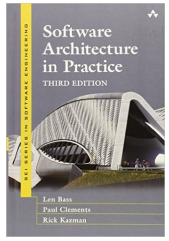
Software Architecture, Process and Management

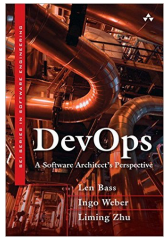
Stuart Anderson
soa@staffmail.ed.ac.uk
http://homepages.inf.ed.ac.uk/soa/

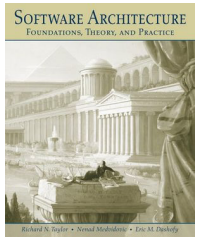
Organisation

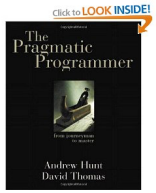
- Me: Stuart Anderson
- Email: soa@staffmail.ed.ac.uk
- Lectures: Mondays and Thursdays 17:10 - 18:00, Teviot LT, MEDS (Teviot Lecture Theatre (Doorway 5), Medical School Teviot)
- Coursework: Will be discussed in the next lecture
- Grades: 25% coursework, 75% exam

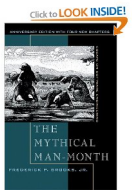
Books

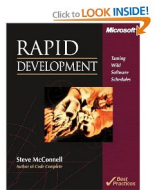


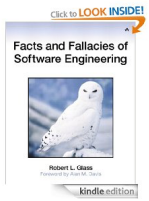












Learning Outcomes…

- Integrate knowledge of Software Architecture to capture Quality Attribute requirements for a system, evaluate proposed architectures against these requirements and synthesise options for improvement.
- Analyse and justify complex tradeoff 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 Software Architecture literature and practice to create a resource that support their learning in Software Architecture.

Outline

SAPM Schedule						
	Week			Topic		
16-Jan-17	1		1	Introduction		
19-Jan-17	1		2	Role of Architecture in Software Engineering		
23-Jan-17	2		3	Basic Concepts		
26-Jan-17	2		4	Architecture and Design		
30-Jan-17	3		5	Describing Architecture		
02-Feb-17	3			NO LECTURE		
06-Feb-17	4		6	Modelling		
09-Feb-17	4		7	Modelling		
13-Feb-17	5			NO LECTURE		
16-Feb-17	5		8	Visualisation		
20-Feb-17	ILW	ILW		NO LECTURE		
23-Feb-17	ILW	ILW		NO LECTURE		
27-Feb-17	6		9	Architectural Analysis		
02-Mar-17	6		10	Architectural Analysis		
06-Mar-17	7		11	Implementation		
09-Mar-17	7		12	Quality Attributes and Architecture		
13-Mar-17	8		13	Quality Attributes and Architecture		
16-Mar-17	8		14	Domain Specific Architectures		
20-Mar-17	9		15	Domain Specific Architectures		
23-Mar-17	9		16	Future of Software Architecture		
27-Mar-17	10		17	Recap and Revision		

What is Success for a Large Project?

- For the purposes of this course, a software project will be considered successful if:
  - The software is delivered on schedule
  - Development costs were within budget
  - The software meets the needs of users

Sometimes the Architecture Analogy is Informative

An Example

- The General Practice Extraction System: https://www.nao.org.uk/report/general-practice-extraction-service-investigation/
- "the General Practice Extraction Service (GPES), an IT system designed to allow NHS organizations to extract data from all GP practice computer systems in England. This data would be used to monitor quality, plan and pay for health services and help medical research.

National Audit Office Conclusion

- **The project has been significantly delayed and many customers have yet to receive data.** The original business case said the service would start in 2009-10, but it took until April 2014 for Health and Social Care Information Centre (HSCIC) to provide the first GPES data extract to a customer.
- **Mistakes in the original procurement and contract management contributed to losses of public funds, through asset write-offs and settlements with suppliers.** The total expected cost of the GPES programme increased from £14 million to £40 million during the planning and procurement stage. Further cost increases have been smaller, but the project has had at least £5.5 million of write-offs and delay costs. The need for the service remains and further public expenditure is needed to improve GPES or replace it.
- **Only one customer, NHS England has so far received data from GPES.** The time needed to design a new type of extract and restrictions in the contracts severely limits HSCIC's ability to provide data to those who request it. It is unlikely that GPES in its current form can provide the NHS-wide service planned.

Customers

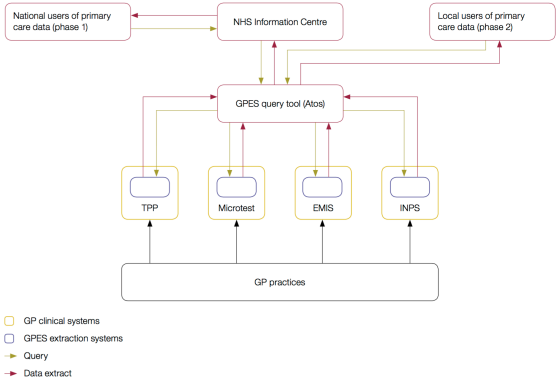
Figure 1
Proposed GPES customers and potential benefits

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

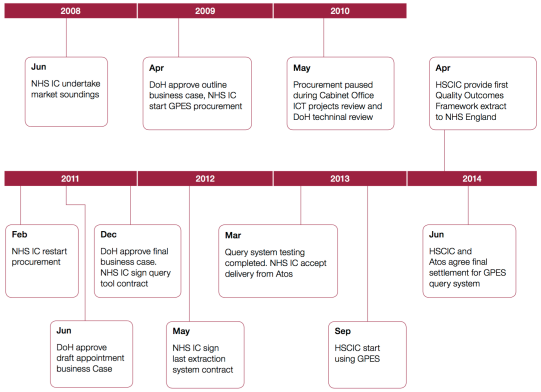
Figure 2
Design of the General Practice Extraction Service



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

Questions

- Looking at the structure (architecture diagram):
  - Can you identify any clear issues arising from the structure?
  - Who would the key stakeholders be in the issues?
  - Why do you think they might take different perspectives on the issue?

Data Extract Issue

- **2.11** NHS Connecting for Health did a technical review of GPES in early 2011, which recommended several signi cant changes to its design. In the original design, each GP system supplier would use a common 'query language' as part of their extraction system. This would allow the NHS IC to design a single extract centrally using the query tool, which all GP clinical systems could understand.
- **2.12** This technical review recommended an alternative where each supplier would be free to develop their own query methods. New queries would no longer be designed in the query tool using a common language, but would instead need to be designed as logical 'business rules' and sent to GP system suppliers to implement.

Data Extract Issues ctd

- **2.13** The NHS IC decided to abandon both the GPSOC contract approach and the common query language, as they could not agree either with the Department and GP system suppliers. They then procured the extraction systems by negotiating direct with the GP clinical system suppliers.
- **2.14** NHS IC's using a non-competitive procurement approach, plus the changes in design, contributed to the restrictive process for designing new extracts. As described in our findings on capability in paragraphs 4.4 to 4.5.
- **2.15** The HSCIC, as successor to the NHS IC and NHS Connecting for Health, has continued to use the GPSOC framework to require data sharing between NHS systems. The new framework, effective from 2014, says that principal clinical system suppliers must provide an interface method for third-party systems to use. This would improve interoperability between systems in GP practices and the health community.

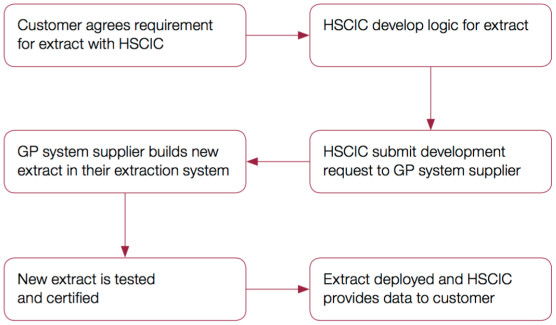
Data Extract Issue ctd

- **4.4** The HSCIC cannot do the wide range and scale of data extracts the NHS requests, because of the design of the GPES system and restrictions in supplier contracts. Customers have requested over 100 different data extracts from GPES, but the HSCIC estimate they will be able to design only 24 new extracts in 2015-16.
- **4.5** Figure 6 shows a summary of the HSCIC's process to develop a new extract, each of which the supplier designs and programmes from scratch. The HSCIC have limited exibility to amend extracts once developed, for example to change a time period and the speci c organisations it will extract data from.

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.

Architecture Definition

- *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)*
- *A software system's architecture is the set of **principal design decisions** about the system (R.N. Taylor et al)*

Architecture is a collection of structures

- We observe three frequently occurring types of structure:
  - **Modular structure:** static structure that focus on how functionality is divided up, structured, and assigned to development and implementation teams.
  - **Component and Connector structure:** runtime structures that focus on how components interact (e.g. information passing, synchronisation, interference,…)
  - **Allocation structures:** mapping to organizational, development, installation, execution environments. (e.g. Components are deployed onto hardware to execute)

Architecture is an Abstraction

- Architecture is used to suppress detail that is unimportant for the sort of reasoning we are doing.
- In particular it abstracts away from the private details of methods used inside modules or components.
- E.g. if we are reasoning about redundancy in an architecture we don't necessarily need to know about the detailed function just that we have a way of delivering it.

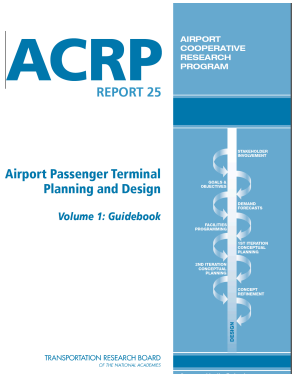
All systems have architectures

- Even if people have forgotten them…
- Complicated systems are embedded in organisations and we can often see architecture through practice:
  - How is a system deployed? This will often provide clues to allocation structures.
  - How is maintenance, evolution, issue reporting dealth 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

Architecture includes Behaviour

- Often we see box and line diagrams but they often do not characterise the behaviour of the boxes of the information passing in the lines (e.g. GPES diagrams).
- Usually we need more than empty boxes but this is often less than some very complete description of the behaviour of the box.
- We look at potentially many structures to capture an architecture so we can use many partial descriptions of behaviour (e.g. if we are concerned with reliability the mean time to failure of a box may be adequate).

Looking at Architectural Analogies may be helpful



- http://onlinepubs.trb.org/onlinepubs/acrp/acrp_rpt_025v1.pdf

Summary

- Architecture is about structure (and key decisions).
- Structure is about a set of elements and their relations.
- For software elements modular, component & connector and allocation structures are commonly useful.
- In architecture elements have abstracted behaviour.
- These abstracted behaviours help us reason about qualities of the system
- All systems have architecture.
- There is no such thing as “good” or “bad” architecture it is a way of describing systems. Analysing the structure in context makes decisions on good or bad.

Some Questions to Think About

- Some people argue that “pipe and filter” is an architecture:
  - Search for “pipe and filter architecture” and review some of the material.
  - What sorts of structures does this architecture allow you to build?
  - What are the main decisions the choice of this architecture makes for you.
  - Can you write down a pipe and filter application (using the linux tools) to take a text as input and create an alphabetical list of all the words that appear in the text where each word appears only once.

The Analogy with Architecture

Stuart Anderson

16 January 2017

Architects

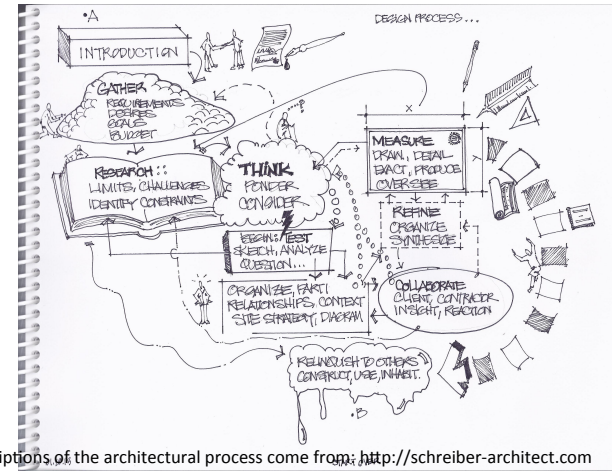
- Architects spend very little of their time designing houses.
- They design special purpose structures like bridges, museums, offices, art galleries, sports venues, …
- Many of these are “one off” and can include innovative and untried elements (not unlike novel software systems).

Scottish Parliament

Elbphilharmonie Hamburg

Berlin Brandenburg Airport

What do architects do?



Descriptions of the architectural process come from: http://schreiber-architect.com

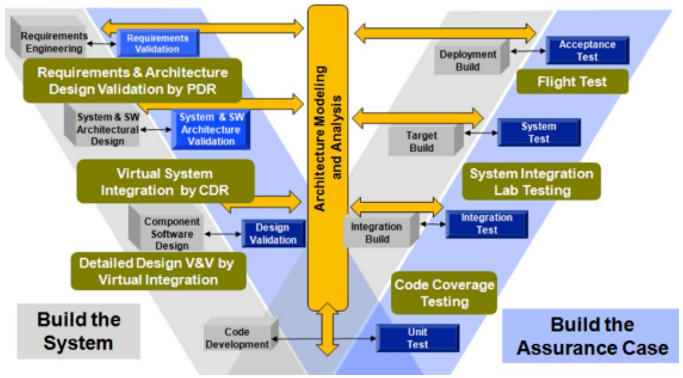
Architectural process

- Pre-design phase
- Schematic Design Phase
- Design Development Phase
- Building Department Submission/Plan Review Phase
- Project Bidding and Award Phase
- Construction Administration Phase
- Celebration

Analogy With Software Process?

- Buildings can be dangerous (collapse, inadequate safety within the building, …)
- Possibly the best analogy is with the sorts of processes used to develop **safety-critical software** (or at least software that has a regulator)
- Buildings provide services when they are in operation so we are looking at "operation" of the building being equivalent to the operation of a software system

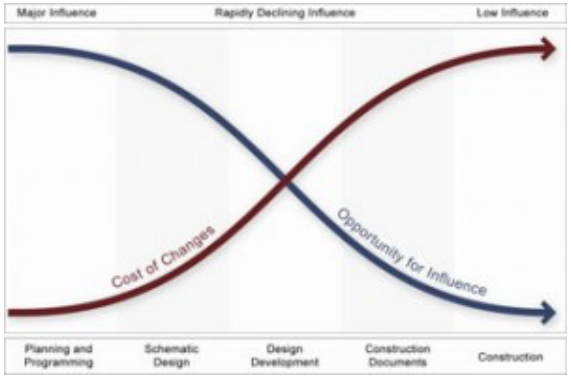
SEI Safety Lifecycle



Pre-Design Phase

- Roughly seems to correspond to requirements, including:
  - The types of spaces frequently included in the building type,
  - The space criteria (number of square feet per person or unit) for those spaces,
  - Typical relationships of spaces for these functions,
  - Typical ratios of net assignable square footage (NASF—areas that are assigned to a function) to gross square footage (GSF—total area to the outside walls) for this building type,
  - Typical costs per square foot for this building type,
  - Typical site requirements for the project type,
  - Regional issues that might alter the accuracy of the data above in the case of this project, and
  - Technical, mechanical, electrical, security, or other issues unique to the project type.

Costs and Ease of change



Schematic Design Phase

- "*The schematic design phase establishes the conceptual design of the project and illustrates the scales and relationships between project components. The primary objective of Schematic Design Phase is to arrive at a clearly defined, feasible concept and to present it in a form that achieves Client understanding and acceptance.*"
- Buildings have an aesthetic aspect often software doesn't (not as architecture).
- This is agreeing that the architect has the correct idea of how to realise the requirement − the right stuff is there and in the right relationship.
- "*Schematic design often produces a site plan, floor plan(s), sections, one or more elevations, and potentially other illustrative materials; computer images, renderings, or models. Typically the drawings include overall dimensions, basic finish materials, and a construction cost is may be estimated.*"

Planning/Zoning Board Submission Phase

- "*For other than single family and two family residences that do not require a variance, a typical Planning, Zoning, or Community Development Department will review projects for conformance to local ordinances and local master plans, which look at allowable site uses, environmental impacts (such as noise, shading, traffic impact, or pollution), density, parking, and other urban planning issues.*"
- This stage asks the question − does the project fit in the proposed context.
- For some safety critical systems (e.g. nuclear plant control) this might take place.

Design Development Phase

- *"The objectives of the Design Development Phase are different from those of Schematic Design Phase. The primary purpose is to define and describe all important aspects of the project so that all that remains is the formal documentation step of construction contract documents."*
- *"During the Design Development phase, the Architect establishes the building's relationships, forms, size and overall appearance through further development of the floor plans, sections, elevations, typical construction details, and equipment layouts. Preliminary specifications, which identify major building materials and systems, and establish quality standards, are also introduced during this phase."*
- The key element will be how to realise the structure that hold together the elements identified in the earlier stages. For software this might be a decision on the sort of architectural patterns to use and how to integrate off-the-shelf and custom components.

Construction Documentation Phase

- *"The Construction Documentation phase completes in detail the requirements for permitting and construction of the project. Floor plans, enlarged plans, wall sections, ceiling plans, power/lighting plans, finish schedules, window and door schedules, elevations, details, and written specification notes are added and refined to further establish the types of materials and systems required for the project. Mechanical, electrical, plumbing, fire protection and other building systems are integrated into the architectural envelope and conflicts are resolved. In larger buildings, communication with the system designers and engineers is expanded to insure a comfortable fit within the building, or request an equipment layout revision, or modify the building design if necessary."*
- Here we are looking at how resource is used and distributed by the building several layers of function and assessment of volumes of resource.
- These are also the basis for contractual matters and for approval.

