International Quality Conference
Toronto, Canada, October 3-7, 2005

Understanding the Specifics of Use-Case-Driven Testing

Presented by Yuri Chernak, Ph.D.
Valley Forge Consulting, Inc.

Email: ychernak@yahoo.com

© 2005 by Valley Forge Consulting, Inc.

Presentation Outline

- Introduction
- Understanding the Use Case Types and Relationships
- **Specific #1** – Decomposing the Test Process into Separate Workflows
- **Specific #2** – Understanding the Level of Detail Captured by Use Cases
- **Specific #3** – Handling the Iterative Evolution of the Use-Case Model
- **Specific #4** – Analyzing the Use Case Types and Relationships
- **Specific #5** – Understanding the Difference in Approaches to Testing Concrete vs. Abstract Use Cases
- Presentation Summary

© 2005 by Valley Forge Consulting, Inc.

Introduction

- Over the last decade the Rational Unified Process (RUP®) has become a mature methodology that enjoys increasing popularity in the software development community.
- The main features of this methodology are:
  - **Use-case-driven development**;
  - Architecture-centric development;
  - Iterative and incremental development;
- Among these three features, the "use-case-driven" feature has had the most profound effect on software testers.
- The term ***use-case-driven*** means that all major project activities are driven by use cases, and all major artifacts are managed around use cases.
- **Use-Case-Driven Testing (UCDT)** has important specifics compared with conventional **Requirements-Based Testing (RBT)** that can present a number of challenges for software testers.

Presentation FOCUS

In this presentation we discuss the use-case-driven testing and its specifics from the **System Test** perspective.

What Are Use Cases?

OMG UML Specification (v.1.5, March 2003) –

"The use case construct is used to define the behavior of a system without revealing its internal structure. Each use case specifies a sequence of actions that the system can perform interacting with its actors."

A. Cockburn, "Writing Effective Use Cases", 2001 –

"A use case captures a contract between the stakeholders of a system about its behavior. The use case describes the system's behavior under various conditions as the system responds to a request from a stakeholder, called primary actor."

K. Bittner, I. Spencer "Use Case Modeling", 2003 –

"Use cases represent the things of value that the system performs for its actors."

On real-world projects use cases can be of different types and can have complex relationships with each other.

Use Case Types and Relationships

Use case types can be classified from the following three perspectives:

- **Perspective #1 – Problem Domain**
  - **Business Use Cases** (*capture business flow and operations*)
  - **System Use Cases (***capture software requirements to build a system***)**
    - **Application Use Cases** *(address functional requirements)*
    - **Infrastructure Use Cases** *(address non-functional requirements)*
- **Perspective #2 – Use-Case Relationships**
  - **Generalization Use Cases (***capture a generalized behavior***)**
  - **Specialization Use Cases** (*capture a specialized behavior*)
  - **Base Use Cases (***can have one or more inclusions***)**
  - **Inclusion Use Cases (***capture functionality included in and reused by other base use cases***)**
  - **Extension Use Cases** (*extend functionality of a base use case*)
- **Perspective #3 – Use Case Instantiation**
  - **Concrete Use Cases (***have an actor who can instantiate a use case***)**
  - **Abstract Use Cases**

    (a) *generalization use cases that cannot be instantiated at all,*

    (b) use case fragments that do not have their own actors to instantiate these use cases.

Similarity Between Use-Case-Driven Testing and Requirements-Based Testing

- Testing strategies of both - UCDT and RBT **-** can be composed of the same levels of testing, for example, the ***unit-integration-system-acceptance*** levels of testing;
- The UCDT and RBT are both critically dependent on the development practices in such process areas as:
  - Project Planning
  - Requirements Management
  - Software Configuration Management
- Regardless, whether it is use-case-driven or requirements-based testing, testing is only as good as the software requirements.

Despite these similarities, use-case-driven testing has its own specifics that can present challenges for software testers.

Use-Case-Driven Testing Specifics

Level of Detail Captured by Use Cases

Use cases are not intended to capture all details of system functionality. Some common examples of supplemental requirements not captured by use cases are:

- User interface (e.g, GUI) requirements;
- Data dictionary requirements (*data-entry field validation rules*);
- Referential integrity rules, concurrency rules, and other database design requirements;
- Requirements for system reports (*report layouts and contents*);
- Calculations and other business rules;
- System interface requirements (*an interface to other systems*).

