SE3Q11: Week 1

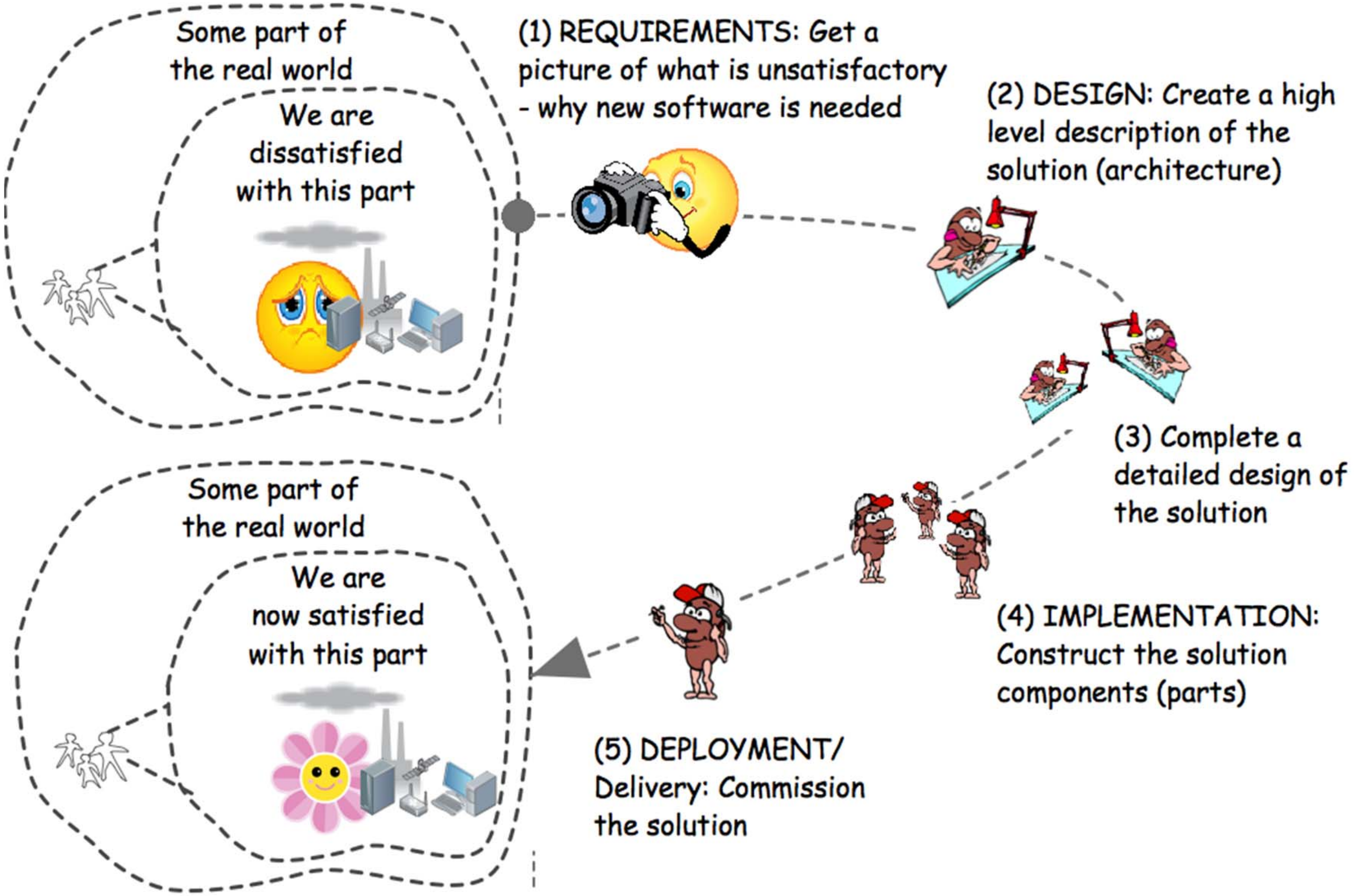
Introduction

Dr. Ken Boness

k.d.boness@reading.ac.uk

Week 1 (a)
Software Quality and Testing

Developing Software



Some part of the real world

We are dissatisfied with this part

(1) REQUIREMENTS: Get a picture of what is unsatisfactory - why new software is needed

(2) DESIGN: Create a high level description of the solution (architecture)

(3) Complete a detailed design of the solution

(4) IMPLEMENTATION: Construct the solution components (parts)

(5) DEPLOYMENT/ Delivery: Commission the solution

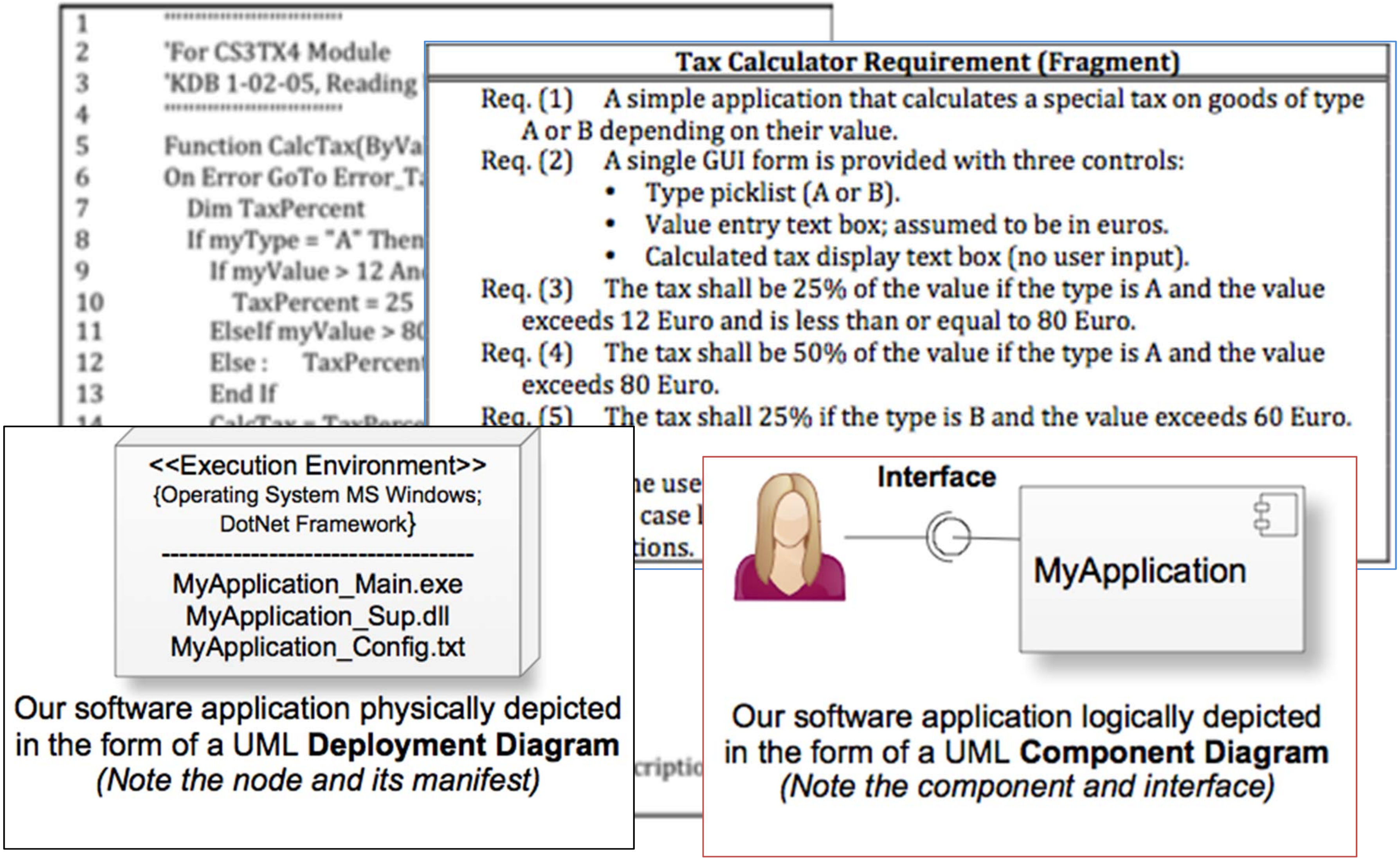
Some part of the real world

We are now satisfied with this part

A simple application

- A Tax calculator application for a PC
- What are the quality concerns?
- What do we test?

Pandora's box: many artefacts involved



```
''''''''''''''''''''''''''''''''
'For CS3TX4 Module
'KDB 1-02-05, Reading
''''''''''''''''''''''''''''''''

Function CalcTax(ByVa
On Error GoTo Error_T
   Dim TaxPercent
   If myType = "A" Then
      If myValue > 12 An
         TaxPercent = 25
      ElseIf myValue > 8
      Else :    TaxPercent
      End If
      CalcTax = TaxPerce
```

Tax Calculator Requirement (Fragment)

Req. (1) A simple application that calculates a special tax on goods of type A or B depending on their value.

Req. (2) A single GUI form is provided with three controls:

- Type picklist (A or B).
- Value entry text box; assumed to be in euros.
- Calculated tax display text box (no user input).

Req. (3) The tax shall be 25% of the value if the type is A and the value exceeds 12 Euro and is less than or equal to 80 Euro.

Req. (4) The tax shall be 50% of the value if the type is A and the value exceeds 80 Euro.

Req. (5) The tax shall 25% if the type is B and the value exceeds 60 Euro.

he use

case l

tions.

criptic

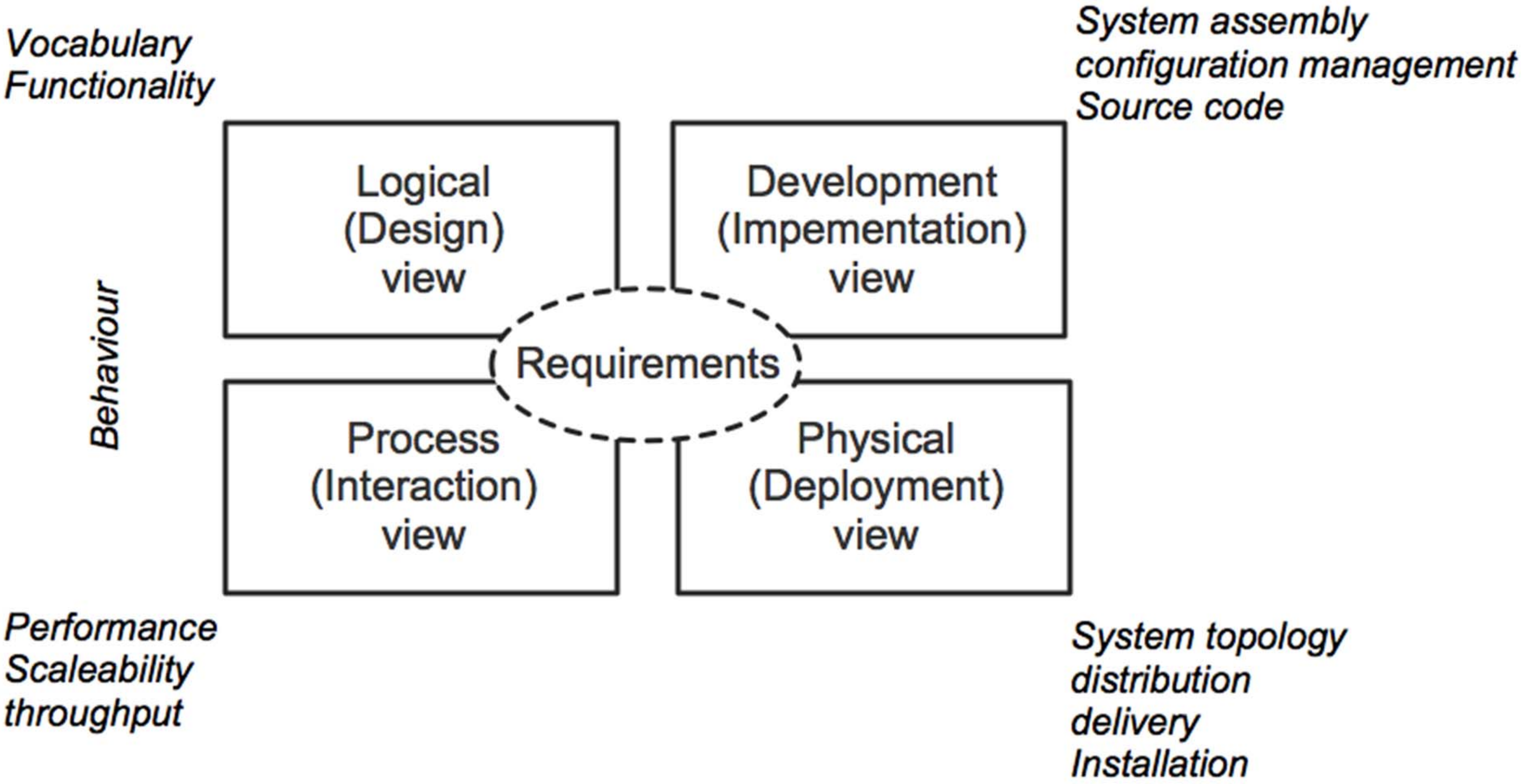
<<Execution Environment>>
{Operating System MS Windows;
DotNet Framework}

MyApplication_Main.exe
MyApplication_Sup.dll
MyApplication_Config.txt

Our software application physically depicted in the form of a UML **Deployment Diagram**
(Note the node and its manifest)

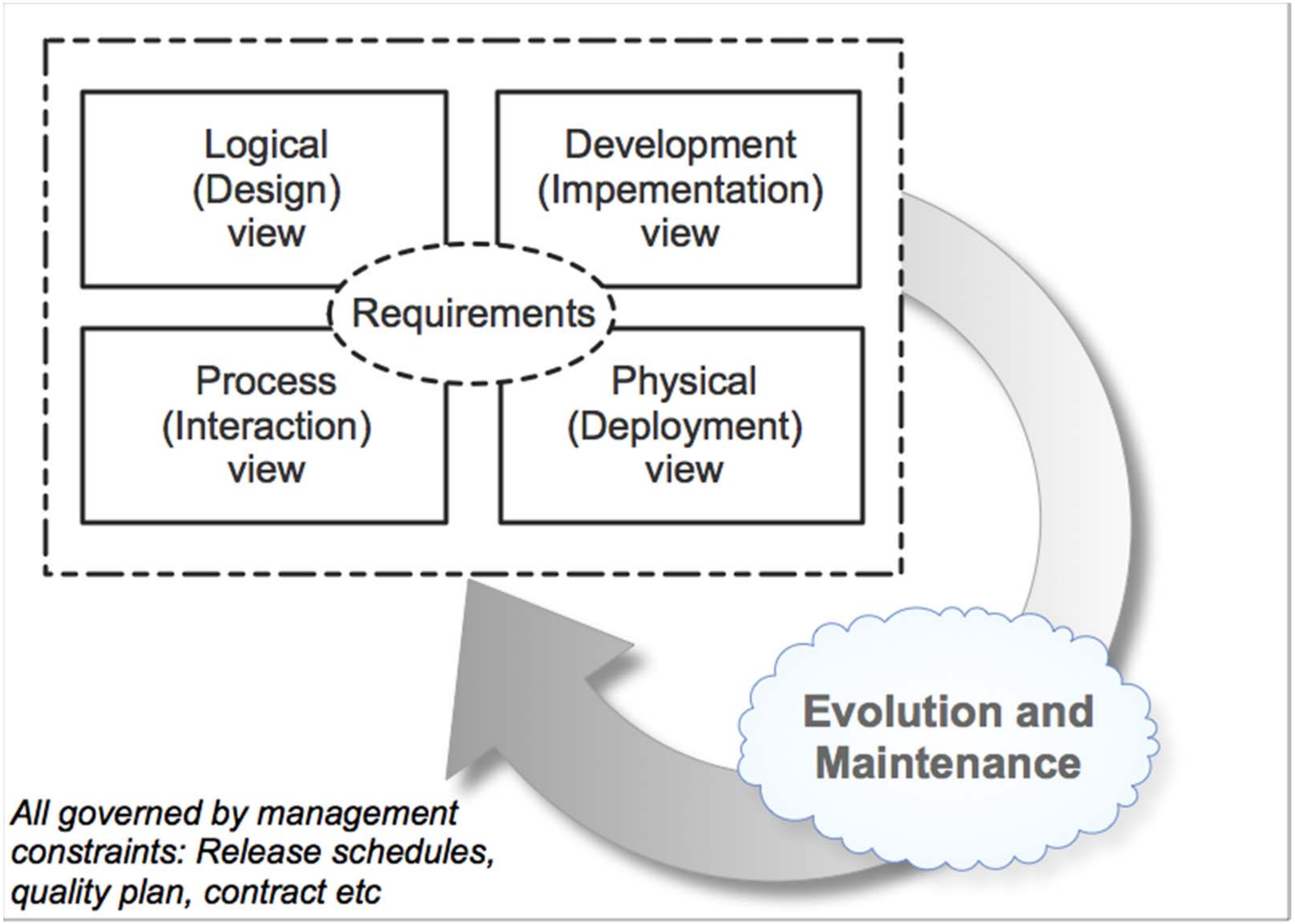
Our software application logically depicted in the form of a UML **Component Diagram**
(Note the component and interface)

An architectural perspective on Pandora’s box



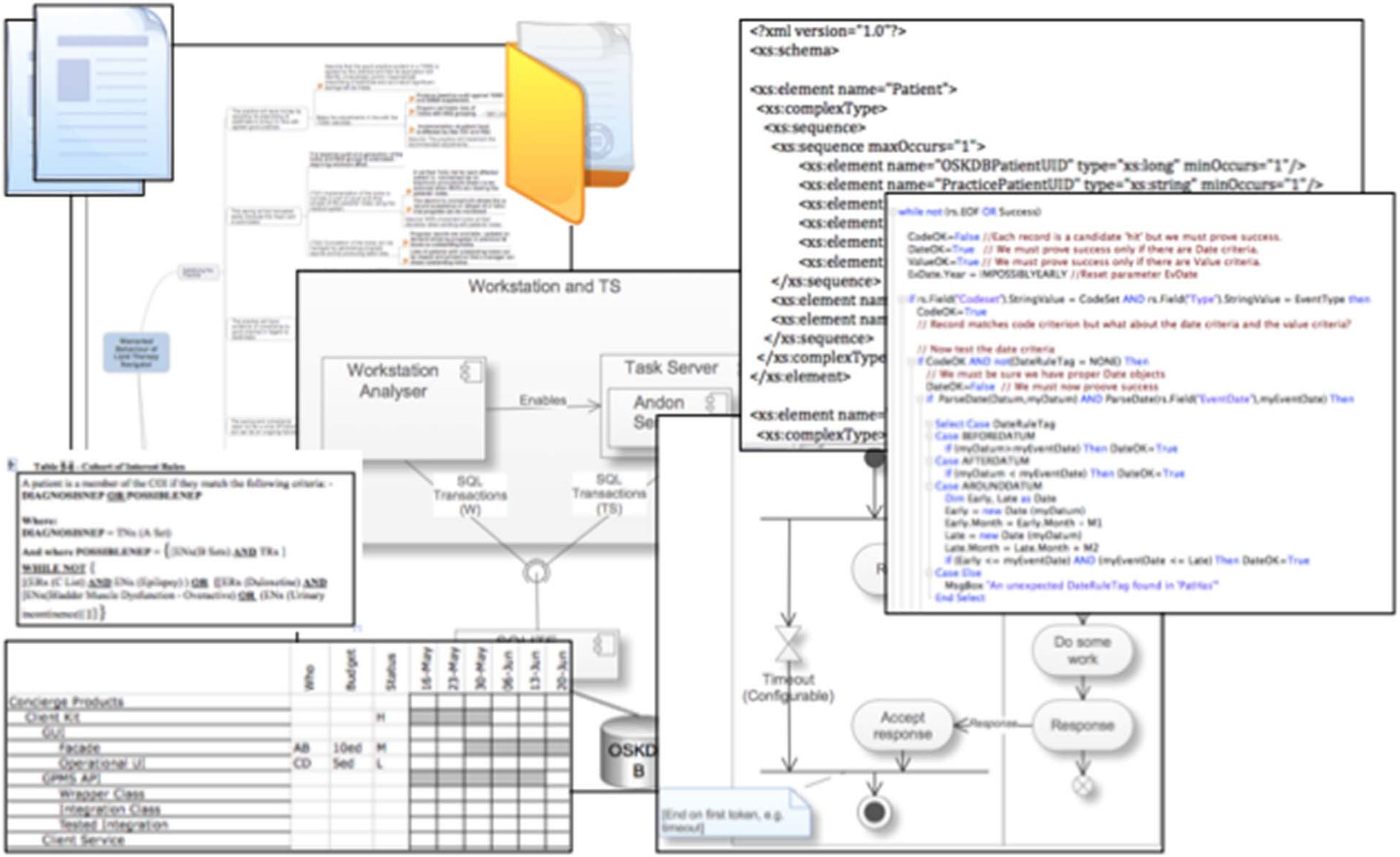
Adapted from
Booch, G., Rumbaugh, J. & Jacobson, I. (1999) *The unified modeling language user guide.* Addison-Wesley.
And Sommerville, I. (2011) *Software engineering*. 9th ed Pearson/Addison-Wesley.