Building Department Submission

- "*Upon completion of the Construction Documentation phase, the drawing package, including specifications, engineering drawings, and structural calculations, along with permit application fees, should be submitted to the Building Department for Plan review.*"
- Here an external authority checks that the proposal meets the building codes and is safe (i.e. it checks that the building will be safe provided it is properly constructed.) For safety critical software there will be restrictions on architectures, programming languages, processors, …

Project Bidding and Award Phase

- *"An architect during the bidding phase helps:*
  - *develop a list of qualified contractors*
  - *submit bid packages to bidders*
  - *reviewing submitted bids, analysing bids to help compare cost figures provided by bidders."*
- This is a procurement phase for software – who is going to construct the software: Detailed coding, V&V etc.

Construction Administration Phase

- *Construction Administration consists of the Architect reviewing shop drawings for various building components such as, roof trusses, floor trusses or engineered joists, steel frames, plumbing systems, electrical systems, and HVAC systems.*
- *Additionally the architect can administer the contract between the project owner and the Contractor*.
- For safety critical software this might be an independent Validation and Verification group who check the quality of the assurance case against the documentation created during the design process.

Summary

- The analogy between construction architecture and software architecture is quite strong.
- Buildings are safety-critical, subject to various standards and codes, and because of aesthetic and amenity constraints they need to demonstrate conformance both to planning and to building codes.
- Just like software architecture, building architects develop a specification for development that integrates custom and off-the-shelf components.

SAPM: Lecture 2
The Practical

Stuart Anderson

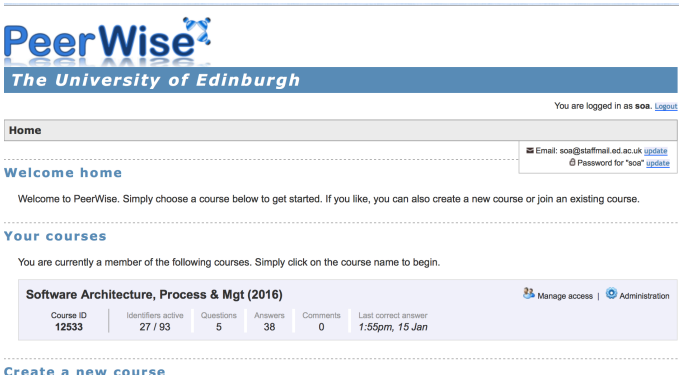
Overview

- Coursework is 25% of the final mark, exam is 75%
- The goal is to build a collection of software architecture case studies for the course.
- Groups of around 10 students will be responsible for developing an architecture case study.
- The goal is to develop a rich resource to support learning about SAPM

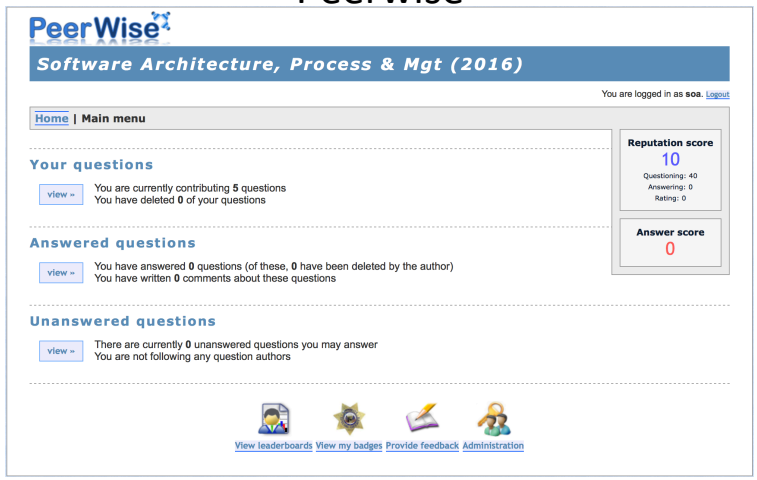
Contributions

- Peerwise questions (15%)
- Comments on other people's contributions (20%)
- Group participation (15%)
- Wiki page(s)(50%)
- Overall you should spend 25-30 hours on the practical.

Peerwise



Peerwise



Comments and Process

- You are expected to comment on other people's contributions. Comments will be graded and should be appropriate and helpful.
- Each group will develop a process to ensure the internal quality of their case study and links to the other case studies are appropriate. The quality of these processes will generate a group mark worth 15% of the overall grade.

Case Studies

- The idea of a case study is to demonstrate the use of the basic concepts of software architecture in a practical setting.
- This will involve you researching a deployed architecture and relating it to the concepts in SAPM and the other case studies as they are developed.

Case Studies

- Each person in the group will develop their own page(s) in the SAPM Wiki (You will have full access shortly).
- The main components are outlined on the next slide

Case Study Structure 1

- Page describing the internal and external review process (constructed by the whole group)
- Internal Review (overview of the case study plus a trail of review actions – mainly focussing on the coherence of the case study) – written by internal reviewer.
- External Review (how the case study relates to other case studies plus a trail of review actions mainly focussing on the relationship of the case study to other case studies. – written by external reviewer.

Case Study Structure 2

- Components and connectors used in the software architecture.
- The structure of the software architecture from different perspectives (static, dynamic, deployment…)
- The attributes (or non-functional properties the software architecture is intended to manage).
- Examples of the deployment of the software architecture in practical situations.
- Strengths and weaknesses of the software architecture.
- Comparison with other approaches to the problem the architecture is used to tackle.

Timeline

- Formative feedback deadline: 2 Mar 2017 – submissions before then will get feedback from me on the basis of the final grading scheme – focus is on how to improve. This is intended only to help you improve.
- Final deadline: 30 March 2017 – your contribution up to this deadline will be graded according to the grading scheme.

Length

- Wiki page contributions should be 500-1000 words roughly and can use diagrams or any other illustrative device (video, animation, audio, …). Think about accessibility of the material (e.g. Suppose someone is visually impaired – can they still access your material?)

Evaluation

- Each component (peerwise, comments, process, contribution) will be graded on four equally weighted factors (I will expand on this in the formal document):
- Relevance - to the course
- Understanding – does it illustrate your understanding?
- Quality – is it well-structured, accurate, comprehensible etc?
- Impact – how well you can relate your work to a practical context?

SAPM Lecture 3
Basic Concepts

Stuart Anderson

What is “good” Architecture

- The architecture is appropriate for the context of use. E.g. a 3-tier e-commerce architecture is not appropriate for an avionics project
- Guidance on “good architecture” focusses on:
  - Process
  - Structure
- Architecture should capture the **principal** design decisions about the system.
- The blueprint – focussing on Structure, Component Behaviour, Component Interaction and how that influences Quality Attributes of Systems

Process

- The architect team is 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
- Document using views that reflect stakeholder interest
- Evaluate the architecture in terms of how well it delivers the quality attributes
- Choose architectures that allow incremental implementation

Structure

- Use good modular structure: hide information, separate concerns, good robust interfaces that are unlikely to change
- Structure depends on perspective/viewpoint: at least **static** (captures e.g. dependency in code), **dynamic** (captures e.g. dependency in data flow), **deployment** (captures e.g. dependency on resources)
- Use well known patterns and tactics (see later) to achieve quality attributes
- Don't depend on particular versions of tools
- Modules producing data should be separate from those consuming data

Structure

- Don't expect simple mapping between modules (static structure) and components (dynamic structure)
- Don't depend on special features in the deployment environment unless essential
- Architecture should use a small number of ways of interaction between components
- Should be clearly identified resource contention issues e.g. if network capacity is a potential issue the architect should budget capacity across components or some other management approach

Importance of Architecture

- Software Architecture:
  - Enables us to manage the key attributes of a system
  - Allows reasoning about and managing change
  - Allows prediction of the key quality attributes
  - Allows better communication among stakeholders
  - Carries the earliest (most fundamental) design decisions
  - Defines constraints on implementation

Importance of Architecture

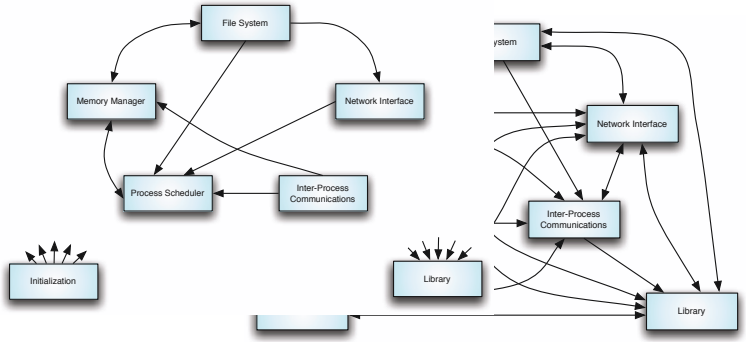
- Software Architecture:
  - Reflects the structure of an organisation
  - Provides the basis for evolutionary prototyping
  - Is the key artifact in reasoning about cost and scheduling
  - Can be used as the transferrable, reuseable model at the heart of a product line
  - Focusses on the assembly of components rather than on the creation of the components
  - Restricts design alternatives and channels developer effort in a coordinated way
  - Provides the basis for training new team members.

Structure

- Structure is slippery to describe
- Sometimes we want to be **prescriptive** (this is how it should be) − often too tidy
- Sometimes we want to be **descriptive** (this is how it is) − often a mess.

What about "Real" Examples?

Linux – Prescriptive Architecture



Linux – Descriptive Architecture

Copyright © Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. All rights reserved.

Managing Attributes

- Enables us to manage the key attributes of a system:
  - If performance is important we need to manage timing behaviour
  - If modifiability is important it is important to try to localise functionality so that modification is localised to change a function.
  - If managing project risk is important it is important to have clear responsibility for delivery of components and good traceability

Managing Change

- Allows reasoning about and managing change:
  - We can see three levels:
    - Inside an element
    - Between elements maintaining the architecture
    - Requiring architectural change
  - These distinctions help in assessing the cost of change
  - We can identify likely changes and try to architect to minimize architectural change

Prediction of Attributes

- Allows prediction of the key quality attributes
  - Should be able to build models on the basis of the architecture that prdict key attributes
  - E.g. if we want a system to be scalable:
    - the architecture should include:
      - capacity information,
      - dynamic component creation, and
      - dynamic deployment characteristics
    - These should be sufficiently detailed to be able to predict how increases in demand will be serviced

Improves Communication

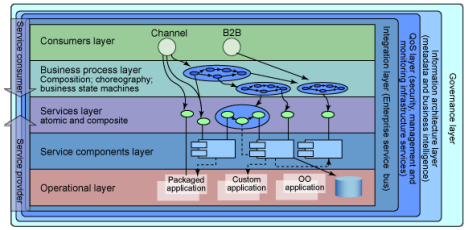
- Allows better communication among stakeholders:
  - User has particular requirements in terms of user experience
  - Customer needs to know about schedule, budget, meeting regulation in their market
  - Project manager needs to know the depndencies in terms of modules and components
- These might be accommodated by different views of the system that are consistent

Early Design

- Carries the earliest (most fundamental) design decisions:
  - What are the key quality attributes?
  - What architecture gives us control over thses attributes?
  - How do we characterise the behaviour of architecture elements?
  - E.g. suppose the attribute is that we want to know whether the system is faulty what does this say about architecture?

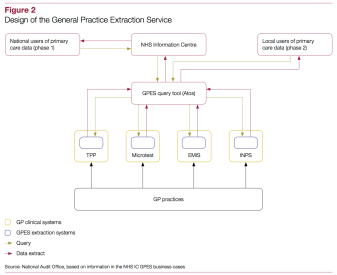
Constraints

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



Structure of Organisation

- Reflects the structure of an organisation:
  - Of the development organisation.
  - Of the overall organisation
  - The work organisation shapes the architecture
  - The architecture shapes the work organisation.



Evolutionary Prototyping

- Provides the basis for evolutionary prototyping:
  - For example:
    - Plug and play – early experience of the base functionality + extensibility
    - Real time architectures – early experience with scheduling – worst case execution times guide design, and deployment

Cost and Scheduling

- Is the key artifact in reasoning about cost and scheduling:
  - Capture dependencies
  - Estimate required efforts etc
  - Allocate effort to elements
  - Understand how elements influence each other
  - Use architecture to interpret bottom-up estimates from teams working on elements

Product Line

- Can be used as the transferrable, reuseable model at the heart of a product line
  - Elements are assets that can copmpose to give new functionality
  - Architecture provides the means to compose the elements
  - Planned approach to the reuse of architectural elements

Component Level

- Focusses on the assembly of components rather than on the creation of the components:
  - With well designed elements and architecture we can combine elements from different producers provided they conform to interface standards. Brings benefits:
    - Decrease time to market
    - More reliability
    - Lower cost
    - Flexibility e.g. using multiple or alternate suppliers for a component.

Channels Development

- Restricts design alternatives and channels developer effort in a coordinated way
  - Provides a defined context for the developer
  - Well defined interfaces and clear ide of the functionality and the quality attributes required
  - Clarity on what is an architectural decision and what is a development decision (e.g if fault tolerance is important we don't want to do that inside a module) we should use architecture.

Training Resource

- Provides the basis for training new team members:
  - Multiple views provide different stakeholder perspectives
  - Records interactions between elements
  - Abstracts from detail to provide an overview

Summary

- This is a long list of good things we get from software architecture.
- This is an overview of some of the potential benefits of architecture.
- We will dig a bit deeper in subsequent weeks

SAPM Design Example Availability

30 January 2017

Availability

- Some systems we need to be there whenever they need to be used.
- These are usually called high availability systems.
- There can be different reasons for high availability:
  - 999 (or 911 or …) telephone system
  - Interplanetary spacecraft systems
  - Electricity supply grid
  - Large (and expensive) computer power supply

Availability

- From hardware there are two key measures:
  - *mtbf – mean time between failures*
  - *mttr – mean time to repair*
- The *availability* of a system is usually defined to be the probability it will be there when you ask it to work: *mtbf/(mtbf+mttr)*
- So there are two ways to make this number bigger (i.e. closer to 1): make the mean time between failures longer, make the system faster to repair

Faults, Errors, Failures

- A *fault* is something in the system (e.g. a broken wire, failed component, wrong bit of code, …) that can cause the system to move into an *error* state when the fault is activated, an error may then eventually cause an externally observable deviation from the intended operation and this is called a *failure.*
- Most high availability systems try to tolerate or mask faults by detecting erroneous conditions before they move into failure conditions.

Faults, Errors, Failures, Example

- A *fault* in a sorting routine means that under some circumstances it fails to sort an array.
- Under there 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.
- 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.

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***

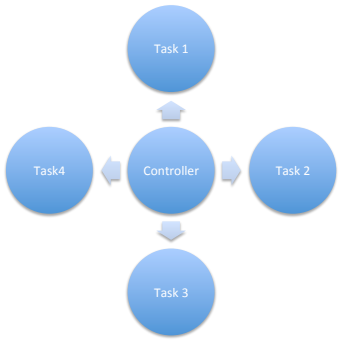
A Concrete Scenario

- 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?
- So the concrete scenario might be as follows.

A Concrete Scenario

- Source: internal task process
- Stimulus: either omission or timing depending on how you look at it but the task does not respond as expected.
- Environment: normal operation
- Response: abandon task
- Response measure: system always responds within 200ms

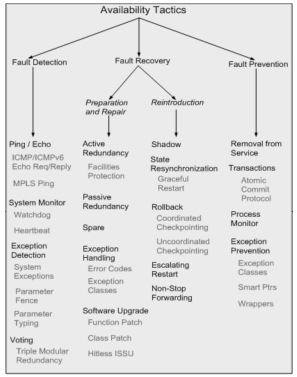
Simple Approach



- Cycles through each of the tasks.
- Passes control to the task
- Waits for control to pass back.
- What happens if a task fails?
- This architecture fails the scenario

Availability Tactics

- Our current architecture fails the scenario, why?
- Because we can't detect the error arising from the fault in a task.
- We look at the tactics to see how we might fix this.



What tactic might we take

- Use a watchdog: https://en.wikipedia.org/wiki/Watchdog_timer
- This would enable a correct response in the absence of a response from a task.
- Does it solve all the problems?
- What else might be necessary?
- Look at the tactic list

Architectural Design Decisions

- We can specialise our 7 categories to consider availability:
  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

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 our example we introduce the watchdog and give it responsibility for responding to error.

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.

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.

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.

Mapping between 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 reinitiallised, 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.

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.

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.

Summary

- Availability is a good example QA.
- We looked at the definition.
- We saw how a scenario might capture a testable Availability requirement.
- We saw how an architecture might fail such a scenario.
- We say how tactics suggest ways to improve architectures.
- We also saw how the seven architecture design decisions map onto Availability as an example.

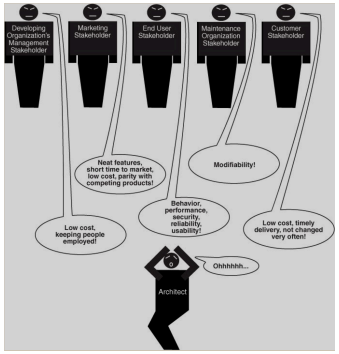
SAPM Meeting 4
Quality Attributes

Architect Influences

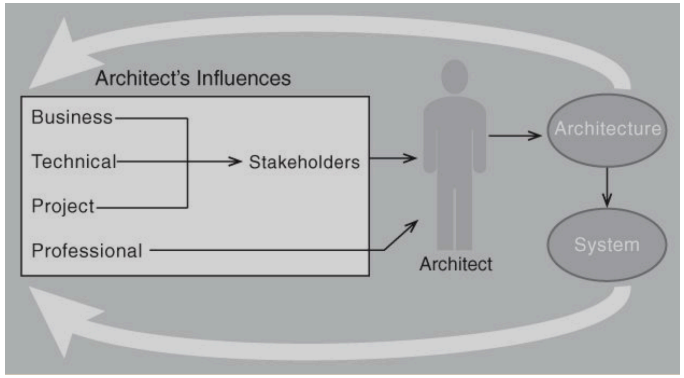
- We saw in the previous lecture that architects are influenced by business, technical, project and professional contexts.
- In the case of business, technical and project contexts these influences are carried by stakeholders.
- Stakeholders represent different perspectives on the system and attempt to influence the architect.
- Later we will see how the architect needs to tradeoff the different influences of stakeholders.

