

Software Architecture Process And Management

Computer Science 4th Year, Created By Monkeys

Contents

1	Introduction	4
1.1	Learning Outcomes of the course	4
1.2	What is Success for a Large Project?	4
1.3	Software Architecture	4
1.4	Case Study: General Practice Extraction System	5
2	Basic Concepts of Architectures	10
2.1	What is good architecture?	10
2.2	Process	10
2.3	Structure	10
2.4	The Importance of Architecture	11
2.5	Managing Attributes and Change	12
2.6	Prediction of Attributes	12
2.7	Communication Between Stakeholders	12
2.8	Early Design and Constraints	12
2.9	Evolutionary Prototyping	13
2.10	Cost and Scheduling	14
2.11	Product Line (Model)	14
2.12	Component Level & Channelling Development	14
3	Context Design	15
3.1	Technical Context	15
3.1.1	Controlling Quality Attributes	15
3.2	Project Life-cycle Context	16
3.2.1	V-Model	17
3.2.2	Spiral Model	18
3.2.3	Agile Development	18
3.2.4	Agile + Devops	20
3.3	Business Context	20
3.4	Professional Context	20
3.5	Domain-Specific Software Architecture	21
3.6	Architectural Patterns	22
4	Quality Attributes	26
4.1	Stakeholders	26
4.2	Functional Requirements	26
4.3	Constraints	26
4.4	Problems with Quality Attributes	27
4.5	Quality Attribute Scenarios	27
4.6	Architectural Tactics	28
4.7	Categories of Architectural Design Decisions	29
4.7.1	Allocation of responsibilities	29
4.7.2	Coordination Model	29
4.7.3	Data Model	29
4.7.4	Management of resources	30
4.7.5	Mapping Among Architectural Elements	30
4.7.6	Binding Time Decisions	30
4.7.7	Choice of technology	31

5 QA: Availability	32
5.1 Fault, Error, and Failure	32
5.2 Generic Scenario	33
5.3 Concrete Scenario	33
5.4 Availability Tactics	33
5.5 WatchDog (Tactic Example)	34
5.6 Architectural Design Decisions (for availability)	35
5.6.1 Allocation of Responsibilities	35
5.6.2 Coordination Model	35
5.6.3 Data Model	36
5.6.4 Management of Resources	36
5.6.5 Mapping among architectural elements	36
5.6.6 Binding Time Decisions	36
5.6.7 Choice of Technology	37
6 QA: Performance	37
7 QA: Security	37
8 QA: Testability	37
9 QA: Modifiability	37
10 Connectors	38
10.1 What is Different About Connectors?	39
10.2 Benefits of Explicit Connectors	39
10.3 Roles Played By Software Connectors	39
10.4 Communication	40
10.5 Coordination	40
10.6 Conversion	41
10.7 Facilitation	41
10.8 Types of Connectors (Talyor, Medvidovic & Dashofy)	42
11 Patterns	43
12 Modelling The Life-cycle	43
13 Dev-Ops	43
14 Product Line Architecture	43
15 Analysis	43
15.1 Evaluation By Designer	43
15.2 Peer Evaluation	43
15.3 External Evaluation	43
15.4 Additional Info	44

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

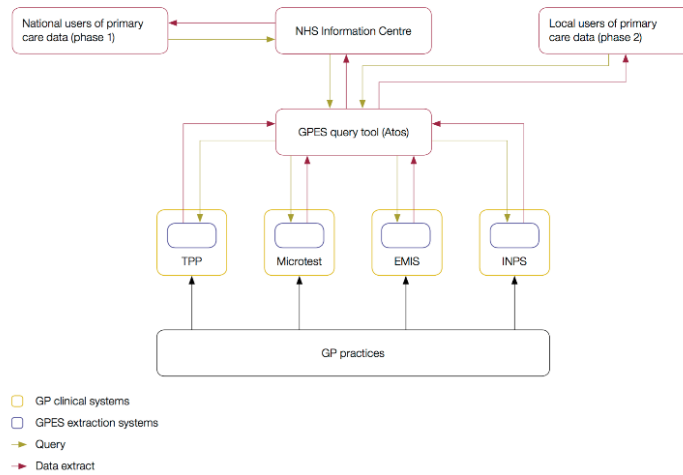
Figure 1
Proposed GPES customers and potential benefits

