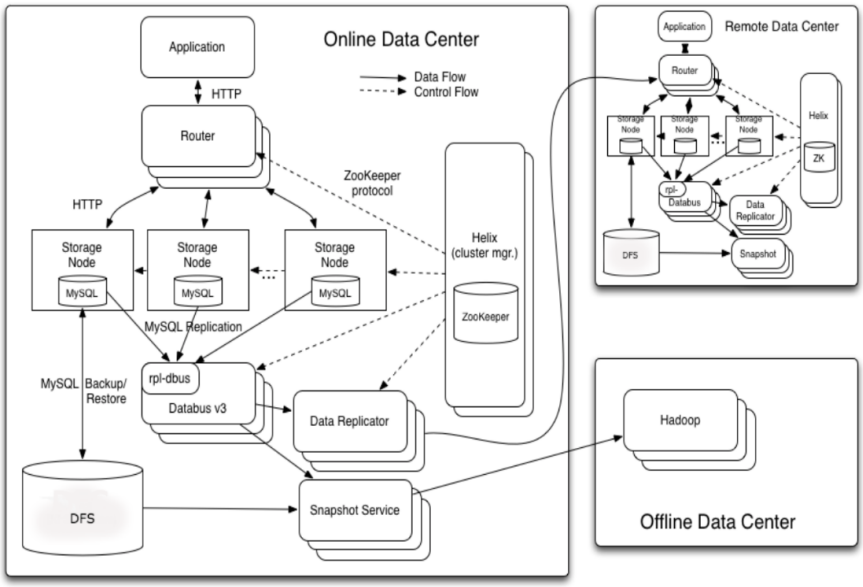
Summary: For better modifiability: 1) Improve cohesion, reduce coupling 2) Use mechanisms that allow you to introduce change late in development cycle 3) Of course, take the cost of all these mechanisms into account

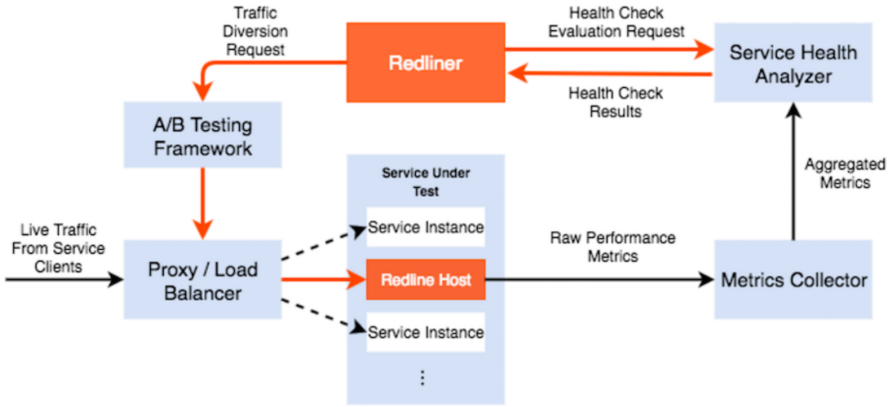
10 Connectors

Software connectors are key elements of the software's architecture. They define the rules of **interaction** between **components**. There are various levels of software connectors that range from *simple* to *complex* connections.

- Simple: shared variable access, method calls ...
- Complex: database access, client-server, scheduler, load balancer ...

In a projects **code base** the connections between components are often implicit and can be noticed easily. In the architecture design we **explicitly identify** them, to allow us to capture **system interactions** (at the level of the components). The specification for interactions are often *complex.* An example for **LinkedIn** is provided below:





10.1 What is Different About Connectors?

Depending on the software project, **components** will have **application-specific** functionality. **Connectors** provide *interaction mechanisms* that are *generic* across different application. **Interaction** may involve **multiple components**, and may have a protocol associated to it.

10.2 Benefits of Explicit Connectors

- **Interaction** is defined by the arrangement of the connectors (as far as possible)
- **Component interaction** is defined by the pattern of connectors in the architecture
- **Interaction** is *"independent"* of the components

10.3 Roles Played By Software Connectors

The specification of the connector protocols determine:

- The types of interfaces
- Properties of interaction
- Rules about ordering interaction
- Measurable features of interactions

Connectors often have multiple roles, the main roles are:

- Communication
- Coordination
- Conversion

- Facilitation

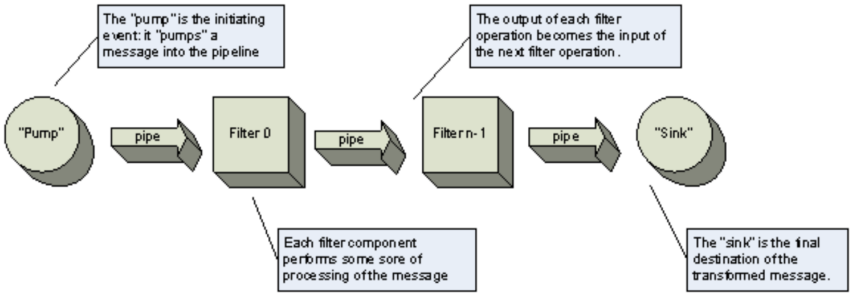
10.4 Communication

Information is transmitted between **components** (e.g. message passing; method calls). **Connectors** constrain:

- **Direction of flow** (The pipes in the image below)
- **Capacity / rate of flow**

** Additional Information **

- Connector providing communication services support **transmission** of data among components
- Data transfer services are a primary building block of component interaction
- Components routinely pass messages, exchange data to be processed and communication results of computations



Connectors influence measurable quality attributes of the system. It separates **communication** from functional aspects.

10.5 Coordination

Coordination controls the timing **relationship** of the functional aspects of the system.

** Additional Information **

- Connectors providing coordination services support transfer of **control** among components
- Components interact by passing the thread of execution to each other

- **Function calls and method invocations are examples of coordination connectors**
- Higher-order connectors, such as signals and load balancing connectors provide richer, more complex interaction built and coordination services

10.6 Conversion

Conversion is how to get components to interact that **do not** have the right means of interaction. **Incompatibilities** might be related to: datatypes, ordering, frequency, structure of parameters etc ...

Examples of types of converters:

- Wrappers: deal with structural issues
- Adaptors: deal with datatype incompatibilities

** Additional Information **

- Connectors providing conversion services **transform the interaction** required by one component to that provided by another
- Enabling heterogeneous components to interact with each other is **not** a trivial task
- Conversion services allow components that have not been specifically tailored for each other to establish and conduct interaction

10.7 Facilitation

Facilitation enables interaction among a group of components that are intended to interact with one and other.