Stakeholders

- Stakeholders typically have conflicting perspectives on systems.
- For example, marketing might like to have as big a market as possible but the maintenance organisation might want the system to be a simple as possible.
- Architecture is the right level of abstraction to resolve these conflicts.



Architecture Influence Cycle



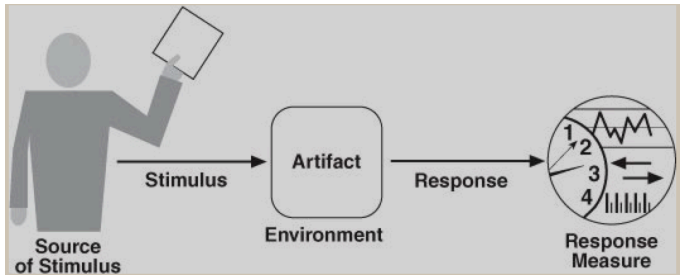
Requirements

- Functional requirements: these specify what the system does, architecture is important to this by structuring the containers that hold functionality.
- Quality Attributes: these specify, usually quantitative, requirements on particular bits of functionality or on the whole systems (e.g. that the system should be available 99% of the time).
- 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

Problems With Quality Attributes

- Often QA requirements are not "testable", for example 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 QA. 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).
- The solution is to use quality attribute scenarios to provide sufficient specificity to avoid some of these issues

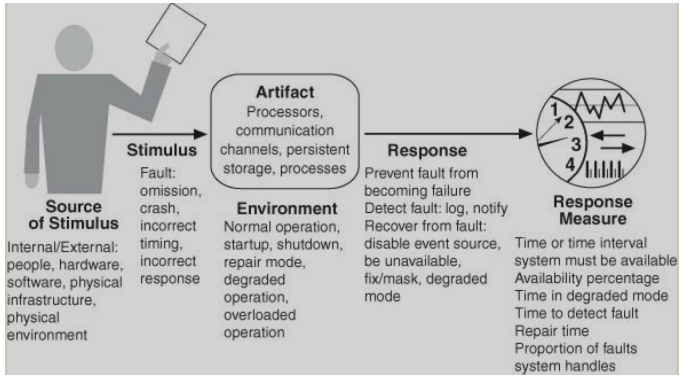
Quality Attribute Scenarios



QA Scenarios

- QA Scenarios have the following components:
  - Source of stimulus: might be a person or another system e.g. a system administrator.
  - Stimulus: this is something that the system needs to respond to e.g. using the wrong configuration specification for the system.
  - Environment: This is the broader context that captures wider aspects of the system e.g. that the system is starting up
  - Artifact: this is the part of the system that is stimulated e.g. the configuration checker in the system
  - Response: The response is the activity resulting from the stimulus e.g. the configuration issue is identified and then repaired
  - Response measure: this is how to measure the response so the scenario is testable e.g. time to detect the wrong configuration and the time to repair.

General Scenario for Availability



General and Specific QA Scenarios

- Each QA has a general scenario associated with it that tries to capture the possible components involved in that particular QA
- This acts as a template or guide for the architect specifying a specific QA Scenario.
- Specific QA scenarios take account of specific stimuli and measures on response, they capture the specification of the QA for a particular system.

Architectural Tactics

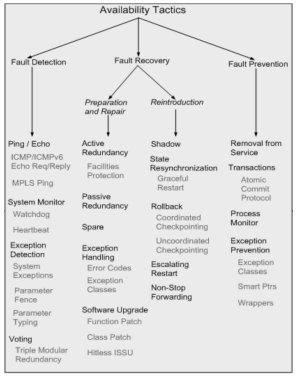
- An *architectural tactic* is a design decision that influences the achievement of a quality attribute response.
- Tactics are focussed on single QAs and don't try to take tradeoffs into account.
- Tactics:
  - Gather the options available to the architect
  - Are more primitive than patterns.
  - Systematically enumerate possible design decisions

Example Tactics

- Tactics are often generic and need to be specialised to a specific context. Example tactics:
  - *Schedule resources:* this is common in refining performance or responsiveness but the particular approach to scheduling will depend on the context.
  - *Use an intermediary:* this is common in modifiablility (e.g. an intermediary might be a layer to control access to some functionality) but the choice of intermediary will depend on the context
  - *Manage sampling rate:* this is already quite specific and will usially apply to real-time systems but it may not be applicable to all real-time systems.

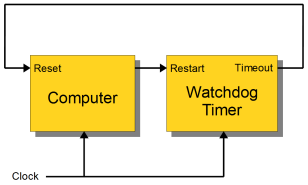
Availability Tactics

- For an example QA and its tactics read: http://www.sei.cmu.edu/reports/09tr006.pdf
- The tactics identify different sorts of responsibilities and how to achieve them



Example Availability Architectures

- Watchdog times ensure the availability of systems by initiating a restart in the event of apparent inactivity.
- Watchdog timers are used extensively by the Jet Propulsion Laboratory in space mission development. Read the following slideset to get an idea of the issues: https://indico.esa.int/indico/event/62/contribution/16/material/1/0.pdf



By Lambtron - Own work, CC BY-SA 3.0, $3

Heartbeat

- Used in high availability clusters.
- Used to provide availability of a shared resource.
- One system "owns" the resource and regularly sends out a heartbeat message indicating the system is alive and in charge of the resource.
- If the heartbeat is missed a backup system takes over and takes over responsibility for the shared resource.
- This is important where the resource can be updated and we want to ensure only one system is the gatekeeper for updating

Categories of Architectural Design Decisions

- There are seven broad categories of design decision:
  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

Allocation of Responsibilities

- Identify the important responsibilities
- Determining how to allocate these to runtime and static elements.
- These are often specific to a QA and so the specifics depend on the QA, for example in the case of availability *fault detection* is an important responsibility that will be further decomposed and distributed in the architecture.

Coordination Model

- Components in the architecture interact with one another via a collection of mechanisms. This is called the coordination model.
- We need to decide on:
  - What elements in the system need to coordinate with one another.
  - What properties the coordination needs to have (e.g. timing properties, security of coordination, …)
  - Choosing the mechanisms (ideally a small number) and these have properties like statefulness, synchrony, delivery guarantees, performance properties.

Data Model

- Choosing abstractions, operations, and properties. How data is created and destroyed, access methods, ...
- Maintaining metadata that controls the interpretation of the data.
- Organising the data, what kind of system will be used to store it, how will it be backed up, how do we recover from data loss

Management of Resources

- Resources can be hard (CPU, memory, battery, …) or soft buffers, processes, …)
- Resource Management includes:
  - Identifying resources to be managed
  - What system element should manage a resource
  - Work out sharing strategies and how to arbitrate in contention situations
  - Consider the consequences of running out of a resource (e.g. Memory).

Mapping among Architectural Elements

- There are two important types of mapping:
  - Mapping between different types of elements in the architecture e.g. from static development structures (modules) to execution elements e.g. threads or processes.
  - Mappings between software elements and environment elements e.g. from processes to specific processors and other hardware.
- The main appings are code to runtime structure; runtime elements to environment; data model elements to data stores

Binding time decisions

- This looks at when you might want to make a decision between a predetermined range of options.
- This can range from design time when a designer might allocate a responsibility through to runtime when an end user might allocate a responsibility.
- Often these are about allowing variability in some of the decisions we have already seen. E.g. we might want some variability in the resources to be managed determined at run time or we might make the coordination model negotiable at runtime if we want to interoperate with a range of systems.

Choice of Technology

- This is critical to being able to realize all the other decisions in a concrete system. We need to consider:
  - What technologies are available
  - What tools are available to support technologies
  - How much training will it take to be able to use a technology?
  - What are the full range of consequences of the choice of a technology (e.g. it may restrict markets because it is incompatible with some other technologies).
  - If the technology is new, how does it fit into the existing preferred technologies for the organisation.

Summary

- We looked at the architecture influence cycle to see the role of stakeholders.
- We then looked at QA Scenarios as a means to make QA requirements testable.
- We looked at architectural tactics as a way of documenting routes to achieve a QA requirement.
- We looked at availability as an example.
- Finally we considered the main areas of architectural design decisions.

SAPM Meeting 4
Contexts for Software Architecture Shaping Design

Stuart Anderson

Contexts for Software Architecture

- 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: where architecture supports technical activity like measurement, V&V, compliance,…
  - Project lifecycle: where architecture interacts with and supports development process
  - Business: where architecture supports organisations e.g customer organisations and development organisations.
  - Professional: where the role of architect defines requirements and constraints on architects.

Technical Context

- Architectures provide a means of controlling quality attributes of the system
  - In the context of design activities we try to choose architectures that enable the attributes we care about.
  - In the case of analysing existing systems as part of reengineering or evolving the system we may find that the architecture can inhibit particular attributes.
- Architectures don't often have much to say about functionality because architectures provide containers for functionality

Controlling Quality Attributes

- Usually we care about several quality attributes at once. For example:
  - The availability of a system where we worry about ensuring there is a system to take over if a system fails.
  - The safety of a system where we have to worry about ensuring that the system only behaves as intended and has no additonal behaviour.
  - The testability of a system where we are concerned about ensuring:
    - elements are clearly able to be isolated
    - we know what behaviour to expect of components of the system
    - we know how components relate to modules so we can track down faulty code
    - We know how components are intended to integrate to give the overall behaviour
- Other quality attributes are things like performance, usability, interoperability,..

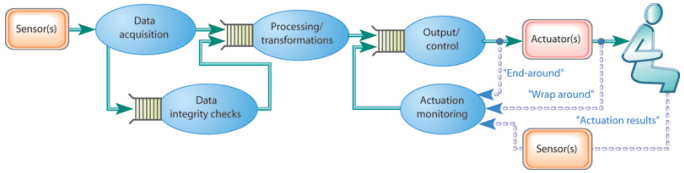
Safety Critical Example

- In the following diagrams:
  - orange blobs are sensors that sense the environment
  - Pink blobs are actuators that change the world around the system by exerting forces on it.
  - Sensors and actuators usually have software embedded in them.
  - blue blobs are components
  - the person icon is the system under control
- Safety systems are constructed as channels that read information from sensors and tell actuators what to do.

Testability Concerns

- In building a safety system we might worry about two aspects of actuators:
  - Actuators are mechanical things so they always fail in the end because they wear out.
  - How to test that that the actuators behave as expected by the safety system.
- To handle this we might us an actuator monitoring architecture.

Actuator Monitoring



http://www.embedded.com/design/prototyping-and-development/4006464/Architecture-of-safety-critical-systems

Actuator Monitoring

- Here we include an actuator monitoring component that compares three things:
  - What the actuators are told do by the system
  - What the actuators do in response
  - How the system responds (this may require additional sensors to detect the response)
- The actuator monitoring will cause the system to fail if it detects an anomaly
- Notice we have said nothing about functionality here

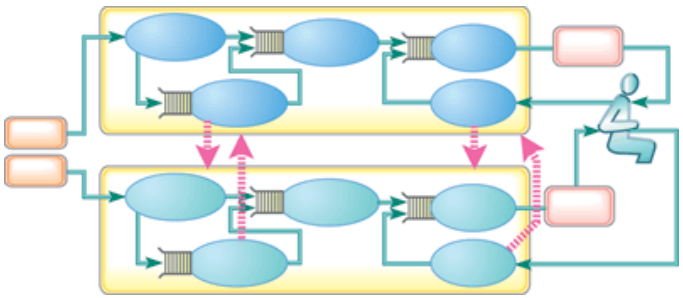
Actuator Monitoring

- At the architectural level:
  - We don't consider the functionality of the monitor or what sensors we might need.
  - We might set a target in terms of how often we are prepared to tolerate an undiagnosed sensor failure (because this is a quality attribute).
  - These sorts of decisions interact with costs (because adding extra sensors has costs)
  - Sensors also fail so we may need to take that into account

Availability

- After looking at the actuator monitoring issue we might worry that the system will fail too often and will take too long to repair.
- The standard approach to this is to use a dual channel architecture where two channels operate concurrently one is in control the other a backup and we switch to the backup if the first system fails.
- This is the architectural approach to managing availability issues. Here there are two copies of our earlier architecture.

Availability



http://www.embedded.com/design/prototyping-and-development/4006464/Architecture-of-safety-critical-systems

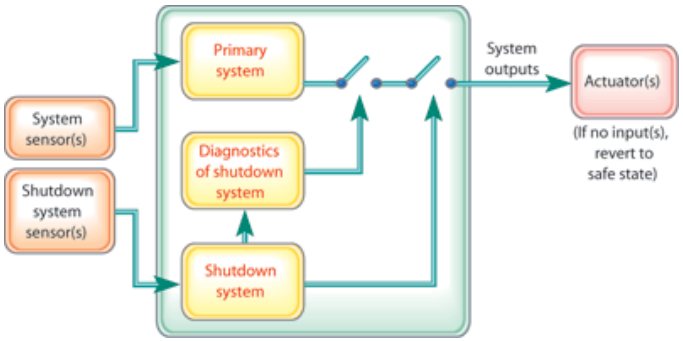
Availability

- We might want to assess how much benefit we would gain from this approach.
- At a detailed design level we might want to look at features in the functionality that cause coordinated failure in both systems
- At the architectural level we might only want to know how correlated failures are between two copies of the system. This would let us assess the impact of the architectural change.

Safety

- Looking at our dual channel, actuator monitored system it might still be the case that the consequences of failure may be dangerous.
- If the system has a "safe state" that we can set the actuators to achieve we might want to use a shutdown system to achieve this.
- See the shutdown architecture to achieve this

Safety



http://www.embedded.com/design/prototyping-and-development/4006464/Architecture-of-safety-critical-systems

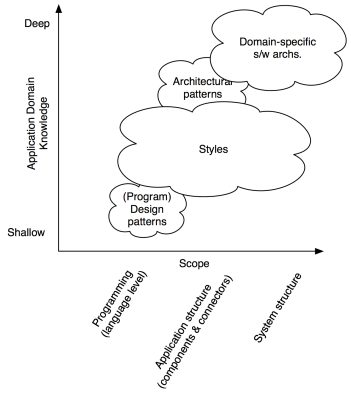
Safety

- Now our dual channel system is the "primary system" this might be quite complex because it attempts to achieve good control over the system.
- If the primary system fails the system under control may become dangerous.
- The shutdown system is very simple and just detects when the system is outside normal operation then it shuts the system down
- Shutdown is achieved by disconnecting from the actuators and they go into safe mode that brings the system under control to its safe state.
- For additional safety we also monitor the shutdown component to detect failure.

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 (at varying levels of completeness) that can be reused.
- These reusable structures also influence other aspects of context, helping to shape processes, organisations and professions.
- We can plot different sorts of architectural structure depending on the degree to which it is specific to a domain and the extent to which it influences the system.

Patterns, Styles, Domain Specific Architecture

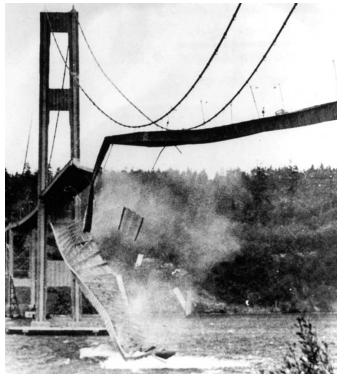


Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

Domain-Specific Software Architectures

- A DSSA is collection of (pre-decided) design decisions that
  - Capture important aspects of particular tasks (domain),
  - Are common across a range of systems in the domain
  - Typically they will have some predefined structures depending on the attributes we want to control and a disciplined way of extending such structures.
- These are not general purpose because they incorporate many specific characteristics of the domain.
- The main benefit is the extent to which such domain-specific architectures capture design knowledge acquired by the community (but we do forget things – see over…).

Tacoma Narrows Bridge



Just four months after Galloping Gertie failed, a professor of civil engineering at Columbia University, J. K. Finch, published an article in Engineering News-Record that summarized over a century of suspension bridge failures. In the article, titled "Wind Failures of Suspension Bridges or Evolution and Decay of the Stiffening Truss," Finch reminded engineers of some important history, as he reviewed the record of spans that had suffered from aerodynamic instability. Finch declared, "These long-forgotten difficulties with early suspension bridges, clearly show that while to modern engineers, the gyrations of the Tacoma bridge constituted something entirely new and strange, they were not new--they had simply been forgotten."

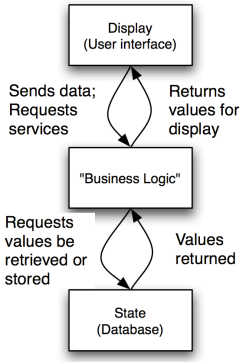
An entire generation of suspension bridge designer-engineers forgot the lessons of the 19th century. The last major suspension bridge failure had happened five decades earlier, when the Niagara-Clifton Bridge fell in 1889.

Architectural Patterns

- An architectural pattern is a set of architectural design decisions that are applicable to a recurring design problem, and parameterized to account for different software development contexts in which that problem appears.
- Architectural patterns are similar to DSSAs but capture less of the behaviour and attributes of the system but are more general because they are intended to abstract a common pattern over several domains.

State-Logic-Display: Three-Tiered Pattern

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

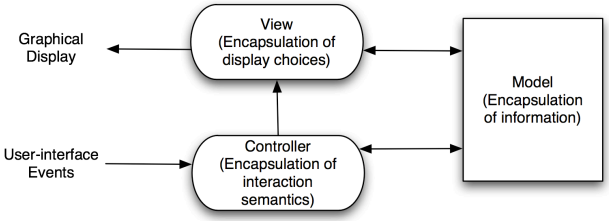


Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

Model-View-Controller (MVC)

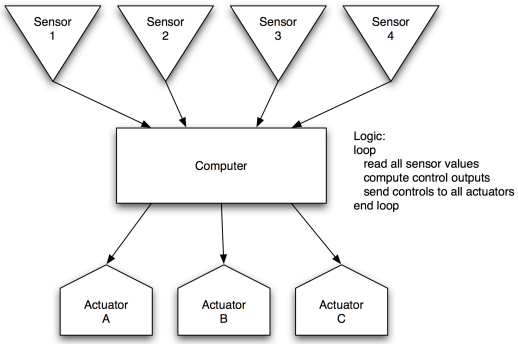
- Objective: Separation between information, presentation and user interaction.
- When a model object value changes, a notification is sent to the view and to the controller.
  - Thus, the view can update itself and the controller can modify the view if its logic so requires.
- When handling input from the user the windowing system sends the user event to the controller.
  - If a change is required, the controller updates the model object.