Use Case Types and Relationships

Unlike traditional functional requirements, use cases can be of different types and can form a complex structure of a use-case model via the <<generalization>>, <<inclusion>>, and <<extension>> types of relationship.

Iterative Evolution of Use-Case Model

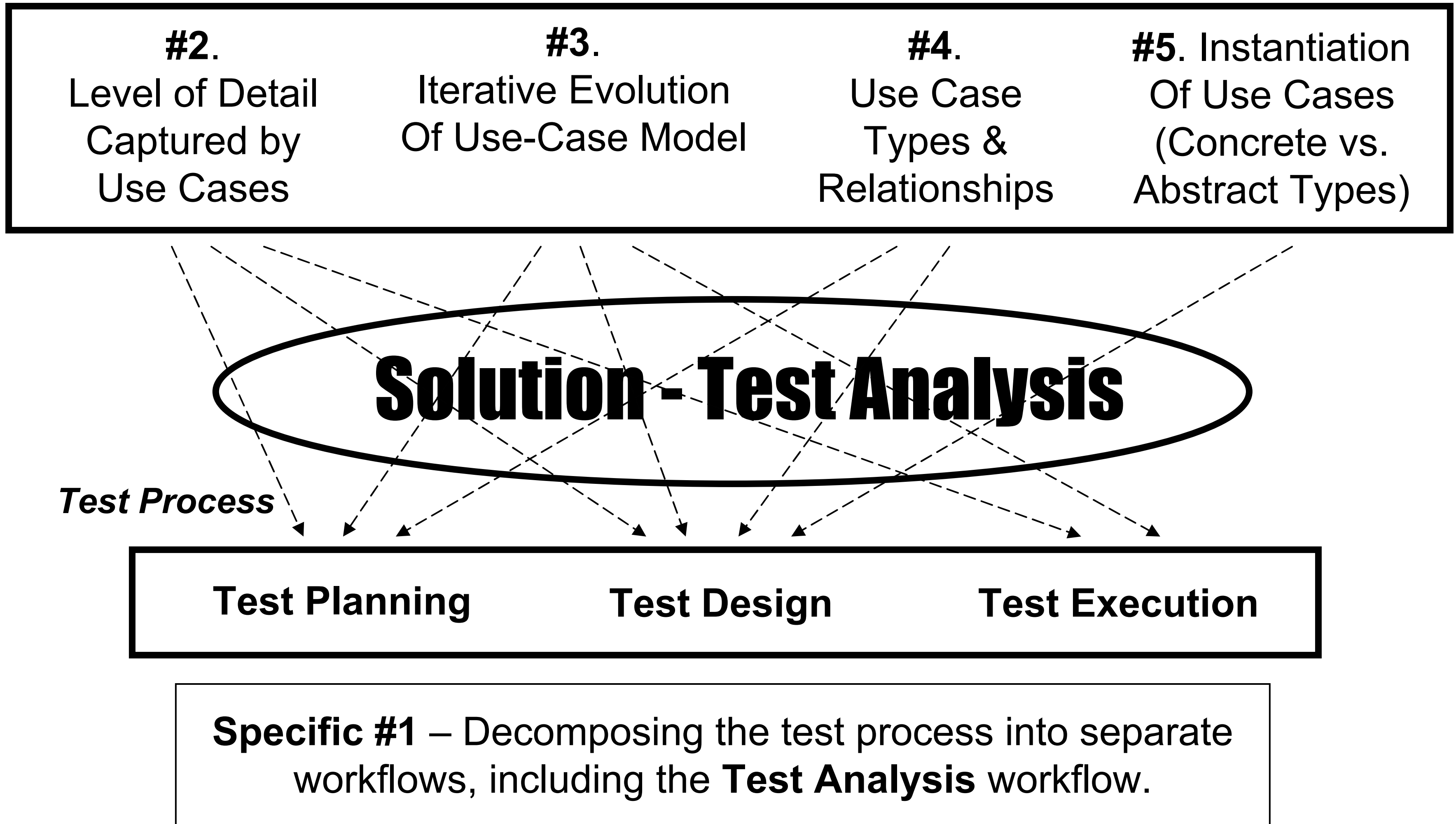
The use-case model can frequently change in the course of a project for the following common reasons:

- new use cases can be incrementally added to the existing use-case model in a new project iteration;
- more details can be added to the existing use cases, for example, alternative or exception scenarios; and
- developers can periodically refactor the use-case model to make its structure better understood and easier maintained.