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

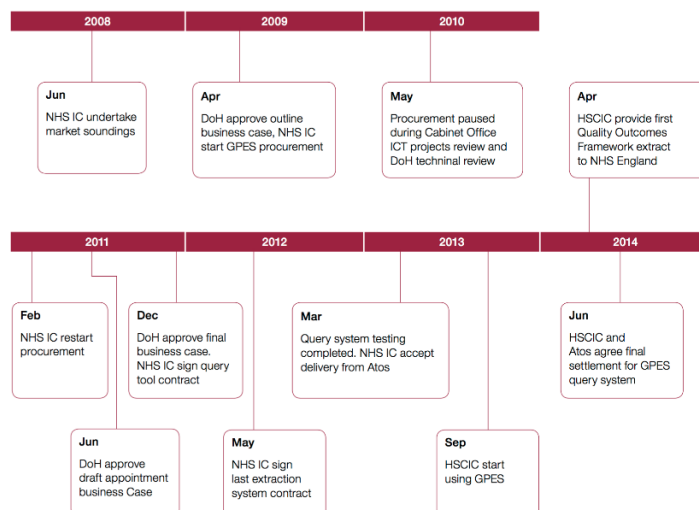


Source: National Audit Office, based on information in the NHS IC GPES business cases

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 a lot to make 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

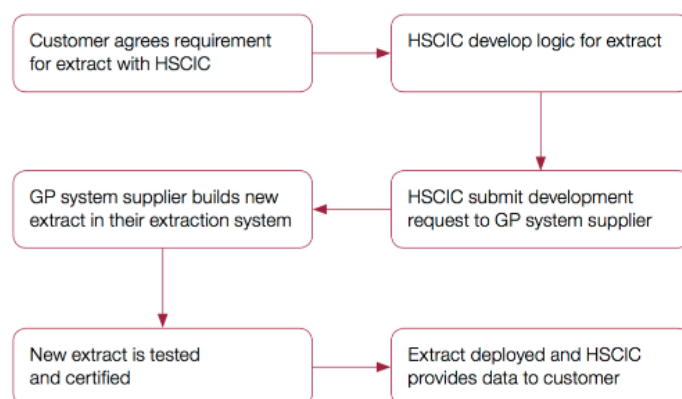
the changes in design both contributed to the restrictive process for *extracts*.

- 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 difficulty 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 head 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

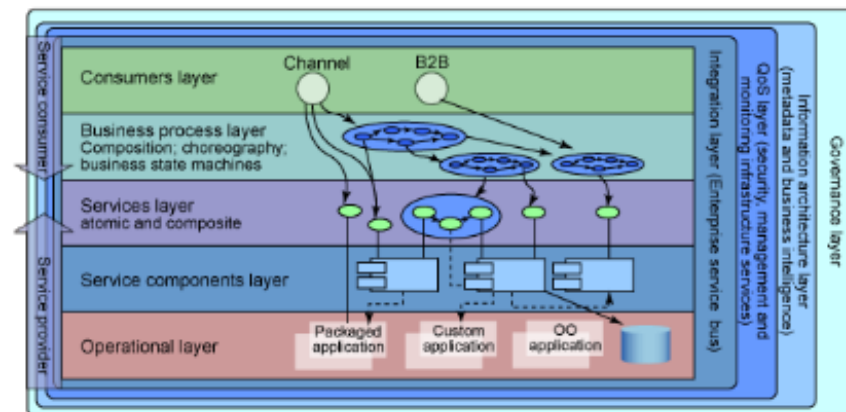
Early design carries the *most fundamental* design decisions, e.g:

- 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.

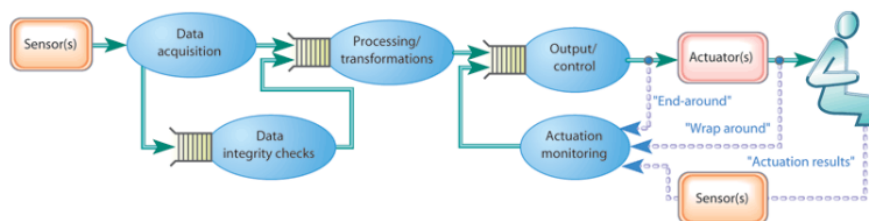
Other examples of quality attributes include **performance**, **usability**, **interoperability** ...

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.