Add management and evolution imperatives to the consideration


Logical (Design) view
Development (Impementation) view
Requirements
Process (Interaction) view
Physical (Deployment) view
Evolution and Maintenance

All governed by management constraints: Release schedules, quality plan, contract etc

Conclusion: There are many artefacts to be managed and tested in a quality aware development regime!



```
<?xml version="1.0"?>
<xs:schema>

<xs:element name="Patient">
 <xs:complexType>
  <xs:sequence>
   <xs:sequence maxOccurs="1">
       <xs:element name="OSKDBPatientUID" type="xs:long" minOccurs="1"/>
       <xs:element name="PracticePatientUID" type="xs:string" minOccurs="1"/>
       <xs:element
       <xs:element
       <xs:element
       <xs:element
   </xs:sequence>
   <xs:element nam
   <xs:element nam
  </xs:sequence>
 </xs:complexType
</xs:element>

<xs:element name=
 <xs:complexType>
```

```
while not (rs.EOF OR Success)

  CodeOK=False //Each record is a candidate 'hit' but we must prove success.
  DateOK=True  // We must prove success only if there are Date criteria.
  ValueOK=True // We must prove success only if there are Value criteria.
  EvDate.Year = IMPOSSIBLYEARLY //Reset parameter EvDate

  If rs.Field("Codeset").StringValue = CodeSet AND rs.Field("Type").StringValue = EventType then
    CodeOK=True
    // Record matches code criterion but what about the date criteria and the value criteria?

    // Now test the date criteria
    If CodeOK AND not(DateRuleTag = NONE) Then
      // We must be sure we have proper Date objects
      DateOK=False  // We must now proove success
      If  ParseDate(Datum,myDatum) AND ParseDate(rs.Field("EventDate"),myEventDate) Then

        Select Case DateRuleTag
        Case BEFOREDATUM
          If (myDatum>=myEventDate) Then DateOK=True
        Case AFTERDATUM
          If (myDatum < myEventDate) Then DateOK=True
        Case AROUNDDATUM
          Dim Early, Late as Date
          Early = new Date (myDatum)
          Early.Month = Early.Month - M1
          Late = new Date (myDatum)
          Late.Month = Late.Month + M2
          If (Early <= myEventDate) AND (myEventDate <= Late) Then DateOK=True
        Case Else
          MsgBox "An unexpected DateRuleTag found in 'PatHas'"
        End Select
```

Table 5.4 - Cohort of Interest Rules

A patient is a member of the COI if they match the following criteria: - **DIAGNOSISNEP OR POSSIBLENEP**

Where:
DIAGNOSISNEP = TNs (A Set)
And where POSSIBLENEP = { [ENs(B Sets) **AND** TRs]
WHILE NOT {
[(ERs (C List) **AND** ENs (Epilepsy)) **OR** ([ERs (Duloxetine) **AND** [ENs(Bladder Muscle Dysfunction - Overactive) **OR** (ENs (Urinary incontinence)]])}

	Who	Budget	Status	16-May	23-May	30-May	06-Jun	13-Jun	20-Jun
Concierge Products									
Client Kit			H	▨	▨	▨			
GUI									
Facade	AB	10ed	M			▨	▨	▨	▨
Operational UI	CD	5ed	L						
GPMS API				▨	▨	▨	▨	▨	
Wrapper Class									
Integration Class									
Tested Integration									
Client Service									

Quality

- The quality attributes of a system are known as non-functional requirements such as:-
  - Correctness
  - Reliability
  - Usability
  - Re-usability
  - Reliability
  - Maintainability
  - Testability
- How well does the system achieve these things.
- We must plan early to test these – leaving them to the end is too late!

Testing

- Testing identifies **faults**, whose removal increases the software quality by increasing the software’s potential reliability.
- Testing is the measurement of software quality. We measure how closely we have achieved quality by testing the relevant factors such as correctness, reliability, usability, maintainability, reusability, testability, etc.
- Other factors that may determine the testing performed may be contractual requirements, or legal requirements, normally defined in industry-specific standards, or based on agreed best practice (or more realistically non-negligent practice).

From BCS ISEB Foundation

Error, Fault, Failure

- An **error** is a human action that produces an incorrect result.
- A **fault** is a manifestation of an error in software (also known as a defect or bug).
- A fault, if encountered, may cause a **failure**, which is a deviation of the software from its expected delivery or service.
- **Reliability** is the probability that software will not cause the failure of a system for a specified time under specified conditions.

From BCS ISEB Foundation

Acceptable errors?

- Errors occur because we are not perfect and, even if we were, we are working under constraints such as delivery deadlines.
- Errors result in faults in the artefacts of software development
- Faults may lead to failure
- A single failure can cost nothing or a lot
  - Web page script missing – reload
  - Software in safety-critical systems can cause death or injury if it fails, so the cost of a failure in such a system may be in human lives. For example aircraft control software.

Testing approaches

- We test primarily to find **faults** rather than demonstrating correctness.
- There are two fundamentally different ways we can discover faults
  - We can execute (run) the software and try to detect faults
    - This we call **dynamic** testing
  - We can put all other development documents under scrutiny without running the software
    - This is called **static** testing
- We will be looking at each of these approaches over the coming lectures.

Desirable Attributes of a Tester

- Good communicator
- Methodical
- Tenacious
- Accurate
- Independent
- Happy when finding faults
- Confident of the positive value of testing

- Often developers and testers have different psychologies
  - Finding faults is not a destructive activity
    - Though sensitive developers may see it so!
    - Must have good ways of communicating with developers

A need for investment?

- We have seen that errors cause faults which may cause failure
- We have also seen that, bearing in mind the architecture and management concerns we can identify a large number of artefacts
    - any can have faults!
- But establishing a quality controlling regime and setting up testing resources is a cost to our organisation
    - Is it worth the investment?

Justifying the Investment

When there is a significant danger of:-

- Litigation arising from software malfunctions
- High development costs arising from unexpected reworking
- High post supply support costs arising from unexpected reworking

And if there is a need to:-

- Comply with customers' mandated quality procedures
- Comply with auditable mandates
    - such as on safety and/or security
- Maintain recognised quality accreditation
    - Such as ISO 9000, TickIT, CMM level above 2
- Maintain a reputation for quality in order to attract business

Week 1 (b)
Capability and Metrics

Capability Maturity

- The Capability Maturity Model was created by US government in order to assess the capabilities of the companies that it employs.
- It sets standards for five levels of process maturity in an organisation.
- It is not a a process improvement system nor a quality system but through its clarity about five ascending levels of maturity it is a useful process improvement reference.
- See www.sei.cmu.edu and the CMM folder on Blackboard.

CMM levels 1&2

Level 1 – Initial (*Ad hoc*)

- No planning
- Non structure
- No expected results
- Destructive testing
- Or: “what shall we test today”

Level 2 – Repeatable

- Basic documentation & standards
- Test plans exist
- Probably manual recording on paper
- Able to repeat / reuse previous project successes on similar applications

CMM Level 3 – Defined

- Good documentation & standards
- Test material retained
- Regression test plans available
- Test material has become an asset that can be used by the company
- Starting to use automation

CMM levels 4&5

Level 4 – Managed (with metrics)

- Detailed metrics of the software process and product quality are recorded.
- Test management tools in regular use.
- Regression test plans available
- Using test automation systematically.

Level 5 – Optimising

- Testing the process not the product
- Rare in software testing environments

Process & Measurement

- CMM provides a structured passage through increasing levels of maturity.
- Brings in:
    - “ repeatable “ , “defined” etc
- Some maturity steps are binary
    - Procedures exist
    - Test plans exist
- Others require measurements.
    - See for example level 4.
- We will need metrics for our processes.
- Thus we can then detect progress (or regress).

Process Metrics

- In our Quality context metrics arise in :-
  - Process improvement / management
  - Product quality reporting
- Some of these overlap.
- For example:
  - Residual Defects per stage
- Others may not
  - Resource Usage per stage.
  - Completion Rates per stage.

Purposeful Metrics

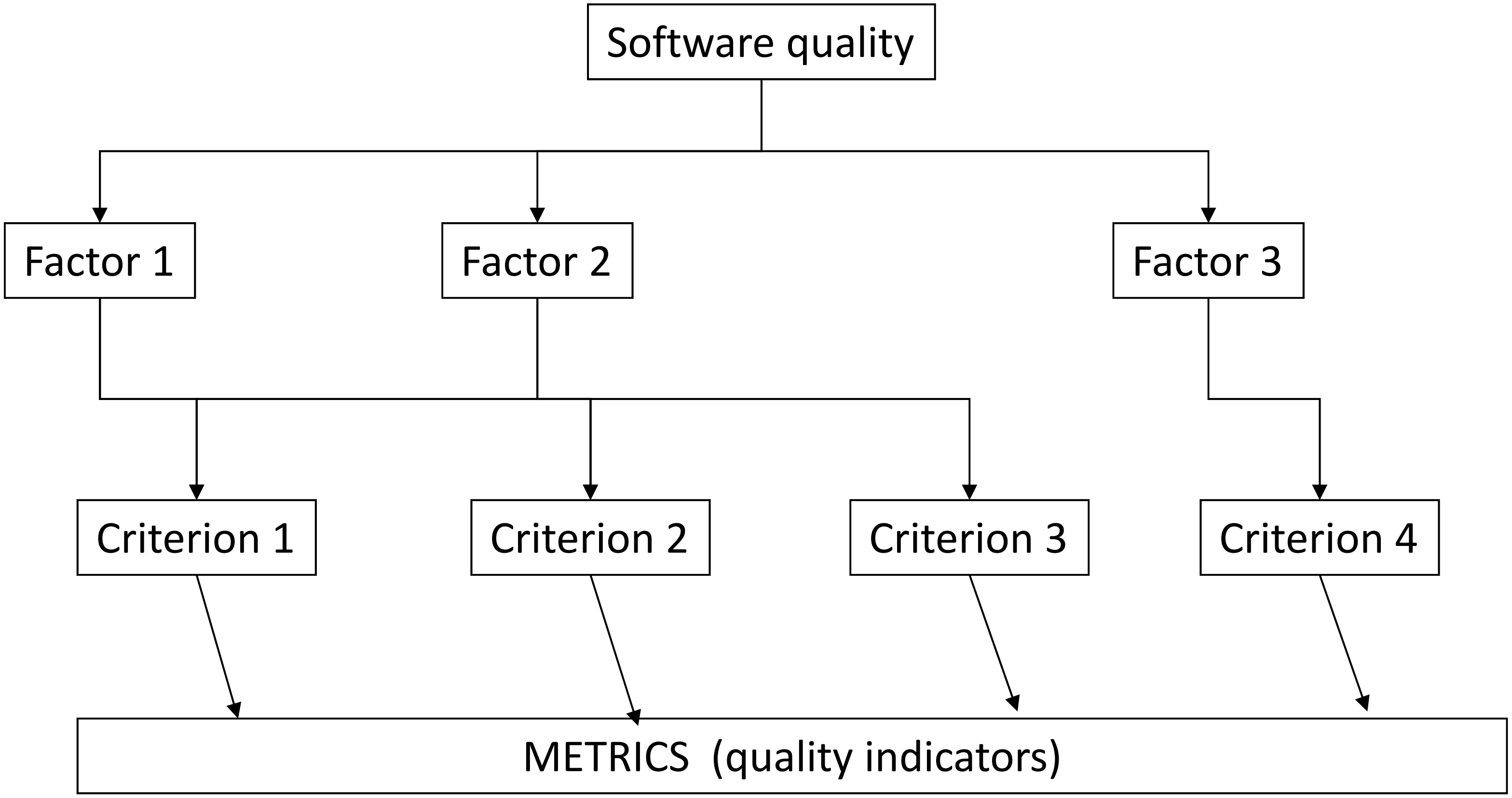
- In order to know what metrics are needed we need to consider why we want them; what are they to show?
- Two approaches are well known:-
  - “Goal Question Metric” (GQM)
  - “Factor-Criteria-Metric” (FCM)
- We will look at FCM and focus on quality as specified in the NASA Software Assurance Technology Center (SATC) quality model.
  - This illustrates the logical application of FCM .

Metrics in Software Quality

- FCM is introduced In the book by Plosch in the context of the SATC quality model.
  - Reinhold Plosch, “Contracts, scenarios and Prototypes; Springer 2004”

- Referring to DIN ISO 9126 [Din91]
  - “Software quality Is the whole of the attributes and attribute values of a software product that refer to its suitability to fulfil defined or assumed requirements.”

The idea of FCM-



FCM applied to The SATC Quality Model

Software Quality

Factors	Criteria	Metrics
Requirements Quality	Ambiguity	Number of weak phrases
	Completeness	Number of TBD's
	Understandability	Document Structure
	Volatility	Count of changes / count of Req
	Traceability	No. of reqs not traced to code/test
Product Code Quality	Architecture	Logic complexity, size
	Maintainability	correlation of complexity/ size
	Reusability	correlation of complexity/ size
	Internal Documentati	Comment percentage
	External Documentat	Readability Index
Implementation Effectiveness	Resource Usage	PM spent on lifecycle activities
	Completion Rates	(planned) task completions
Testing Effectiveness	Correctness	Errors and Criticality
		Time finding errors
		Time (effort) of error fixes

SATC Requirements Criteria

- Ambiguity: Requirements that may have multiple meanings, or those that leave the developr the decision whether or not implementation should take place.
- Completeness: A requirement document is complete if it contains all the requirements specified in adequate detail to allow design and implementation to proceed.
- Understandability: Relates to the ability of the developers to understand what is meant by the requirements document.
- Volatility: a Mouth requirements document is one that is changed frequently. Additionally, the impact of changes to the requirements Increases with the project time.
- Traceability: Software requirements must be traceable to the customer/system-requirements (backward) and to the design and code (forward).

SATC Product Code Quality Criteria

- Architecture: This criterion deals with the evaluation of modules to Identify possible error-prone modules and to indicate' potential problems in reusability and maintainability.
- Maintainability: This is the suitability of the software for ease of locating and fixing a fault In the Implementation.
- Reusability: This is the ability of the software to be reused in a different content or application.
- Internal Documentation: This refers to the adequacy of the internal code documentation to be able to maintain the software product.
- External Documentation: This refers to the adequacy of the external code documentation to be able to maintain the software product; taking perspective of system users.

SATC Impl. And Testing Effectiveness Criteria

- Resource Usage: This is a process related criterion that indicates whether the resource usage (eg. Personnel hours) correlates with the appropriate phase of the project.
- Completion Rates: This is a process related criterion that deals with the completion of deliverables - in reference to defined schedules.
- Correctness: This is the extent to which the product fulfils the underlying specifications.