** Additional Information **

- Improve interaction of components that were intended to interoperate (usually **optimise** or streamlines interactions)
- Ensure proper performance profiles (load balancing or scheduling)
- Synchronization mechanisms (monitors $\rightarrow$ enforce mutex access to resources)

10.8 Types of Connectors (Talyor, Medvidovic & Dashofy)

Connector	Description
Method/Procedure Call	Producre call connectors model the flow of control among components through various invocation techniques. They are thus **coordination connector**. [Examples: fork and exec]
Data Access	Data access connectors allow components to access maintained by a data store component. Therefore they provide **communication services**. [Example: JDBC → java SQL driver]
Event	An even as the instantaneous effect of the termination of the invocation of an operation on an object, which occurs at that object's location. [Example: windows with GUI inputs]
Stream	Streams are used to preform transfer of large amounts of data between autonomous processes. Thus they provide **communication services** in a system. [Examples: UNIX pipes, TCP/UDP sockets, client-server protocols]
Distributor	Distributor connectors perform the identification of interaction paths and subsequent routing of communication and coordination information among components along these paths. They provide **facilitation** services. *[Distributor connectos never exist by themselves, but provide assistance to other connectors, such as steams or procedure calls)*
Arbitrator	When components are aware of the presence of other components but cannot make assumptions about their needs and state, arbitrators streamline system operation and resolve any conflicts (providing **facilitation**). They also redirect the flow of control (providing **coordination**)
Adaptor	Adaptor connectors provide facilities to support interaction between components that have not been designed to interoperate. *(adopters involve matching communication polices and interaction protocols among components, thus providing **conversion** services.*

11 Architectural Patterns

An architectural pattern is a **general, reusable solution** to a **commonly occurring problem** in software architecture within a given context.

It comprises of:

- **Context** that provides a frame for a problem.
- **Problem** that is a generalized description of a class of problems with QA requirements that need to be met.
- **Solution** that is suitably generalized in the same way as the problem.

We consider:

- **Static Patterns** designed to solve issues around **development and maintenance** of the code base.
- **Connector and component models** that consider patterns of **interaction, monitoring** at runtime and responses to stimuli.
- **Deployment patterns** that consider the **allocation of elements to hardware resources** and the **availability** of resources in the system.

11.1 Static Patterns

11.1.1 Layer Pattern

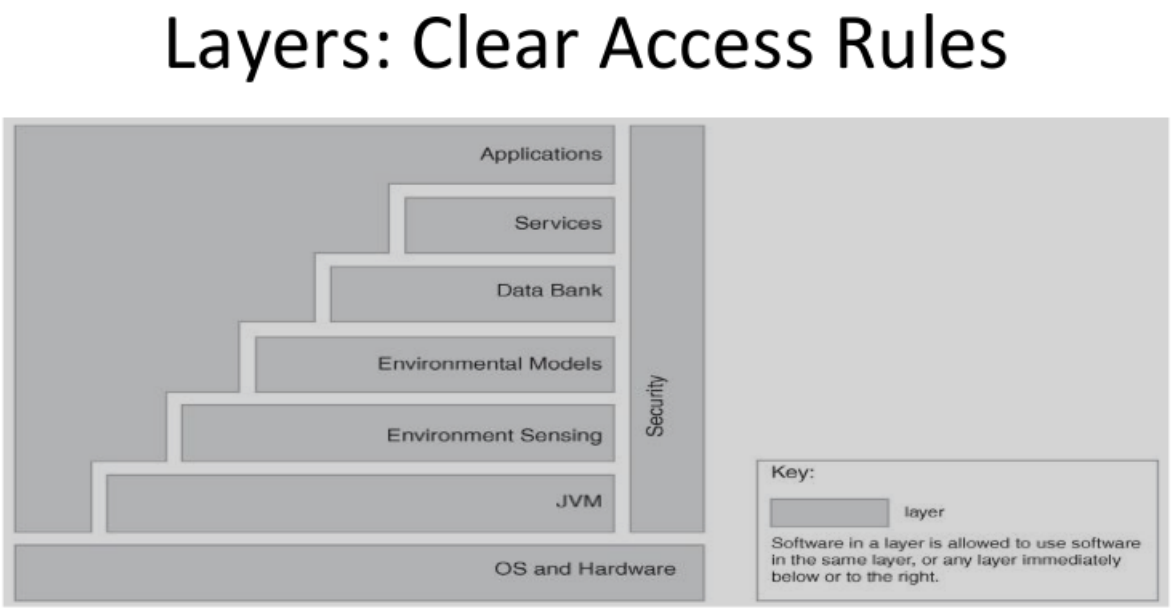
Description:

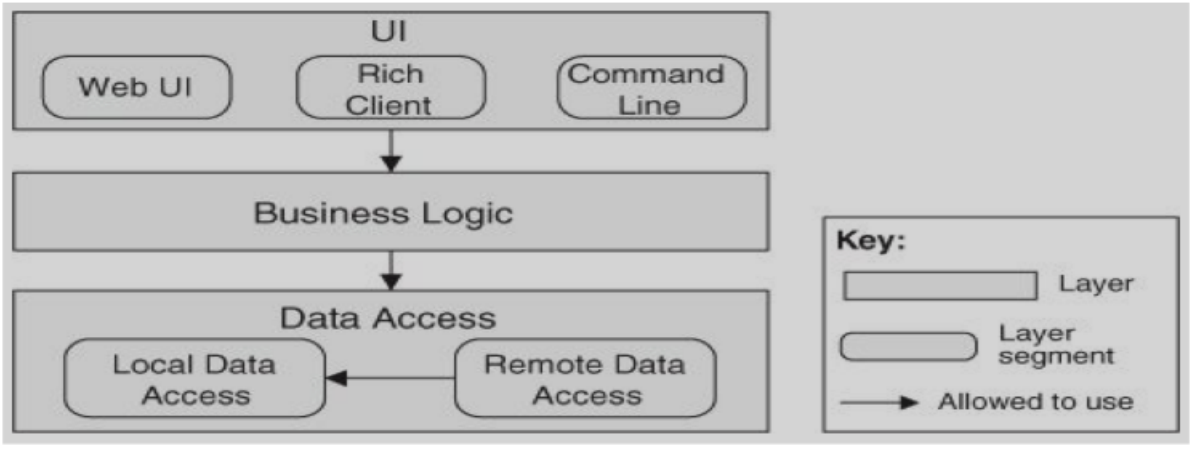
Overview	The layered pattern defines layers (groupings of modules that offer a cohesive set of services) and a unidirectional *allowed-to-use* relation among the layers. The pattern is usually shown graphically by stacking boxes representing layers on top of each other.
Elements	*Layer*, a kind of module. The description of a layer should define what modules the layer contains and a characterization of the cohesive set of services that the layer provides.
Relations	*Allowed to use*, which is a specialization of a more generic *depends-on* relation. The design should define what the layer usage rules are (e.g., "a layer is allowed to use any lower layer" or "a layer is allowed to use only the layer immediately below it") and any allowable exceptions.
Constraints	▪ Every piece of software is allocated to exactly one layer. ▪ There are at least two layers (but usually there are three or more). ▪ The *allowed-to-use* relations should not be circular (i.e., a lower layer cannot use a layer above).
Weaknesses	▪ The addition of layers adds up-front cost and complexity to a system. ▪ Layers contribute a performance penalty.