Actuator Monitoring



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*):

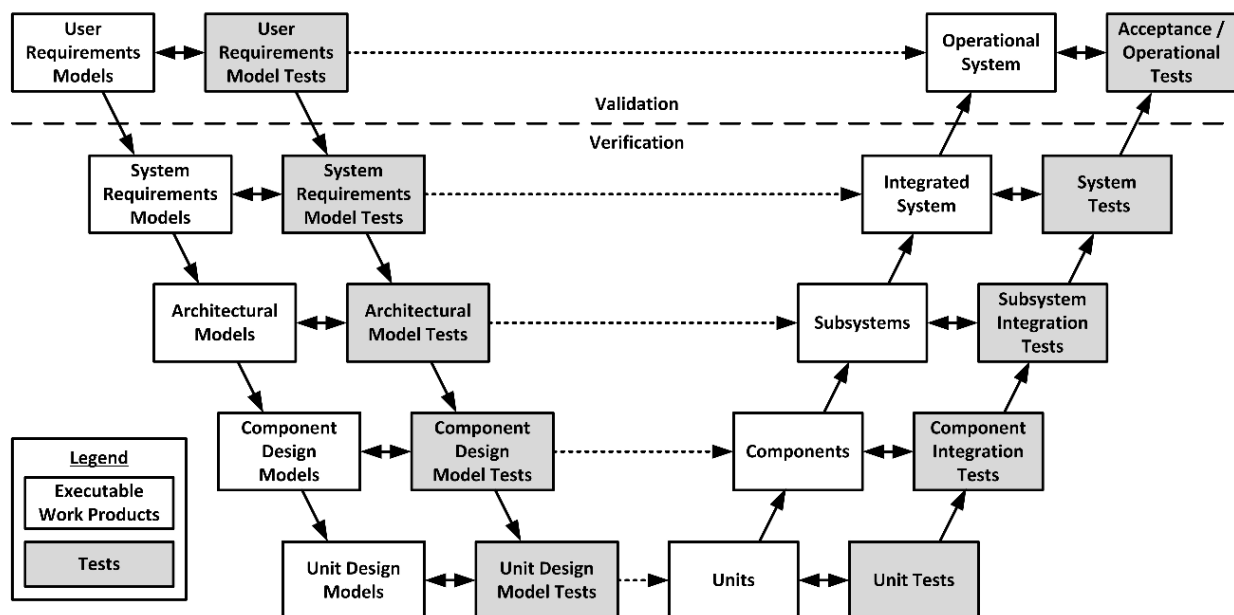
- Making a business case for the system
- Understanding the requirements that concern quality attributes

- 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

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!