Finding More Information

- look up on www
  - “Software Assurance Technology Center (SATC)” and NASA
  - Factor-criteria-metric (FCM)
  - Goal Question Metric (GQM)

Collecting Metrics - Problems

- How do we collect metrics?
  - This is a very practical and possibly expensive problem.
  - Can worry managers
- Entering data needed is boring !
  - Especially to developers
  - And can easily be sabotaged

Collecting Metrics - Remedies

- Ensure corporate buy-in
  - – The bosses will champion the required effort
  - – The staff see the point.
- Collect only the data you need !
- Be seen to use the data !
- Automate where possible.
  - – Special tools
  - – Side effects of normal process

Managing Metrics

- Plan - decide the purpose with management
- Specify – apply FCM reasoning
  - Ensure chosen/ specified metrics are realistic
- Execute - Have timely means to gather the metrics in place
  - Use forms to collect data for metrics
  - Use special tools; such as :
    - Process management
    - Computer Aided Software Testing (CAST)
- Record
  - Use a database to store metrics
    - For use in this project
    - and future projects
- (Note the reuse of our **Fundamental Test Process**)

SE3SQ11: Week 2
Practical 1
Visual Studio Metrics

Dr. Pat Parslow

k.d.boness@reading.ac.uk

SE3SQ11: Week 3
Practical 2
Looking at automated and static testing

Dr. Pat Parslow

k.d.boness@reading.ac.uk

University of Reading

SE3Q11: Week 4

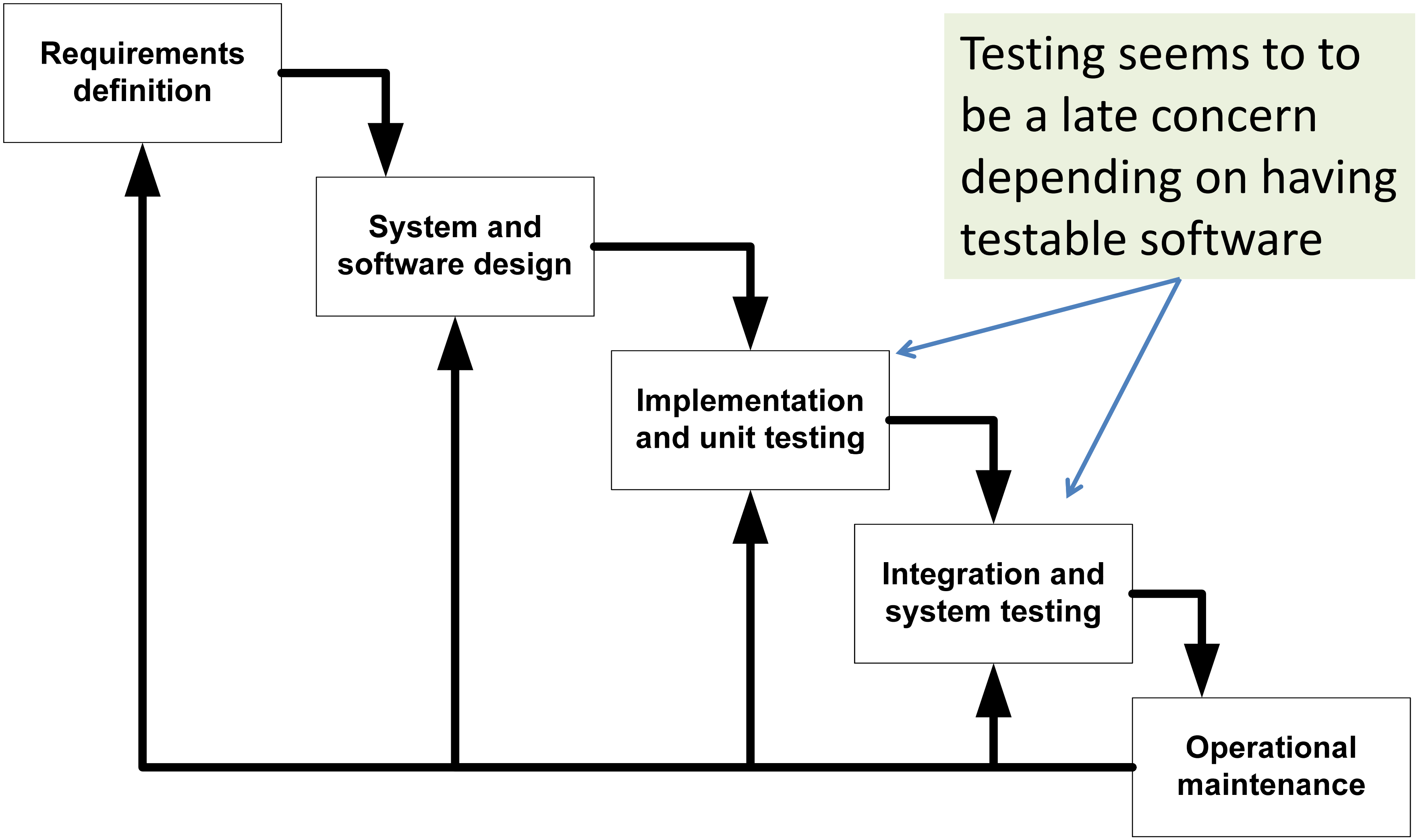
The V-Model & Test Cases

Dr. Ken Boness

k.d.boness@reading.ac.uk

(a) The V-Model & Testing Opportunities

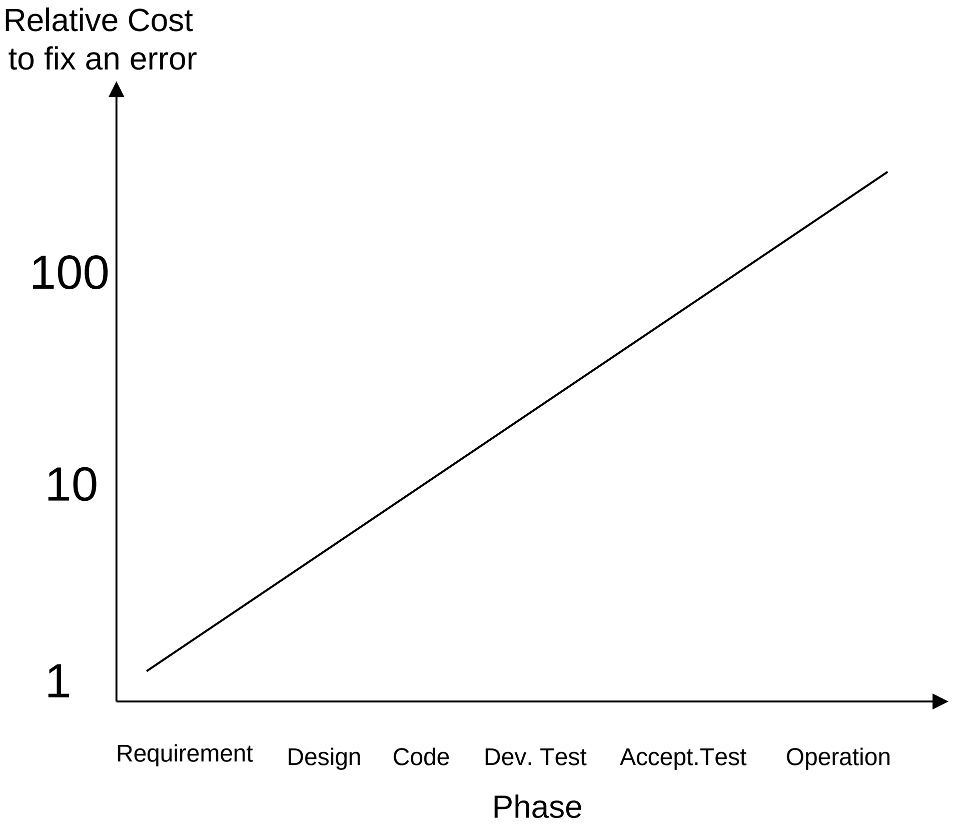
c.f. the waterfall development lifecycle


Requirements definition
System and software design
Implementation and unit testing
Integration and system testing
Operational maintenance

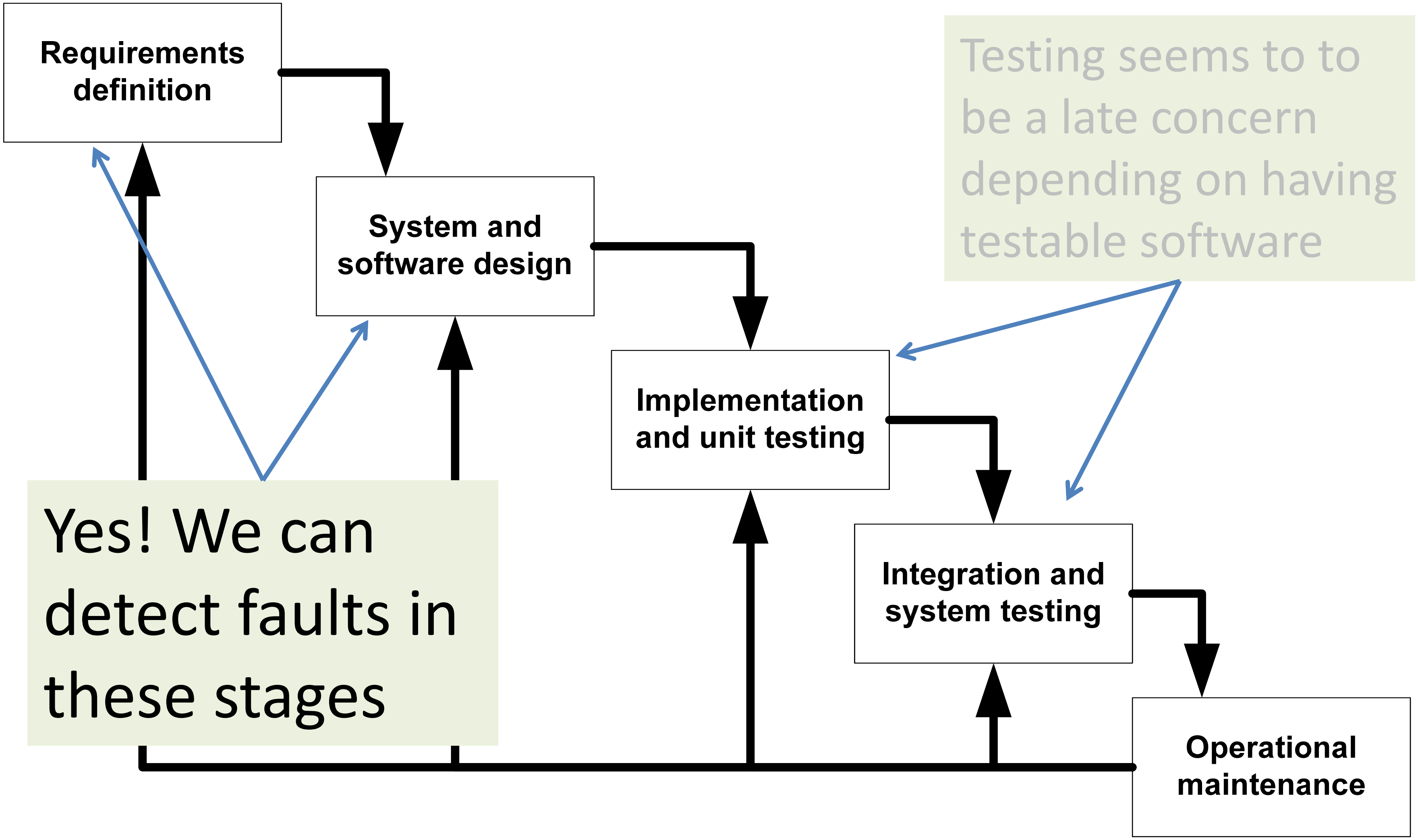
Testing seems to to be a late concern depending on having testable software

Fault remedy economics

- Fixing an error in operational software it likely to be 10-90 times more expensive than fixing it in the design phase.
- Also: 60% error introduced at design and 40% in implementation
- See: Barry Boehm “Software Engineering Economics” 1981 – fig 4.2



Can we detect faults earlier?



Requirements definition

System and software design

Implementation and unit testing

Integration and system testing

Operational maintenance

Testing seems to to be a late concern depending on having testable software

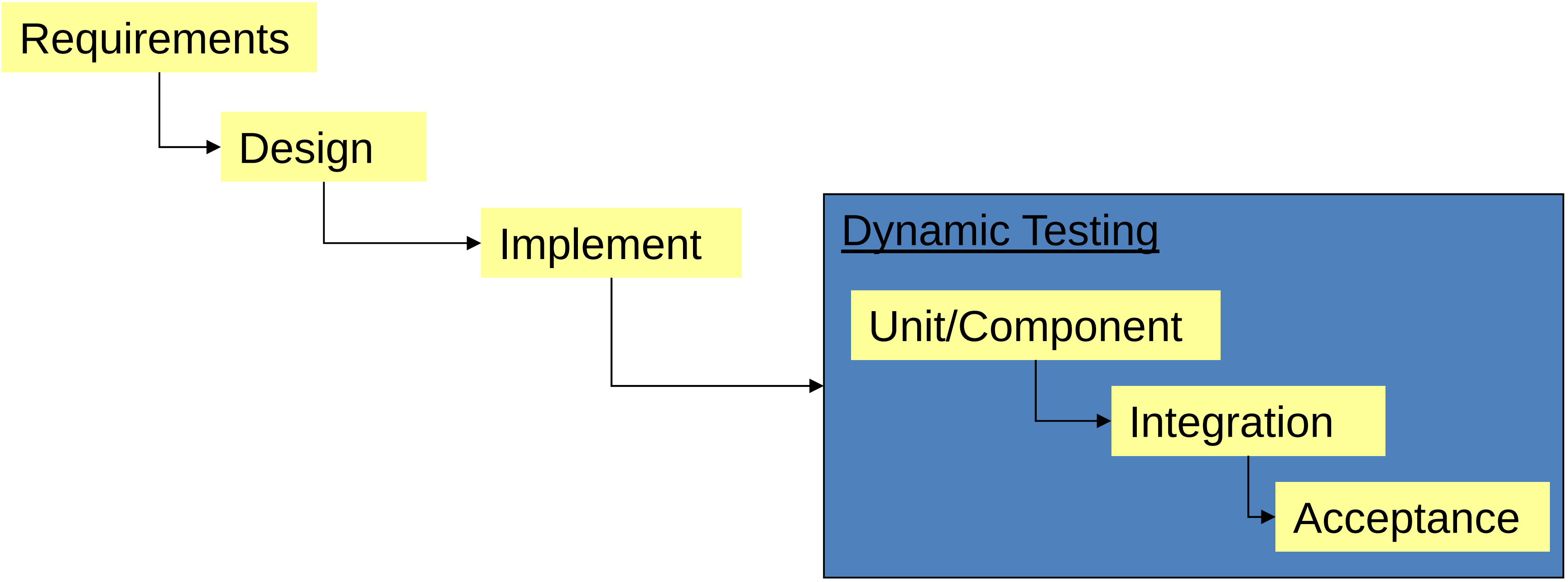
Yes! We can detect faults in these stages

Two kinds of test

- **Static tests:** Detecting faults in software development artefacts other than the running software
  - Specifications
  - Plans
  - Data sets
  - Source code
  - etc
- **Dynamic tests**: Detecting faults by running the software.
  - Unit tests
  - Integration tests
  - Acceptance tests
  - Etc
- Applying static tests ahead of dynamic tests is our tactic for early fault detection

Can we save time in preparing dynamic tests?

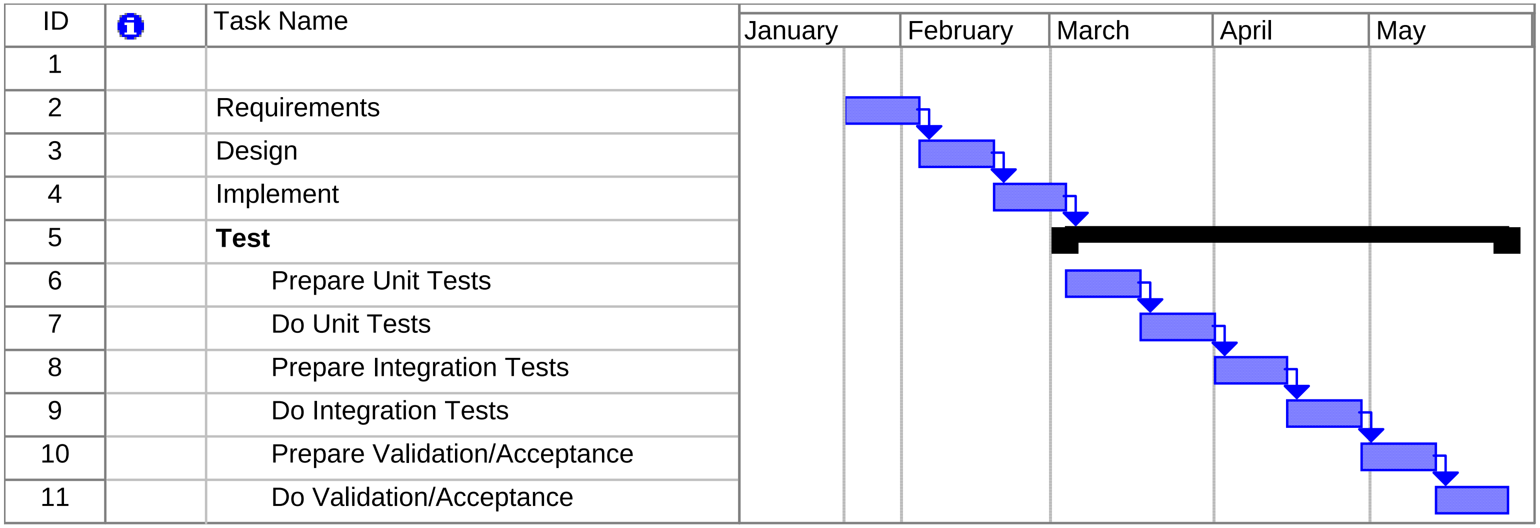
- Must we wait until software is code complete before we prepare dynamic tests? (see next slide for a project perspective)



Is this a good plan?

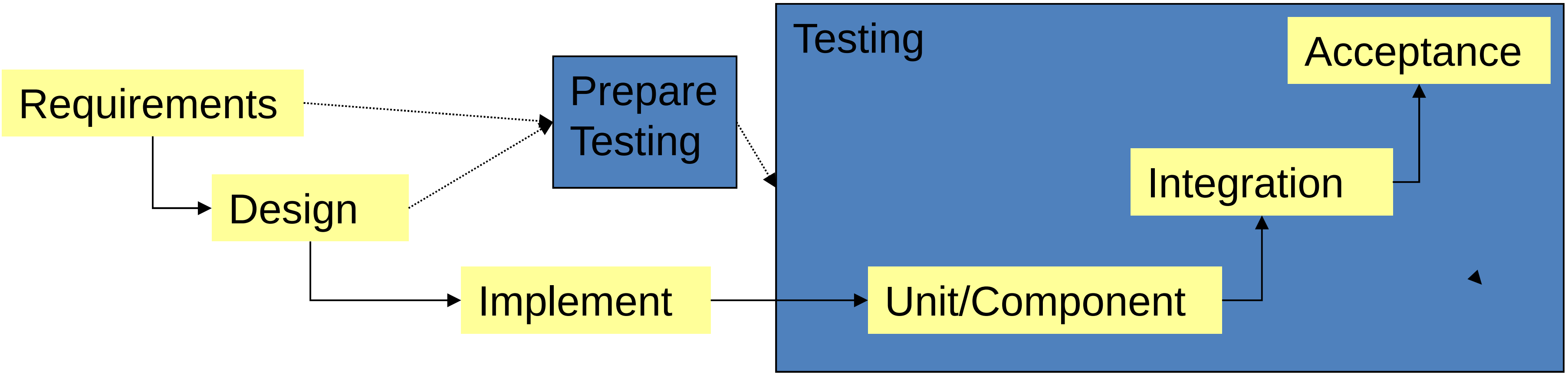
– No because we prepare late and, worse,

– Discover errors late.

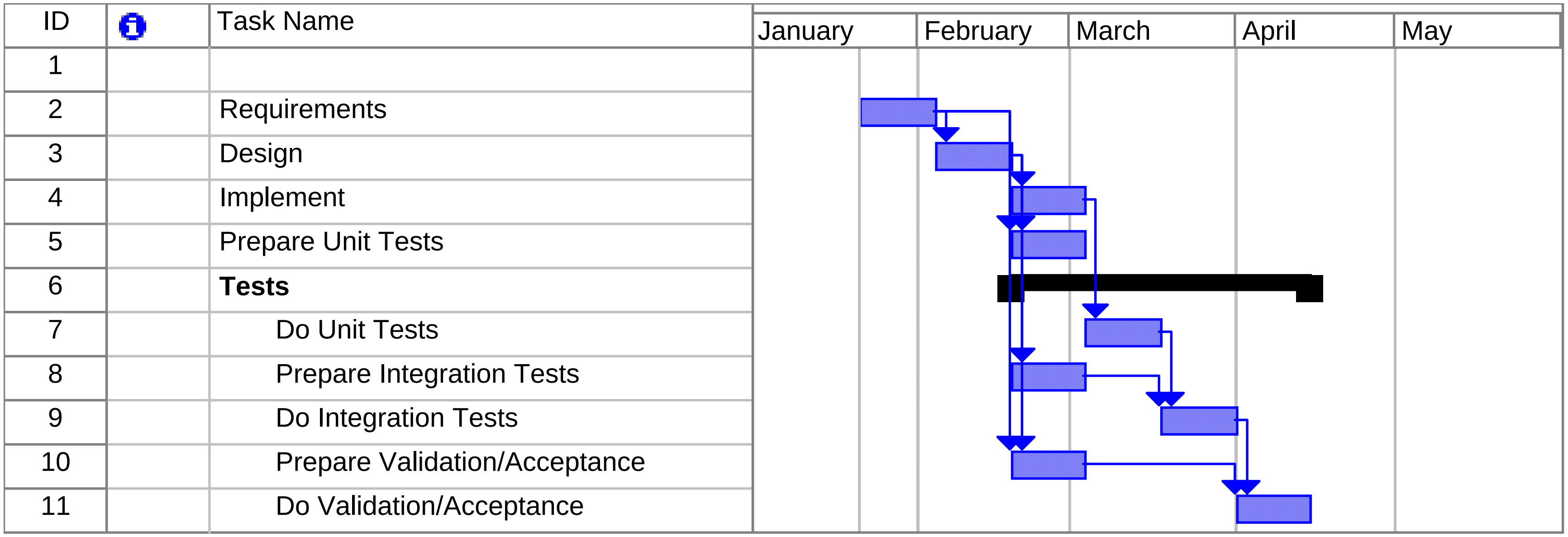


ID	ℹ	Task Name
1		
2		Requirements
3		Design
4		Implement
5		**Test**
6		Prepare Unit Tests
7		Do Unit Tests
8		Prepare Integration Tests
9		Do Integration Tests
10		Prepare Validation/Acceptance
11		Do Validation/Acceptance

A better use of time – prepare early!