In a logical multilayered architecture for an information system with an object-oriented design, the following four are the most common:

- Presentation layer (a.k.a UI layer, view layer ...)
- Application layer (a.k.a service layer)

- Business layer (a.k.a business logical layer (BLL), domain layer)
- Data access layer (a.k.a persistence layer, logging, networking, database)





11.2 Connector and component models

11.2.1 Model-View-Controller

Context: The user interface is subject to continuing change to either meet the needs of the application or diversity in the user group.

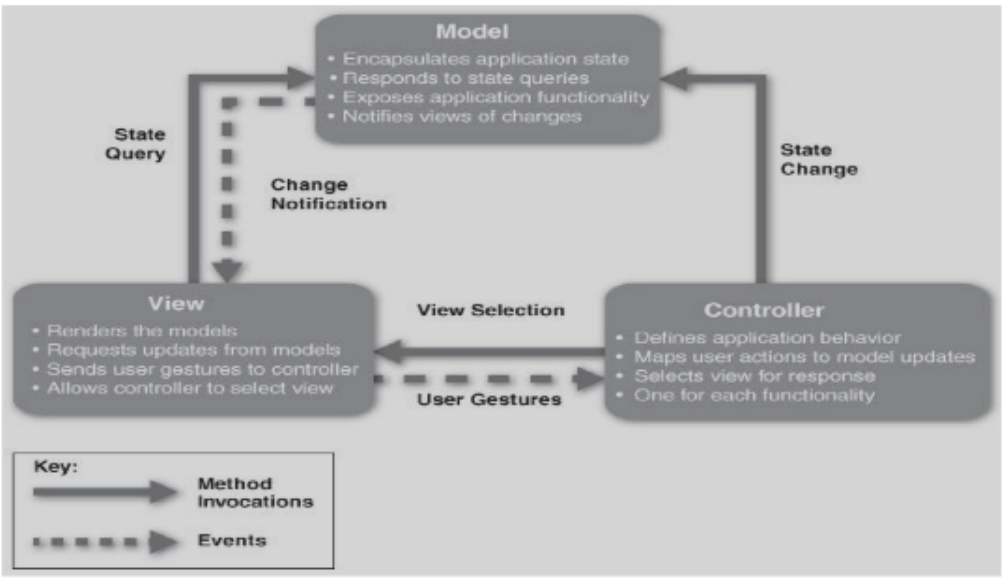
Problem:

- Isolating the UI functionality from the Application functionality.
- Maintaining multiple views in the presence of change in the underlying data.

MVC: Solution

Overview	The MVC pattern breaks system functionality into three components: a model, a view, and a controller that mediates between the model and the view.
Elements	The *model* is a representation of the application data or state, and it contains (or provides an interface to) application logic. The *view* is a user interface component that either produces a representation of the model for the user or allows for some form of user input, or both. The *controller* manages the interaction between the model and the view, translating user actions into changes to the model or changes to the view.
Relations	The *notifies* relation connects instances of model, view, and controller, notifying elements of relevant state changes.
Constraints	There must be at least one instance each of model, view, and controller. The model component should not interact directly with the controller.
Weaknesses	The complexity may not be worth it for simple user interfaces. The model, view, and controller abstractions may not be good fits for some user interface toolkits.

MVC Pattern

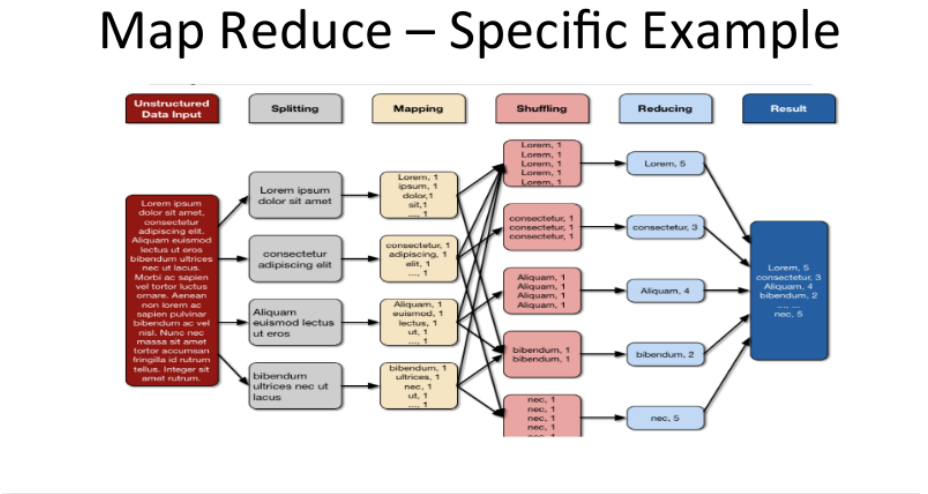


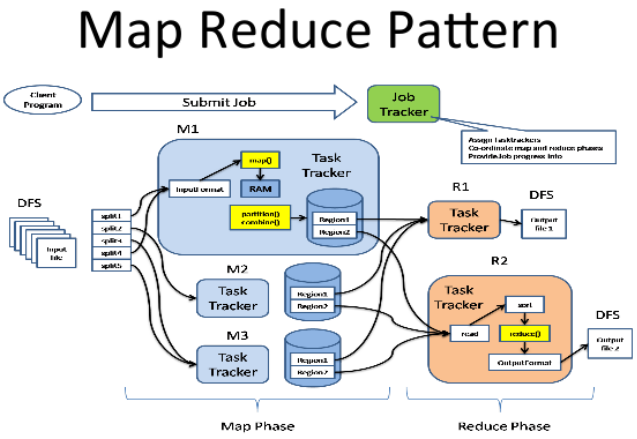
11.3 Deployment/Allocation patterns

11.3.1 Map-reduce pattern

Context:

- We have large quantities of data that we want to process in parallel.
- This encourages an approach that involves significant amounts of independent processing.





11.3.2 Other Allocation Patterns

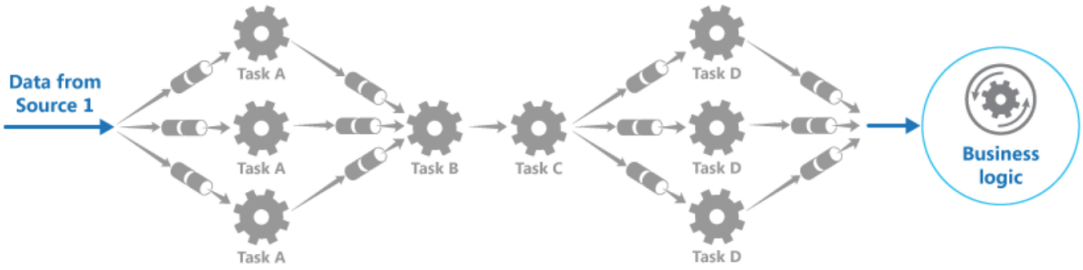
- Multi-tier architecture pattern
- Cloud architectures

11.4 Other patterns

- Pipe and Filter Pattern
- Broker Pattern
- Client-Server Pattern
- Peer-to-Peer Pattern
- Service-Oriented Architecture Pattern
- Publish-Subscribe Pattern
- Shared Data Pattern