Model-View-Controller



Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

Sense-Compute-Control



Objective: Structuring embedded control applications

Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

Contexts Shape Design

- The technical context identifies features we want to control and packages a range of other properties.
- Standard architecture (patterns, domain specific architectures etc.) package these. The other context we consider also help to shape the choice of architecture.
- Design uses pre-decided structures and then alters and extends structure as necessary.

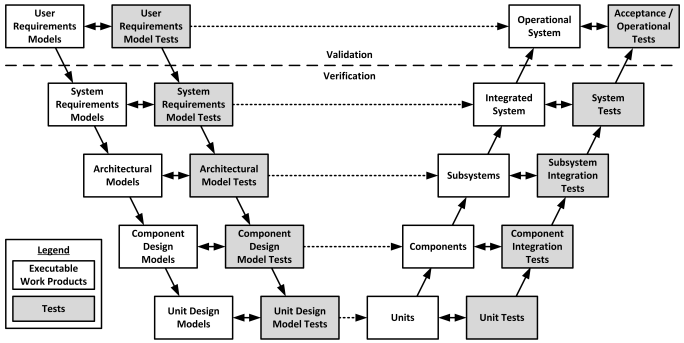
Project Lifecycle Context

- We will consider this in more detail later in the course. We know that whatever lifecycle we use we will need to do the following (and these are all done best by talking about architecture:
  - Make a business case for the system
  - Understanding 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
- These take place in all lifecyles but in different places

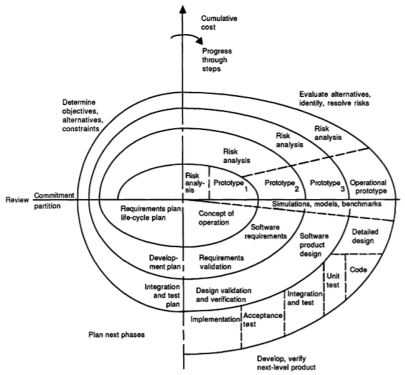
Lifecycle Models

- We will consider three models (or classes of models in future lectures):
  - The V-model which is a development of waterfall and explicitly includes architectural design as a stage.
  - Iterative models as exemplified by Boehm's spiral model that focusses on project risk management. Here architecture that gives us vision of how development tasks are interrrelated will play a key role as well as ensuring quality requirements are met.
  - Agile – here some decisions are more stable than others and the more stable ones concern architectures. We will return to this later in the lecture series.

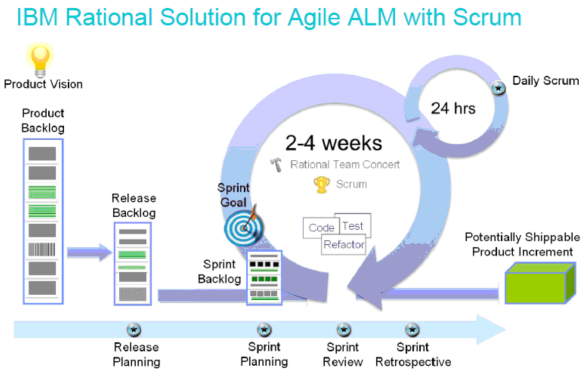
V-model



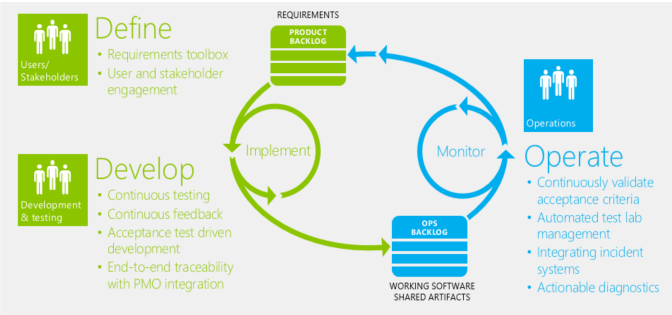
Spiral Model



Agile



Agile + Devops



https://agilepearls.wordpress.com/tag/devops/

Business context

- Here we will consider two aspects in later lectures:
  - How the organisational structure of stakeholders can drive architectural decisions and shapes decision taking around architecture.
  - How architectural expertise drives the structure of development organisations in terms of their functional units and interrelationships.
- This is considered in depth in the SEI report: "*Relating Business Goals to Architecturally Significant Requirements for Software Systems*" linked to from the SAPM wiki.

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 with.
  - You can use architecture as a way of describing your past experience.
  - You can specialise in particulr classes of architecture e.g. financial architecture.

Summary

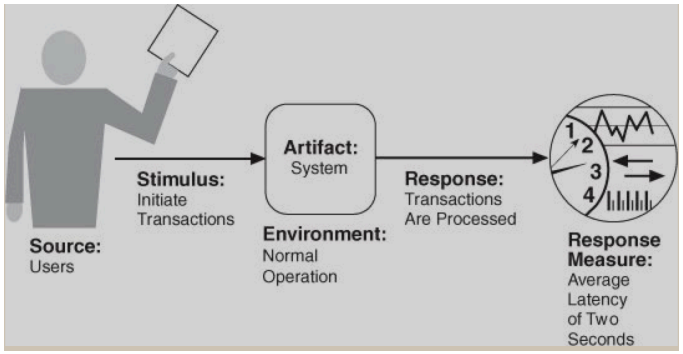
- We identified the four main contexts for architecture: technical, lifecycle, business, and professional.
- Architecture plays a different role in each of these contexts.
- We will expand on this in later lectures.

SAPM Performance

Performance General Scenario

Portion of Scenario	Possible Values
Source	Internal or external to the system
Stimulus	Arrival of a periodic, sporadic, or stochastic event
Artifact	System or one or more components in the system
Environment	Operational mode: normal, emergency, peak load, overload
Response	Process events, change level of service
Response Measure	Latency, deadline, throughput, jitter, miss rate

Specific Scenario



Making the Scenario Specific

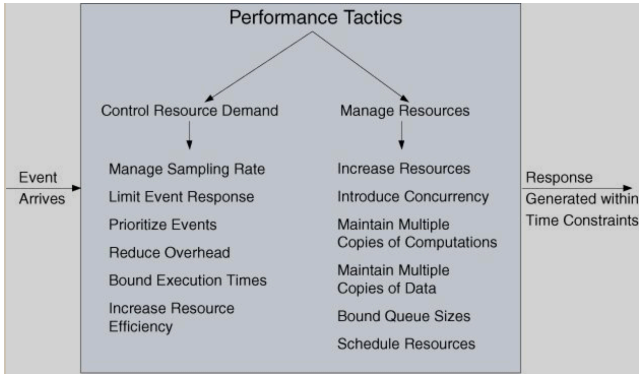
- We need to say something about the distribution of the arrival of the stimuli
  - E.g. The inter-arrival time is always greater than 1.0 secs
  - How is this different from the arrival rate is less than 1 per second?
- Any stimulus needs to be processed within 2 seconds of arriving.
- The responses should appear in the same order as the stimuli

A possible architecture

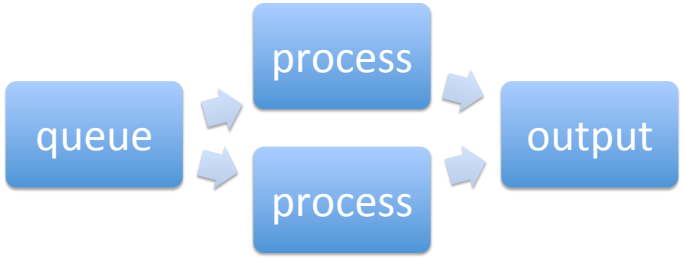
Be specific about the architecture

- 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 (why?)

Performance Tactics



A possible architecture



New Architecture

- This passes the scenario – why?
- Suppose the processing time for the stimulus was much more variable (e.g. 0.2 secs to 1.5 secs) – does the architecture still satisfy the scenario?

Control Resource Demand Tactics

- **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

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)

Checklist: 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:
  - Responsibilites arising from threads crossing boundaries of responsibility
  - Responsibilities for thread management
  - Responsibilities for scheduling shared resources

Checklist: 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?

Checklist: Data Model

- Determine what parts of the data model will be heavily loaded or have tight time constraints.
- Then:
  - 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?

Checklist: Mapping Among Architecture Elements

- Does colocation 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)

Checklist: Resource Management

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

Checklist: Binding time

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

Checklist: 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.

Summary

- For performance you need to ensure resource is effectively monitored and managed.
- Architecture gives you a good level to do this.
- Next we consider Security.

SAPM Meeting Security

What QAs are Important

- **Confidentiality:** Only those who should have access are given access.
- **Integrity:** Data or services are not subject to unauthorised manipulation
- **Availability:** the system is available for legitimate use.
- **Mechanisms that support CIA:** Authentication, nonrepudiation, authorisation.

General Security 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

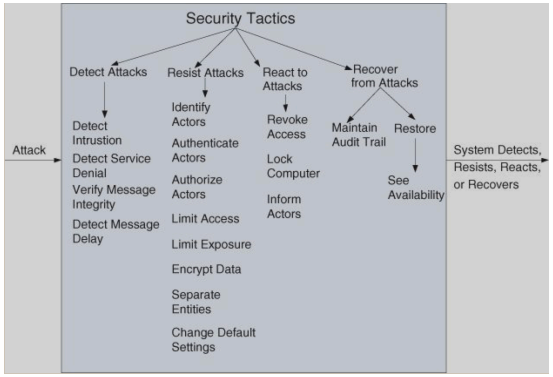
Generic Security Scenario

- **Response:** Transactions are carried out so that:
  - Data or services are protected from unauthorized access
  - Data or services are not manipulated without authorization
  - Parties in a transaction are identified with assurance
  - Parties cannot repudiate their participation
  - Data, resources etc are available for legitimate use
  - System records: access, modification of data, resources or services.
  - 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

Concrete Security Scenario: Denial of Service

- **Source:** A wide range of systems with different IP addresses
- **Stimulus:** Accesses to the service provided
- **Artifact:** The service we are concerned with.
- **Environment:** Normal operation
- **Response:** Detect abnormal load
- **Response Measure:** Mode of operation is changed to ensure normal service to trusted IP addresses.

Tactics



Concrete Scenario

- Consider a service with no protection.
- This would fail our scenario.
- If we require the system to maintain service to key users. We require:
  - Some means of detecting anomalous service demand.
  - Service now operates in two modes: normal mode and priority users mode
  - We add a layer to the architecture that filters out non-priority user service demands in priority users mode.

Allocation of Responsibilities

- For all security-sensitive system responsibilities, do the following:
  - Ensure all actors have identities
  - Authenticate identities
  - Check authorizations
  - Ensure authorization is required for all such actors
  - Log attempts, successes and failures on all sensitive operations
  - Ensure data is encrypted
  - Ensure responsibilities are allocated to appropriate actors.

Coordination Model

- Ensure coordination mechanisms use authentication and authorisation.
- Ensure coordination mechanisms are not vulnerable to impersonation, tampering, interception, …
- Ensure data involved in coordination is protected using encryption.
- Monitor level of demand for communication to identify excessive demands

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 and at rest using appropriate encryption.
- Ensure appropriate backup/recovery mechanisms are in place.

Mapping among Architectural Elements

- Explore how different mappings change the way users can access resources.
- Ensure for all of these mappings the models of access and authorisation are preserved.
  - Actors should be identified and authenticated
  - Use appropriate authorisation mechanisms
  - Ensure logging is enabled
  - Ensure data is protected by encryption
  - Recognise impact of attack on resources
- Ensure recovery from attack is possible

Resource Management

- Explore the overheads resulting from monitoring, detecting, preventing and recovering from attacks.
- Analyse how a user can access and make demands on critical resources.
- Manage resource access to ensure malicious use of resource is detected and managed.
- Identify the potential for corruption/contamination and how to manage this.
- 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.

Binding Time

- Explore the consequences of varying binding times on the ability to trust an actor or component.
- Put in place appropriate mechanisms to ensure trust given binding time.
- Explore potential impact on resource use, capacity/throughput, response time,
- Ensure appropriate encryption of all data around binding.
- Explore the potential of variation in binding time as a covert channel.

Choice 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.

Summary

- We need to consider attacks on Confidentiality, Integrity and Availability.
- Attacks need to be monitored and detected, resisted where possible, otherwise we may need some other form of reaction, eventually we should be able to fully recover.
- Tactics are a high-level way of categorizing possible protection against attack.
- In this area there is need for considerable domain expertise, see for example the NIST Framework for infrastructure cybersecurity:
  - http://www.nist.gov/cyberframework/upload/cybersecurity-framework-021214.pdf

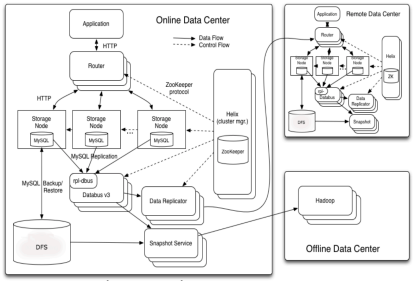
Software Connectors 1

Based on Nenad Medvidovic's slides

Software Connectors

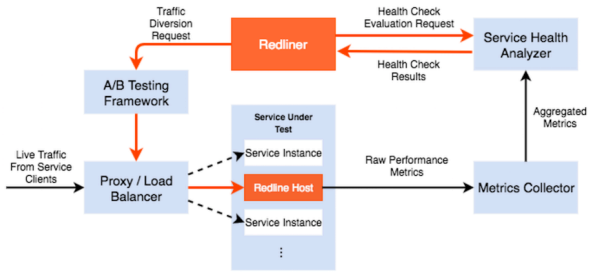
- Key part of Architectures (static, dynamic, deployment views)
- Connect components and define the rules of interaction between components
  - Simple: shared variable access; method calls; …
  - Complex: database access; client-server; scheduler; load balancer
- Connectors provide: Interaction ducts;

LinkedIn Espresso Doc Store



https://engineering.linkedin.com/espresso/introducing-espresso-linkedins-hot-new-distributed-document-store

LinkedIn Redliner



https://engineering.linkedin.com/blog/2017/02/redliner--how-linkedin-determines-the-capacity-limits-of-its-ser

Connectors in Software Arch

- In coding often connectors are implicit
- In software architecture:
  - They are identified and have an identity
  - Capture system interaction (at the level of components)
  - They have a specification that can be complex

What is Different about Connectors?

- Components have application-specific functionality.
- Connectors provide interaction mechanisms that are generic across different applications.
- Interaction may involve multiple components
- Interaction may have a protocol associated to it.

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

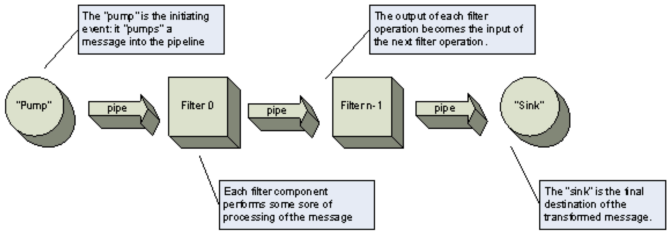
Roles played by Software Connectors

- The specification of the connector protocols determine: the types of interface that it works with; properties of interaction; rules about ordering of interaction; measurable features of interaction.
- The main roles are: Communication; Coordination; Conversion; Facilitation
- Connectors often have multiple roles

Communication

- Information is transmitted between components (e.g. message passing; method call; remote procedure call,…).
- Connectors constrain things:
  - Direction of flow (e.g. pipes)
  - Capacity, rates of flow, etc.
- May have other effects e.g. coordination (e.g. blocking I/O)
- Influences measurable Quality Attributes of the system
- Separates communication from functional aspects (components do the functional part).

Pipes



Coordination

- Controls the timing relationship of functional aspects of the system
- E.g. coordinating the arrival of data at a collection of components
- Different from communication, conversion or facilitation.
- Some connectors in other classes also include coordination in their definition.