A better plan – prepare tests early



ID	ℹ	Task Name
1		
2		Requirements
3		Design
4		Implement
5		Prepare Unit Tests
6		**Tests**
7		Do Unit Tests
8		Prepare Integration Tests
9		Do Integration Tests
10		Prepare Validation/Acceptance
11		Do Validation/Acceptance

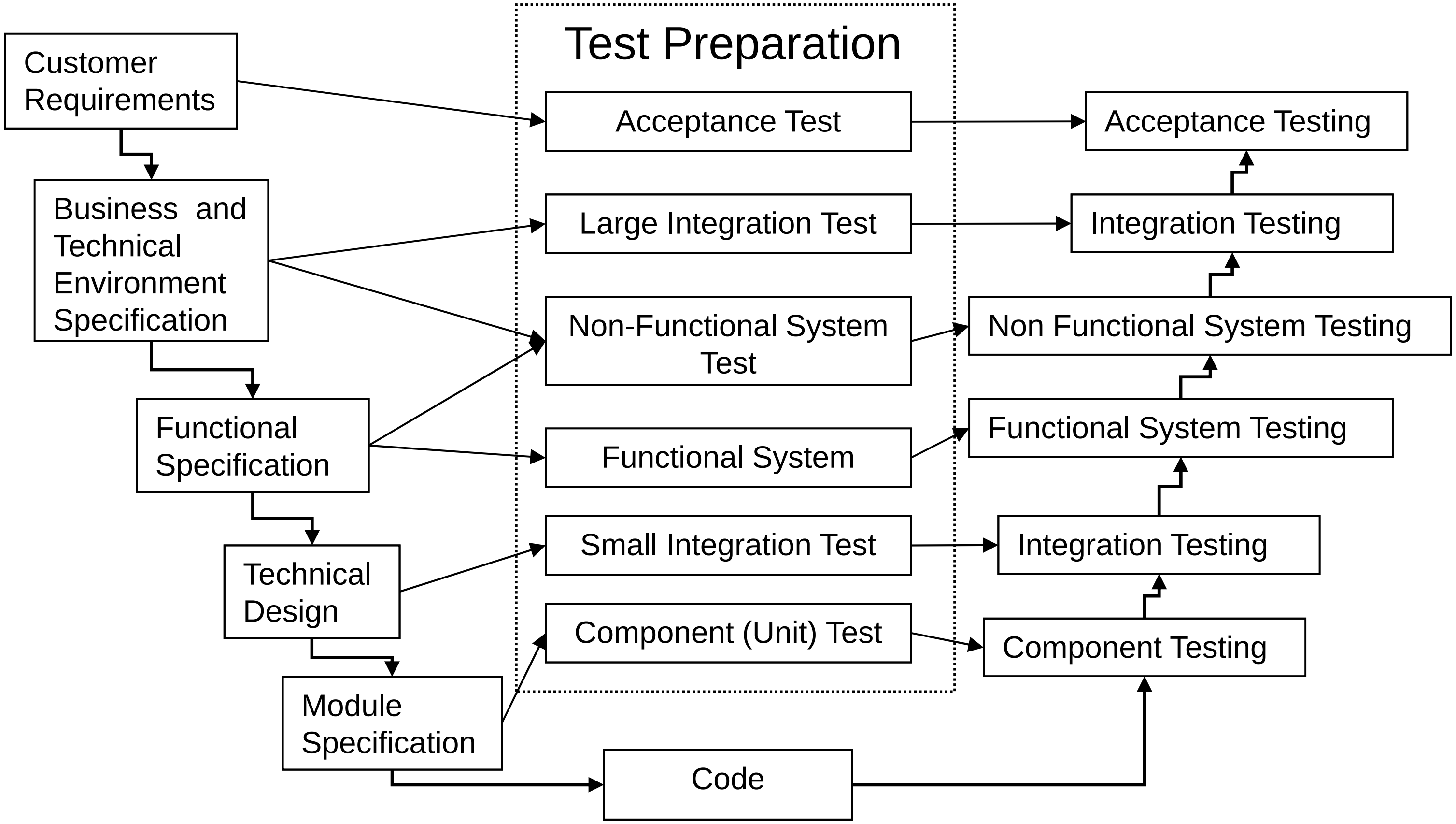
This leads to V-Model…

Bringing these things together

- The need for early detection of faults
- The use of static and dynamic tests
- Early preparation of dynamic tests

- These are all present in the wisdom of the so called V-Model for testing.
  - A classic approach!

V Model - Detailed



Possibilities for Static Testing

- Every box in the V-model can be reviewed and effectively tested for defects
    - Customer requirements
    - Acceptance tests (preparation)
    - Acceptance tests (results)
    - Functional specification
    - Code
    - etc (see diagram of V-model)
- Other things not in the figure can also be submitted for static testing
    - Quality plans
    - Development plan
    - etc

Different stages of dynamic testing

- The V-model indicates different stages of testing
    - Component
    - Integration (small)
    - System testing (F and NF)
    - Integration (large)
    - Acceptance
- Each has a different role and should be done in sequence

Component Testing

- This is sometimes also known as unit testing
- The important thing is that in our development project we can identify the piece of executable software as the
  - “lowest level of independently testable software” See BCS SIGIST Standard for Software Component Testing
- The idea is that we must verify such components before we consider integrating them.
- Verification is performed against the V-Model “Module Specification”

Integration Testing

Small

- Detecting faults that could not be found at an individual component testing level.
    - Joining individual components that have already been tested in isolation.
    - The goal is to test and verify that that a given set of components will function together as specified in their mutual interfaces.
- Verification is performed against the V-Model "Technical Design Specification"

Large

- Detecting find that could not be found at an individual component testing level.
    - Joining individual components that have already been tested in isolation.
    - The goal is to test and verify that that a given set of components will function together as specified in their mutual interfaces.
- Verification is performed against the V-Model "Business and Technical Environment Specification"

System Testing

Functional

- Running the software as a complete ‘system’ of components in order to detect inconsistencies with the required overall functionality.
    - Functional requirements are present or absent
- Verification is performed against the V-Model “Functional Specification”

Non-Functional

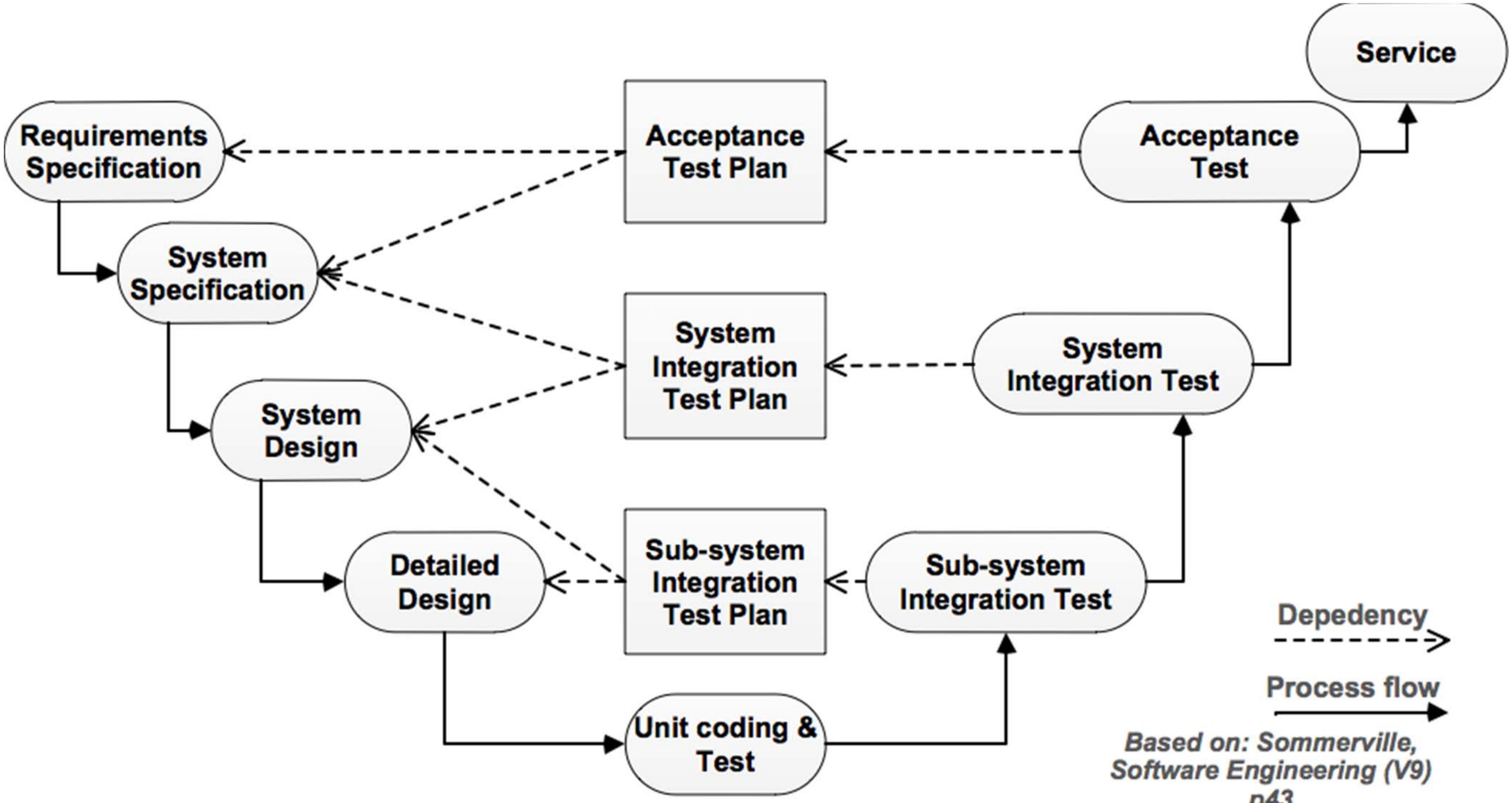
- As functional testing but concentrating on the non functional requirements
    - Non-functional requirements may concern how well a function is executed (performance, precision etc) of how the function has been implemented
- Verification is performed against the V-Model “Functional Specification” AND the “Business and Technical Environment Specification”

System Testing - Acceptance

- Running the software as a complete ‘system’ of components in order to detect inconsistencies with the customer requirements.
- Verification is performed against the V-Model “Customer Requirements Specification”

Simplification

- Some organisations merge some of the of tests to end up with the following list
    - Component
    - Integration
    - System
    - Acceptance
- This is illustrated in the following model



Requirements Specification

System Specification

System Design

Detailed Design

Unit coding & Test

Acceptance Test Plan

System Integration Test Plan

Sub-system Integration Test Plan

Service

Acceptance Test

System Integration Test

Sub-system Integration Test

Depedency

Process flow

Based on: Sommerville, Software Engineering (V9) p43

Other dynamic tests

- Other tests are often named by test groups such as:
    - Regression
    - Smoke
    - Bash
    - Load
- Please look these up – there are many internet references

Regression Tests

- These are tests run after some modification has been made to previously released software.
- The aim is to ensure that no behaviour previously established (functional or non functional) has not unintentionally been lost or damaged.
- These are typically conducted by re-running system tests (functional and non-functional)
- These tests become crucial in iterative-incremental development methods (e.g. Agile).

(b) Test Cases

Dr. Ken Boness

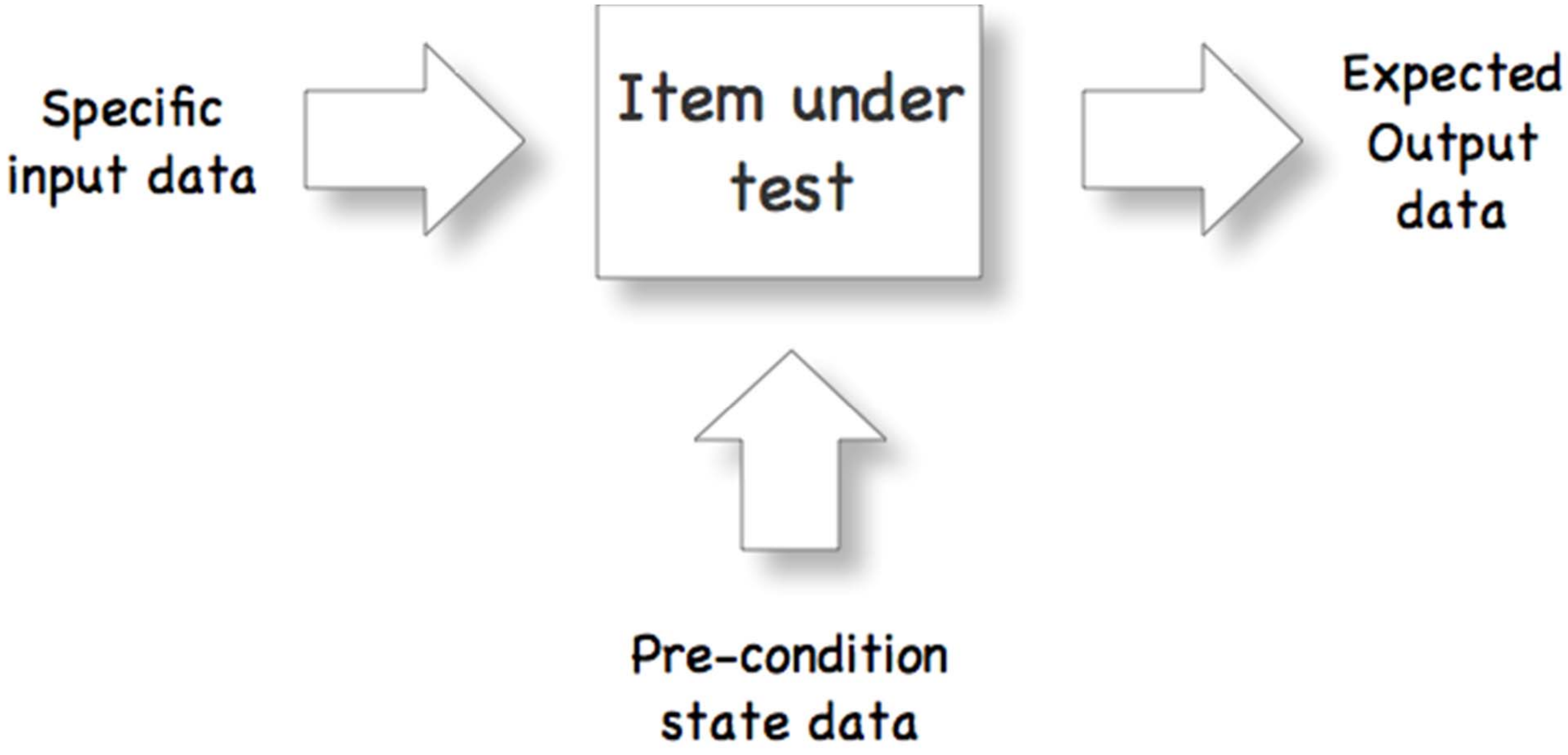
k.d.boness@reading.ac.uk

Test Cases

- Test case = A set of inputs, execution preconditions, and expected outcomes developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

- Test case suite = A collection of one or more test cases for the software under test.

Test Case



Specific input data

Item under test

Expected Output data

Pre-condition state data

Dynamic Testing

- By applying a series of specific **test cases** to a software **item under test** we may show that:-
  - The item is fit for purpose
  - It does what is expected of it – conforms with specification.
  - It does not behave in unexpected ways when used properly.
- As a matter of practicality dynamic testing can not prove the absence of faults!

Example

- Suppose the item under test is a component that satisfies the following specification

Accept a real number X as the input

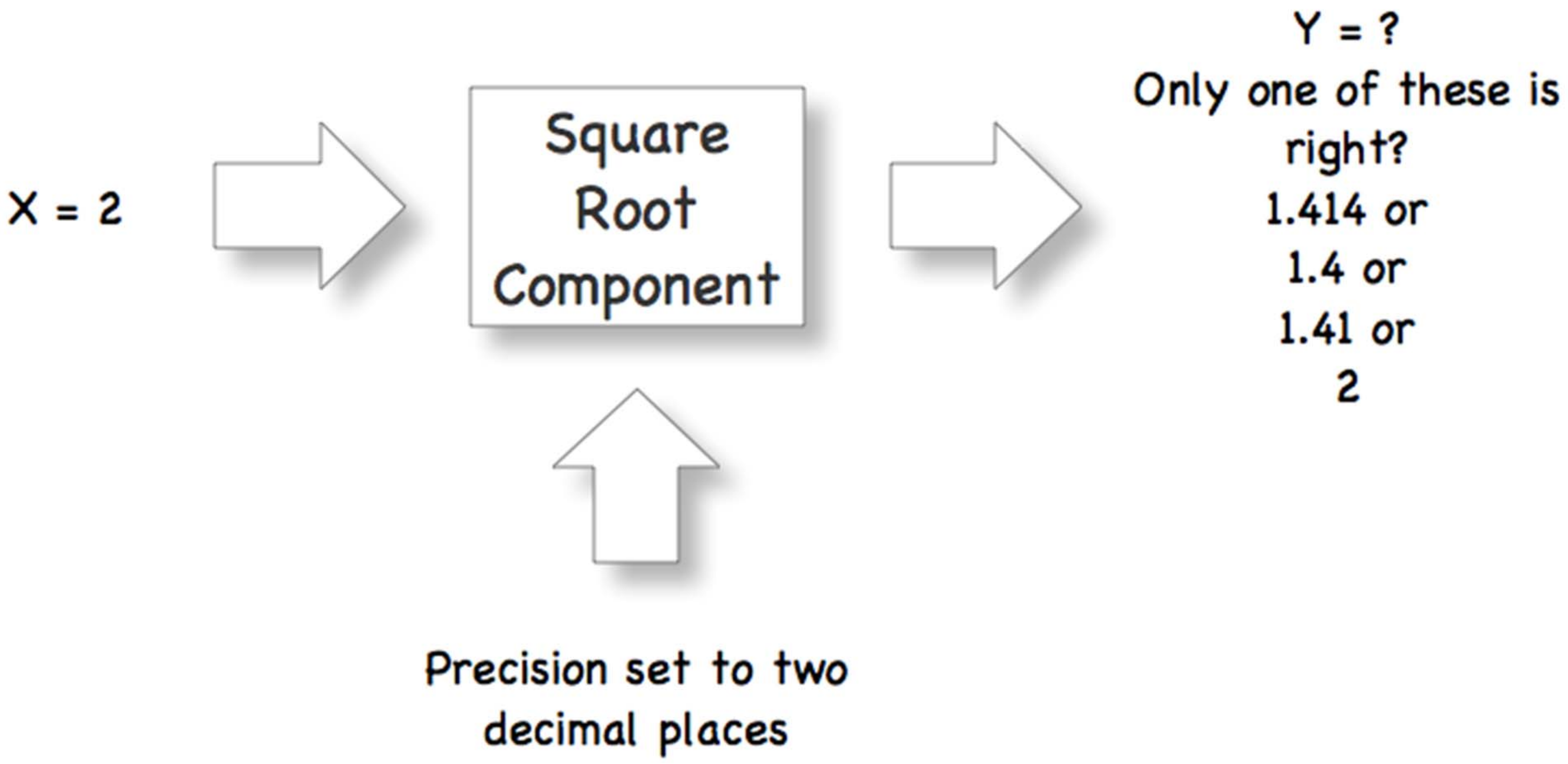
Calculate Y= $X^{1/2}$ to precision P

    Where P is a pre-settable user preference

Return the value of Y.