11.4.1 Pipe and Filter Pattern

- It consists of any number of components (filters) that transform or filter data, before passing it on via connectors (pipes) to other components.
- The filter transforms or filters the data it receives via the pipes with which it is connected. A filter can have any number of input pipes and any number of output pipes.
- The pipe is the connector that passes data from one filter to the next. It is a directional stream of data, that is usually implemented by a data buffer to store all data, until the next filter has time to process it.
- This architecture is great if you have a lot of transformations to perform and you need to be very flexible in using them, yet you want them to be robust.
- amazing for parallelisation



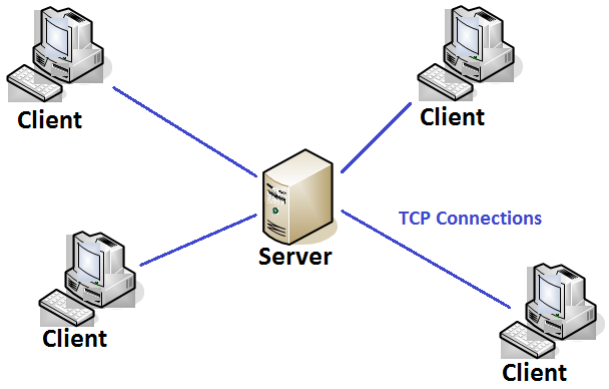
11.4.2 Broker Pattern

- The Broker architectural pattern can be used to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions.

- Makes communication among remote and heterogeneous classes easy.
- Keep in mind this is not very scalable, so it loses effectiveness for larger systems.

11.4.3 Client-Server Pattern

- The client–server model is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients. Often clients and servers communicate over a computer network on separate hardware, but both client and server may reside in the same system. A server host runs one or more server programs which share their resources with clients.



- The computing power, memory and storage requirements of a server must be scaled appropriately to the expected work load.

11.4.4 Peer-to-peer pattern

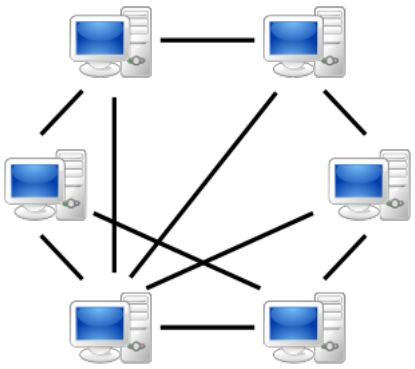
- Peer-to-peer (P2P) computing or networking is a distributed application architecture that partitions tasks or workloads between peers. Peers are equally privileged, equipotent participants in the application. They are said to form a peer-to-peer network of nodes.

11.4.5 Service Oriented Architecture Pattern

A service-oriented architecture (SOA) is a style of software design where services are provided to the other components by application components, through a communication protocol over a network. The basic principles of service oriented architecture are independent of vendors, products and technologies. A service is a discrete unit of functionality that can be accessed remotely and acted upon and updated independently, such as retrieving a credit card statement online.

Properties:

1. It logically represents a business activity with a specified outcome.



2. It is self-contained.
3. It is a black box for its consumers.
4. it may consist of other underlying services.
5. loose coupling between services

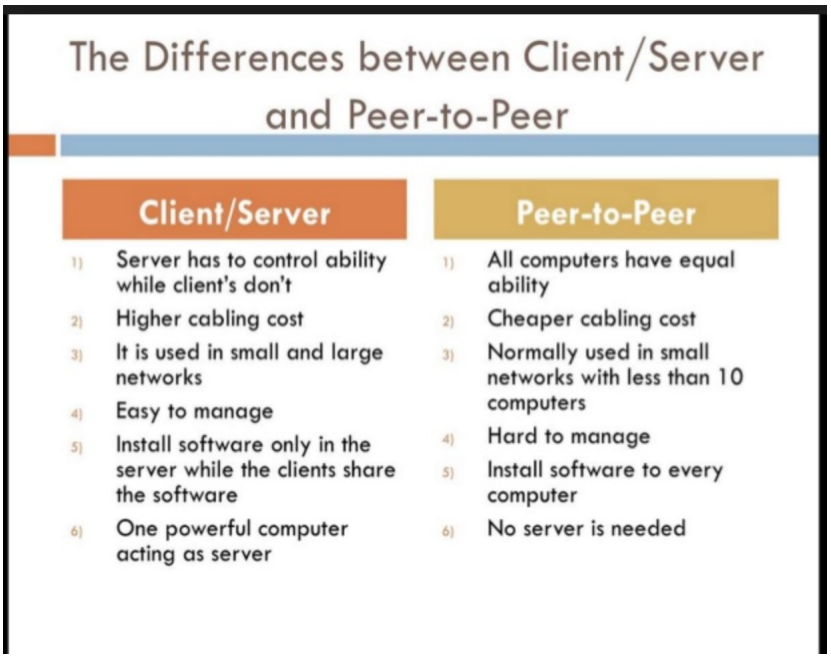
Defining concepts

1. Business value is given more importance than technical strategy.
2. Strategic goals are given more importance than project-specific benefits.
3. Intrinsic inter-operability is given more importance than custom integration.
4. Shared services are given more importance than specific-purpose implementations.
5. Flexibility is given more importance than optimization.
6. Evolutionary refinement is given more importance than pursuit of initial perfection.

11.4.6 Publish Subscribe Pattern

publish–subscribe is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead characterize published messages into classes without knowledge of which subscribers, if any, there may be. Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are.

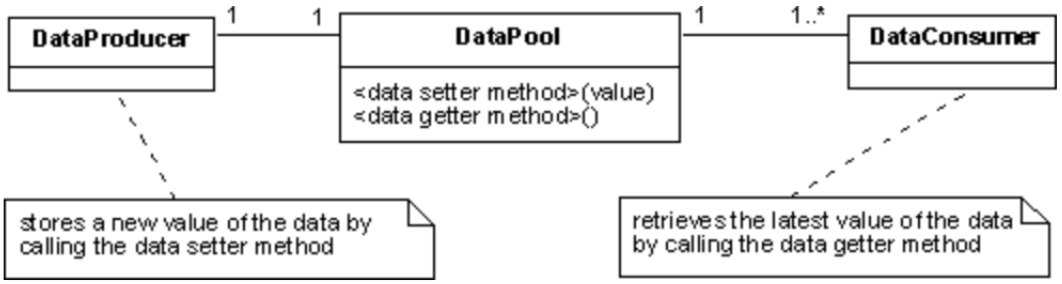
Properties



1. Loose coupling between publishers and subscribers- can be both an advantage and disadvantage
2. better scalability that client server as parallel operation possible.