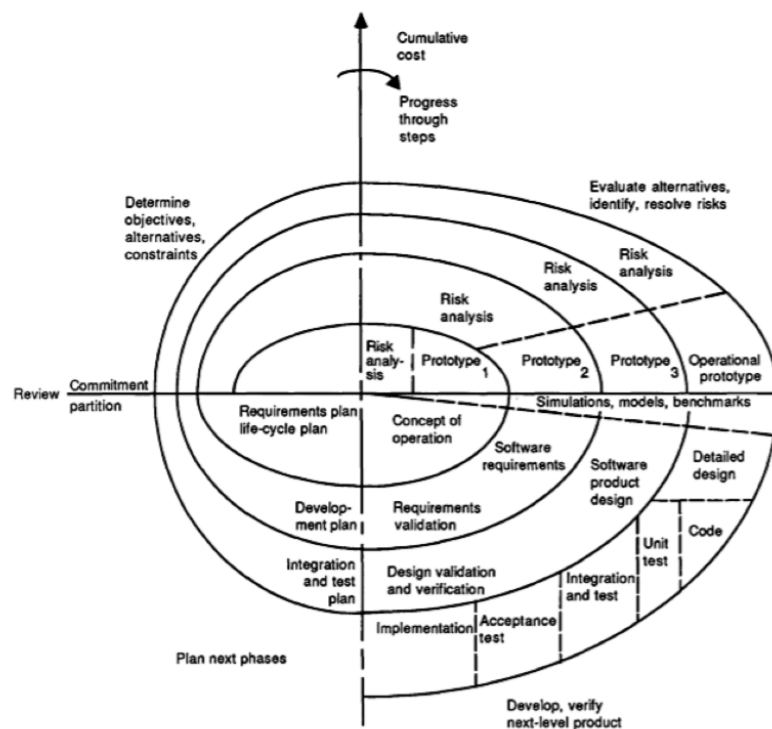
V-model



3.2.2 Spiral Model

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

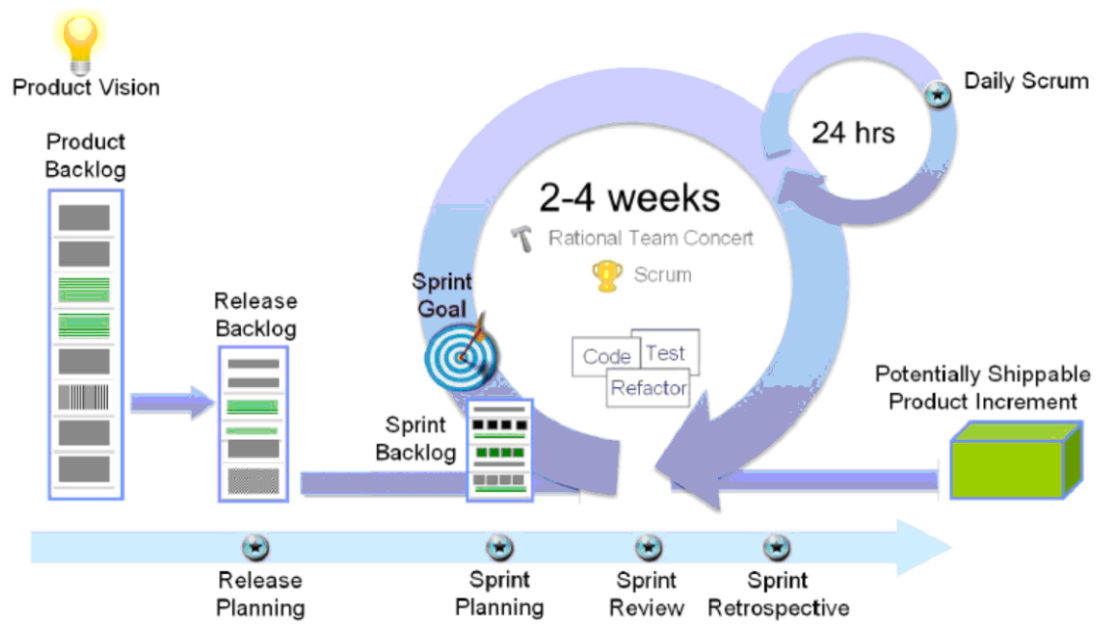


3.2.3 Agile Development

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.

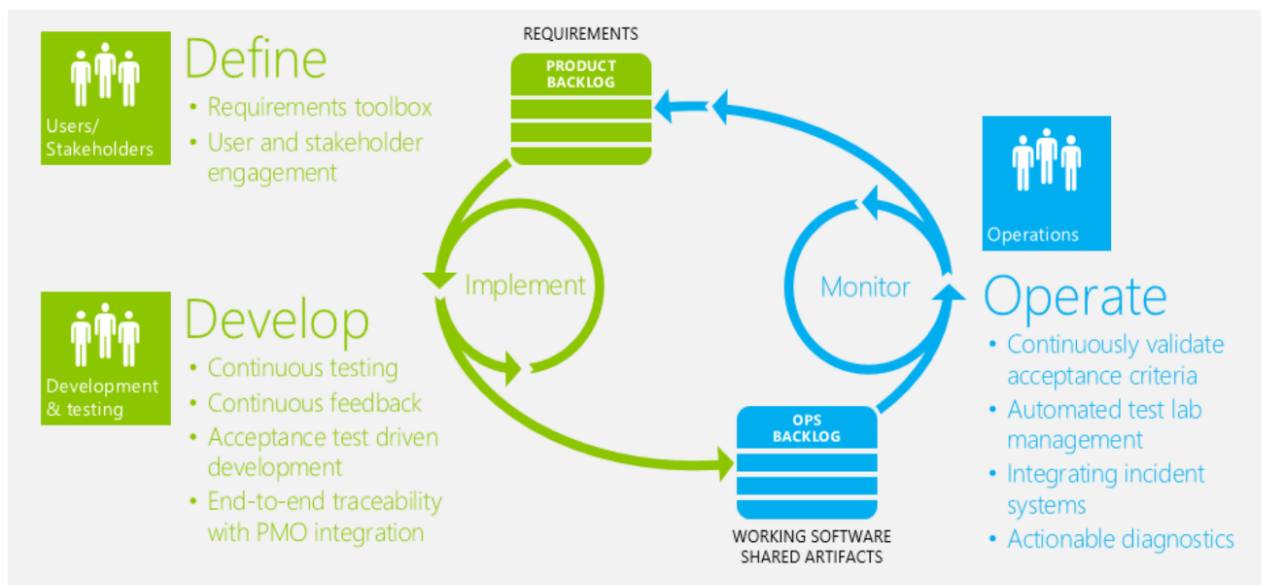
Agile

IBM Rational Solution for Agile ALM with Scrum



3.2.4 Agile + Devops

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:

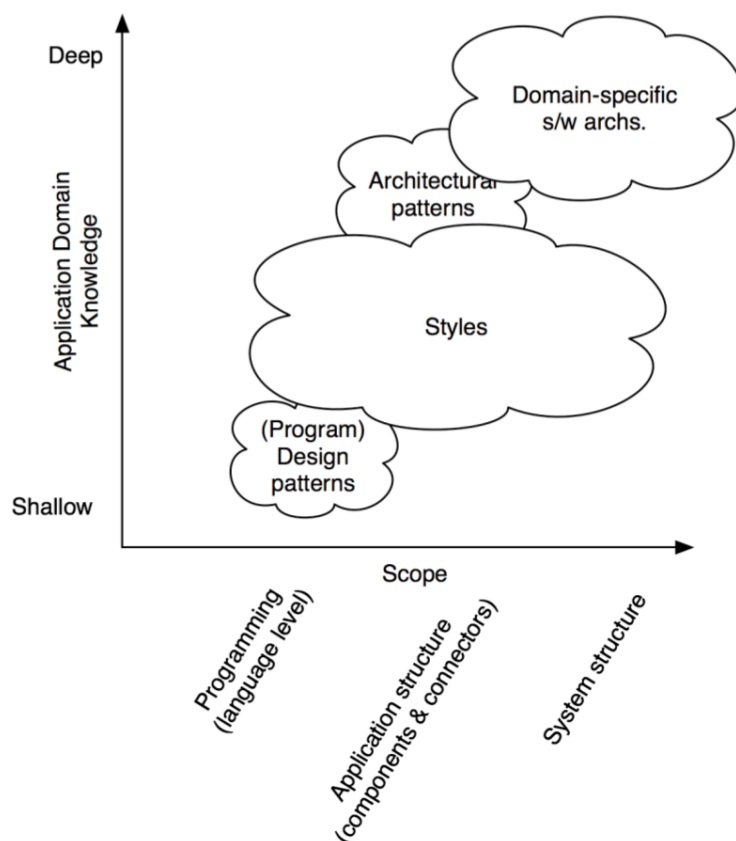
- A way of describing your expertise

- 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 impor-

tant 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:

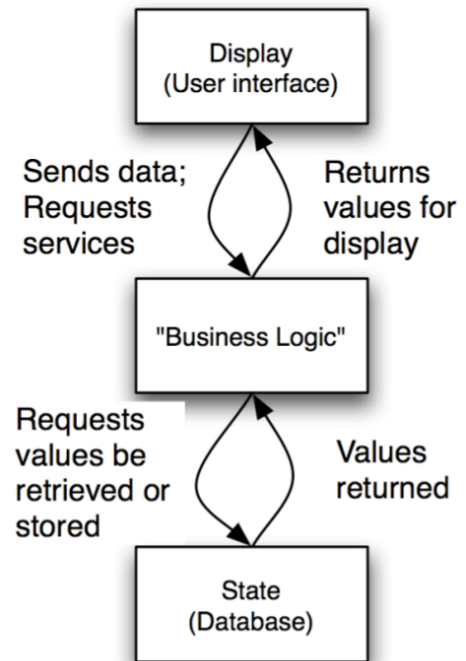
1. State Logic Display: Three-Tiered Pattern
2. Model View Controller Pattern
3. Sense Compute Control Pattern

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.

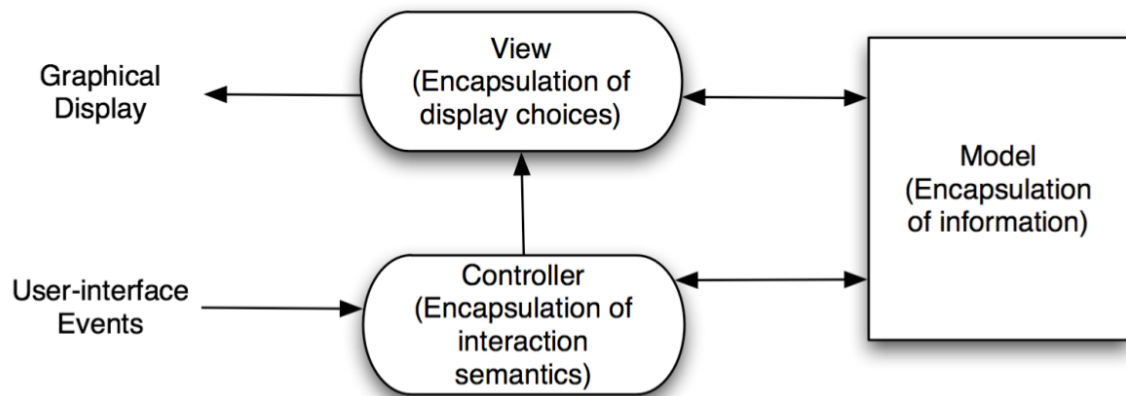
**** In design we use pre-decided structures and then alter/extend them as and when we need too. ****