- We can see that it has an input (X) a predictable output (Y) which depends on the state operation decided at some earlier time by a user (P)

Test Case for the example



X = 2

Square Root Component

Precision set to two decimal places

Y = ?

Only one of these is right?

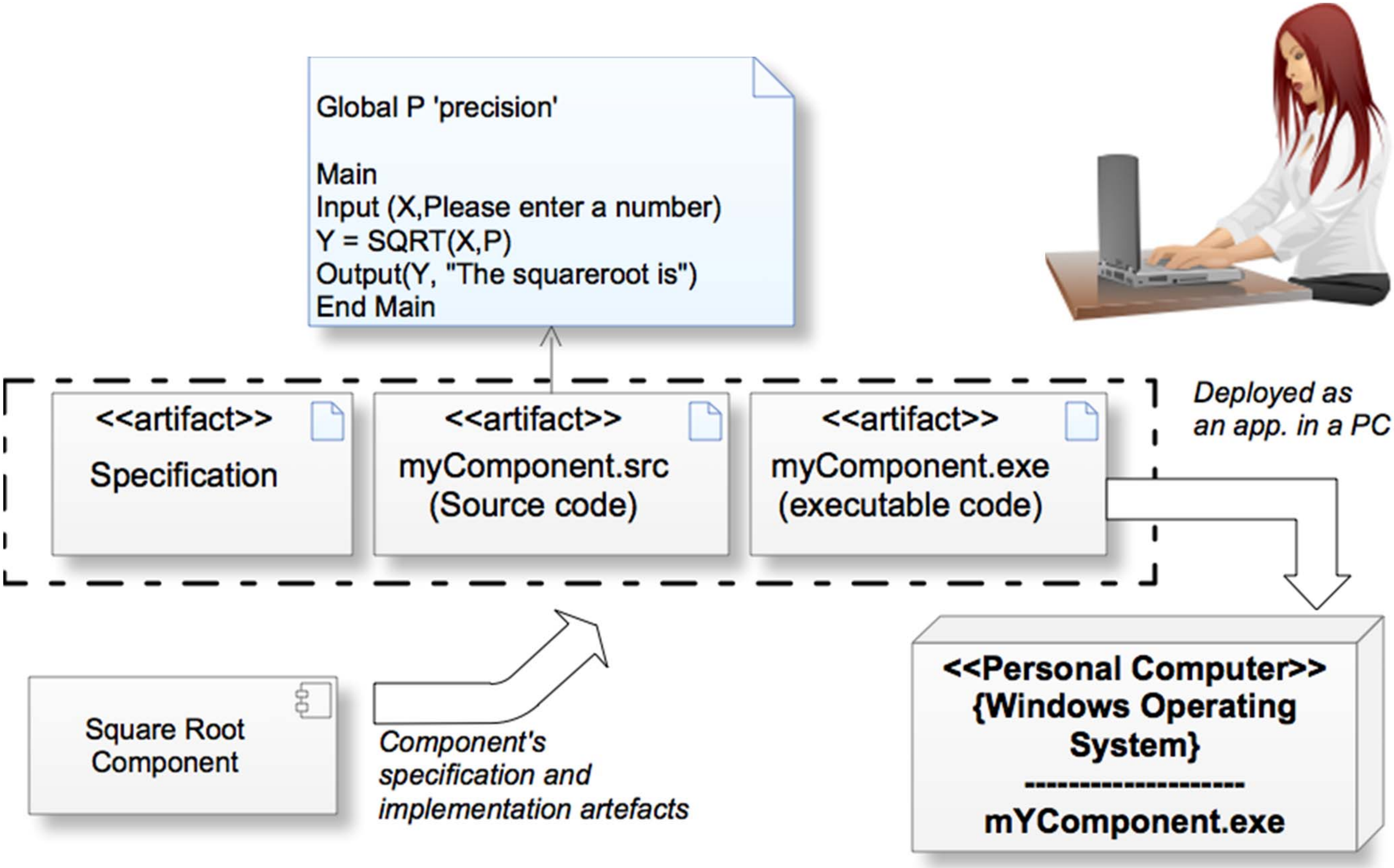
1.414 or

1.4 or

1.41 or

2

Probable architectural artefacts involved in the example



```
Global P 'precision'

Main
Input (X,Please enter a number)
Y = SQRT(X,P)
Output(Y, "The squareroot is")
End Main
```

<<artifact>>
Specification

<<artifact>>
myComponent.src
(Source code)

<<artifact>>
myComponent.exe
(executable code)

Deployed as an app. in a PC

Square Root Component

Component's specification and implementation artefacts

<<Personal Computer>>
{Windows Operating System}
----
mYComponent.exe

Manual Testing

Tester (after setting the app running AND having set precision to 2)

Enter X=4, Expect Y=2.00, See Y=2.00 on screen
Compoenent has passed the test

Enter X=2, Expect Y=1.41, See Y=1.42 on screen
Component has failed the test

The two test cases used

Test ID	Description	State	Input	Expected Output
TC1	Test easy SQRT	P=2	X=4	Y=2.00

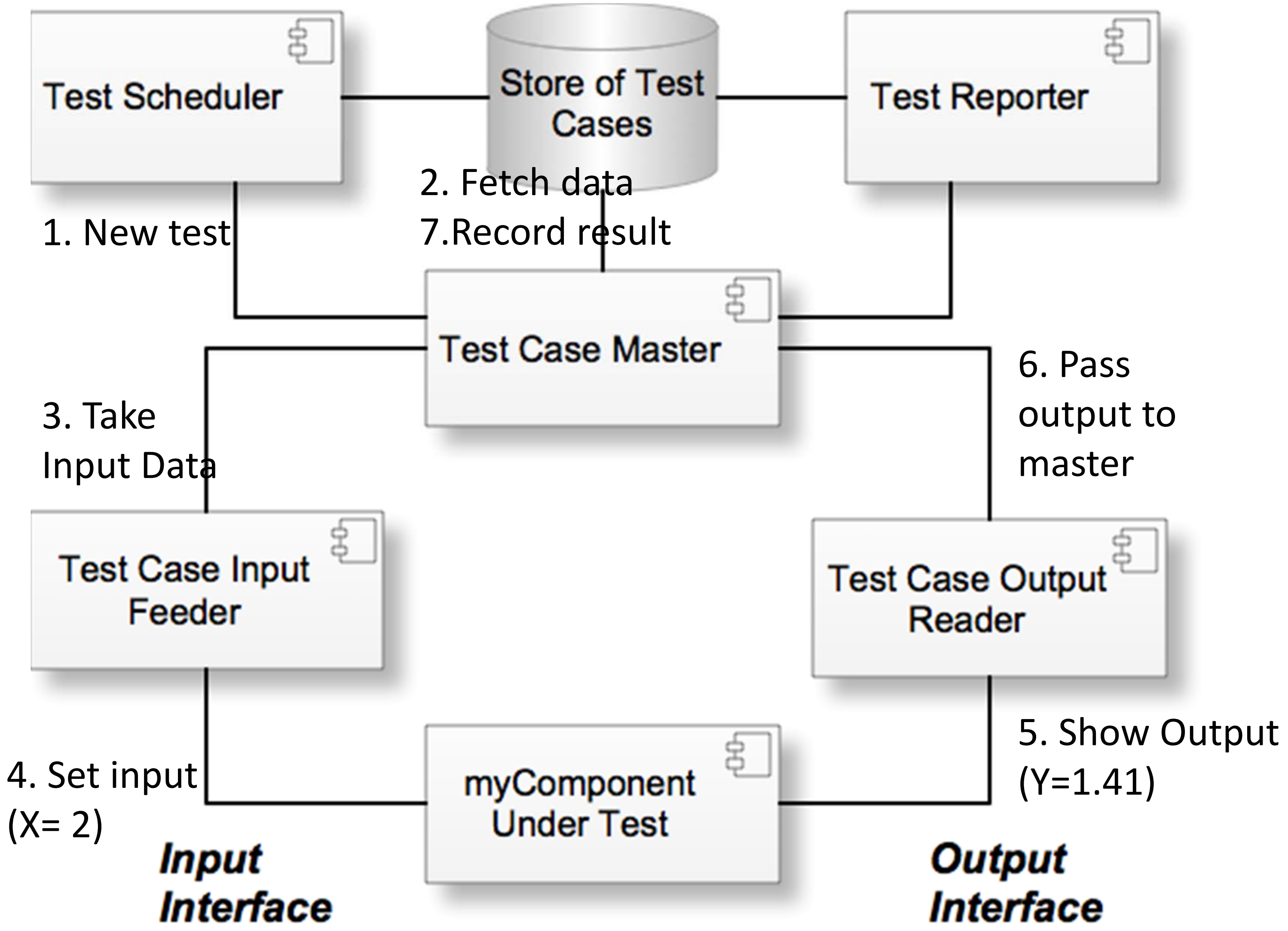
Test ID	Description	State	Input	Expected Output
TC2	Test irrational SQRT	P=2	X=2	Y=1.41

Automating the test?

- Should we simply rely on a tester typing the test case inputs and reading the results on the screen?
    - Sometimes!
- For large applications we need to do better

Fragment of a possible test automation system

(Sketched as a collaboration diagram)



Notes on the test automation system

- The test cases and test results are stored on a database
- There is set of components to control test schedule, execution and reporting
- The test item must be inserted into this system
- Three BIG Issues
  - How do the data feeder and output reader actually interface with the component under test?
  - Nothing has been said about deployment
  - Nothing has been said about version control for the test

Issue: How does the test kit connect to the component’s interfaces?

- For a windows app such as we have been discussing testers can use products such as ‘WinRunner’ to feed data in and get data out
- An alternative tactic is to ask the developers to include some alternative I/O to the component (eg API’s or using data files)
  - But still need to test the GUI !
- Could construct a special test harness with privileged connection to the component
  - (rather like the first idea above)

Issue - Deployment

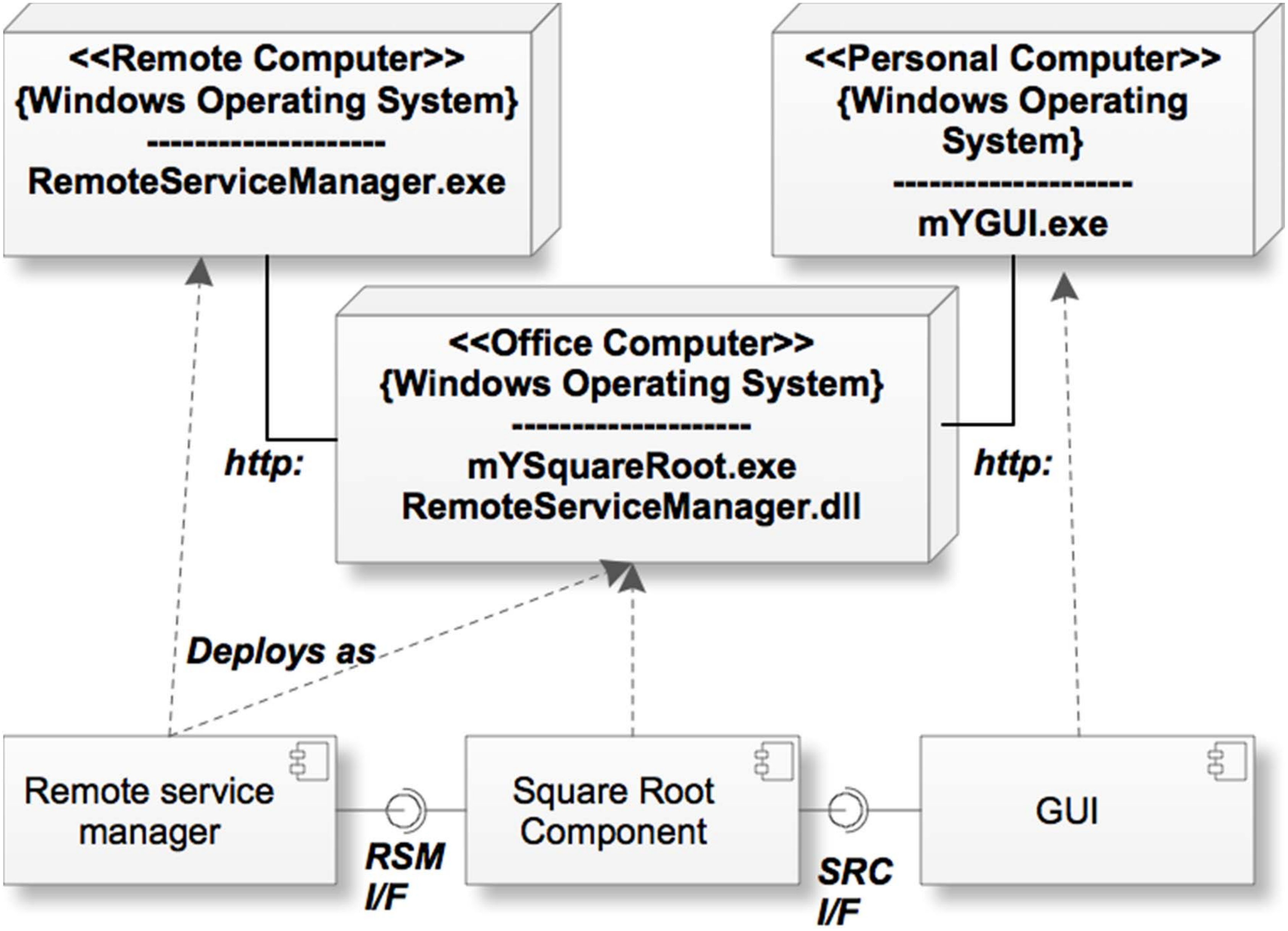
- We must specify what hardware and operating systems are needed.
- It may all be on one computer
- Or there may be a complex arrangement of networked computing
- It may be necessary to deploy into different operating systems (depending on what your product is required to work on)!

Issue – Version Control

- We must be able to test the component in any of the previous versions that may be warranted to customers or needed for other quality management purposes
  - Consider
    - Sources
    - Libraries
    - Test data
    - Compilations/builds
    - Operating environment (type and version)
    - etc.

Suppose the item to be tested has a more complex environment?

- Not just a single personal computer
- A remote server, an office computer and a personal computer
- As well as a full test kit we might envisage a couple of helpful ‘test harnesses’ or ‘stubs’.



<<Remote Computer>>
{Windows Operating System}

RemoteServiceManager.exe

<<Personal Computer>>
{Windows Operating System}
----
mYGUI.exe

<<Office Computer>>
{Windows Operating System}
----
mYSquareRoot.exe
RemoteServiceManager.dll

http:

http:

Deploys as

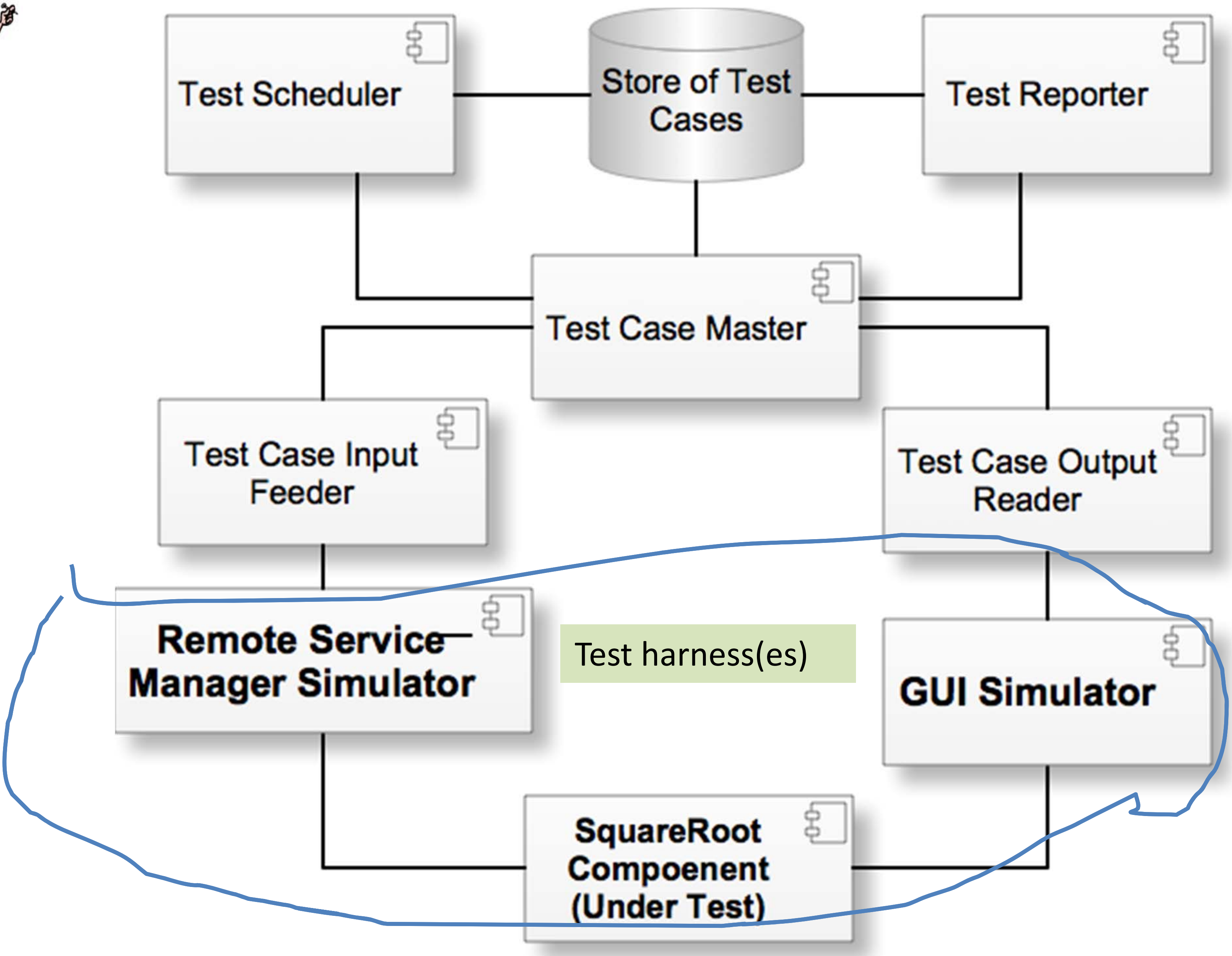
Remote service manager

RSM I/F

Square Root Component

SRC I/F

GUI



Test harness(es)

A note on quality planning

- Reflecting on this lecture reminds us that there are many artefacts to be brought to bear in the right versions and at the right time
- Traceability and reproducibility, usually required in a quality system, demand that we have provision for the accurate reconstruction of previously conducted tests
- We need to design test cases, we may need to procure hardware and software for test infrastructure.
  - Particular attention should be given to special test harnesses and special data that might be needed in the test cases

Preparation for Next Week

- There will be lots of test cases – how do we design them?
- Look up component testing in the Glossary.
- Familiarise yourself with the Component Test Guide (Standard for Component Testing - BCS SIGIST on Blackboard).
  - Section 1 gives scope and context. Section 1.5 points out that the document is not prescriptive of choice of appropriate testing but defines the tests that may be used in component testing.
  - Section 1.6 indicates that conformity will require following the process of section 2.

