# Chapter 1

# Testability

## 1.1 Overview

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

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

## 1.2 General Scenario of Testing

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

  These tests can be run manually or using automated testing tools
- **Stimulus:** Why is the set of tests being executed? Could be due to:
  - completion of a certain part of the code, or of an entire system

  - complete integration of subsystem into larger system
  - completion of entire system
  - time to deliver the system to the customers
- **Environment:** Which environment we are dealing with:
  - Design time
  - Development time
  - Compile time
  - Integration time
  - Deployment time
  - Run time
- **Artifacts:** The portion of the system that is being tested
- **Response:** What is the outcome of running the tests:
  - Execute test suite and capture results
  - Capture activity that resulted in the fault
  - Control and monitor the state of the system
- **Response Measure:** How exactly we will measure the results and if we were successful in the testing Could be:
  - Find a fault or type of faults
  - Achieve a certain percentage of code coverage
  - Time limit for running the tests

Figure 1.1 on the next page shows an example of this general scenario. Testability deals with making this process easier to carry out.

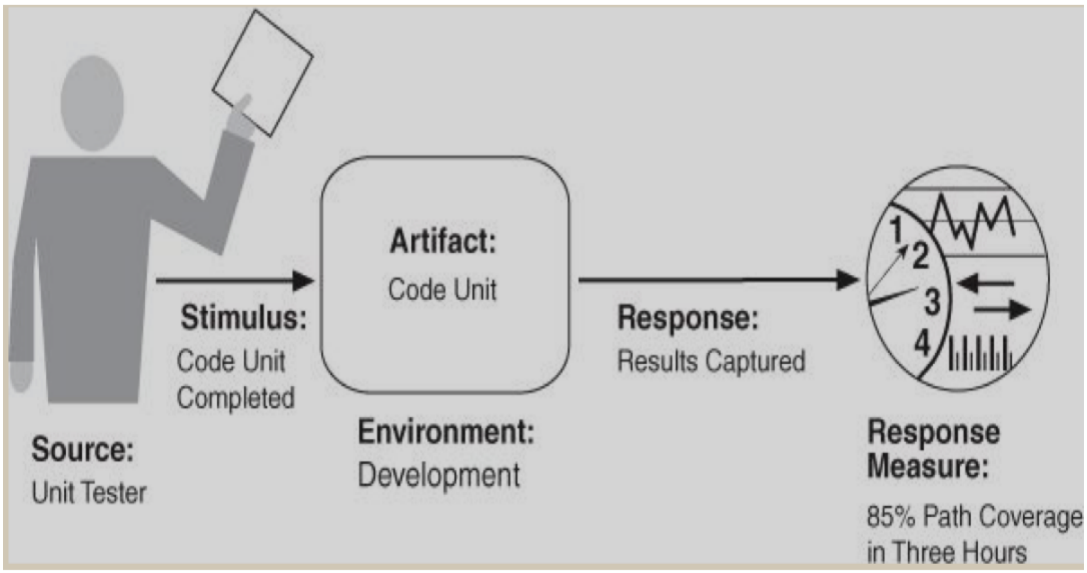


Figure 1.1: An example of such a scenario

## 1.3 Main Tactics Towards Increasing Testability

- Limit complexity in modules themselves - for example, split a large, complex module into smaller, less complex ones
- Limit complexity in system structure - for example, introduce specific interfaces in a system that give us better access to a log database
- Limit nondeterminism

## 1.4 Checklist Towards Achieving Testability

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

- **Allocation of responsibilities:** This means determining which system responsibilities are most critical and thus need the most testing. Make sure that system responsibilities have been created to actually perform the testing and capture the results, capture(log) activity that resulted in fault or unexpected behaviour, control and observe relevant system state for testing.
- **Coordination Model:** Ensure system's coordination and communication mechanisms support the execution of the test suite, support capturing activity that resulted in a fault, support injection and monitoring of state, and do no introduce unecessary nondeterminism
- **Data Model:** Determine the major data abstractions that must be tested to ensure the correct operation of the system.
- **Mapping Among Architectural Elements:** Determine how to test possible mappings of architectural elements so that the desired test response is achieved.
- **Resource Management:** Ensure that there are sufficient resources available to execute the test suite and capture the results. Ensure test environment is representative of the environment in which the system will run in.
- **Binding Time:** Ensure that components that are bound later than compile time can be tested in the late-bound context.
- **Choice of Technology:** Determine what technologies are available to help achieve the testability scenarios that apply to your architecture.