11.4.7 Shared Data Pattern

1. intent is to Decouple the production of data from their consumption.
2. Consider the case of an application built as a collection of components. These components interact by exchanging data. In a data exchange, the component that generates the data is called the data producer and the component that receives the data is called the data consumer.Data pools are introduced to act as shared memory areas through which data are exchanged among application components. The data pools physically contain the data items. The producers of data deposit the data into the data pool and the consumers of data retrieve the data they need from the data poo



11.5 Relationship between Patterns and Tactics

	Modifiability									
	Increase Cohesion		Reduce Coupling					Defer Binding Time		
Pattern	Increase Semantic Coherence	Abstract Common Services	Encapsulate	Use a Wrapper	Restrict Comm. Paths	Use an Intermediary	Raise the Abstraction Level	Use Runtime Registration	Use Startup-Time Binding	Use Runtime Binding
Layered	X	X	X		X	X	X			
Pipes and Filters	X		X		X	X			X	
Blackboard	X	X			X	X	X	X		X
Broker	X	X	X		X	X	X	X		
Model View Controller	X		X			X				X
Presentation Abstraction Control	X		X			X	X			
Microkernel	X	X	X		X	X				
Reflection	X		X							

12 Architectural Modelling

When building a software architecture, we want to ideally have control over the quality attributes (example – testability, modifiability, etc) and even to be able to predict how well our system will be able to do in each one in advance.

So – we want to be able to build a predictive model of the software architecture and then use this model to predict the quality attributes – how well our software will perform on each of them.

When we model a specific architecture to be used for a software project, we wish to test before implementing if the architecture will comply with the quality attributes we need/ specify. Architectural modelling gives us a means to do so.

12.1 Performance: Queueing Model

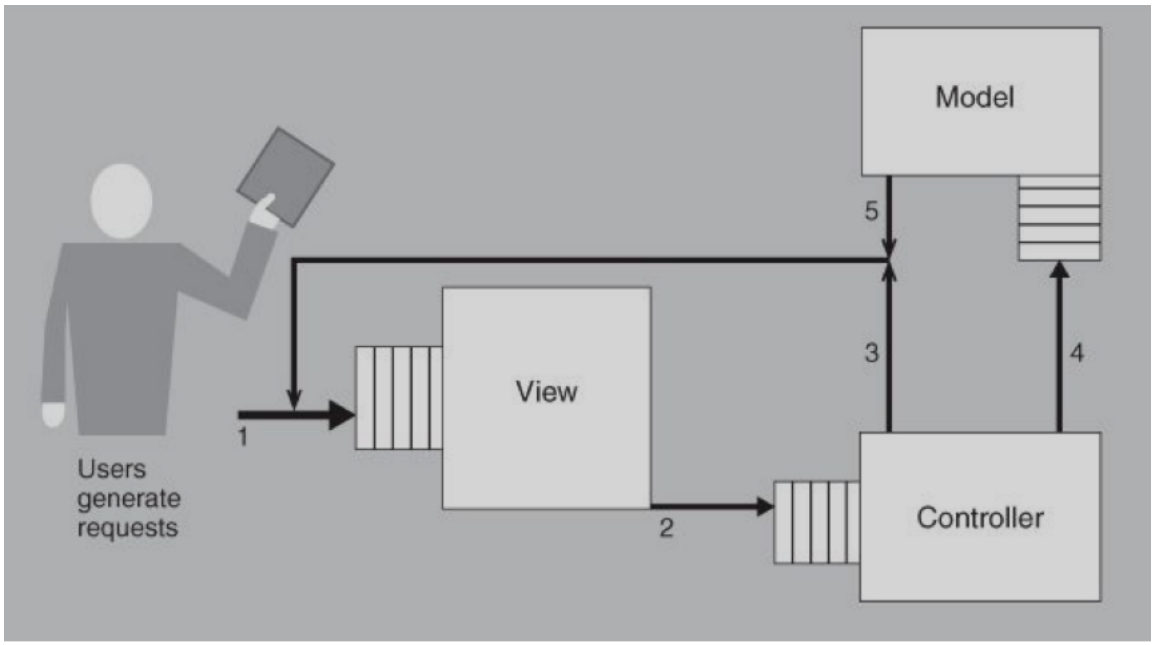
Because the queuing model is **highly specific** we can model this scenario well and actually test before the **implementation** if the architecture will meet the quality attributes we specify.

*This example is also provided in the **performance** QA section*

12.1.1 How the model looks like:

1) The beginning specifies the distribution of arrivals of service requests.
2) The queuing discipline
3) The scheduling algorithm
4) Routing of messages coming from the algorithm
5) The results – could be the quality attributes the model is predicting

12.1.2 MVC Model – model to test the performance quality attribute



1) The view component will service the service requests at some rate.
2) The view translates these into service requests for the controller.
3) The there are service requests from the controller to the view, from the controller to the model, and from the model to the view component – all are interleaved.

If we have good estimates for all the different components of the model (distribution of external service demands, queuing disciplines, network latencies, transfer characteristics), then it can be a very good predictor of quality attributes.

12.2 Availability: Broker Model

For the availability quality attribute we give the example us the usage of the **broker model**. Availability is not as well defined / specific as performance, thus it is harder to model and therefore test this quality attribute before the implementation of the system.

The key issue is testing how long it takes to detect that a failure has taken place

12.2.1 Main Quality Attributes and their Analysis Techniques

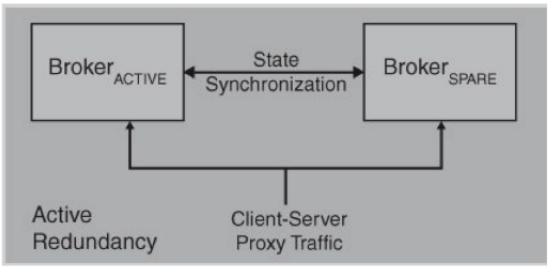
Availability is a measure of software quality defined as:

$$MTBF \div (MTBF + MTTR)$$

MTBF is Mean Time Between Failures
MTTR is Mean Time To Repair

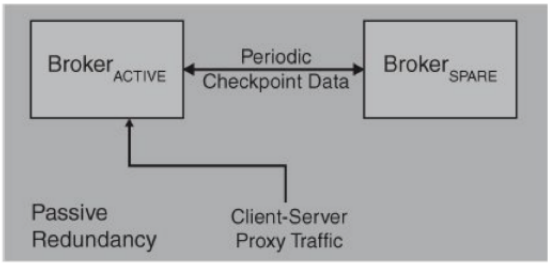
There are three different techniques to test availability using the broker model these are:

Hot Spare



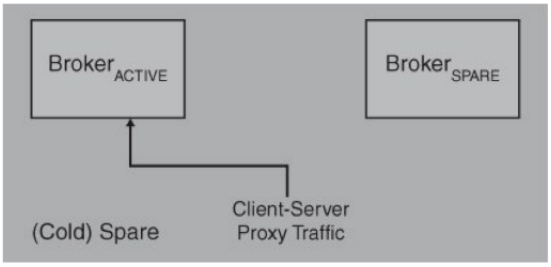
- Active and redundant both receive identical request stream.
- Synchronous maintenance of broker state.
- Fast failover in the event of failure of the active system.