- We will be working on this for a few weeks.

SE3Q11: Week 5

Designing Test Cases

(BCS Component Testing)

Dr. Ken Boness

k.d.boness@reading.ac.uk

Types of Dynamic Test

- Black Box = Testing that is based on an analysis of the specification of the component without reference to its internal workings.

- White Box = Testing that is based on an analysis of the specification of the component with reference to its internal workings

Contrasting Black and White Box Testing

- Black box is relevant throughout the life cycle whereas, in general, additional white box is appropriate for sub-system testing (unit, link) but becomes progressively less useful towards system and acceptance testing.

- System and acceptance testers will tend to focus more on specifications and requirements than on code.

(a) Black Box Test Cases

Dr. Ken Boness

k.d.boness@reading.ac.uk

Lots of Tests

- The use of systematic techniques (and corresponding measures) are essential to provide confidence.
- On a complex system this may lead to many test cases.
- Tools can increase productivity and quality and are particularly useful for white box testing.

Tactics for defining test cases

- The methods described in BCS Standard on Component testing (e.g. equivalence partitioning and boundary value analysis)
  - These are specification based and we will try out in the following slides
    - – Typical specifications enabling this approach are often based on IEEE std 830-1998
- Other tactics we will mention in a later lecture include:
  - Taking *scenarios* out of use cases (e.g. Cockburn, sea level, fully dressed)
  - Reverse engineering of “warranted behaviour”

A Taxing Example

(An extract from a Specification artefact)

- A simple application that calculates a special tax on goods of type A or B depending on their value.
- A single GUI form is provided with three controls:
  - Type picklist (A or B).
  - Value entry text box; assumed to be in euros.
  - Calculated tax display text box (no user input).
- The tax shall be 25% of the value if the type is A and the value exceeds 12 Euro and is less than or equal to 80 Euro.
- The tax shall be 50% of the value if the type is A and the value exceeds 80 Euro.
- The tax shall 25% if the type is B and the value exceeds 60 Euro.
- On this system the user has a keyboard allowing the user to type numbers, upper and lower case letters and four other characters. A mouse allows focus and selection options.

Possible test cases

Type	Value (Euros)	Description	Result (Euros)
B	80	Normal tax	20
B	10	Normal Tax	0

Study the example

- List more possible test cases.
- How many test cases are needed to test this from the point of view of user input?

Study the example

- Is your list of test cases efficient and effective?
- How many test cases do you need.

- The following slides show two approaches described in the BCS SIGIST Standard for Component Testing.
    - Section §3.1 – Equivalence partitioning
    - Section §3.2 – Boundary value analysis

  Please study these sections of the standard

Equivalence Partitioning

- It is inefficient to write test cases for every possible value of the data.
- Equivalence partitioning is a formalisation of an approach that is used intuitively by many testers. The idea is that:-
  - Exhaustive testing is impractical to carry out due to time and resource constraints.
  - Software tends to generalise the way it deals with subsets of data; i.e. for a range of values for a given variable the software should behave the same (eg If X>100 then).
  - Hence it makes sense to identify these ranges, called partitions, for all of the control variables and then define test cases based on these ranges.

Boundary value analysis

- Boundary value analysis is based on the idea that:
  - The inputs and outputs of a component can be partitioned into classes that, according to the component's specification, will be treated similarly by the component and
  - that developers are prone to making errors in their treatment of the boundaries of these classes.
  - Therefore it is prudent to identify such boundaries and associated test cases.

Tutorial Exercises

- Work through the equivalence partitioning analysis of the taxing example and list a set of test cases achieving at least 50% coverage.
- Work through the boundary value analysis of the taxing example and list a set of test cases achieving at least 50% coverage.

(b) White Box Test Cases

Dr. Ken Boness

k.d.boness@reading.ac.uk

Objectives

- Look at White Box Methods for Dynamic Testing
- Start investigating white box testing as described in the “Standard for Component Testing” - BCS SIGIST
- Move on to look at the “taxing example” as a case study for testing:-
  - Executable statements
  - Producing a control flow graph for:-
    - Branch coverage
    - Decision coverage

Objectives

- “Standard for Component Testing” - BCS SIGIST

- This a tutorial/practical session to look at:-
- Statement Testing
- Executable source code
- Coverage and %Coverage
- Control Graphs
- Blocks in control graphs
- Branch Coverage
- Decision Coverage

Work through BCS SIGIST ref

- Statement Testing: section 3.6 (starts p10)* B6(starts p44)

- Branch/decision testing: section 3.9 (starts p11)* B7 (starts p45)

- State Transition Testing: section 3.3 (starts p9)* B3 (starts p26)

c.f. the “taxing example”

- Look at the “taxing example” in the lecture on black box testing.
- The next slide contains a possible program code implementation we will use it to explain:-
  - Executable statements
  - Control flow graph
  - Branch coverage
  - Decision coverage

- Look more into White Box Methods for Dynamic Testing in the “Standard for Component Testing” - BCS SIGIST

c.f. A Taxing Example

```
1     '''''''''''''''''''''''''''''
2     'For CS3TX4 Module
3     'KDB 1-02-05, Reading University
4     '''''''''''''''''''''''''''''
5     Function CalcTax(ByVal myType, myValue) As String
6     On Error GoTo Error_TaxCalc
7         Dim TaxPercent
8         If myType = "A" Then'Deal with the A Types here
9             If myValue > 12 And myValue < 80 Then
10                TaxPercent = 25
11            ElseIf myValue > 80 Then :   TaxPercent = 50
12            Else :      TaxPercent = 0
13            End If
14            CalcTax = TaxPercent
15        ElseIf myType = "B" Then
16            If myValue > 60 Then:      TaxPercent = 60
17            Else :TaxPercent = 0
18            End If
19            CalcTax = TaxPercent
20        Else
21            CalcTax = "Type not recognised"
22        End If
23    Exit Function
24    Error_TaxCalc:
25        CalcTax = "Error in Calc Tax: " & Err.Description
26    End Function
```

Note: The line numbers
Are not part of the source. They are added solely for reference.

c.f. A Taxing Example

The executable statements excluding function framework are below:-

S1	**6,24**
S2	**8,*15,20,22***
S3	**9,*11,12,13***
S4	**10**
S5	**11 (b)**
S6	**12 (b)**
S7	**14**
S8	**16,*17,18***
S9	**16b**
S10	**17**
S11	**19**
S12	**21**
S13	**23**
S14	**25**

```
1    ''''''''''''''''''''''''''''''
2    'For CS3TX4 Module
3    'KDB 1-02-05, Reading University
4    ''''''''''''''''''''''''''''''
5    Function CalcTax(ByVal myType, myValue) As String
6    On Error GoTo Error_TaxCalc
7        Dim TaxPercent
8        If myType = "A" Then 'Deal with the A Types here
9            If myValue > 12 And myValue < 80 Then
10               TaxPercent = 25
11           ElseIf myValue > 80 Then :  TaxPercent = 50
12           Else :      TaxPercent = 0
13           End If
14           CalcTax = TaxPercent
15       ElseIf myType = "B" Then
16           If myValue > 60 Then:      TaxPercent = 60
17           Else :TaxPercent = 0
18           End If
19           CalcTax = TaxPercent
20       Else
21           CalcTax = "Type not recognised"
22       End If
23   Exit Function
24   Error_TaxCalc:
25       CalcTax = "Error in Calc Tax: " & Err.Description
26   End Function
```

c.f. A Taxing Example

The italics mean “structural” parts of a multi part statement such as “If”.
The label “b” indicates the second statement on the physical line.

This is my own notation.

S2	**8,*15*,*20*,*22***
S3	**9,*11*,*12*,*13***
S5	**11 (b)**
S6	**12 (b)**
S8	**16,*17*,*18***
S9	**16b**

```
1     ''''''''''''''''''''''''''''
2     'For CS3TX4 Module
3     'KDB 1-02-05, Reading University
4     ''''''''''''''''''''''''''''
5     Function CalcTax(ByVal myType, myValue) As String
6     On Error GoTo Error_TaxCalc
7        Dim TaxPercent
8        If myType = "A" Then'Deal with the A Types here
9           If myValue > 12 And myValue < 80 Then
10             TaxPercent = 25
11          ElseIf myValue > 80 Then :  TaxPercent = 50
12          Else :     TaxPercent = 0
13          End If
14          CalcTax = TaxPercent
15       ElseIf myType = "B" Then
16          If myValue > 60 Then:     TaxPercent = 60
17          Else :TaxPercent = 0
18          End If
19          CalcTax = TaxPercent
20       Else
21          CalcTax = "Type not recognised"
22       End If
23    Exit Function
24    Error_TaxCalc:
25       CalcTax = "Error in Calc Tax: " & Err.Description
26    End Function
```

% Statement Coverage
Suppose we have test cases

Test case	**Inputs**			**Expected Outcome**
	Type		**Value**	
TC1	**A**		**50**	**25%**
TC2	**A**		**10**	**0%**

These exercise statements : S2, S3, S4, S6

ie 4 out of 14 statements

%Statement coverage = 4/14 * 100% = 28%

Control Flow Graph

(Step 1 Identify Blocks)

- We identify blocks as sequences of code such that:-
  - They make a complete sequence of instructions with no branches in (except the beginning) and no branches out (except the end).
  - They are guaranteed to be executed in total or not at all; depending on whether we enter the block or not.

Marking of Blocks

Blocks Marked as Bn

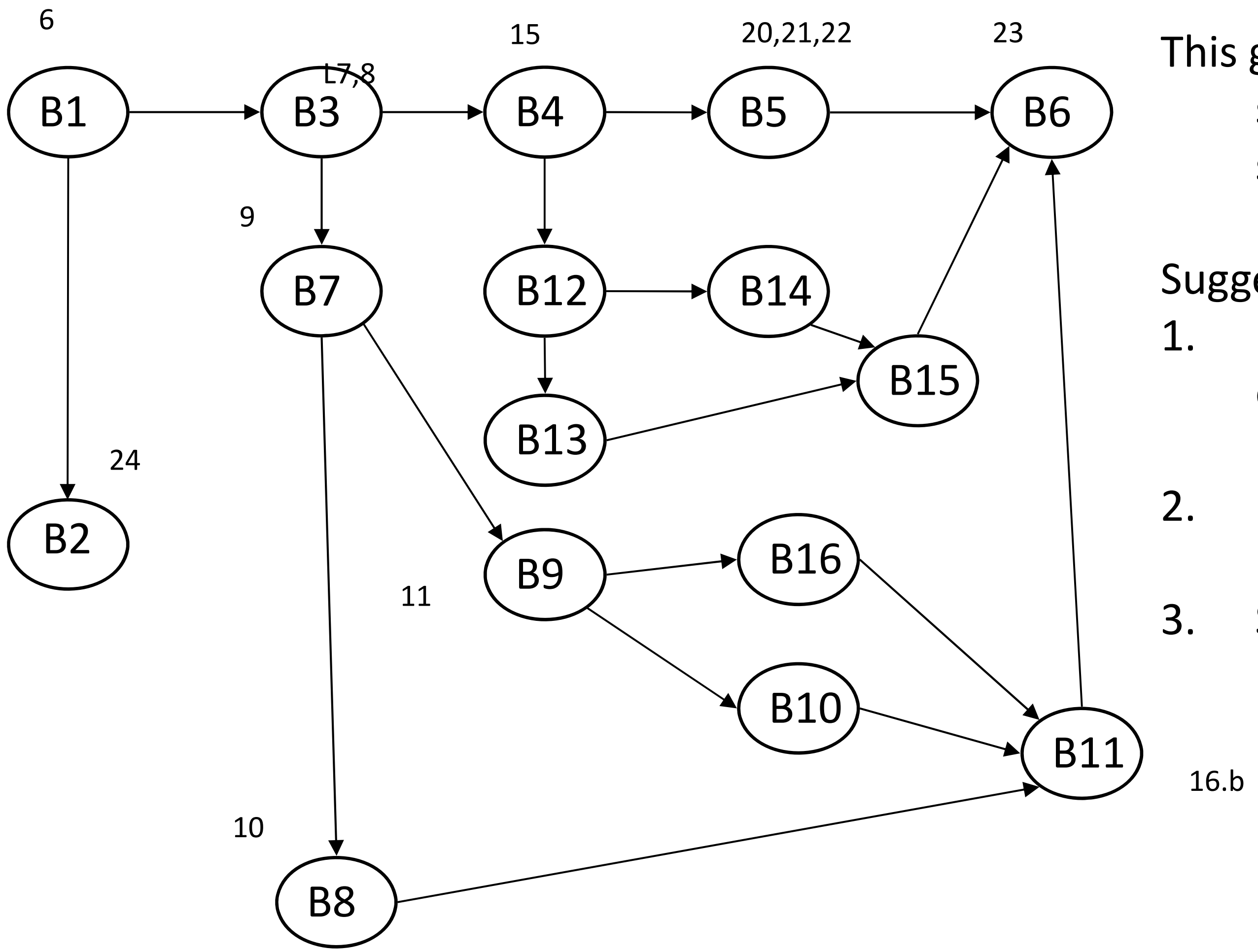
Check this and See whether you agree.

```
5          Function CalcTax(ByVal myType, myValue) As String
6    B1    On Error GoTo Error_TaxCalc
7    B3       Dim TaxPercent
8             If myType = "A" Then'Deal with the A Types here
9    B7          If myValue > 12 And myValue < 80 Then
10   B8             TaxPercent = 25
11   B9          ElseIf myValue > 80 Then
11b  B16                                   TaxPercent = 50
12   B10         Else :
12b                                         TaxPercent = 0
13               End If
14   B11         CalcTax = TaxPercent
15   B4       ElseIf myType = "B" Then
16   B12         If myValue > 60 Then:
16b  B13                                    TaxPercent = 60
17   B14         Else :
17b                                         TaxPercent = 0
18               End If
19   B15         CalcTax = TaxPercent
20   B5       Else
21               CalcTax = "Type not recognised"
22            End If
23   B6    Exit Function
24   B2    Error_TaxCalc:
25            CalcTax = "Error in Calc Tax: " & Err.Description
26         End Function
```

Control Flow Graph (Step 2 render the blocks)

- We construct a kind of flow diagram from the identified blocks
- Take the first block and identify which block or blocks flow from it.
    - These are joined by directed arrows.
- The result is the desired Control Flow Graph
- It represents branches and decisions which we should consider testing

Resulting Control Graph



This graph based on the block shown on the previous slide of taxing example.

Suggestions:

1. put line numbers above each node (as shown for B3).
2. Recognise decisions as blocks.
3. See next slide

Branches and Decisions

- On a control graph
  - – Decisions are given by basic blocks that have more than one exit arrow.
    - Count them (6 are shown) B1,B3,B4,B7,B9,B12
  - – The number of branches is given by the number of arrows.
    - Count them (18 are shown)

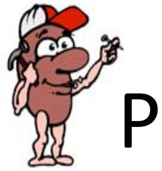
Test Case Analysis

Test case	Inputs			Blocks Exercised	Branches Exercised	Decisions Exercised	Statements Exercised	Expected Outcome
	Type		Value					
TC1	A		50	B1>B3>B7>B8 >B15>B6	#Blocks -1=5	B1,B3,B7	S2,S4	25%
TC2	A		10				S6b	0%

1. c.f. (slide 6) 3 statements exercised out of 14 statements; therefore %Statement coverage = 4/14 * 100% = 28% .
2. Inspect slide 9. There are d decisions out of D exercised; therefore % Decision coverage = d/D x 100%.

   (Hint: TC1 achieves 3/6 = 50%)
3. Inspect slide 9. There are b branches out of B exercised; therefore % Branch coverage = b/B x 100%.

   (Hint: TC1 achieves 5/18 = 27%)



Practice exercise in white box statement, branch and coverage testing

- Create a set of test cases to expand the set on the previous slide to assure 100% coverage each of decisions, branches and statements.

- Discuss the results

SE3Q11: Week 6

Study Week – No Lectures

SE3SQ11: Week 7
Practical 3
Working with test cases and automation

(a) Exploring automatic code generation

(b) Exploring Test-first development

Dr. Pat Parslow

k.d.boness@reading.ac.uk

SE3SQ11: Week 8
Practical 4
Working with test cases and automation

(a) Evaluating tests

(b) Reflection and General Discussion

Dr. Pat Parslow

k.d.boness@reading.ac.uk

University of Reading

SE3Q11: Week 9

High level approaches to designing dynamic tests

Dr. Ken Boness

k.d.boness@reading.ac.uk

University of Reading

(a) High level approaches to designing dynamic tests

Dr. Ken Boness

k.d.boness@reading.ac.uk

Tactics for defining black box test cases

- We have looked at the methods described in BCS Standard on Component testing (e.g. equivalence partitioning and boundary value analysis)
- This lecture looks at two more tactics:
  - Taking *scenarios* out of use cases (e.g. Cockburn, sea level, fully dressed)
  - Reverse engineering of “warranted” behaviour

Use Cases

The usual structure of a use case includes:-

- Pre-conditions
  - These define the state of the system (running software) that must pertain before the test case behaviour may be required
- Narrative
  - Goal-oriented steps defining input/output transactions between actors and the system (running software)
- Post-conditions
  - Defining the state of the system (running software) following the completion of the narrative steps

These define the envelope of many possible scenarios

Use Case Example

Use Case UC1: Calculate square root

Primary Actor: User

Scope: mYComponent, black box

Level: sea

Pre-Conditions

1. The precision of calculations (P) has been set

Post-conditions

1. Nothing stored
2. P is unchanged
3. Result is on display to user

Narrative

1. User enters value (X)
2. Component calculates square root (Y) of value (X) to precision (P)
3. Component displays (Y)

Scenario as test case

- Is a particular passage through a use case
- Where the use case is defined with general steps the scenario enters specific values
- For example
  - Assuming that the precondition has been satisfied with P=2
  - The narrative becomes
    - Step 1: User enters value 4
    - Step 2: Component calculates square root as 2.00
    - Step 3: Component displays 2.00
- We have the ingredients of a test case
  - Specific state, inputs and outputs

Use case and scenario

- Follow the use case
- Define the state of each element of the pre-conditions
- Follow the steps of the narrative in turn
    - At each step
        - enter a specific value for each input
        - Check the expected output
- When the narrative is complete check the state of each element of the post conditions
- You have now stepped through a particular scenario
    - Bearing in mind equivalence partitioning and boundary value testing you might need to repeat the above for different scenarios

Scenario as a set of test cases

- The example we used led to one test case
- It had a very simple narrative and there were no extensions (correcting deviations from the narrative flow)
- It is more usual for a scenario to lead to a a set of test cases that must be used in a particular sequence
    - Consider the next example: How many test cases would we need?