Use Case Instantiation

Some inclusion/extension use cases can be just fragments of functionality that do not have their own actors who can instantiate their scenarios. Such fragments are called **abstract** use cases,as opposed to **concrete** use cases that have their own actors. In addition, abstract use cases may not have any flow of events.

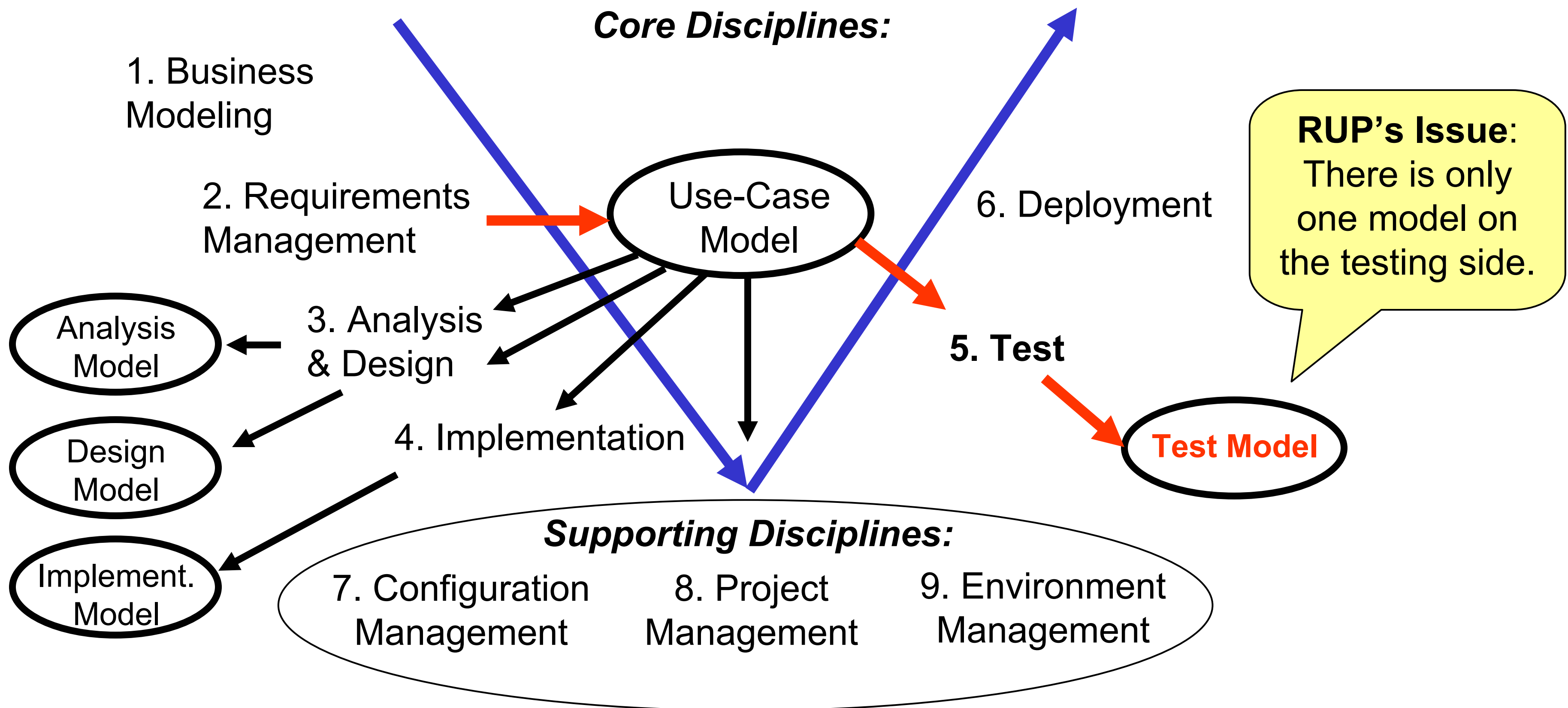
Impact of UCD Development on the Test Process