Warm Spare



- Warm broker is maintained at the most recent checkpoint state.
- In the event of failure the system rolls back to the most recent checkpoint.
- This is slower than the hot spare approach

Cold Spare



- Here there is no attempt to synchronise.
- In the event of failure the cold spare is started.
- The system state is recovered via interaction with other systems (so they have to be resilient to failure in the broker)

The main quality attributes and their analysis techniques are:

- **Availability** – Markov models, statistical models
- **Interoperability** – Conceptual framework
- **Modifiability** – Coupling and cohesion metrics, cost models
- **Performance** – Queuing theory, real-time scheduling theory
- **Security** – No architectural models
- **Testability** – Component interaction metrics
- **Usability** – No architectural models

Some QAs have good, well established analysis techniques, others do not,

12.3 Architectural Analysis

12.3.1 Analytical Model Landscape QUALITY ATTRIBUTES

Quality Attribute	Intellectual Basis	Maturity/Gaps
Availability	Markov models; statistical models	Moderate maturity; mature in the hardware reliability domain, less mature in the software domain. Requires models that speak to state recovery and for which failure percentages can be attributed to software.
Interoperability	Conceptual framework	Low maturity; models require substantial human interpretation and input.
Modifiability	Coupling and cohesion metrics; cost models	Substantial research in academia; still requires more empirical support in real-world environments.
Performance	Queuing theory; real-time scheduling theory	High maturity; requires considerable education and training to use properly.
Security	No architectural models	
Testability	Component interaction metrics	Low maturity; little empirical validation.
Usability	No architectural models	

12.3.2 Analysis In The Life-Cycle

Life-Cycle Stage	Form of Analysis	Cost	Confidence
Requirements	Experience-based analogy	Low	Low–High
Requirements	Back-of-the-envelope	Low	Low–Medium
Architecture	Thought experiment	Low	Low–Medium
Architecture	Checklist	Low	Medium
Architecture	Analytic model	Low–Medium	Medium
Architecture	Simulation	Medium	Medium
Architecture	Prototype	Medium	Medium–High
Implementation	Experiment	Medium–High	Medium–High
Fielded System	Instrumentation	Medium–High	High

12.3.3 Types of Analysis

- **Thought experiment:** just a sort of discussion using informed people.
- **Back of the envelope:** using very approximate techniques with unreliable assumptions
- **Check-list:** collated experience.
- **Analytic Model:** based on sound abstractions –heavily dependent on estimates being correct
- **Simulation:** higher level of detail – less analytic, more concrete
- **Prototype:** approximate system in an experimental setup
- **Experiment:** fielded system, simulated load
- **Instrumentation:** measuring the variable of interest

12.4 Summary

Architecture is the correct level to deal with quality attributes. Analysis can be costly, depending on how accurate you want it to be. Analysis throughout the lifecycle helps with decision taking.

13 The Life-cycle

They impose some discipline on the development process – order stages and activities of the development process. Usually it's an ongoing process improvement cycle that focuses on making this process better.

13.1 Typical Examples

There are some typical stages of the software lifecycle.

Life-Cycle Stage	Architecture-Based Activity
Business needs and constraints	• Create a documented set of business goals: issues/environment, opportunities, rationale, and constraints using a business presentation template.
Requirements	• Elicit and document six-part quality attribute scenarios using general scenarios, utility trees, and scenario brainstorming.
Architecture design	• Design the architecture using ADD. • Document the architecture using multiple views. • Analyze the architecture using some combination of the ATAM, ARID, or CBAM.
Detailed design	• Validate the usability of high-risk parts of the detailed design using an ARID review.
Implementation	
Testing	
Deployment	
Maintenance	• Update the documented set of business goals using a business presentation template. • Collect use case, growth, and exploratory scenarios using general scenarios, utility trees, and scenario brainstorming. • Design the new architectural strategies using ADD. • Augment the collected scenarios with a range of response and associated utility values (creating a utility-response curve); determine the costs, expected benefits, and ROI of all architectural strategies using the CBAM. • Make decisions among architectural strategies based on ROI, using the CBAM results.

The images on the next few pages show the typical stages of the software lifecycle and then typical examples of different software lifecycles.