Use Case Example

Use Case UC1: Calculate square root

Primary Actor: User

Scope: Component, black box: Level: fish

Pre-Conditions

1. The precision of calculations (P) has been set

Post-conditions

1. P is updated
2. Result is on display to user

Narrative

1. User enters value (X)
2. Component calculates square root (Y) of value (X) to precision (P)
3. Component displays (Y) and invites the user to change precision (P)
4. User optionally enters a new value for P and component returns operation to step 2

We would need a test sequence including steps with different test cases - something like

Test Step	Use case steps involved	Test Case	State	Input	Expected Output
1. Set up preconditions with P=2 using procedure defined in ref: XYZ.					
2. Run Test Case	Narrative steps 1,2,3	TC1: First Calc	P=2 No display	X=2	SQRT of 2 is 1.41 Do you wish to change precision?
3. Run Test Case	Narrative steps 4,2,3	TC2: Change precision	P=2 Display shows last result	P=3	SQRT of 2 is 1.414 Do you wish to change precision?
4. Run Test Case	Narrative step 4 Post Conditions 1,2	TC3:Skip new value	P=3 Display shows last result	Skip (Means TBD)	SQRT of 2 is 1.414

Reverse Engineering

- Sometimes there is no *a priori* specification
- This can happen for many reasons
- Often the case in *backlog driven* developments (e.g. agile approaches)
- When you ask: “What percentage of expected user experience and stakeholder interests do you test”
    - You get the reply: “we are not sure”
- We must fix this, and can usually fix it by reverse engineering from the “warranted” behaviour

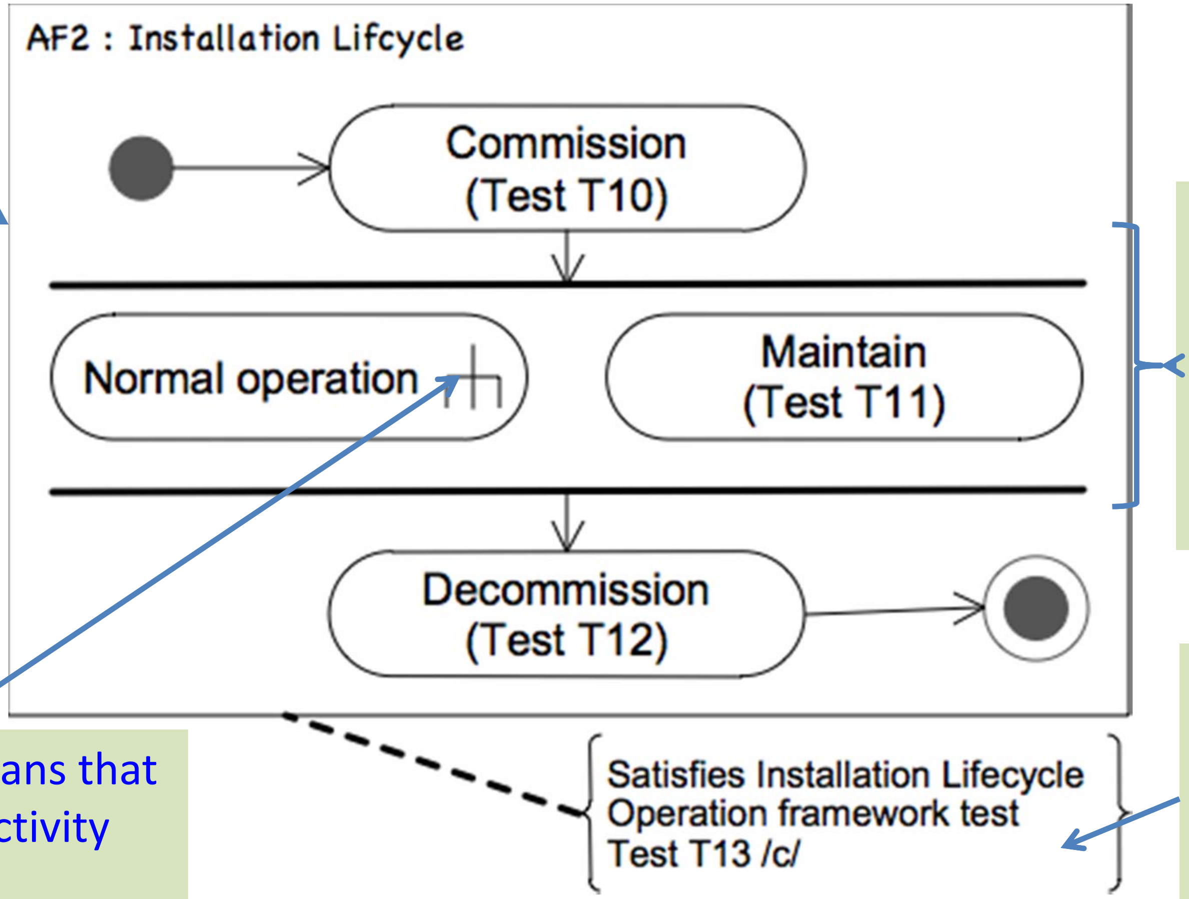
Reverse engineering from the "warranted" behaviour

- We must be careful about the term warranted
- Here it is used in the sense of the things the customers would reasonably be expecting rather than a statement couched in legal concerns
- This approach is not in the text books but it is one that I have applied in a series of small and medium sized software product developments

The approach

- Simplified UML Activity Diagrams are used in a format we will refer to as Activity Frames
- Activity frames are hierarchical
    - An activity can be expanded into more detail by replacing it with a lower level activity frame
- The hierarchy is important to us
- Every activity shown includes its acceptance test or we expect to see a sub hierarchy
    - Hierarchies are terminated only by acceptance tests

Activity Frames


AF2 : Installation Lifcycle
Commission (Test T10)
Normal operation
Maintain (Test T11)
Decommission (Test T12)
Satisfies Installation Lifecycle Operation framework test Test T13 /c/

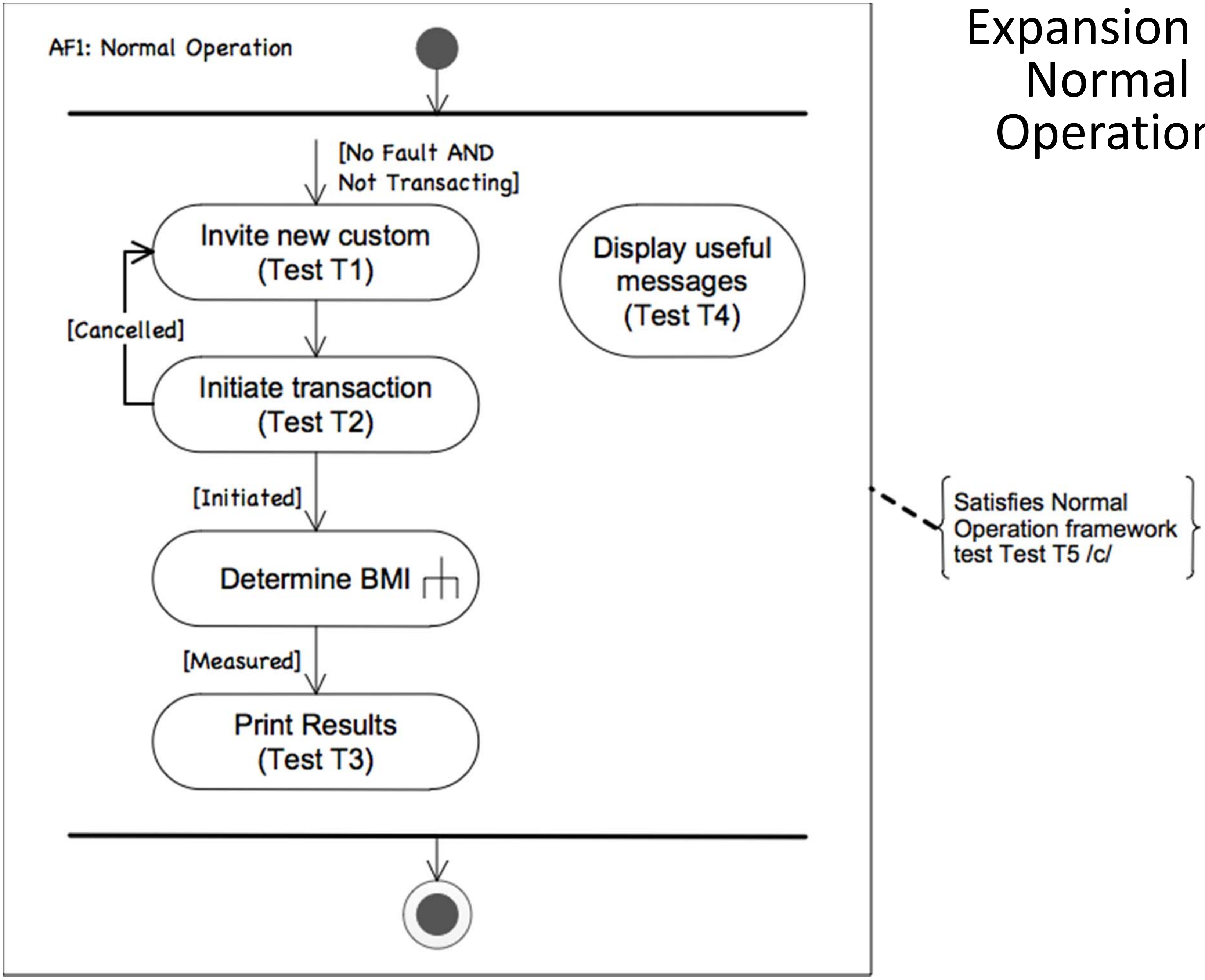
The ‘Frame’

Non UML standard simplification of concurrent activity affordance

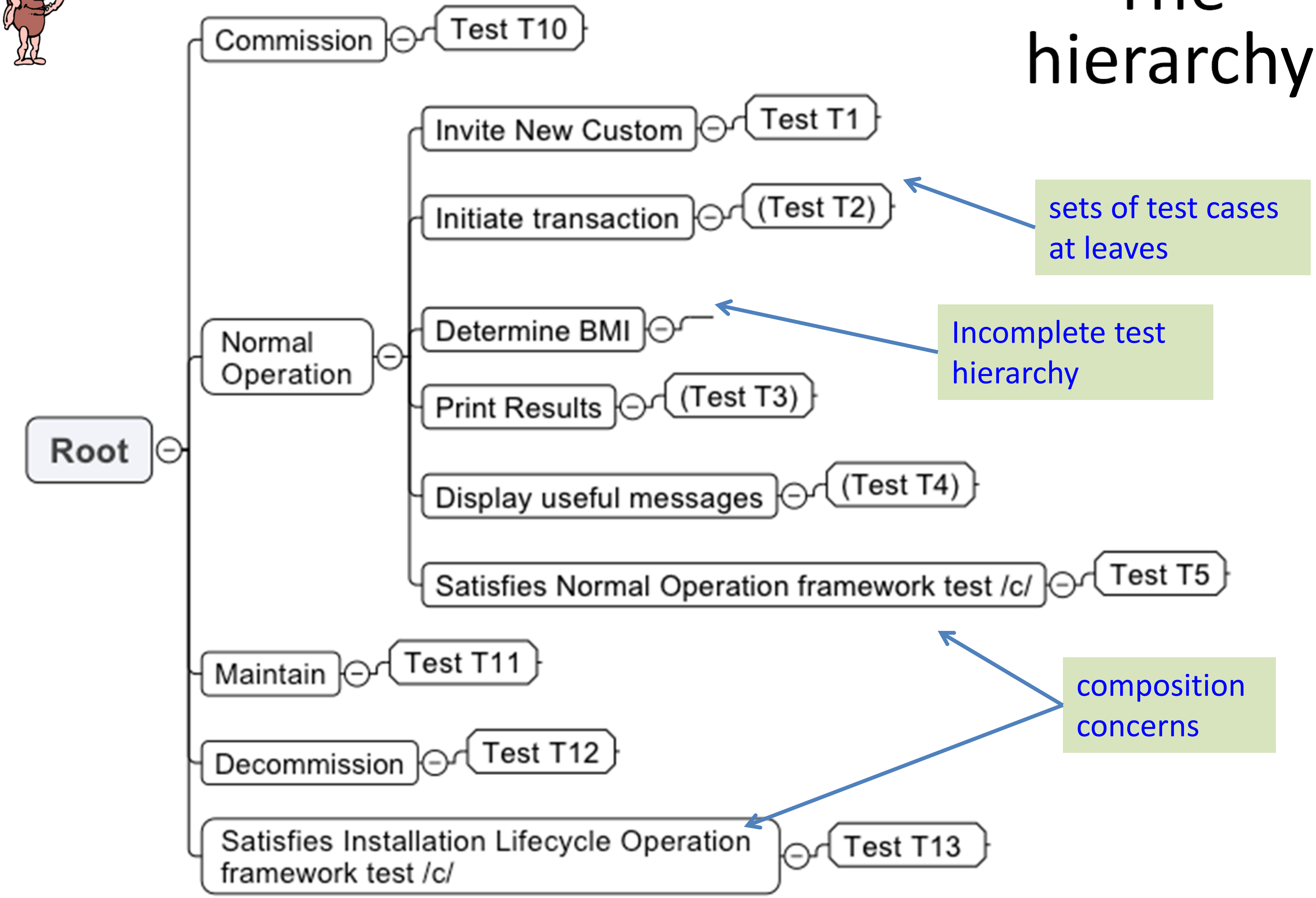
This symbol means that there is a sub activity frame available

A condition binding on the whole frame – and its sub frames

Expansion of Normal Operation


AF1: Normal Operation
[No Fault AND Not Transacting]
Invite new custom (Test T1)
[Cancelled]
Initiate transaction (Test T2)
[Initiated]
Determine BMI
[Measured]
Print Results (Test T3)
Display useful messages (Test T4)
Satisfies Normal Operation framework test Test T5 /c/

The hierarchy



- Root
  - Commission
    - Test T10
  - Normal Operation
    - Invite New Custom
      - Test T1
    - Initiate transaction
      - (Test T2)
    - Determine BMI
    - Print Results
      - (Test T3)
    - Display useful messages
      - (Test T4)
    - Satisfies Normal Operation framework test /c/
      - Test T5
  - Maintain
    - Test T11
  - Decommission
    - Test T12
  - Satisfies Installation Lifecycle Operation framework test /c/
    - Test T13

sets of test cases at leaves

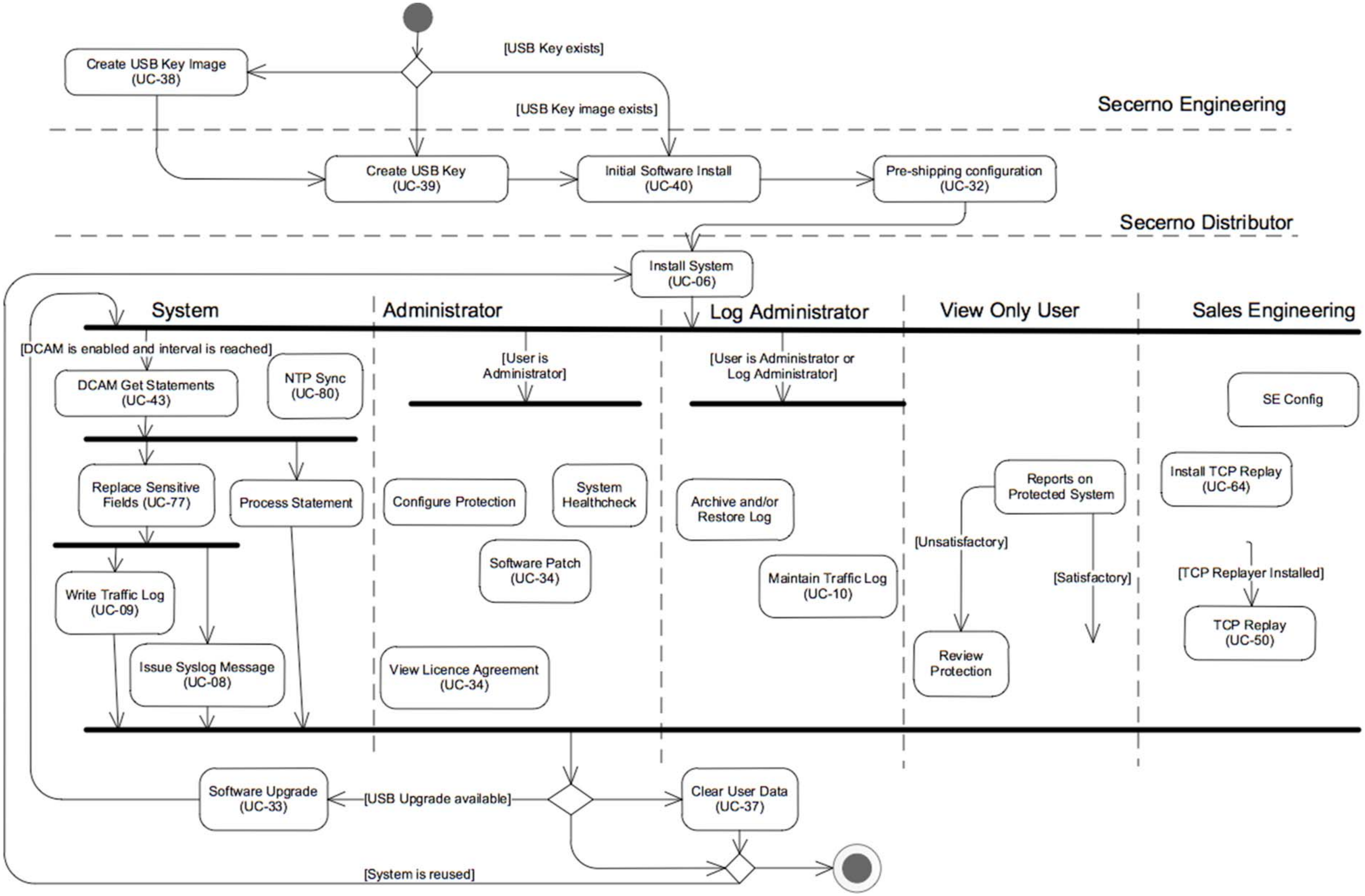
Incomplete test hierarchy

composition concerns

Consider the hierarchy

- We can see all the warranted behaviour
- We can check that is what is meant
- Every leaf is a test (or set of test cases)
- We have tests for compositional logic often neglected
- We have a basis for expressing the percentage coverage of our testing
- We have a systematic approach to the identification of all necessary dynamic tests