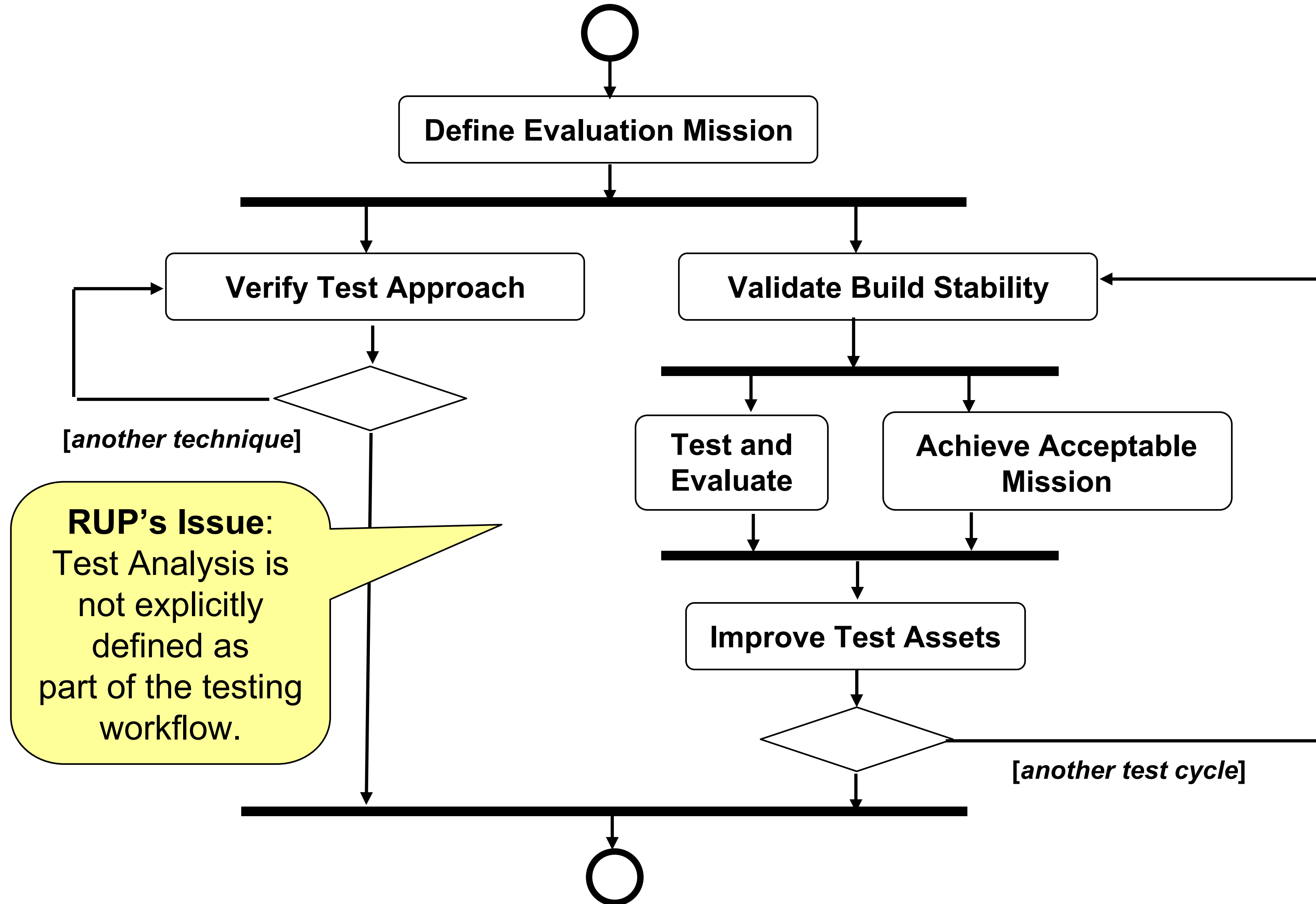
Specific #1 – Decomposing the test process into separate workflows, including the **Test Analysis** workflow.