## 1.5 Summary

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

# Chapter 2

# Modifiability

## 2.1 Overview

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

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

## 2.2 Four key questions that are asked when making a system:

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

## 2.3 A specific scenario with same "general scenario" terms used as in the testability section:

Source: Developer
Stimulus: Wishes to change the UI
Artifact: The code itself
Environment: Design time
Response: Change made and unit tested

Response Measure: This should all be done in 3 hours

## 2.4 Important Terms

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

## 2.5 Tactics of Modifiability:

- **Reduce size of a module** – split it up
- **Increase cohesion within a module** – increase semantic coherence
- **Decrease the coupling in the module** – this can be done in the following ways:
  - Encapsulate
  - Use an intermediary
  - Restrict dependencies
  - Refactor
  - Abstract common servies – think of OOP

- **Defer binding** – we can use the "reduce coupling" tactics later in the process so that they are more likely to be done by a computer rather than a human

## 2.6 Checklist Towards Achieving Modifiability

- **Allocation of Responsibilities:** Try to work out how things are likely to change – work out what responsibilities change. Try to modularize so that change does not affect responsibilities that span many modules.
- **Coordination Model:** See how changes are likely to affect coordination and make it so that most probable changes impact coordination across a small numebr of modules.
- **Data Model:** Data model changes impact as few modules as possible.
- **Mapping Among Architectural Elements:** Look at potential changes and see if some may best be responded to by changing the mapping to elements.
- **Resource Management:** Determine how a change in responsibility will change resource.
- **Binding Time:** Control choice of binding times so there are not too many combinations to consider. Defer binding to later.
- **Choice of Technology:** Choose technologies that make the most likely changes easier – of course, balance with costs.

## 2.7 Summary

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

# Chapter 3

# Architectural Modeling:

## 3.1 Overview

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

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

## 3.2 How the model looks like:

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

## 3.3 MVC Model – model to test the "performance" quality attribute

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

If we have good estimates for all the different components of the model (distribution of external service demands, queuing disciplines, network laten-

cies, transfer characteristics), then it can be a very good predictor of quality attributes.

## 3.4 Main Quality Attributes and their Analysis Techniques

The main quality attributes and their analysis techniques are:

- **Availability** – Markov models, statistical models
  Availability is a measure of software quality defined as MTBF / (MTBF + MTTR) MTBF is Mean Time Between Failures MTTR is Mean Time To Repair
- **Interoperability** – Conceptual framework
- **Modifiability** – Coupling and cohesion metrics, cost models
- **Performance** – Queuing theory, real-time scheduling theory
- **Security** – No architectural models
- **Testability** – Component interaction metrics
- **Usability** – No architectural models

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

## 3.5 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

## 3.6 Summary

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

# Chapter 4

# Lifecycle

## 4.1 Overview

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

## 4.2 Typical Examples

There are some typical stages of the software lifecycle.

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

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

| 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.<br>• Document the architecture using multiple views.<br>• 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.<br>• Collect use case, growth, and exploratory scenarios using general scenarios, utility trees, and scenario brainstorming.<br>• Design the new architectural strategies using ADD.<br>• 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.<br>• Make decisions among architectural strategies based on ROI, using the CBAM results. |

Figure 4.1: A diagram showing the typical lifecycle stages and what software engineers do to the architecture at each stage.