Conversion

- How to get components to interact that don't 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)

Facilitation

- Enable interaction among a group of components that are intended to interact.
- Help manage the interaction
- Examples: load balancer; replication management; redundancy management; scheduler
- Can also relate to coordination e.g. synchronization: critical sections; monitors;

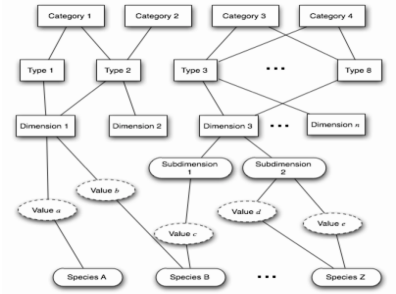
Taylor, Medvidovic and Dashofy

- Have a complex categorization of conectors
- This has some utility since it helps us understand potential different types and uses of connectors.
- We will not go into great depth but it is worth considering briefly

Types of Connector

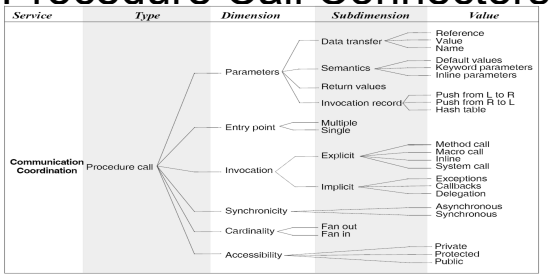
- Method/Procedure call
- Data access
- Events
- Stream
- Distributor
- Arbitrator
- Adaptor

A Framework for Classifying



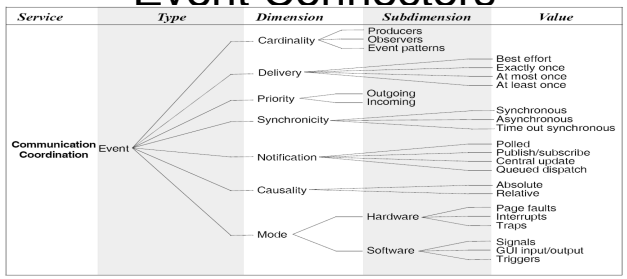
Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

Procedure Call Connectors



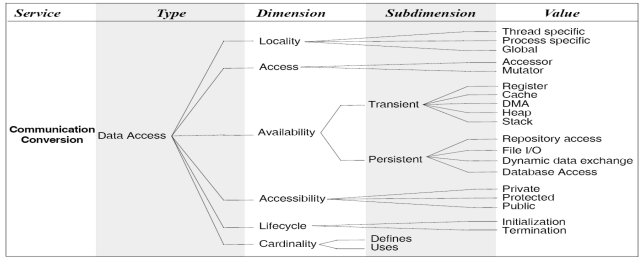
Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

Event Connectors



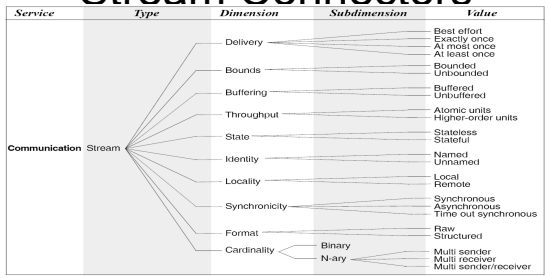
Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

Data Access Connectors



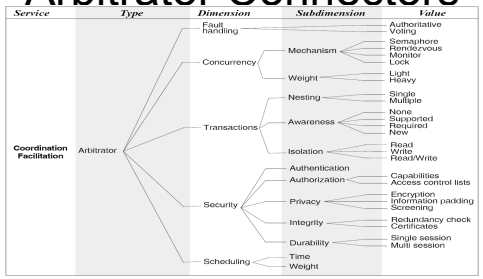
Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

Stream Connectors



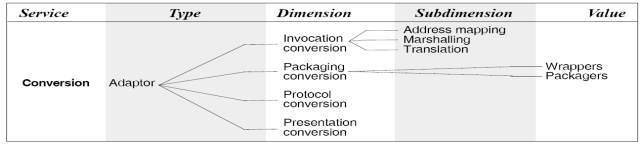
Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

Arbitrator Connectors



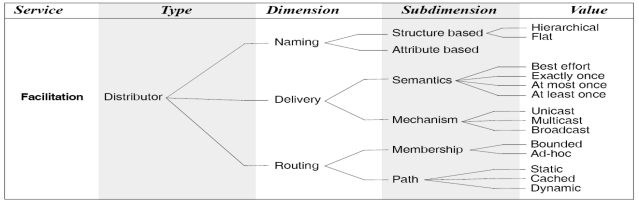
Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

Adaptor Connectors



Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

Distributor Connectors



Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

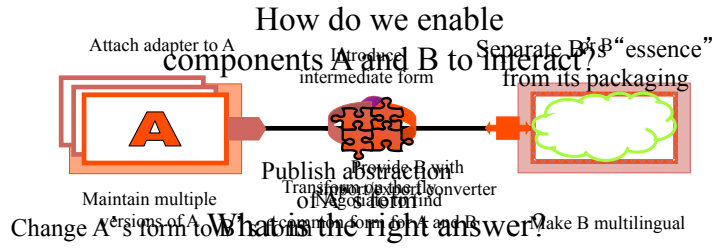
Discussion

- Identifying connectors makes the interactions in the system explicit – allows reasoning, modelling.
- Defines the context a component operates in:
  - Places requirements on the component.
  - Flexible connectors that can be adapted easily give us freedom to substitute components with somewhat different interaction capabilities.
- Having an explicit structure allows us to consider replace some or all of the connectors:
  - Aids system evolution – replacing deprecated connectors with connectors that are fully supported.
  - May not require changes in component functionality and leave system functionality unchanged

Discussion

- Libraries of OTS connector implementations allow developers to focus on application-specific issues
- Difficulties
  - Rigid connectors
  - Connector "dispersion" in implementations
- Key issue
  - Performance vs. flexibility

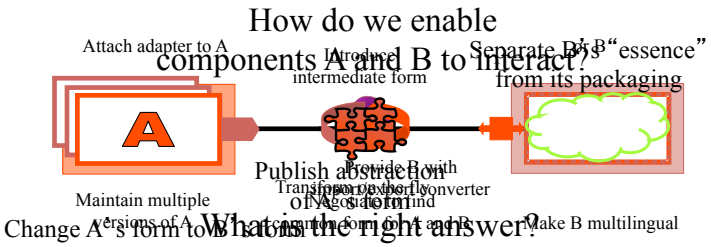
Next time: Choosing Software Connectors



Borrowed from Nenad Medvidovic's slides

Software Connectors 2

Role and Challenge of Software Connectors



How to Select Connectors

- Determine a system's interconnection and interaction needs
- Determine roles to be fulfilled by the system's connectors
  - Communication, coordination, conversion, facilitation
- For each connector
  - Determine its appropriate type(s)
  - Determine its dimensions of interest
  - Select appropriate values for each dimension
- For multi-type, i.e., composite connectors
  - Determine the atomic connector compatibilities

Example

- A system to integrate consumer generated wellness data from multiple sources:
  - Sources of consumer data – with different access protocols and data format
  - Need to integrate
  - Access should be secure

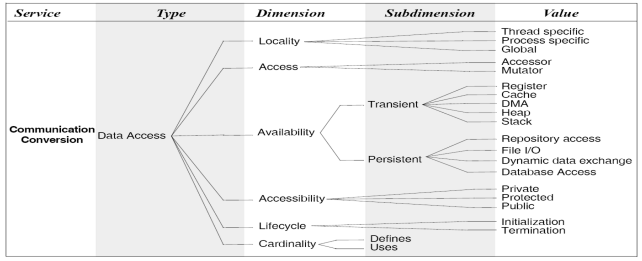
Example

- We need to convert the diverse data formats.
- We need to have a uniform access protocol.
- We need to communicate them to the integration component.
- We need to ensure privacy of the individual

Example

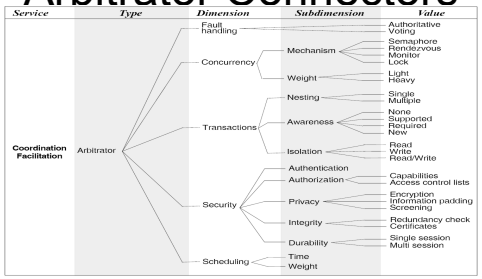
- We need a different types of conversion adaptors
  - One to convert to uniform protocols
  - One to convert data formats
- We need communication connectors to connect the sources to an integrator component and to connect from the integrator to users.
- We need an arbitrator component to ensure users are authenticated and authorised.
- We need to think about what values to choose for our connectors
- We need to think about what constraints the environment might place on the choice of connectors.

Data Access Connectors



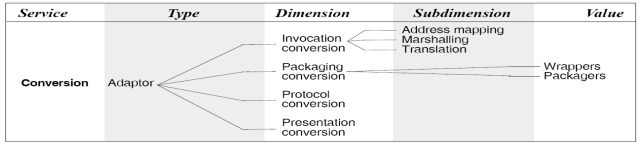
Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

Arbitrator Connectors



Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

Adaptor Connectors



Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

Summary

- Connectors are a key element in defining a software architecture.
- There is a wide variety of connectors and there is considerable diversity in their definition.
- Often connectors are combinations of various types.
- Analysing the requirements on an interconnection helps us to decide on the characteristics of a connector.

Architectural Patterns and Tactics

Overview

- Architectural patterns:
  - Package common combinations of design decisions
  - Has known ways of reusing
  - Describes a class of architectures
- Tactics are simpler and more atomic – they capture a step to take for a particular Quality Attribute to change behaviour with respect to that QA.
- Tactics can be seen as the building blocks of patterns

Architectural Patterns

- An architectural pattern comprises:
  - A **context** that provides the frame for a *problem.*
  - A **problem** that is a generalised description of a class of problems often with QA requirements that should be met.
  - A **solution** that is suitably generalised in the same way as the problem. A solution:
    - Describes the architectural structures that solve the problem that balances the forces at work in the solution.
    - The solution might be static, runtime or deployment oriented.

Architectural Solution

- An architectural solution comprises:
  - A set of element types (e.g. data repositories, processes, objects)
  - A set of interaction mechanisms (e.g. static relationships, events and handlers, …)
  - The relationship between the elements.
  - Constraints on the relationship between elements, element behaviour and communication mechanisms
- This wikipedia page covers the basic architectural patterns: https://en.wikipedia.org/wiki/List_of_software_architecture_styles_and_patterns

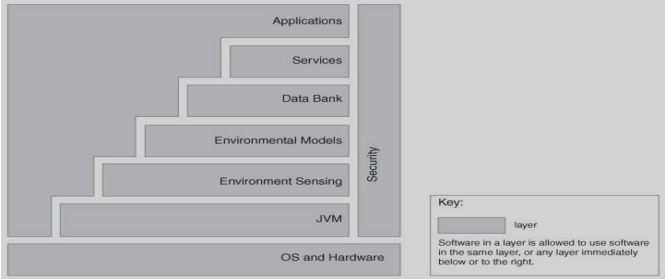
Architectural Patterns

- 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

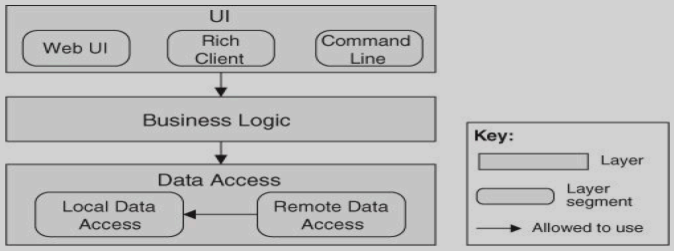
Static Pattern: Layered Architecture

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.

Layers: Clear Access Rules



Clear Access Rules



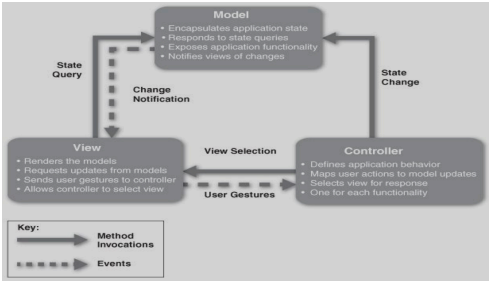
Component Connector: Model-View-Controller

- Context:
  - Here we have a situation where the user interface is subject to continuing change either to 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



Other Component-Connector Patterns

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

Deployment/Allocation Patterns

- Here we are looking at contexts:
  - Where we are concerned with resource use
  - We might consider flexible deployment of resource
  - The QAs we care about are sensitive to the pattern of deployment and the use of resources.

Allocation: Map-Reduce Pattern

- Context:
  - We have large quantities of data we wish to treat as "population" data.
  - This encourages an approach that involves significant amounts of independent processing.
- Problem:
  - Where for ultra-large data sets doing some individual processing to a portion of the data set and then sorting and analyzing grouped data, map reduce provides a simple way of doing this processing.

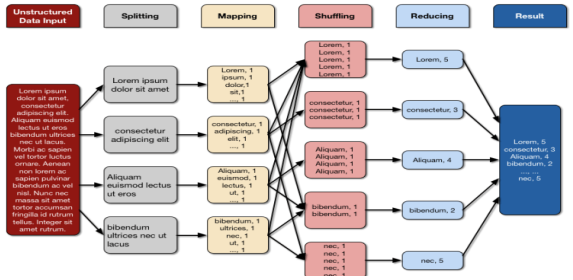
Map Reduce - Solution

Overview	The map-reduce pattern provides a framework for analyzing a large distributed set of data that will execute in parallel, on a set of processors. This parallelization allows for low latency and high availability. The map performs the *extract* and *transform* portions of the analysis and the reduce performs the *loading* of the results. (*Extract-transform-load* is sometimes used to describe the functions of the map and reduce.)
Elements	*Map* is a function with multiple instances deployed across multiple processors that performs the extract and transformation portions of the analysis. *Reduce* is a function that may be deployed as a single instance or as multiple instances across processors to perform the load portion of extract-transform-load. The *infrastructure* is the framework responsible for deploying map and reduce instances, shepherding the data between them, and detecting and recovering from failure.

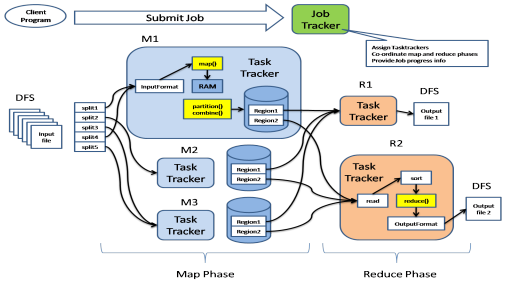
Map-Reduce: Solution

Relations	*Deploy on* is the relation between an instance of a map or reduce function and the processor onto which it is installed. *Instantiate, monitor, and control* is the relation between the infrastructure and the instances of map and reduce.
Constraints	The data to be analyzed must exist as a set of files. The map functions are stateless and do not communicate with each other. The only communication between the map instances and the reduce instances is the data emitted from the map instances as <key, value> pairs.
Weaknesses	If you do not have large data sets, the overhead of map-reduce is not justified. If you cannot divide your data set into similar sized subsets, the advantages of parallelism are lost. Operations that require multiple reduces are complex to orchestrate.

Map Reduce – Specific Example



Map Reduce Pattern



Other Allocation Patterns

- Multi-tier architecture pattern
- Cloud architectures

Relationships Between Tactics and Patterns: Patterns package 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	×	×	×		×	×	×			
Pipes and Filters	×		×		×	×			×	
Blackboard	×	×			×	×	×	×		×
Broker	×	×	×		×	×	×	×		
Model View Controller	×		×			×				×
Presentation Abstraction Control	×		×			×	×			
Microkernel	×	×	×		×	×				
Reflection	×		×							

Summary

- We have seen the use of architectural patterns.
- Patterns are more problem focussed than Tactics
- Pattersn can help in static, dynamic and deployment contexts.
- Patterns capture experience – they are discovered NOT invented…
- Next time we will consider some other patterns and their relationship to tactics and analysis.

SAPM
Testability

Testability

- Testability is **not** testing.
- A system or element of a system is testable if it is possible to test it in the way required by a particular development or maintenance process.
- The concern of the software architect is that the architecture is structured in order to ensure desired testing can take place.

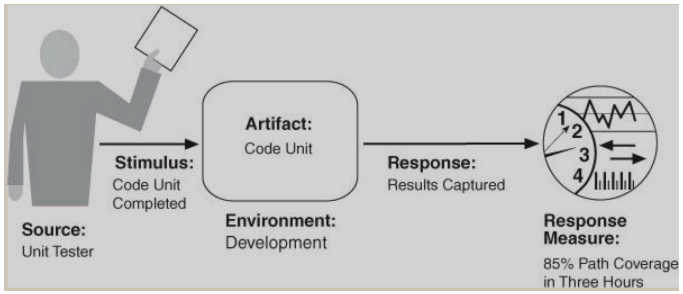
Testability Concerns

- Unlike the other examples we have considered (availability, performance and security) testability is concerned with the code structure rather than the connector/ component view or deployment view.
- The system elements we consider are code modules and the relationships are dependencies involved in building the code for components.

Testability: General Scenario

Portion of Scenario	Possible Values
Source	Unit testers, integration testers, system testers, acceptance testers, end users, either running tests manually or using automated testing tools
Stimulus	A set of tests is executed due to the completion of a coding increment such as a class layer or service, the completed integration of a subsystem, the complete implementation of the whole system, or the delivery of the system to the customer.
Environment	Design time, development time, compile time, integration time, deployment time, run time
Artifacts	The portion of the system being tested
Response	One or more of the following: execute test suite and capture results, capture activity that resulted in the fault, control and monitor the state of the system
Response Measure	One or more of the following: effort to find a fault or class of faults, effort to achieve a given percentage of state space coverage, probability of fault being revealed by the next test, time to perform tests, effort to detect faults, length of longest dependency chain in test, length of time to prepare test environment, reduction in risk exposure (size(loss) × prob(loss))

Concrete Scenario



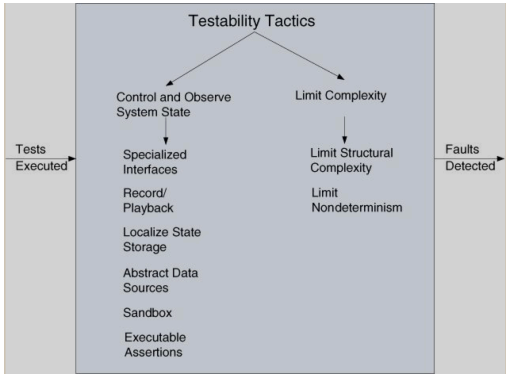
Coverage Concrete Scenario

- **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

Development Concrete Scenario

- **Source:** Developer/Tester
- **Stimulus:** Completion of new functionality to add transaction logging to a system.
- **Artifact:** Logging Modules and log database
- **Environment:** Development
- **Response:** Results from tests of simulated operation of the system.
- **Response Measure:** Consistency between log and the simulated data.

Testability Tactics



Coverage Scenario

- We might identify that there is one particularly large module with a very large number of loop-free paths that confounds the regression test.
- If it could be broken into smaller modules with lower complexity that could allow the regression test to achieve higher path coverage.
- This is an example of a tactic based on limiting complexity (or at least attempting to control it)

Development

- It may be that we cannot pass the scenario because we do not have adequate access to the log database from the tests.
- If we restructure the software architecture to introduce specific interfaces that provide better access to the database this may resolve the issue.
- This is an example where we are attempting to change what can be observed to it is possible to check if the system passes the test.