Specific #1: Decomposing the Test Process into Separate Workflows

The Rational Unified Process (RUP) methodology is composed of the nine disciplines:



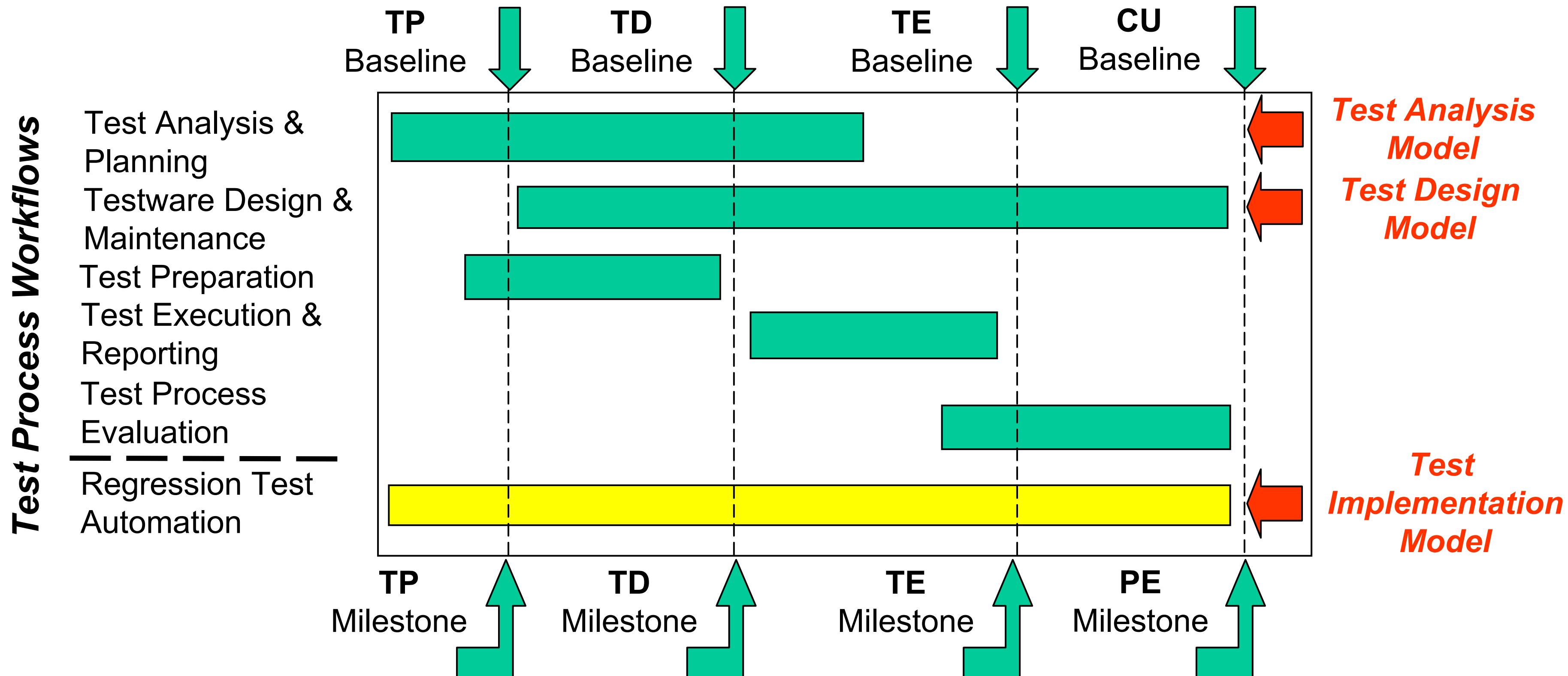
Specific #1: RUP's Testing Workflow



© 2005 by Valley Forge Consulting, Inc.

Specific #1: Tips & Guidelines

The test process can be logically decomposed into the following six workflows:



Acronyms for Milestones and Baselines: TP – Test Planning; **TD** – Test Design; **TE** – Test Execution; **PE** – Project-End; **CU** – Cumulative;

(*source - S. Boycan, Y. Chernak "Case Study: Implementing Configuration Management for Software Testing Projects", CrossTalk, July, 2005*)

© 2005 by Valley Forge Consulting, Inc.

Specific #1: Tips & Guidelines (cont'd)

Objectives of the Test Process Workflows

- **Test Analysis and Planning** – The main objectives of this workflow are a) to define the test strategy, and b) to analyze the use-case model to plan testing and to identify all the quality concerns within the context of each use case.
- **Testware Design and Maintenance** – The objectives of this workflow are to refine test ideas about what must be tested for each use case and to provide details about how to execute these tests. Also, this workflow includes maintenance of the existing test designs.
- **Test Preparation** – The objectives of this workflow are to set up a test environment and a defect tracking system, to generate required test data, and to develop test supporting utilities (if required).
- **Test Execution and Reporting -** The objectives of this workflow are to execute tests and evaluate the quality of the software product, to find and report software defects, and to report the product's testing progress and status.
- **Test Process Evaluation** – The objectives of this workflow are to evaluate the test process completeness, effectiveness, and efficiency, to perform a causal analysis of *test escapes*, and recommend improvements to the test process.
- **Regression Test Automation** – The objective of this workflow is to develop the test automation architecture and automated regression scripts.