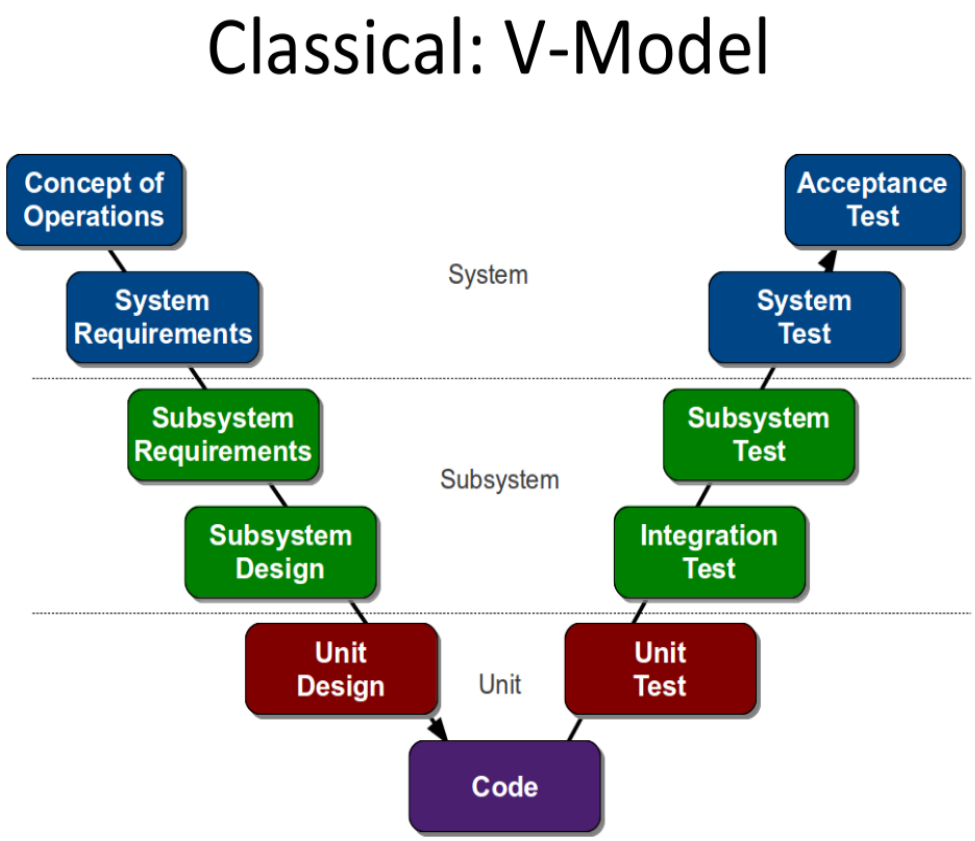


Figure 4.2: A diagram showing the V model

# Transitional: Spiral Model

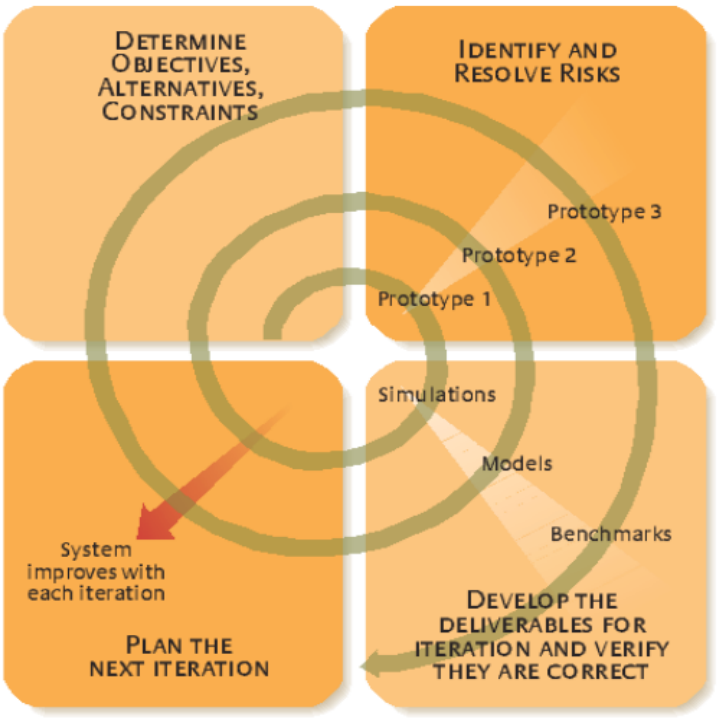


Figure 4.3: A diagram showing the spiral model

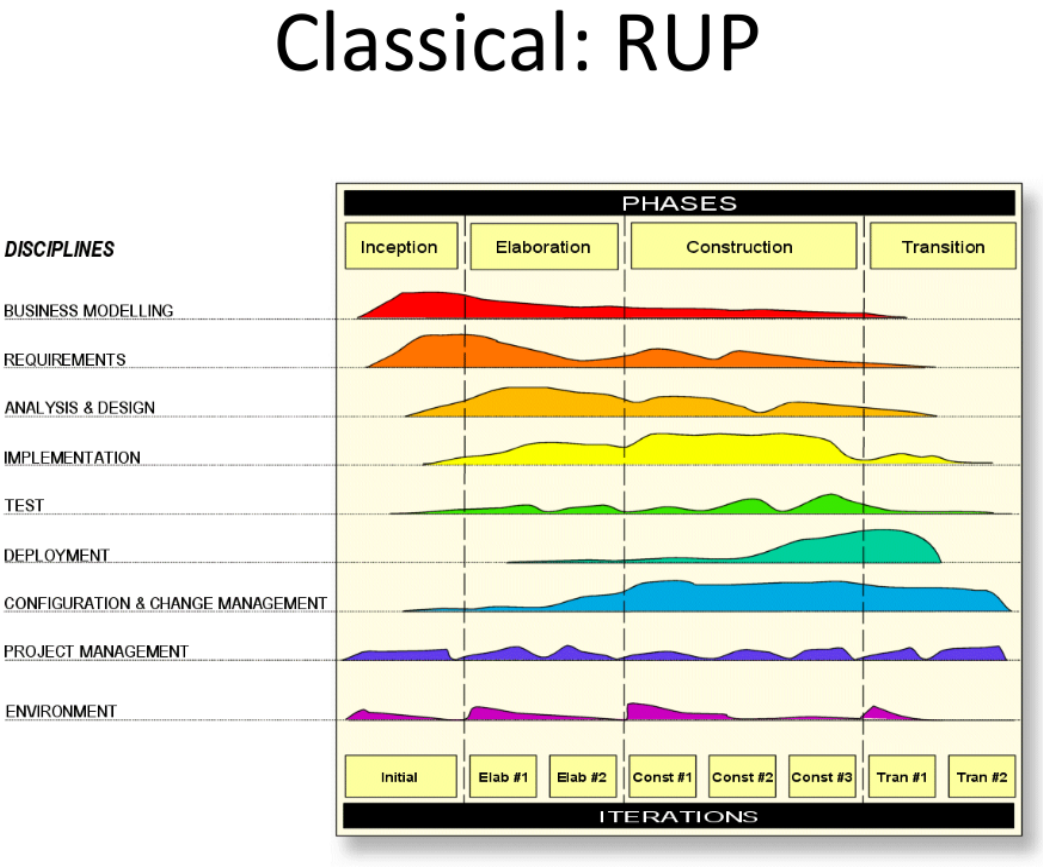


Figure 4.4: A diagram of the RUP model

## 4.3 Agile Programming

Agile Programming Practice:

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

Manifesto for Agile Software Development:

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

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

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

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

## 4.4 Comparison of Agile versus Plan-Driven Approach

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

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

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

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

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

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

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

# Analysis Techniques

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

Figure 4.5: Analysis Techniques

# Analysis Techniques and Stage

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

Figure 4.6: Analysis Technique and Stage

## 4.5 Summary

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

# Chapter 5

# Product Line Architecture

## 5.1 Overview

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

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

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

## 5.2 Key Properties

Key properties of this product line architecture:

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

## 5.3 Benefits to organization

There are several benefits to the organization:

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

## 5.4 Main Terms

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

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

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

## 5.5 Different Techniques in Introducing Product Lines

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

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

## 5.6 Pros of Using a Product Line:

- Increased competitiveness on the market because of reduced hardware resource consumption and reduced time to market for new features
- Development efficiency – reuse, easy configuration of software products, increased planning accuracy
- Quality - interface integrity, reuse of core assets

- Customer needs – differentiation by individual software solutions, clear feature-cost mapping

## 5.7 Architectural Features of Product Lines:

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

## 5.8 Phased Introduction

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

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

# Chapter 6

# DevOps

## 6.1 Overview

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

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

## 6.2 OSLC

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

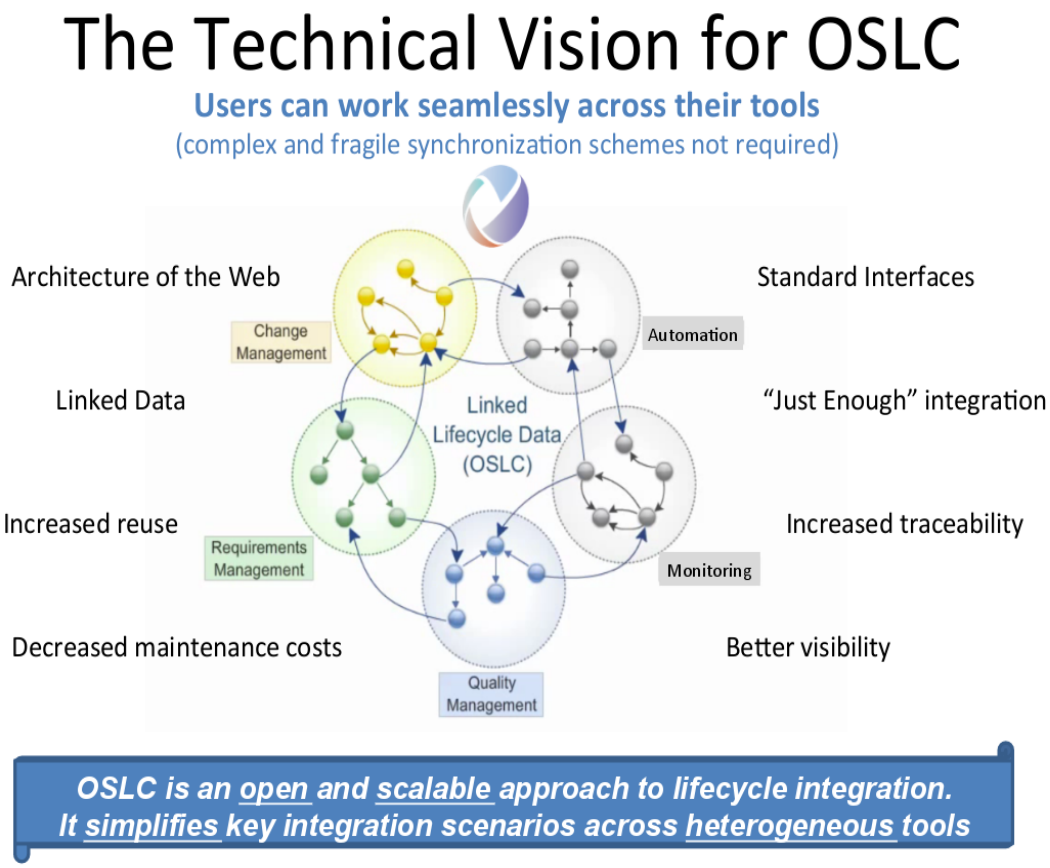


Figure 6.1: Summary of OSLC

## 6.3 Critical Points in the Lifecycle of Software

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

## 6.4 Microservices Architectural Pattern

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

## 6.5 How to make development easier, faster, and better

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

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

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

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

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