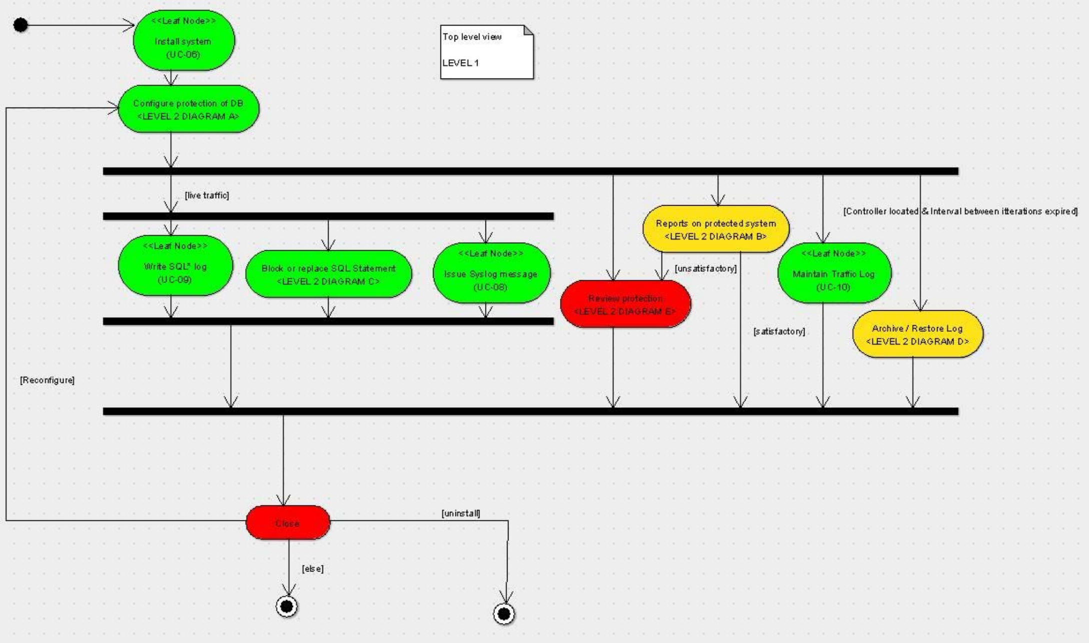
A real world example activity frame



The real world example

- The starting point was sprint 30 of a Scrum style backlog driven development
  - already 25 engineer years of development
  - Could not determine percentage coverage
  - Testing failing to give confidence on produce releases
- The maximum depth of hierarchy was 5 levels
- All leaves of the hierarchy were covered by acceptance condition test cases
- The rigour of the approach detected key compositional tests that had been missed
- The root of the hierarchy allowed a traffic light dashboard overview of test hot spots

Hot spots shown by testing



Summary

- We have looked at ways of identifying black box test cases based on
    - Use Cases
    - Reverse engineering with simplified UML like Activity Diagrams
        - Activity Frames
- These are practical, real world approaches but not likely to be found in text books - yet

(b) Static Testing

Dr. Ken Boness

k.d.boness@reading.ac.uk

Reviews

- “A process or meeting during which a work product, or set of work products, is presented to project personnel, managers, users or other interested parties for comment or approval.”
- BS 7925-1

Goals of Reviews

- The verification of the review item against specified standards and criteria.
  - Also validation in the case of requirements.

- To achieve consensus opinion.

- The review team should work together to improve the quality of the item being reviewed.

Why Review

- There are different levels of review from very informal to very formal, they all have their benefits and should be used throughout the SDLC.

- Reviews are very effective at finding faults.
- Very cost effective.
- Easy to set up.
- Effective at finding problems and issues very early in the SDLC.
  - Recall the cost escalation discussed in lecture 1.

What may be Reviewed?

- Requirements
- Functional Specifications
- Designs
- Program Specifications
- Interface Specifications
- Installation instructions
- Code
- Test Plans
- Test Cases
- Any other documents such as user guides.

Benefits of Reviews

- Quality of product improved
- Process improvement
- Productivity improved
- Reduced project timescales
- Reduced testing and rework
- Reduced levels of faults
- Reduced product lifetime costs

Review Activities

- Planning
- Overview meeting
- Preparation
- Review meeting
- Report on review
- Follow-up meetings

Types of Review

- Informal reviews
- Walkthroughs
- Formal technical reviews
- Inspections

Informal review

- These are the least formal of the review methods

- Done at any time – largely unplanned
- Peer to peer
- Builds team spirit
- Two-way communication
- Undocumented
- Cheap and widely used

- Often not even aware you are doing it? Assuming you have good working practices!

Inspections

- Led by trained moderator
- Each participant has a specific role
- Extensive use made of formal processes (such as checklists, entry and exit criteria)
- The document is read aloud and followed word for word.
  - Based on the work of Michael Fagan in 1970’s and 80’s
  - Essential that the moderator is trained and/or experienced.

Summary of the Procedural Rules (1/2)

- Inspections often done at a number of points in the process of project planning and system development; i.e. through the SDLC.
- All classes of defects in documentation are inspected; not merely logic or function errors.
- Inspections are carried out by colleagues at all levels of seniority except the big boss.
- Inspections are carried out in a prescribed series of steps such as individual preparation, meeting, error rework etc.
- Inspection meetings are limited to two hours.
- Inspections are led by a trained moderator.

Summary of the Procedural Rules (2/2)

- Inspectors are assigned specific roles to increase the effectiveness.
- Checklists of questions to be used by inspectors are often used to define the tasks and to stimulate increased defect finding. (there has been some research discussion on the benefits on the benefits of checklists of versus ad hoc detection – See Basili et al – this is provided as a handout for discussion).
- Material is inspected at a particular rate which has been found to give maximum arrow finding ability.
- Statistics on types of defect are kept and used the reports which are analyzed in a manner similar to financial analysis. These reports are vital for improving process of inspections themselves but most importantly the quality process of the company involved.

Fagan Inspection Process Operations.

- PLANNING
  - Material, Inspectors, schedule. Material, Inspectors, schedule.
- OVERVIEW
  - Introduce the subject matter and context to team.
  - Optional depending on case.
- PREPARATION
  - Learn material, prepare role,
  - Do not focus on finding defects.
- INSPECTION MEETING
  - Find and classify defects.
- PROCESS IMPROVEMENT
  - Identify systematic defects in the process and recommend fixes.
- REWORK
  - Author reworks all fixes.
- FOLLOW-UP
  - Moderator verifies all fixes and fitness of the material to pass.

We are usually Looking for

- Errors of Omission
- Errors of Commission

- The following two slides are taken from:
- Comparing Detection Methods for Software Requirements Inspections: A replicated experiment” Porter A. A, Votta L, Basili V.R. 1994 IEEE transactions.

Errors of Omission

- Missing Functionality: Information describing the desired internal operational behaviour of the system has been omitted from the SRS.
- Missing Performance: Information describing the desired performance specifications has either been omitted or described in a way that is unacceptable for acceptance testing.
- Missing Interface: Information describing how the proposed system will interface and communicate with objects outside the scope of the system has been omitted from the SRS.
- Missing Environment: Information describing the required hardware, software, database, or personnel environment in which the system will run has been omitted from the SRS.

Errors of Commission

- Ambiguous Information: An important term, phrase or sentence essential to the understanding of system behaviour has either been left undefined or defined in a way that can cause confusion or misunderstanding.
- Inconsistent Information: Two sentences contained in the SRS directly contradict each other or express actions that cannot both be correct or cannot both be carried out.
- Incorrect Fact: Some sentence contained in the SRS asserts a fact that cannot be true under the conditions specified in the SRS.
- Wrong Section: Essential information is misplaced within the SRS

Phantom Inspector

- The synergy of the team produces more benefits than accounted for by the individuals present.

- Additional defects found by the phantom inspector
- 28% more in requirements
- 55% more in code

University of Reading

SE3Q11: Week 11

Test Planning and Discussion

Dr. Ken Boness

k.d.boness@reading.ac.uk

University of Reading

(a) Planning Tests

Dr. Ken Boness

k.d.boness@reading.ac.uk

Testing

- Detecting **faults** so that they can be remedied
- Early detection is a key factor in minimising the cost of remedy
- Testing is like an empirical science
  - Test conditions controlled and can be reproduced
  - Test specimen defined and can be reproduced
  - Test Results captured, analysed and kept available for future reference
  - Analysed test results communicated for action
  - All the above managed under a plan

How much can we expect to test?

- Testing requires an **investment** of time and resource
- The timing of tests with respect to the development programme may be crucial
    - Is there time to perform exhaustive tests?
- The cost of providing equipment needed for tests may be unacceptable
    - For example: Multiple computers with many permutations of operating system and collaborating 3rd party products
- We may be limited by inescapable uncertainties

A simple example

Consider the simple program

- What would amount to exhaustive testing?
- Is it sufficient to test the source code without running an executable version?

```
For I = 1 to 100
      If A[i] = True then
           Print I
     Else
           Print 0
     Endif
```

‘ (Example based on van Vliet.)

Working on the source

- We can read the program code and check that it conforms to programming quality standards.
- We can read the program code and test its logic without running the code.

These are both examples of what we call **static** testing

i.e. we test without running the code in its expected computing environment

```
For I = 1 to 100
        If A[i] = True
then
              Print I
      Else
              Print 0
      Endif
```

Working on the executing program

- We can build the executable file and deploy it and then confirm that it performs according to its specification**.
- This would be what we call **dynamic** testing
- But how feasible will it be to test it exhaustively?

***We might need to read the Specification as another static test.*

```
For I = 1 to 100
        If A[i] = True then
              Print I
      Else
              Print 0
      Endif
```

Over-ambition for dynamic testing

Consider the simple program

- What assumptions are there? (Consider A[i])
- How many outcomes to be tested (2100)?
- How many deployment permutations would need to be repeated?

```
For I = 1 to 100
        If A[i] = True then
            Print I
      Else
            Print 0
      Endif
```

Being practical

- Consider a large industrial product development
- Say 20 person years of software development
  - Perhaps 1 million lines of Code!
  - The simple example had 6 lines of code!
- Need to be cunning and commensurate
  - Plan what is important to test in the circumstances (considering safety, reliability, support cost etc).
  - Choose key tests based on a mixture of sampled and exhaustive tests.

Some Numbers

- About 35-80 errors per 1000 lines of code should be expected.
- Extensively tested software may be expected to have 0.5-3 errors per 1000 lines of code.
- A fault in the seat allocation software of an airline cost the company $50m in one quarter.
- Fixing an error in operational software it likely to be 10-90 times more expensive than fixing it in the design phase.

(see Van Vliet “Software Engineering”)

Conclusion: Exhaustive testing would in most cases take an enormous amount of resource and is therefore usually impractical.

- We must be **selective** and **commensurate**!

Selective

- If there is not enough time to test everything then we need to be smart about what choose to test.
- We must consider
  - what we must test and
  - how we test

  in order to do
  - Enough, and
  - Not expect to do too much.

Commensurate

- Determining how much testing is enough depends upon the circumstances and the risks involved. For example:
    - Obligations for maintaining the software after its release
    - Possibility of schedule delays by late discovery of errors
    - Extent to which testing is contractually mandated
    - Criticality of safety and / or security guarantees
    - Possible litigation
- Risk must be used as the basis for allocating the test time that is available and for selecting what to test and where to place emphasis
    - For example safety critical software would usually require much more rigorous testing than most commercial web sites. It is a matter of risk management

High level planning of tests

- Set the testing scope.
- Analyse the risks especially of things deliberately omitted from the testing scope.
- Define suitable test stages.
    - Define entry and exit criteria for each stage.
- Define the requirements on test environment.
- Identify the sources of test data.
- Define the documentation requirements

Test planning

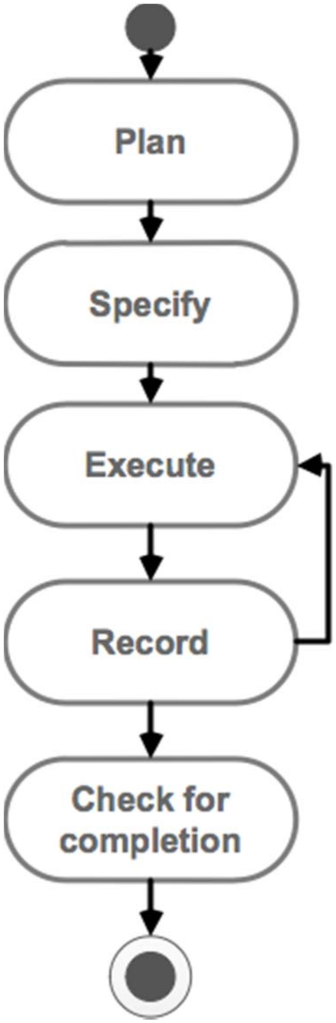
- A considered strategy defining what is to be tested and what is not
  - Include risk assessment of what is deliberately omitted.
  - Relate to the development schedule
- Static Tests
  - Decide on the artefacts, test methods and criteria (entry, fault type, exit)
- Dynamic tests
  - Component Tests (aka unit tests)
  - Integration tests
  - System tests
  - Regression Tests
  - Acceptance tests
  - Other (smoke tests, stress tests, bash tests)
- Definitions of the tests
  - Entry-conditions, Exit-conditions and explicit tests
- Resources
  - Data
  - Test harnesses
  - Environmental hardware and software
- Who, when where
  - Traditional planning embracing all the above tests and resources as the 'what'

Planning static tests

- We will consider Inspections
  - Other static tests are less prescriptive
- Follow the Fagan operations
  - see lecture on static tests
- For each anticipated artefact inspection
  - Identify the artefact (including version)
  - Write a schedule
  - Specify entry criteria (pre-conditions)
  - Fault criteria (test categories and inspection theme)
  - Specify exit criteria (post-conditions)
  - Specify any resources needed beyond the artefact being inspected

Planning dynamic tests

- The so-called **Fundamental Test Process**
- A series of steps to follow for any particular static or dynamic test
  - Echoes of good practice in experimental science!
- Done diligently
  - Meaningful results
  - Reusable results
  - Results can be reproduced



…The Fundamental Test Process

- As the objective of a test should be to detect faults, a 'successful' test is one that does detect a fault.
  - This is counter-intuitive, because faults delay progress: a successful test is thus also one that may cause delay.
  - The successful test reveals a fault which, if found later, may be many times more costly to correct so, in the long run, is a good thing.
- Completion or exit criteria are used to determine when testing (at any test stage) is complete. These criteria may be defined in terms of:-
  - Cost, time, faults found or coverage criteria.
  - Coverage criteria are defined in terms of items that are exercised by test suites, such as branches, user requirements, most frequently used transactions, etc.

Testing throughout the lifecycle

- There are different development process models for the software development lifecycle (SDLC). They suit different situations. The V-model is particularly effective for testing.
- The overall test plan should be appropriate to the development process
    - For example Agile processes have different needs compared with V-model development

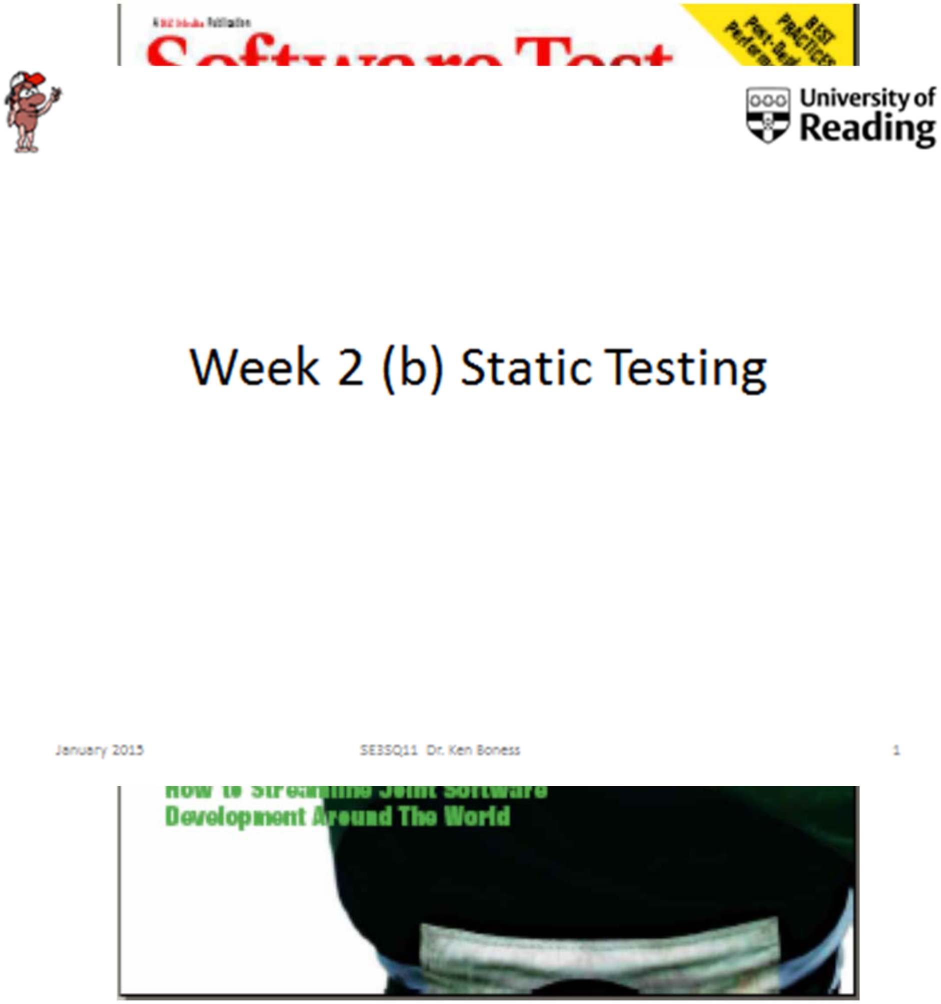
(b) Discussion

Dr. Ken Boness

k.d.boness@reading.ac.uk

Testing is an organisation (e.g. the combination of supplier and sponsor) owned system of human activity which takes place in the engineering environment of software product development and is participated in by many members of the organisation and its customers and notably by people called testers. It takes various software engineering artefacts as inputs and produces evidence of their quality as outputs and it does this by means of specification, preparation, execution, results analysis and reporting among other things. It benefits the organisation by avoiding and minimising the adverse effects of artefact failure (such as delay, loss of funds, injury to people). The worldview that makes this valuable is that it will save unnecessary expense, increase the organisation’s standing and will support the organisation’s quality system.

That’s all folks!



Software Test
BEST PRACTICES
Post-Depl
Perform
How to Streamline Joint Software
Development Around The World

University of Reading

Week 2 (b) Static Testing

January 2015 SE3SQ11 Dr. Ken Boness 1