Checklist: Allocation of responsibilities

Allocation of Responsibilities	Determine which system responsibilities are most critical and hence need to be most thoroughly tested. Ensure that additional system responsibilities have been allocated to do the following: ▪ Execute test suite and capture results (external test or self-test) ▪ Capture (log) the activity that resulted in a fault *or* that resulted in unexpected (perhaps emergent) behavior that was not necessarily a fault ▪ Control and observe relevant system state for testing Make sure the allocation of functionality provides high cohesion, low coupling, strong separation of concerns, and low structural complexity.

Checklist: Coordination Model

Coordination Model	Ensure the system's coordination and communication mechanisms: ▪ Support the execution of a test suite and capture the results within a system or between systems ▪ Support capturing activity that resulted in a fault within a system or between systems ▪ Support injection and monitoring of state into the communication channels for use in testing, within a system or between systems ▪ Do not introduce needless nondeterminism

Checklist: Data Model

Data Model	Determine the major data abstractions that must be tested to ensure the correct operation of the system. ▪ Ensure that it is possible to capture the values of instances of these data abstractions ▪ Ensure that the values of instances of these data abstractions can be set when state is injected into the system, so that system state leading to a fault may be re-created ▪ Ensure that the creation, initialization, persistence, manipulation, translation, and destruction of instances of these data abstractions can be exercised and captured

Checklist: Mapping among elements

Mapping among Architectural Elements	Determine how to test the possible mappings of architectural elements (especially mappings of processes to processors, threads to processes, and modules to components) so that the desired test response is achieved and potential race conditions identified. In addition, determine whether it is possible to test for illegal mappings of architectural elements.

Checklist: Resource Management

Resource Management	Ensure there are sufficient resources available to execute a test suite and capture the results. Ensure that your test environment is representative of (or better yet, identical to) the environment in which the system will run. Ensure that the system provides the means to do the following: ▪ Test resource limits ▪ Capture detailed resource usage for analysis in the event of a failure ▪ Inject new resource limits into the system for the purposes of testing ▪ Provide virtualized resources for testing

Checklist:Binding Time

Binding Time	Ensure that components that are bound later than compile time can be tested in the late-bound context. Ensure that late bindings can be captured in the event of a failure, so that you can re-create the system's state leading to the failure. Ensure that the full range of binding possibilities can be tested.

Checklist: Choice of Technology

Choice of Technology	Determine what technologies are available to help achieve the testability scenarios that apply to your architecture. Are technologies available to help with regression testing, fault injection, recording and playback, and so on? Determine how testable the technologies are that you have chosen (or are considering choosing in the future) and ensure that your chosen technologies support the level of testing appropriate for your system. For example, if your chosen technologies do not make it possible to inject state, it may be difficult to re-create fault scenarios.

Summary

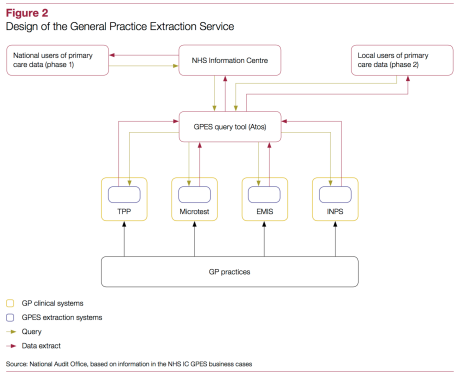
- Testability becomes increasingly important in an environment where development is test driven.
- Software Architecture has a role to play in helping with the choice of static structure for the code of a system.
- The main ways we can improve testability is via changing the observability of phenomena or by limiting complexity these both have an impact on our ability to test.

SAPM Modifiability

Modifiability

- Just as testability is not about testing, modifiability is not about modifying – it's about how easy it might be to modify.
- This is a key area because change incurs cost.
- Four key questions:
  - What can change?
  - How likely is something to change?
  - When, where, how and by whom will changes be made?
  - What is the cost of making the change?

Look at the GPES Example



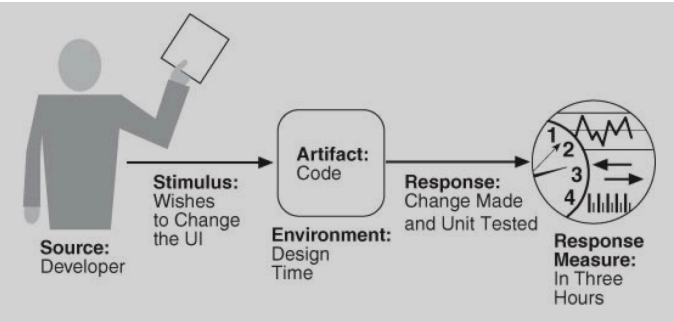
GPES: Two versions

- Version 1: General purpose query facility in each GP system.
- Version 2: Building a specific piece of business logic for each different query.
- Think about:
  - What changes can happen?
  - How likely is a change?
  - When, where, how and by whom?
  - How mush will it cost?

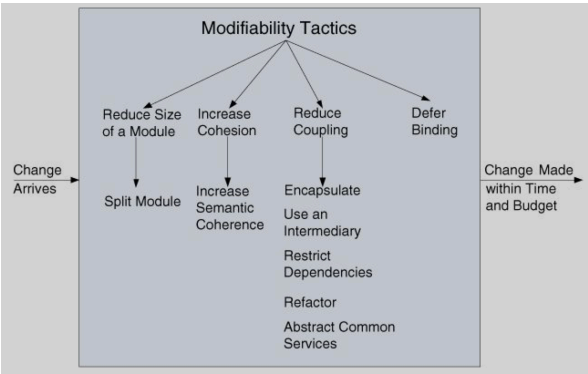
General Scenario

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

Specific Scenario



Modifiability Tactics



GPES-relevant Scenario

- **Source:** One of the stakeholders e.g. Medicines and Healthcare Products Regulatory Agency
- **Stimulus:** Wants prescribing data on NSAIDs
- **Artifacts:** Code (but depending on the architecture this could be configuration data)
- **Environment:** Operation
- **Response:** Develop the code
- **Response Measure:** Data available 5 weeks after request

GPES Version 1

- Design and validate the query with the Medicines agency.
- Code the query.
- Test on some systems to ensure it does not have bad effects.
- Rollout to all systems.
- Make the query available to Medicines agency.

GPES Version 2

- Design and validate the query with Medicines agency.
- Negotiate with the GP system providers on the design of the business logic (different in all systems?)
- Are the providers the only vendor of such services? Should it go to a procurement?
- Validate the queries on each system
- Integrate the results
- Roll out to all systems
- Make the query available to the Medicines Agency.

Modifiability Tactics

- It seems likely that the GPES V2 architecture will not pass the modifiability scenario we describe.
- Are any of the modifiability tactics appropriate to change the architecture to enable it to pass the scenario?

Modifiability Tactics

- Reduce Coupling is the category of tactics we need to consider.
- Each of the following offer potential routes with slightly different emphases:
  - Use an intermediary
  - Restrict dependencies
  - Refactor
  - Abstract common services
- "Defer Binding" means can we do this later in the process so it is more likely to be done by a computer than a human? Here this is unlikely.

More on Binding Time

- Compile time/Build Time: component replacement, compile time parameters,…
- Deployment time: configuration scripts that bind at deployment, …
- Initialization time: resource files
- Runtime: dynamic lookup, service lookup, name servers, plugins, publish-subscribe, shared repositories, (Maybe just in time compilation fits here too)

Modifiability Design Checklist

- **Allocation of responsibilities**: work out how things are likely to change e.g. technical, legal, organisational, social, markets, customers..:
  - Work out what responsibilities change.
  - Try to modularise so a change does not affect responsibilities that span many modules.
- **Coordination model**: look at how changes are likely to affect coordination and try to ensure that the most likely changes impact coordination across a small number of modules (look at GPES example)
- **Data model**: Similar to coordination model – see how a change impacts on data models and try to esnure data model changes span as few modules as possible.

Modifiability Design Checklist

- **Mapping among architectural elements**: looking at potential changes to the system, assess whether some may best be responded to by changing the mapping to elements. Explore issues such as dependencies between elements, data holdings in elements, assignment of elements to processes, threads or processors.
- **Resource Management**: Determine how a change in responsibility or quality attribute will change resource. Attempt to localise resourcing change resulting from a likely change to a small number of modules. Look at ways of using policies or configuration to manage resource change more effectively

Modifiability Design Checklist

- **Binding Time:** Control choice of binding times so there are not too many combinations to consider. Consider attempting to defer binding to later, balance this against the cost of providing a later binding mechanism. (e.g. GPES example again).
- **Choice of Technology:** Choose technologies that make the most likely changes easier (e.g. choose a technology that allows runtime alteration of critical parameters rather than one where parameters are chosen at compile time) but balance this agains the cost of the different technologies.

Summary

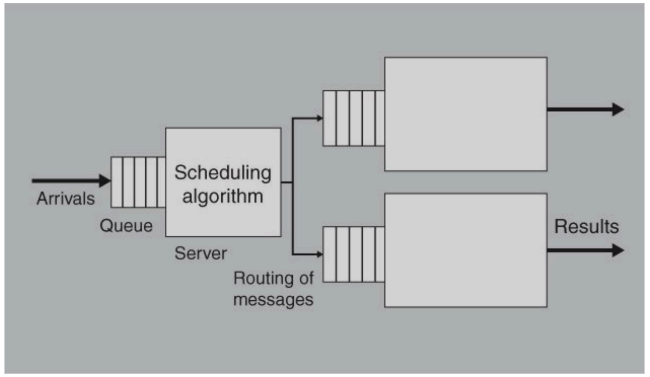
- The standard approaches look to improve cohesion and reduce coupling in the structure in all of the views (static, dynamic, deployment).
- Costs need to be taken into account, making something easier to change later in the process may be useful but excessively expensive.
- Often introducing mechanisms that allow us to make changes late in the development cycle helps ease modifiability costs.

Architectural Modelling

Overview

- Software Architecture is intended to give us control over Quality Attributes.
- Ideally we'd like to be able to use Software Architecture to predict Quality Attributes.
- We should be able to build a predictive model of the Software Architecture and use the model to predict Qas.
- The current situation is patchy…

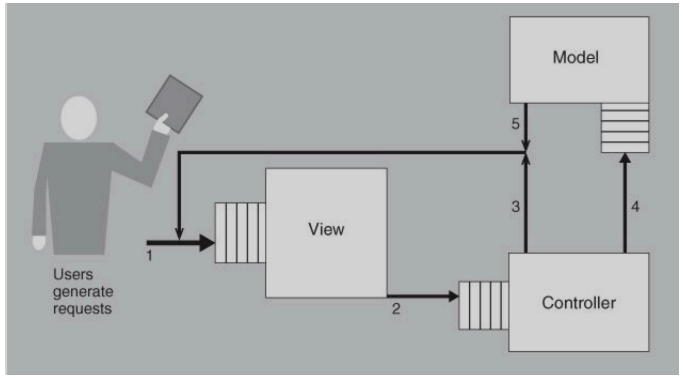
Performance: Queueing Models



Data Needed for the Model

- We need the following information in order to model effectively:
  - The distribution for the arrival of service requests
  - The queuing discipline
  - The scheduling algorithm
  - The distribution of service times for service requests
  - Network characteristics
- The theory places restrictions on the distributions
  - Arrivals are usually expected to be Poisson Distributions specified by arrival rate
  - Service times are usually exponentially distributed on the service rate.
  - Some queuing behaviors are excluded such as reneging or jockying

Queueing Model of MVC



Data for MVC

1. Rate of service requests: the View component will service them at some rate.
2. Service requests to the Controller are generated by the View component.
3. Service requests from the Controller to the View component
4. Service requests from the Controller to the model
5. Service requests from the Model to the View Component

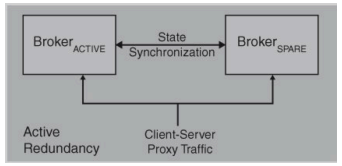
Modelling MVC

- We need estimates of:
  - Distribution of external service demands
  - Queuing Disciplines within the queues in front of each component.
  - Network latencies
  - Transfer characteristics:
    - View – Controller
    - Controller – View
    - Controller – Model
    - Model – View
- This is a well-studied area and can produce good predictions if the estimates are good.
- More sophisticated techniques are available.
- Scaling to large numbers of components is an issue

Availability

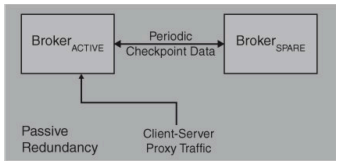
- Recall availability is defined as:
  - *mtbf/(mtbf+mttr)*
- Not as well developed as Performance
- Range of different architectural solutions result in different availabilities.
- One key issue is how long it takes to detect that a failure has taken place…
- Example is a Broker system.

Hot Spare



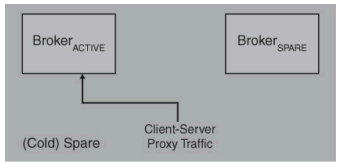
- Active and redundant both receive identical request stream.
- Synchronous maintenance of broker state.
- Fast failover in the event of failure of the active system.

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)

Analytical Model Landscape

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	

Analysis in the Lifecycle

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

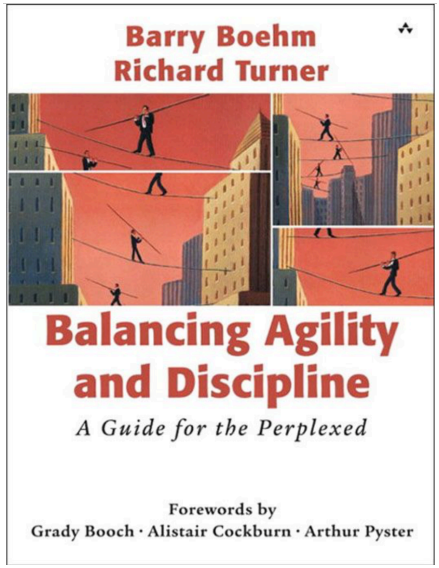
Types of Analysis

- Thought experiment: just a sort of discussion using informed people.
- Back of the envelope: using very approximate techniques with unreliable assumptions.
- Checklist: 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.

Summary

- Architecture is the correct level to deal with Quality Attributes.
- Some Qas have good, well established, analysis techniques, others don't.
- Analysis can be costly depending on how accurate you want it to be.
- Analysis through the lifecycle helps with decision taking.

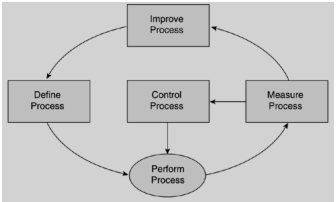
Architecture and Lifecycles



Balancing Agility and Discipline

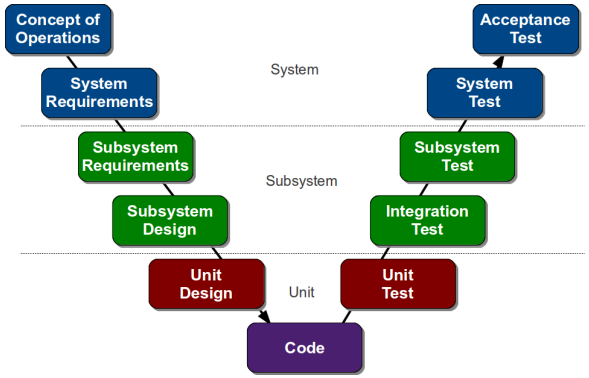
- Lifecycles generally impose some discipline on the development process.
- Software Architectures often feature in Lifecycles as a stage or support for analysis or design
- Lifecycles exist because they codify useful patterns of activity and save us time and effort
- Agility focusses on getting adequate solutions to stakeholders with less time and effort
- We need to balance the discipline of lifecycles against the delivery focus of agility
- Boehm and Turner explore this balance in their book

Lifecycles



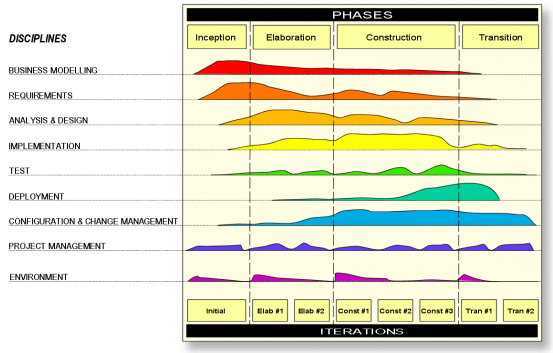
- Lifecycles underpin development processes by ordering stages and activities.
- Any good organisation is always looking to improve its processes so there is usually an ongoing process improvement cycle focussed on making the process better.

Classical: V-Model



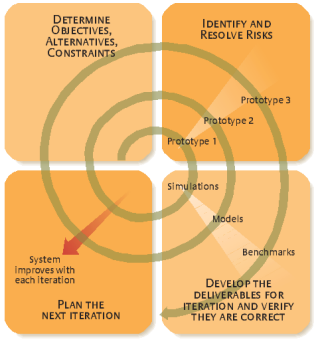
https://davenicolette.files.wordpress.com/2012/02

Classical: RUP



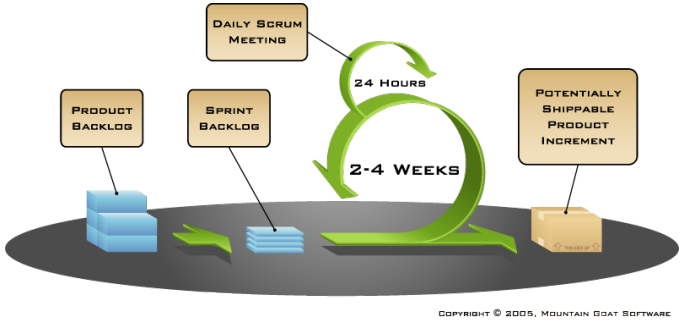
https://davenicolette.files.wordpress.com/2012/02/rup.png

Transitional: Spiral Model



Adapted from B. Boehm, "A Spiral Model of Software Development and Enhancement," ACM SIGSOFT Engineering Notes 11, no. 4 (1986): 25.