State-Logic-Display: Three-Tiered Pattern

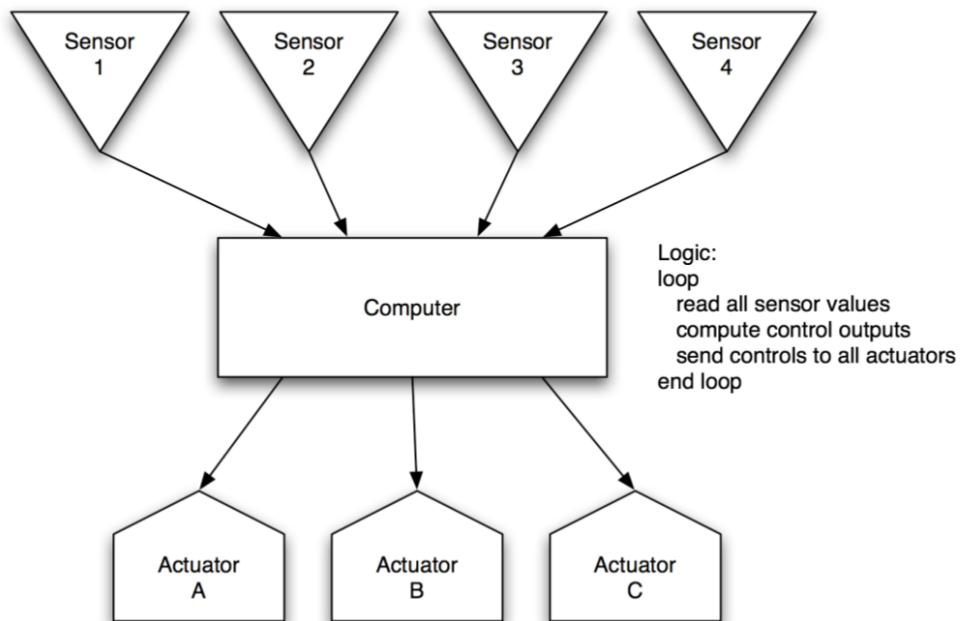
- Application Examples
 - Business applications
 - Multi-player games
 - Web-based applications



Model-View-Controller



Sense-Compute-Control



Objective: Structuring embedded control applications

4 Quality Attributes

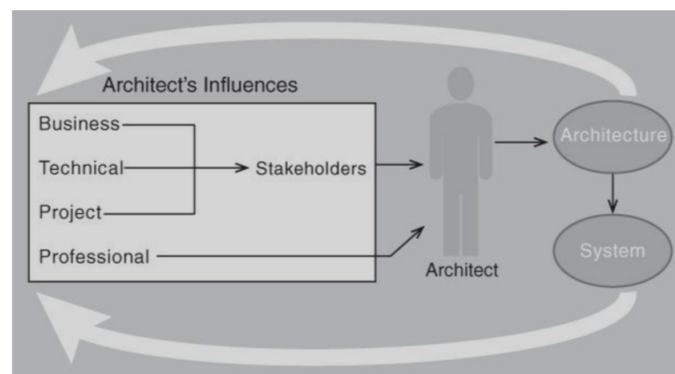
Quality Attributes specify, usually quantitatively, the **requirements** on particular parts of **functionality** or on the **whole system**. Software Quality Attributes are the **benchmarks that describe systems intended behaviour** within the environment for which it was built. The quality attributes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

Architecture Influence Cycle



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 mean 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*)

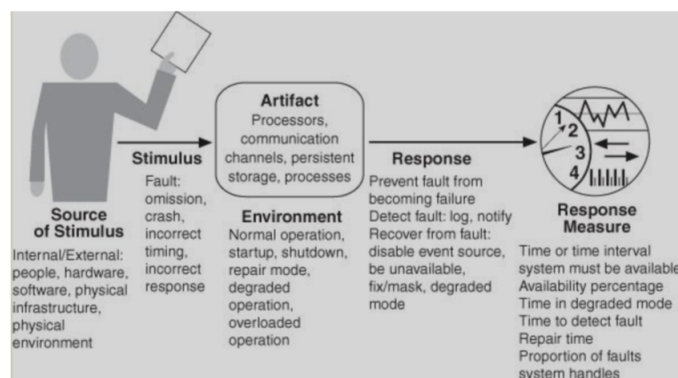
One Solution: explicitly state the use case of a quality attribute scenario, this aids the avoidance of some issues.

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*)

General Scenario for Availability



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**.