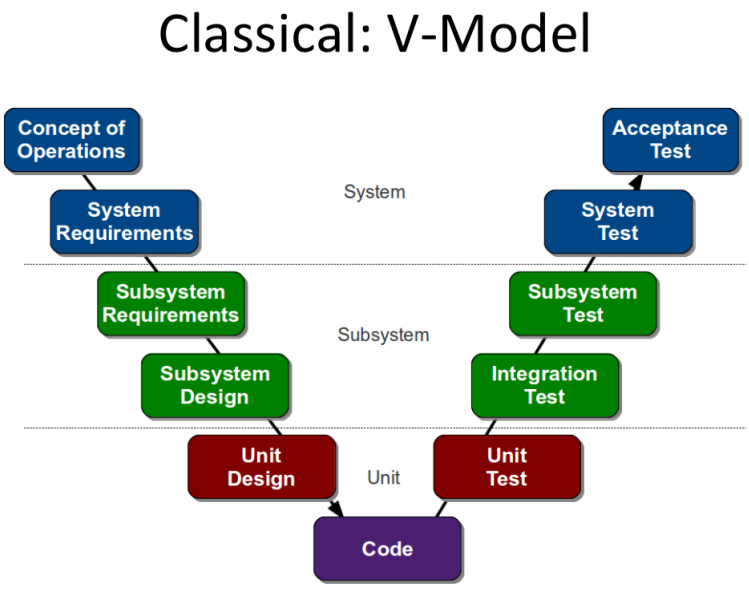


Figure 19: A diagram showing the V model

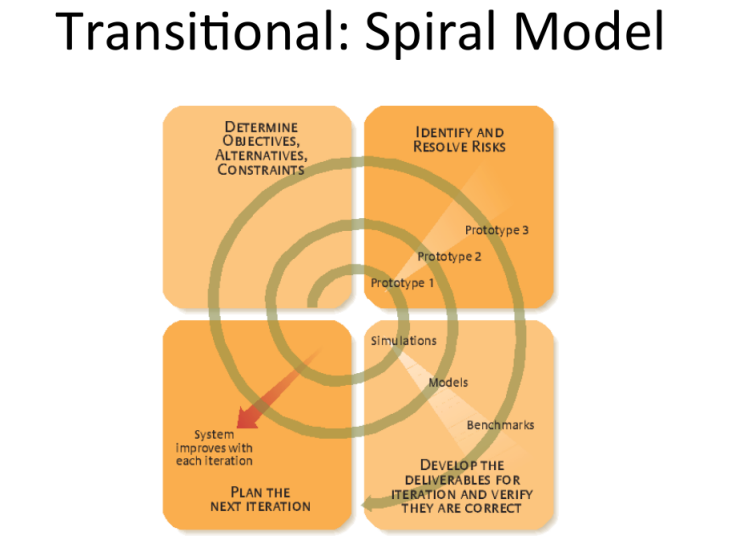


Figure 20: A diagram showing the spiral model

Classical: RUP

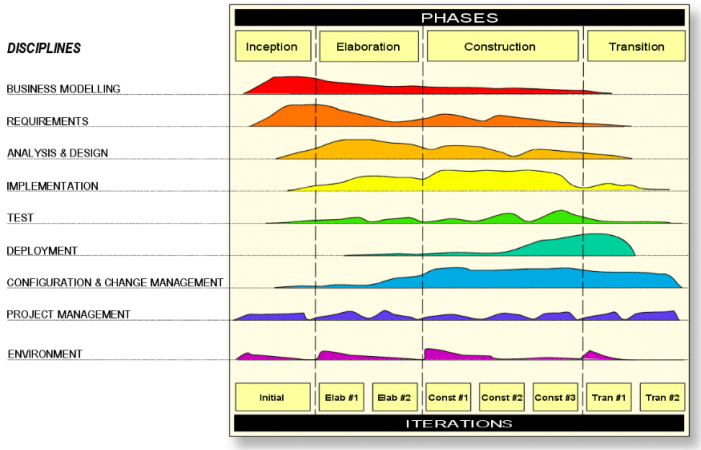


Figure 21: A diagram of the RUP model

Rational Unified Process(RUP) explained
Rational Unified Process (RUP) establishes four phases of development, each of which is organized into a number of separate iterations that must satisfy defined criteria before the next phase is undertaken: in the inception phase, developers define the scope of the project and its business case; in the elaboration phase, developers analyze the project's needs in greater detail and define its architectural foundation; in the construction phase, developers create the application design and source code; and in the transition phase, developers deliver the system to users.

13.2 Agile Programming

Agile Programming Practice:

- Test first programming
- Refactoring
- Continuous integration
- Simple design
- Pair programming
- Common codebase
- Coding standards
- Open work area

Manifesto for Agile Software Development:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

The terms on the right are valued, but those on the left are valued more.

Scrum is an Agile framework for completing complex projects. Scrum originally was formalized for software development projects, but it works well for any complex, innovative scope of work. The possibilities are endless. The Scrum framework is deceptively simple:

- A product owner creates a prioritized wish list called a product backlog.
- During sprint planning, the team pulls a small chunk from the top of that wish list, a sprint backlog, and decides how to implement those pieces.
- The team has a certain amount of time — a sprint (usually two to four weeks) — to complete its work, but it meets each day to assess its progress (daily Scrum).
- Along the way, the ScrumMaster keeps the team focused on its goal.
- At the end of the sprint, the work should be potentially shippable: ready to hand to a customer, put on a store shelf, or show to a stakeholder.
- The sprint ends with a sprint review and retrospective.
- As the next sprint begins, the team chooses another chunk of the product backlog and begins working again

13.3 Comparison of Agile versus Plan-Driven Approach

The tables in the following images compare the agile to a plan-driven approach.

Characteristics	Agile	Plan Driven
Application		
Primary goals	Rapid value; responding to change	Predictability, stability, high assurance
Size	Smaller teams and projects	Larger Teams and projects
Environment	Turbulent; high-change; project-focus	Stable; low-change; project/ organisation focus

Characteristics	Agile	Plan Driven
Management		
Customer relations	Dedicated on-site customers; focus on prioritized increments	As-needed customer interactions; focus on contract provisions
Planning and Control	Internalized plans; qualitative control	Documented plans; quantitative control
Communication	Tacit interpersonal knowledge	Explicit documented knowledge

Characteristics	Agile	Plan Driven
Technical		
Requirements	Prioritized informal stories and test cases; undergoing unforseeable change	Formalized project, capability, interface, quality, forseeable evolution requirements
Development	Simple design; short increments; refactoring assumed to be inexpensive	Extensive Design; longer increments; refactoring assumed expensive
Testing	Executable test cases define requirements	Documented test plans and procedures

Characteristics	Agile	Plan Driven
Personnel		
Customers	Dedicated, collocated CRACK* performers	CRACK* performers – not always collocated
Developers	High level skills required throughout – less scope for lower performers	High skill people needed at critical points, more possibility to use lower skilled people
Culture	Comfort and empowerment through freedom (chaos?)	Comfort and empowerment through policies and procedures (order).

Key Points: When developing software, work top-down and bottom-up at the same time – the balance of this depends on the size and complexity of the project.

General Rule: As the size of the error/loss increases, the probability of the loss decreases. When the loss is small, the probability of it occurring is high. The relationship varies slightly depending on the project, but the general trend is as described above.

Analysis Techniques

Acronym	Name	Inputs	Outputs
QAW	Quality Attribute Workshop	Mission drivers; system architectural plan	Raw scenarios; consolidated scenarios; priorities; refined scenarios
ADD	Attribute Driven Design	Constraints; functional and QA requirements	Module; component; and deployment architectural views
ATAM	Architecture Tradeoff Analysis Method	Business/mission drivers; existing architecture documentation	Potential arch approaches; scenarios; QA questions; risks; themes; sensitivities; tradeoffs
CBAM	Cost-benefit analysis method	As ATAM + existing scenarios	Architectural strategies; priorities; risks
ARID	Active Reviews for Intermediate Designs	Seed scenarios; existing Arch documentation	Issues and problems for Architecture

Figure 22: Analysis Techniques

Analysis Techniques and Stage

Life-Cycle Stage	QAW	ADD	ATAM	CBAM	ARID
Business needs and constraints	Input	Input	Input	Input	
Requirements	Input; output	Input	Input; output	Input; output	
Architecture design		Output	Input; output	Input; output	Input
Detailed design					Input; output
Implementation					
Testing					
Deployment					
Maintenance				Input; output	

Figure 23: Analysis Technique and Stage

13.4 Summary

- There are many possible lifecycles and variants on these lifecycles.
- For any partcular area of actvity we need to find the balance between agility and discipline.
- Risk links discipline and agility
- QAs, scenarios and tactics help link architecture to more agile practice.
- Architectural analysis techniques provide useful information for lifecycle activities, however they are arranged in a process.
- Processes like the Spiral Model or ICM provide processes that are sensitive to project risk.

14 Dev-Ops

This is the process of seeing development and operations as a unified entity. It oversees the whole process and all stages of developing software. It is the set of practices intended to reduce the time between communicating a change to a system and the change being placed into normal operation while ensuring necessary quality.

The Whole Application Lifecycle: 1) Define - ideation 2) Develop - idea to working software 3) Operate - working software in production, value realization 4) Measure - actionable learning

14.1 OSLC

OSLC standards simplify lifecycle integration, which leads to cost savings and increased flexibility. It is the foundational technology for all integration, and is the natural choice for standardizing loosely-coupled integrations in new domains.