Agile: SCRUM



Agile Manifesto

Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

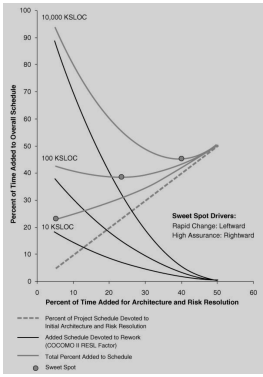
Individuals and interactions	over	processes and tools
Working software	over	comprehensive documentation
Customer collaboration	over	contract negotiation
Responding to change	over	following a plan

That is, while there is value in the items on the right, we value the items on the left more. [agilemanifesto.org]

Agile Practice

- Test-first programming
- Refactoring
- Continuous integration
- Simple Design
- Pair Programming
- Common Codebase
- Coding Standards
- Open Work Area

Bigger needs more structure



From Boehm and Turner

Architecture, Agility and Structure

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

Architecture, Agility and Structure

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

Architecture, Agility and Structure

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

Architecture, Agility and Structure

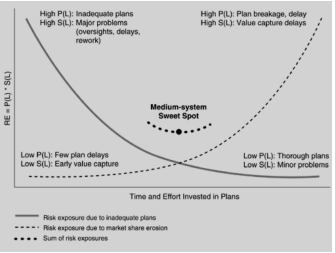
Characteristics	Agile	Plan Driven
Personnel		
Customers	Dedicated, collocated CRACK* performers	CRACK* performers – not always collocated
Developers	High level skills required throughout – less scope for lower performers	High skill people needed at critical points, more possibility to use lower skilled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

Architecture, Agility and Structure

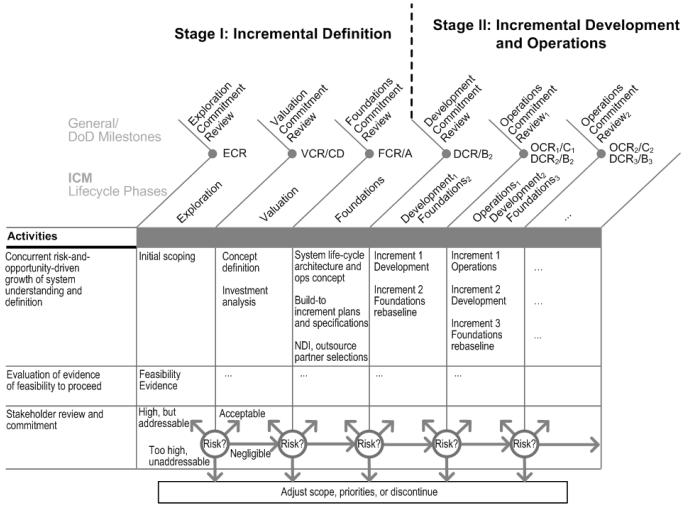
- Work top-down and bottom-up simultaneously – balance will depend on the size and complexity of the project.
- Top-down does architectural workbased on things like patterns.
- Bottom-up develops implementation and environment-specific constraints and solutions.
- Focus on QAs, scenarios, tactics and processes to reconcile competing aspects provides a bottom-up/top-down link
- Balancing commitment and flexibility

Factor	Agility Discriminators	Plan-Driven Discriminators
Size	Well-matched to small products and teams. Reliance on tacit knowledge limits scalability.	Methods evolved to handle large products and teams. Hard to tailor down to small projects.
Criticality	Untested on safety-critical products. Potential difficulties with simple design and lack of documentation.	Methods evolved to handle highly critical products. Hard to tailor down to low-criticality products.
Dynamism	Simple design and continuous refactoring are excellent for highly dynamic environments, but a source of potentially expensive rework for highly stable environments.	Detailed plans and Big Design Up Front excellent for highly stable environment, but a source of expensive rework for highly dynamic environments.
Personnel	Requires continuous presence of a critical mass of scarce Cockburn Level 2 or 3 experts. Risky to use non-agile Level 1B people.	Needs a critical mass of scarce Cockburn Level 2 and 3 experts during project definition, but can work with fewer later in the project —unless the environment is highly dynamic. Can usually accommodate some Level 1B people.
Culture	Thrives in a culture where people feel comfortable and empowered by having many degrees of freedom. (Thriving on chaos)	Thrives in a culture where people feel comfortable and empowered by having their roles defined by clear policies and procedures. (Thriving on order)

Risk links Plans and Agility



- P(L) – the probability of loss.
- S(L) – the size of loss
- These curves vary depending on the type of project.



Incremental Commitment Model: http://csse.usc.edu/csse/research/ICM/

Incremental Commitment Model

- Boehm, Barry, and Jo Ann Lane. "Using the incremental commitment model to integrate system acquisition, systems engineering, and software engineering." *CrossTalk* 19.10 (2007): 4-9.
- Koolmanojwong, Supannika, and Barry Boehm. "The incremental commitment model process patterns for rapid-fielding projects." *New Modeling Concepts for Today's Software Processes*. Springer Berlin Heidelberg, 2010. 150-162.
- Boehm, Barry. "Applying the Incremental Commitment Model to Brownfield Systems Development." *Proceedings, CSER* (2009).

Architecture, Analysis and Lifecycle

- Kazman, R., R. L. Nord, and M. Klein. "A Life-Cycle View of Architecture Analysis and Design Methods (CMU/SEI-2003-TN-026). Pittsburgh, PA: Software Engineering Institute." (2003).
- Kazman *et al* look at how analysis techniques fit into lifecycle stages.
- They refer to several analysis techniques and then outline how they contribute to lifecycle stages.

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

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	

Lifecycle Stages and Architecture Activity

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.

Summary

- There are many possible lifecycles and variants on these lifecycles.
- For any particular area of activity 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.

Product Line Architecture

Systems

- Systems often come in families: basic, regular, professional, enterprise,…
- Can we share components?
- Is architecture useful is encouraging an approach that uses components as building blocks for systems.

Software Product Line

- A collection of software-intensive systems sharing a common, managed, set of features that satisfy the specific needs of a market segment or mission that are developed from a set of core assets in a prescribed way.
  - Product lines are a feature of many physical products (e.g. cars)
  - Useful in the context of software engineering

Software Product Lines

- Are directed by business goals in a particular application domain.
- The products in a product line share a software product line architecture
- Products are structured by the product line architecture and are built form services and components.
- Architercture and components are the core assets used to satisfy the business goals.
- Product line leverage commonality and limit variability of the product.

Product lines spread costs over several products

- Requirements and requirements analysis
- Domain model
- Architecture and design
- Performance engineering
- Documentation
- Test cases, data, and plans
- Skills
- Processes, methods and tools
- Defect fixing
- Components and services

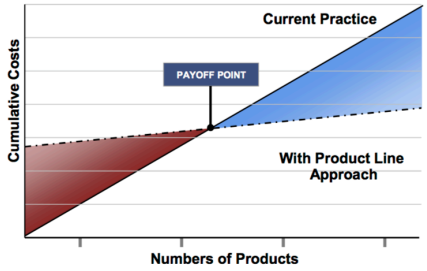
Benefits to the organisation

- Large-scale productivity gains
- Improve time to market
- Maintian market presence (rapidly evolving variants)
- Sustain growth
- Improved market agility
- Better use of skills
- Enable mass customisation
- Gain control of configuration
- Improve product quality
- Better predictability of cost, schedule and quality

Costs of a product line

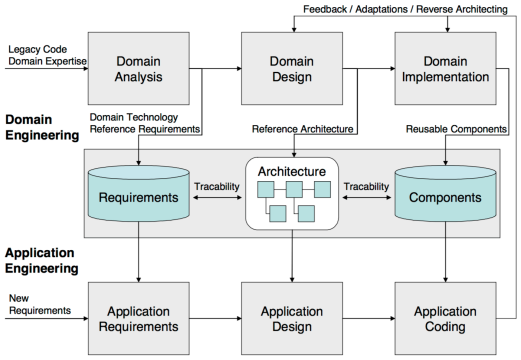
- Architecture: flexible enough to support variation in the products
- Software components: general enough to support variability
- Test plans, cases, data: take account of variation in components
- Business cases: must operate at the level of a product family
- Project plans: generic and extensible to deal with variation
- Tools and processes: must support architecture, variation, configuration, ..
- People, skills, training: need to be skilled in architecture and product lines.

Product Line Economics



Weiss, D. M. & and Lai, C. T. R.
Software Product-Line Engineering: A Family-Based Software Development Process.
Reading, MA: Addison-Wesley, 1999.

Product line summary



Van der Linden, F., (2002). Engineering Software Architectures, Processes and Platforms for System Families—ESAPS Overview, Proceedings of the Second International Conference on Software Product Lines (SPLC2), pp. 383-397, San Diego, CA, USA, August 2002.

Core Process Activities

- 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.

Introducing Product Lines

- **Proactive:** Up-front investment to develop the core assets – need to know the market well (maybe have an already established set of products)
- **Reactive:** Start with one or two products and use them to generate core assets.
- **Incremental:** Develop core assets as the business need evolves.

Example: Bosch Gasoline Systems

Based on: Software Product Lines in Action, Chapter 9, Frank van der Linden, Klaus Schmid, Eelco Rommes
Springer 2007, ISBN: 978-3-540-71436-1

Bosch Gasoline Systems

Company facts of Bosch Gasoline Systems

Organisational size: ~ 1,000 developers.
Starting Mode: Strategic focus, based on existing assets.

Experienced improvements:
- Reduction of calibration effort (–20%) and maintenance.
- Reduction of the resource consumption (20–30%).
- Product line definition reflecting market variance.

Business: Addressing new business challenges was a major driver. Three market segments were identified as a starting point.

Architecture: A new software architecture was developed, but assets were derived from the existing asset base.

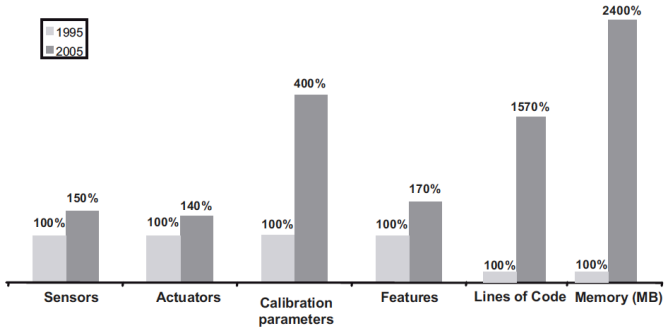
Process: Bosch works on CMMI level 3. Systematic process engineering provided an important basis for product line development.

Organisation: A restructuring of the organisation was necessary to reflect the different roles in a product line organisation.

Bosch

- Employed 250,000 people in 2006, now 375,000
- Revenue €70.6 bn in 2015
- Gasoline systems is one of the largest divisions in 2006, employing 1000 developers.
- Every new engine controller leads to a new variant
- Thousands of controllers are in use.

Rapid Change



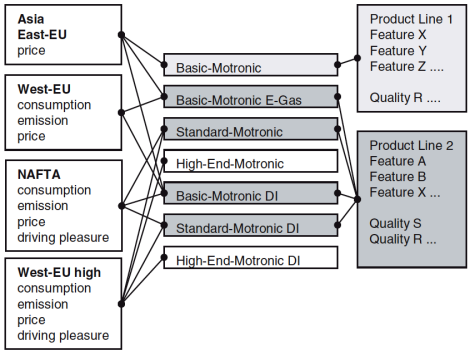
Goals

- Competitiveness:
  - Reduced hardware resource consumption
  - 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

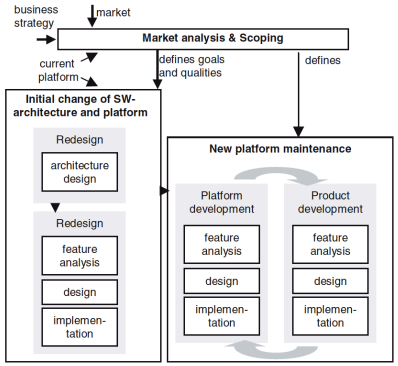
Approach

- Process:
  - Consider business strategy
  - Consequences for products
  - Consequences for processes and methods
  - Consequences for tools
  - Consequences for the organisation

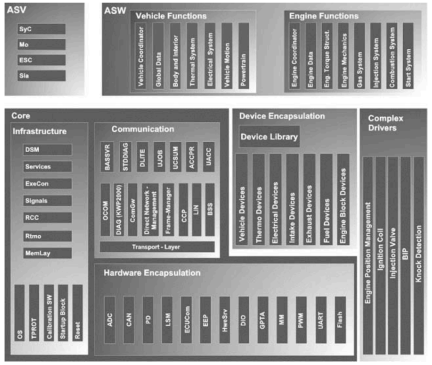
Business Strategy



Production



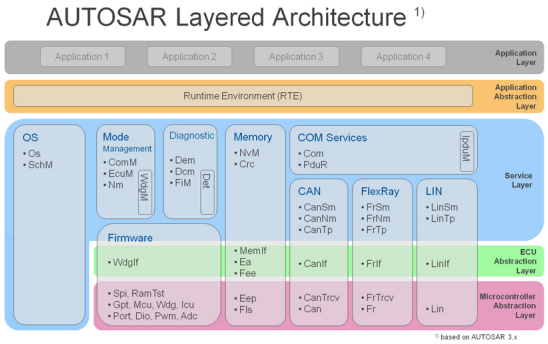
Architecture



Architectural Features

- Control of resource consumption e.g. memory (because of cost).
- Distributed development requires good interface management.
- Layers provide the possibility to share applications without knowing the details of particular sensors or actuators
- Reuse goals: Applications can be used across different generations of system; “core” software is highly configurable and is reused via reconfiguration; “Vehicle functions” can be used across gasoline and diesel engines
- Standardisation is important, Bosch play an important role in AUTOSAR: http://www.autosar.org

Architecture Standardisation



Component Redesign

- Focussed on: reuse; simplification of calibration; resource consumption; stabilisation of interfaces (within the architecture)
- Redesign progressed by:
  - Analysing existing software inventory: features, sources of variability; relation to product line; document interdependency.
  - Concept development and design of components: simplification; configurability; architecture driven structure; document relations between features and components;
  - Baselines for variants of software components: document baselines; implement; maintain up-to-date document and implementation.

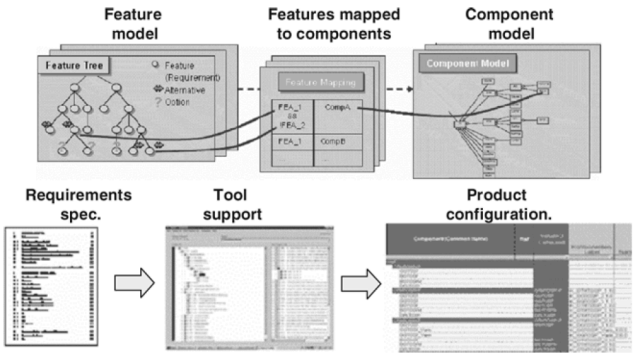
Phased Introduction

- Investigate and customise product line engineering.
- Design and pilot adequate processes and methods.
- Roll out and institutionalise in the standard development process.

Roll out

Action:	**Roll out by redesign of existing platform.**
Purpose:	Initial development of work products like interfaces, feature trees, overview functions, etc.
Helpful:	Product line engineering coaches.
Action:	**Series of product line process workshops with middle management.**
Purpose:	Understanding, acceptance and management support.
Helpful:	Management commitment.
Action:	**Embody new process steps in standard development process.**
Purpose:	Visibility of product line engineering integration in process infrastructure.
Helpful:	Management commitment, existing process infrastructure.
Action:	**Training program for product line engineering and architecture.**
Purpose:	Understand product line engineering and internalise new methods.
Helpful:	Use of domain-specific examples.

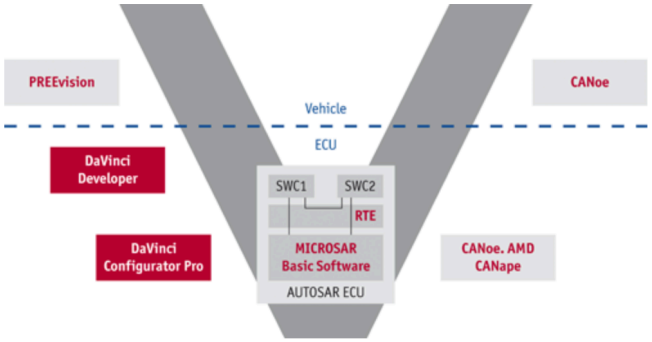
Toolchains



Toolchains

- Tools were primitive and not well adopted – strong requirements for: feature modelling; architecture documentation; interface documentation, checks and management; linkage between features and implementation; feature-based product derivation.
- Strong drive to standardise to develop tools and ease interchange between Bosch as a supplier and purchasers of Bosch products

Maturing Toolchains

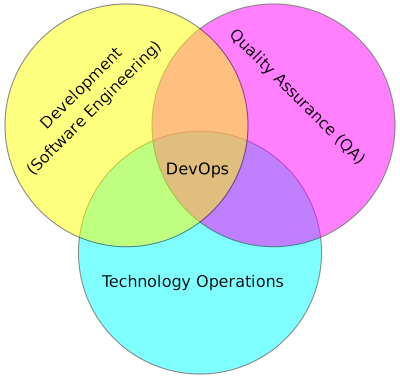


Key Elements

- Management commitment to: build up product line knowledge; establishing a product line business unit to drive product line engineering; commitment to rollout.
- Process excellence: in parallel to adoption of product line Bosch to achieve Capability Maturity Model Integration level 3 (i.e defined process characterised for the organisation and is proactive)

DevOps and Architecture

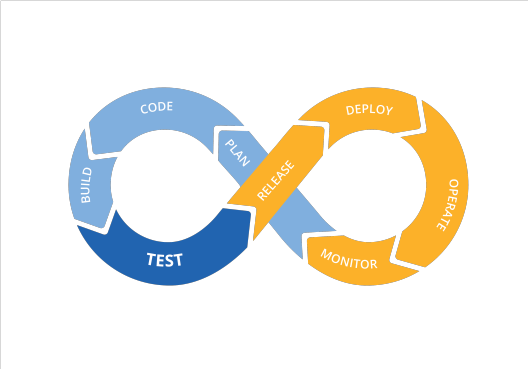
Where does DevOps Live?