Example of Availability Tactics

Different tactics identify different responsibilities and how to achieve them. For exam for **availability**:

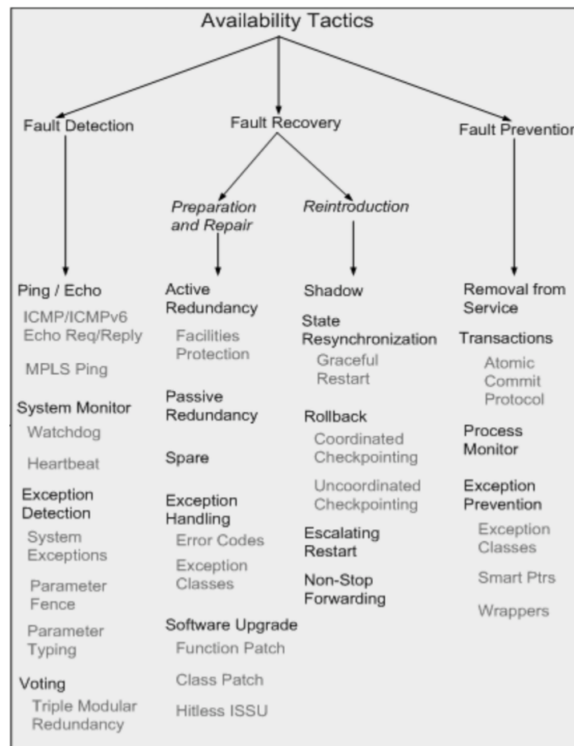


Figure 16: Availability Tactics

(Watch dog in next section!)

- Example₁ - Heartbeat is used in clusters to **provide availability of a shared resource**.

One system owns the resource and regularly sends out heartbeat to indicate system is alive. If heartbeat is missed the backup system takes over responsibility of the shared resource.

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 Generic 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

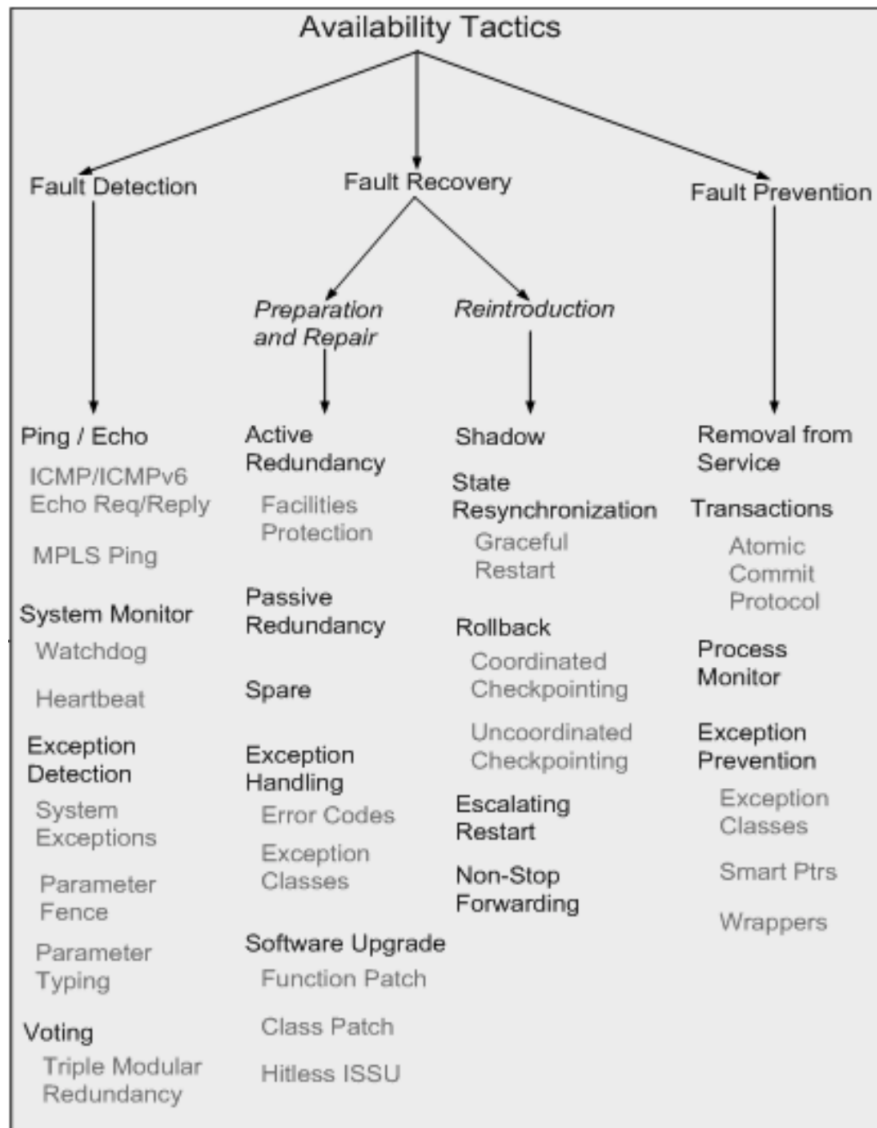
5.3 Concrete Scenario

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

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:



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 task 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

7 QA: Security

8 QA: Testability

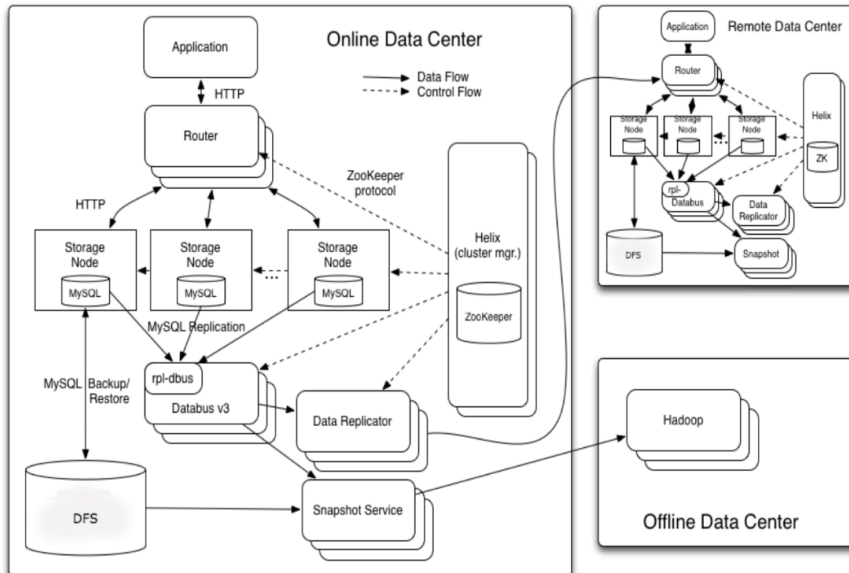
9 QA: Modifiability

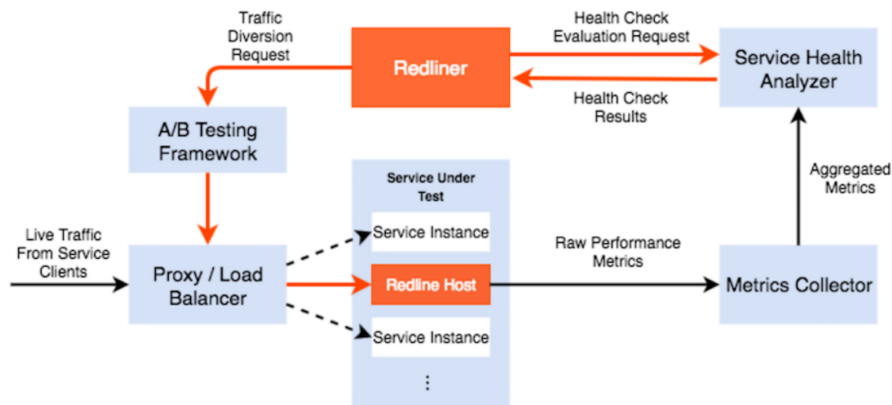
10 Connectors

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

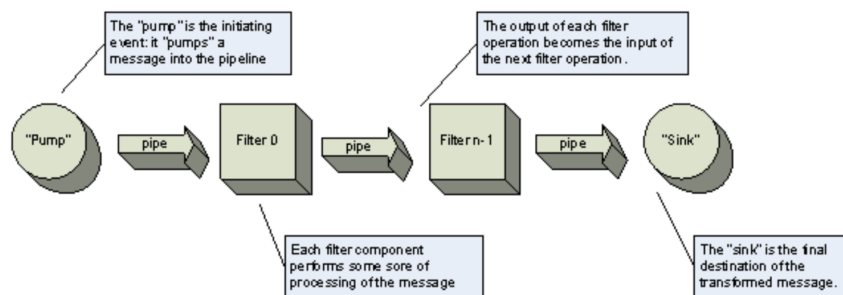
10.4 Communication

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 → enforce mutex access to resources)

10.8 Types of Connectors (Talyor, Medvidovic & Dashofy)

Connector	Description
Method/Procedure Call	Produce 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. <i>[Distributor connectos never exist by themselves, but provide assistance to other connectors, such as steams or procedure calls]</i>
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. <i>(adopters involve matching communication polices and interaction protocols among components, thus providing conversion services.</i>

11 Patterns

12 Modelling The Life-cycle

13 Dev-Ops

14 Product Line Architecture

15 Analysis

Why we Evaluate:

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

There are three types of Evaluation:

- Evaluation By Designer
- Peer Evaluation
- External Evaluation

15.1 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)*

15.2 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.

15.3 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?

15.4 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.