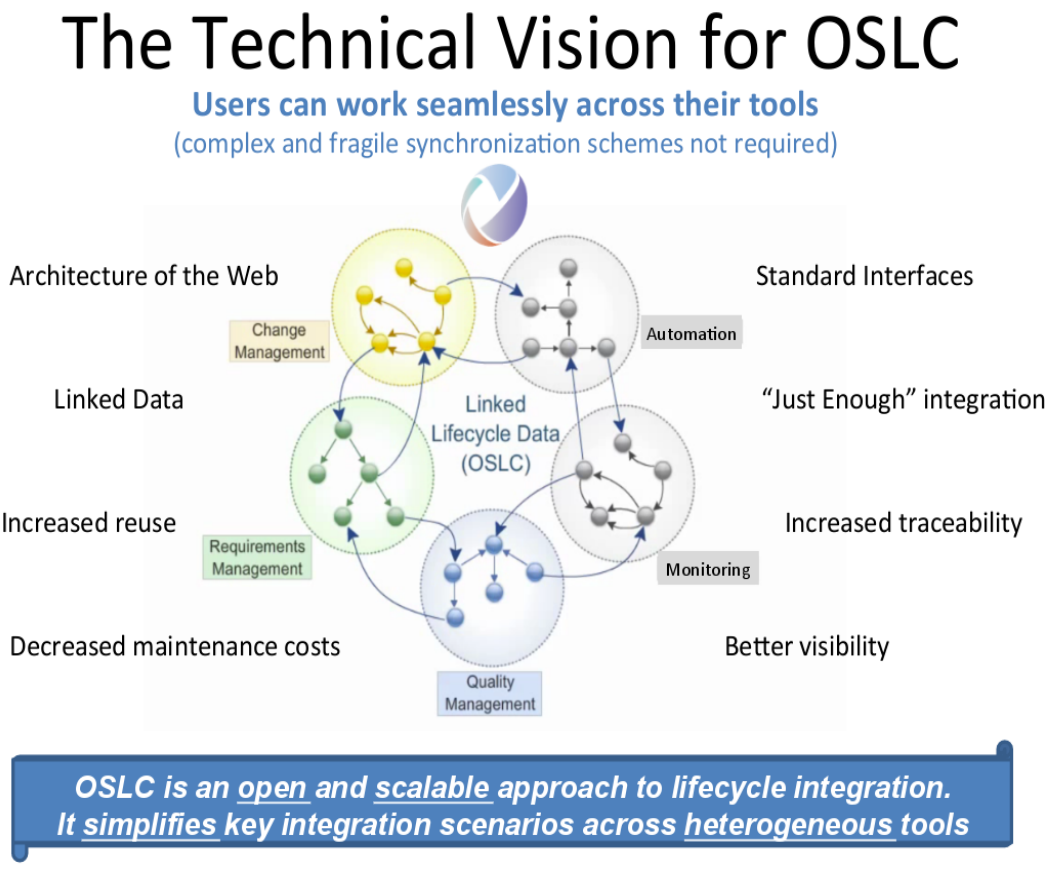


Figure 24: Summary of OSLC

14.2 Critical Points in the Lifecycle of Software

- Make the decision to commit the code to be introduced into the system.
- Transition from something being just under consideration to actually being a key part of the system.
- Do we have enough confidence to be able to make such a transition?
- We want to ensure each of these transitions are as reliable as possible.

14.3 Microservices Architectural Pattern

Puts each element of functionality (small, single purpose) into a separate service and scales by distributing these services across servers, replicating as needed. They are loosely coupled and asynchronous.

14.4 How to make development easier, faster, and better

- One step environment creation – need a common environment build process
- Code, environment, and configuration all in one place
- Automation is essential
- Feedback loops – understanding and responding to the needs of all customers (both internal and external) – automate these feedback loops

Having such a consistent process and effective feedback results in agility – then can experiment, fail, but learn and recover quickly.

It is important to see your mistakes before you actually start producing the software on a wide scale:

- Be consistent in the code, environments, and configuration
- Try to catch misconfiguration and inconsistencies early
- Testing every feature as you are building the system
- Try to limit technical debt

Technical Debt - concept in programming that reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution

15 Product Line Architecture

Software often comes in families. Thus, it makes sense to try to share components. Many real-world products are produced in a product line – such as cras, for example.

A software product line is a collection of software-intensive systems that share a common, managed set of features that are developed from a set of core assets in a prescribed way.

Think of different, successive versions of the same product released by a company – they are all based on the same baseline architecture. When the number of products in it increase beyond a certain number, it starts to become profitable for the company using this technique.

15.1 Key Properties

Key properties of this product line architecture:

- usually are directed by the business goals in the application domain
- all products share a software product line architecture
- new products are all structured by this product line architecture and are built from services and components
- product lines spread costs over several products
- Architecture must be flexible enough to support variation in the products
- Software components – general enough to support variability
- All other components of the software must be general enough to deal with variation
- People need to be skilled in architecture and product lines

15.2 Benefits to organization

There are several benefits to the organization:

- much better productivity
- quicker delivery time to the market – so maintain market presence and sustain growth
- enable mass customization
- improve product quality
- better predictability of cost, schedule, and quality

15.3 Main Terms

Core Asset Development: improving the base components in terms of qualities, products they support, and architecture

Product Development: identifying and building products to meet market need inside the product line

Management: monitoring and improving the processes, tools, and practices

15.4 Different Techniques in Introducing Product Lines

- Proactive – up front investment to develop the core assets
- Reactive – start with one or two products and use them to generate core assets – so see how well they do and then "react"
- Incremental – develop core assets as the business need evolves

The general process involves considering the business strategy, consequences for products, consequences for processes and methods, and consequences for tools and the organization.

15.5 Pros of Using a Product Line:

- Increased competitiveness on the market because of reduced hardware resource consumption and reduced time to market for new features
- Development efficiency – reuse, easy configuration of software products, increased planning accuracy
- Quality - interface integrity, reuse of core assets
- Customer needs – differentiation by individual software solutions, clear feature-cost mapping

15.6 Architectural Features of Product Lines:

- Control of resource consumption, such as memory
- Good interface management
- Layers provide the opportunity to share applications without knowing the details of some components
- Software is reusable so that component redesign is easy and quick
- Standardization is important
- Important to standardize and develop tools that ease interchange between the company and it clients (toolchains)

15.7 Phased Introduction

This is how a company would start applying a software product line (called phased introduction) :

- Investigate and customize product line engineering
- Design and apply adequate processes and methods
- Roll out and institutionalize the standard development process

16 Analysis

Why we Evaluate:

- To inform decision making
- To allow progression to later stages in to process (think sprial/ incremental development methodologies)

There are three types of Evaluation:

- Evaluation By Designer
- Peer Evaluation
- External Evaluation

16.1 Evaluation By Designer

The consequences of the decision regulate how much effort to put into the process → more importance means more effort in evaluation. Try to use iterative approaches that get deeper in order to eliminate unpromising alternatives early. *(Don't strive for perfection, good enough for the context is usually good enough)*

16.2 Peer Evaluation

Fix the *Quality Attributes* to consider as part of the review → maybe determined by the process or the business case. The **architect** presents the architecture to the reviewers (the questions are for information). The review is driven by the relevant scenarios the architect talks the review team through a scenario demonstrating the architecture meets the requirements captured in a scenarios.

16.3 External Evaluation

Means to bring in additional expertise. Maybe represent some stakeholder interests. It is a more **expensive** evaluation method and difficult to organise so this will often correspond to some major hurdle in the process.

These may include, but are not limited to **(contextual factors)**:

- What artefacts are available?
- Who sees the results of the review?
- Who preforms the evaluation?
- Which stakeholders will participate?
- How does the evaluation relate to business goals of the system?

16.4 Additional Info

When we evaluate shit we must consider **Contextual Factors** as well.

In summary, the larger and more complex the system the more likely you are to have done explicit architectural design and any design should be evaluated.