Key Issues

- Quality is the key – normally operations staff guard that but development are responsible for designing in the Quality Attributes that ensure quality.
- Operations have the direct experience of use of the system – monitoring that use is a way of empirically verifying quality – operations have the data that is used to regulate operations and is essential information for development.
- Development is responsible for building in the right monitoring to ensure operations can operate effectively

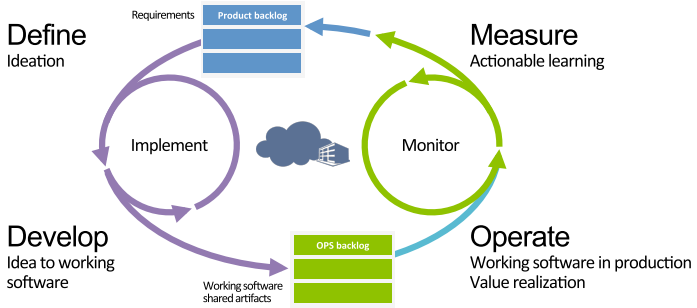
DevOps Sees Development and operations as a unified entity



The Essence of DevOps

- DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal operation while ensuring necessary quality.
- Movement Comes from Open Source (OSLC: http://open-services.net)
- When Development and Operations Synergize
- Covers the *entire* Application LifeCycle

The Whole Application LifeCycle



Why OSLC and Standardizing Matters

Software runs the world

But it is heterogeneous and disjoint

And it needs to be integrated

Traditional approaches to software integration require custom software

Custom software is expensive to maintain and a limit on future choice

Custom integrations drive software TCO higher and limit choice

Open standards are key enablers for broad and large-scale integration

OSLC standards simplify lifecycle integration leading to cost savings and increased flexibility

OSLC is helping the world run more efficiently

Standardizing OSLC will increase adoption and acceptance of OSLC

The world will benefit from standardized integration through OSLC

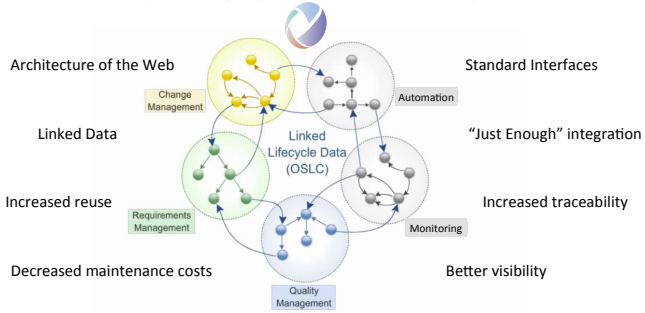
Aspirations for OSLC

- Foundational technology for all integration
- The household name for integrations
- Natural choice for standardizing loosely-coupled integrations in new domains

The Technical Vision for OSLC

Users can work seamlessly across their tools

(complex and fragile synchronization schemes not required)



OSLC is an open and scalable approach to lifecycle integration. It simplifies key integration scenarios across heterogeneous tools

Key issues for DevOps

- Traditionally we use test as the way of delivering quality change but we can "shepherd" committed change into use by controlling quantities of change, users experiencing change, results of monitoring than this may offer a better way.
- Delivery mechanism needs to be high quality: reliable, repeatable, available,

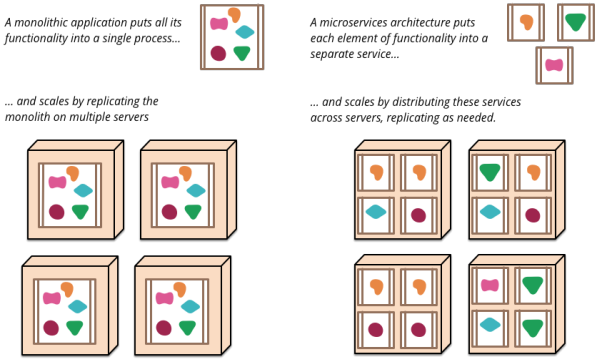
Critical points

- Making the decision to commit the code to be introduced into the system.
- Transitioning from being under consideration into part of the production deployment that will be used by all users.
- Issues is how to have enough confidence to make each of these transitions. Monitoring is critical.
- The question is how to ensure the transitions are as reliable as possible.

The extent of the lifecycle

- Involves all people involved in the delivery of the service/application
- Operations and development people are in continuous interaction.
- We need architecture to achieve this.
- Microservices architectural pattern is often used.

Microservices

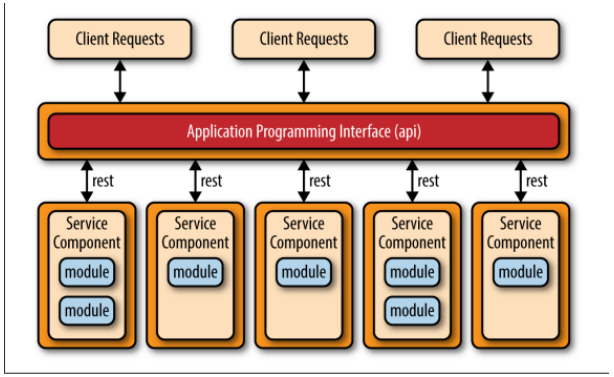


http://martinfowler.com/articles/microservices.html

Attributes of Microservice Architecture

- Separately deployed units
- Very small service components
- Single purpose function or an independent portion of functionality
- Distributed
- Loosely coupled
- Multiple versions are acceptable
- Asynchronous
- No Orchestration

Example: Providing and API



Thanks to Assaf Gannon

A Practical Example

"Do painful things more frequently, so you can make it less painful…"
https://www.eiseverywhere.com/file_uploads/c6b285b64a18cc2742c0fb20061d6530_OBC_BreakoutSession_AdrianCockcroft_1pm.pdf

Adrian Cockcroft, Architect, NetFlix

NETFLIX

The First Way – Systems Thinking

- Understand the entire flow of work
- Seek to increase the flow of work
- Stop problems early and often
  - Don't let them flow downstream
- Keep everyone thinking globally
- Deeply understand your systems

Defining Work and Make It Visible

- Business Projects
  - The new inventory system
- Internal IT Projects
  - Deployment Automation
- Changes
  - Database Performance Tuning
- Unplanned Work
  - Web Site Outage

Richa

One Step Environment Creation

- Need a common environment build process
  - For development, qa and production
- The environment will evolve as development proceeds
- The longer you wait to have a common environment build process, the harder it is to create one

The First Way Goals

- One source of the truth
  - Code, environment and configuration in one place
- Consistent release process
  - Automation is essential (one click)
- Decreasing cycle times, Faster release cadence

The Second Way – Feedback Loops

- Understanding and responding to the needs of all customers (internal and external)
- Shorten and amplify all feedback loops
- With feedback comes quality

Crossteam Connections

- Development is embedded in the Ops incident escalation process
- Dev and Ops collaborate on post-mortems and root cause analysis
- Monitoring and metrics become essential

A Word about Metrics

- Avoid vanity metrics
  - Giving you numbers that make happy noises
- Real metrics are actionable
  - What do you do when it goes up?
  - What do you do when it goes down?
- Real metrics reflect business, not technology

Automating Feedback Loops

- Capture as much data as possible at the incident
- Avoid interpretation
- The issue becomes the data, not the people who gathered it

Second Way Goals

- Defects and performance issues fixed faster
- Ops and InfoSec user stories appear as part of the application
- Everyone is communicating better
- More work getting done

The Third Way – Synergy

- Consistent process and effective feedback result in agility
- Now use that agility to experiment
- You only learn from failure
  - So fail often, but recover quickly

Break Things Before Production

- Consistency in code, environments and configuration
- ASSERTs to catch misconfigurations and inconsistencies
- Static code analysis, and testing become part of the continuous integration and deployment

Battling Technical Debt

- Allocate 20% of cycles to technical debt reduction
  - Before you end up allocating 100%
- Remember they're still visible stories with measurable metrics

Fighting Against One Right Way

- Rapid cycling encourages experimentation
  - Every feature can be split-tested
  - Use Metric Driven Development
  - If you can't tell which test is better, what's the point?

Third Way Goals

- Ability to anticipate, even define new business needs through visibility in the systems
- Ability to test and optimize new business opportunities in the system while managing risk

Resources

- Visible Ops Handbook (Gene Kim)
- The Phoenix Project (Gene Kim)
- Web Operations (Allspaw/Robbins)
- Continuous Delivery (Humble/Farley)
- Lean Startup (Eric Reis)

Architecture Evaluation

Why Evaluate?

- By the designer to inform decision making.
- By peers to inform decision making or to allow progression to later stages in the process.
- External evaluation when the architecture is closer to being complete.

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 enough.

Peer Evaluation

- Fix on the QAs to consider as part of the review – may be determined by the process or the business case.
- The architect presents the architecture to the reviewers – 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 the scenario.
- The outcome is a list of potential issues with actions: fix, mitigate, tolerate, …

External Evaluation

- Means to bring in additional expertise.
- May represent some stakeholder interests.
- More expensive and difficult to organise so this will often correspond to some major hurdle in the process.

Contextual Factors

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

Example: ATAM – The Architecture Tradeoff Analysis Method

- Designed by the Software Engineering Institute: http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=5177
- Designed to be usable where:
  - Evaluators are not expert in the architecture
  - Evaluators need not be familiar with the business goals.
  - The system need not be fully developed
  - There may be large numbers of stakeholders

Participants in ATM

- **The evaluation team:** 3-5 people with designated roles (people may have multiple roles). Team members should be seen to be neutral with respect to the project.
- **Project decision takers:** manager of the project, funder of the project, main architect
- **Architecture stakeholders:** developers, testers, integrators, maintainers, performance engineers, …

The Evaluation Team

Role	Responsibilities
Team Leader	Sets up the evaluation; coordinates with client, making sure client's needs are met; establishes evaluation contract; forms evaluation team; sees that final report is produced and delivered (although the writing may be delegated)
Evaluation Leader	Runs evaluation; facilitates elicitation of scenarios; administers scenario selection/prioritization process; facilitates evaluation of scenarios against architecture; facilitates on-site analysis
Scenario Scribe	Writes scenarios on flipchart or whiteboard during scenario elicitation; captures agreed-on wording of each scenario, halting discussion until exact wording is captured
Proceedings Scribe	Captures proceedings in electronic form on laptop or workstation: raw scenarios, issue(s) that motivate each scenario (often lost in the wording of the scenario itself), and resolution of each scenario when applied to architecture(s); also generates a printed list of adopted scenarios for handout to all participants
Questioner	Raises issues of architectural interest, usually related to the quality attributes in which he or she has expertise

ATAM Outputs

- **Concise presentation of the architecture** – needs to be presentable in around one hour.
- **Articulation of the business goals –** clearly communicated to all participants
- **Prioritized QA requirements expressed as scenarios –** testable QA requirements.
- **Risks and non-risks –** architecture decision that carries risks (or not).
- **Risk themes –** attempt to identify systemic risk by grouping risks into themes.
- **Mapping of Architecture Decisions to QA requirements –** motivating architecture decisions by QA requirements
- **Identified sensitivity and tradeoff decisions –** critical decisions that have significant impact on QA requirements.

Phases of ATAM

Phase	Activity	Participants	Typical Duration
0	Partnership and preparation	Evaluation team leadership and key project decision makers	Proceeds informally as required, perhaps over a few weeks
1	Evaluation	Evaluation team and project decision makers	1–2 days followed by a hiatus of 1–3 weeks
2	Evaluation (continued)	Evaluation team, project decision makers, and stakeholders	2 days
3	Follow-up	Evaluation team and evaluation client	1 week

Phases of ATAM

- **Phase 0:** Getting the schedule, agendas and list of stakeholders prepared and preparing necessary documents and presentations and gettting documents to the evaluation team
- **Phase 1 :** Evaluation team + decision makers – information gathering and clarification.
- **Phase 2:** Stakeholders join in the process.
- **Phase 3:** Follow up.

Evaluation Phase Steps

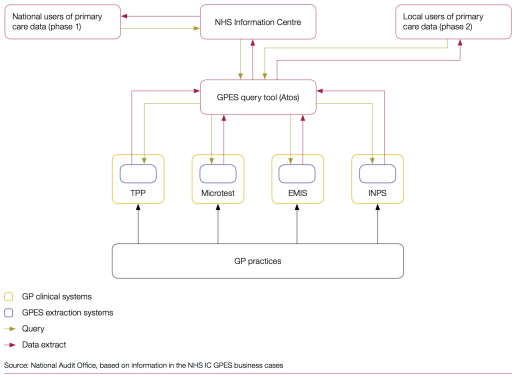
1. **Presentation of the ATAM approach –** remind participants of the approach
2. **Business drivers presentation –** functions; constraints; business goals; major stakeholders; architectural drivers
3. **Architecture presentation:**
   - Context for the system
   - Static modular view
   - Component and connector view
   - Deployment view
   - Main QA requirements and how the architecture addresses them:
     - What has been reused
     - Trace of key use cases
     - Trace of key change scenarios
     - Main issues/risks driving architectural change

Evaluation Phase Steps

4. **Identify architectural approaches –** create a catalogue of patterns and tactics used in the architecture.
5. **Generate Quality Attribute Utility Tree -** this is an approach to identifying architecturally significant requirements (ASR) by looking through the QAs – identifying particular aspects of the QA that are relevant and any requirements related to that aspect of the QA. Each ASR is ranked High, Medium or Low in importance.
6. **Analyze architectural approaches –** look at the most important QA requirement scenarios as identified at stage 5 and probe how the architecture meets the QA scenario.

- **At this point the first analysis phase is complete**

Recall the GPES Structure



Example: ATAM for GPES

- Consider some aspects of ATAM for GPES:

2. Business drivers: the need to be able to meet a large number of varying requests from varied stakeholders with good response time.
3. Present the static view, component view and deployment view if available: trace through how a request for a new extraction of data would work. A key risk would be failing to respond to a request for new data in a reasonable time.
4. For GPES we would look at the use of the query subsystem as a mediator.
5. In doing this we would consider modifiability and consider how easily it is to adapt the system to a new information request.
6. We might look at a scenario where a stakeholder wants to have a new extraction request implemented quickly.

Elaborating on Stage 6 for GPES

- Scenario: Implementing a new extract request
- Attribute: Modifiability
- Environment: Normal Operation
- Stimulus: Stakeholder request for a new extract.
- Response: New extract available within 2 months of request.

Elaborating on Stage 6 for GPES

Arch Decision	Sensitivity	Tradeoff	Risk	Non-risk
Use Mediator				N1
Implement in business logic	S1	T1	R1	
Depend on Vendors	S2		R2	

Elaborating on Stage 6 for GPES

- Reasoning:
  - Mediator ensures each modification can be executed concurrently so this avoids delays in individual systems.
  - Is there a chance that the development time for any of the business logic components could exceed 2 months – the tradeoff is that any alternative is unacceptable to some stakeholders (the vendors in particular).
  - Depending on the vendors accurately to implement the modification may be an issue.

Phase 2 Evaluation

- **Repeat the summary of ATAM**

7. **Brainstorm prioritization of scenarios –** revisit the prioritization for additional scenarios e.g. a particular stakeholder (performance engineer) might propose a scenario on the response time of the system.
8. **Analyze Architectural Approaches –** revisit stage 6 but with an expanded and reprioritized set of scenarios
9. **Present results –** the evaluation group tries to group risks into risk themes to identify systemic issues and results are presented.

ATAM Results

- Documentation of architectural approaches taken by the project.
- Prioritized list of scenarios
- Utility tree
- Risks discovered
- Non-risks identified
- Sensitivity and Tradeoff points identified

Lightweight Version

Step	Time Allotted	Notes
1: Present the ATAM	0 hrs	The participants are familiar with the process. This step may be omitted.
2: Present Business Drivers	0.25 hrs	The participants are expected to understand the system and its business goals and their priorities. Fifteen minutes is allocated for a brief review to ensure that these are fresh in everyone's mind and that there are no surprises.
3: Present Architecture	0.5 hrs	Again, all participants are expected to be familiar with the system and so a brief overview of the architecture, using at least module and C&C views, is presented and 1 to 2 scenarios are traced through these views.
4: Identify Architectural Approaches	0.25 hrs	The architecture approaches for specific quality attribute concerns are identified by the architect. This may be done as a portion of step 3.

Lightweight version (ctd)

4: Identify Architectural Approaches	0.25 hrs	The architecture approaches for specific quality attribute concerns are identified by the architect. This may be done as a portion of step 3.
5: Generate Quality Attribute Utility Tree	Variable 0.5 hrs – 1.5 hrs	Scenarios might exist: part of previous evals, part of design, part of requirements elicitation. If you've got 'em, use 'em and make them into a tree. Half hour. Otherwise, it will take longer. A utility tree should already exist; the team reviews the existing tree and updates it, if needed, with new scenarios, new response goals, or new scenario priorities and risk assessments.
6: Analyze Architectural Approaches	2–3 hrs	This step—mapping the highly ranked scenarios onto the architecture—consumes the bulk of the time and can be expanded or contracted as needed.

Lightweight version (ctd)

7: Brainstorm and Prioritize Scenarios	0 hrs	This step can be omitted as the assembled (internal) stakeholders are expected to contribute scenarios expressing their concerns in step 5.
8: Analyze Architectural Approaches	0 hrs	This step is also omitted, since all analysis is done in step 6.
9: Present Results	0.5 hrs	At the end of an evaluation, the team reviews the existing and newly discovered risks, non-risks, sensitivities, and tradeoffs and discusses whether any new risk themes have arisen.
TOTAL	**4–6 hrs**	

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.
- ATAM is comprehensive and attempts to capture project risks.
- ATAM is one approach to this – the standard process is not particularly agile – lightweight process is better.