Specific #1: Tips & Guidelines (cont'd)

- By developing three test models (Analysis, Design, Implementation), testers implement a top-down approach to better handle the complexity of the use-case model.
- The Test Analysis workflow is very important. It delivers the Test Analysis model that is a basis for developing the Test Design and Test Implementation models.
- The Test Analysis workflow overlaps with the Test Design and Test Execution workflows to feed test ideas to test design. During test execution, testers should continue exploring a software product and look for new test ideas.
- As the test process is defined as a collection of continuous workflows, it is important that a test team has some defined project milestones where testers make important project decisions.

Examples of Test Project Milestones

- **TP** - Test Planning Milestone *(a decision – a project is feasible, a test team confirms the project's plan and schedule);*
- **TD** - Test Design Milestone *(a decision – test design is sufficiently complete to start test execution);*
- **TE** - Test Execution Milestone *(a decision – testers have met test exit criteria and can stop testing)*;
- **PE** - Project-end Milestone *(a decision to end a project cycle and to select test artifacts to maintain and reuse in future cycles)*.

© 2005 by Valley Forge Consulting, Inc.

Specific #2: Understanding the Level of Detail Captured by Use Cases

Conventional Functional Requirements vs. Use Cases

- Conventional functional requirements are commonly broken down into small pieces and they tend to present low-level functional details. Hence, the task of designing test cases for a given requirement can be straightforward.
- In contrast, use cases - a) capture large chunks of functionality that produce the results of value to stakeholders, and b) are not intended to capture all of the functional details of system implementation.

Supplemental requirements commonly not captured by use cases are:

- User interface (GUI) requirements
- Data dictionary requirements (*data-entry field validation rules*)
- Semantic integrity rules, referential integrity rules, concurrency rules, and other database design requirements
- Requirements for system reports (*report layouts and contents*);
- Calculations and other business rules
- System interface requirements (*an interface to other systems*)

Specific #2: Understanding the Level of Detail Captured by Use Cases (cont'd)

Issues with Use-Case-Driven Testing

- ❑ Implementation of supplemental requirements can tangle and mix with the implementation of use-case scenarios.
- ❑ Implementation of the same supplementation requirement can be scattered across the application and crosscut multiple use cases. On the other hand, the implementation of the same scenario can mix with multiple supplemental requirements.

Testers' Challenges in Use-Case-Driven Testing

- ❑ How to produce test designs that are structured around use cases and cover all related supplemental requirements as well?
- ❑ How to encapsulated test designs for supplementation requirements that they can be effectively reused for applicable use cases and efficiently maintained when supplemental requirements have changed?

By performing **test analysis**, testers can better handle these challenges and identify all quality concerns they need to address in system testing.

Specific #2: Tips & Guidelines

Test Analysis in RBT

Test analysis is performed for an entire application, is a part of test planning, and is commonly documented in a Test Plan document, for example, IEEE Std. 829 Test Plan document - sections “Features to be Tested”, “Test Approach”.

Two Levels of Test Analysis in UCDT

- ❑ Test analysis is performed for an entire application and it can be documented in a System Test Plan document;
- ❑ Test analysis is also performed for a given use case and it can be documented in a Test Analysis specification (*we will discuss later*).

The Purposes of Test Analysis in UCDT

- ❑ The purpose of test analysis, performed for the entire application, is to develop ideas about what in general can go wrong with this kind of system and identify crosscutting quality concerns, i.e., concerns related to multiple use cases.
- ❑ The purpose of test analysis, performed for a use case, is to help a tester better understand and identify all of the use case’s quality concerns that should be addressed in system testing and provide a basis for test effort estimation and guidelines for detailed test design.

© 2005 by Valley Forge Consulting, Inc.

Specific #2: Tips & Guidelines (cont'd)

What are Crosscutting Quality Concerns (CQC)?

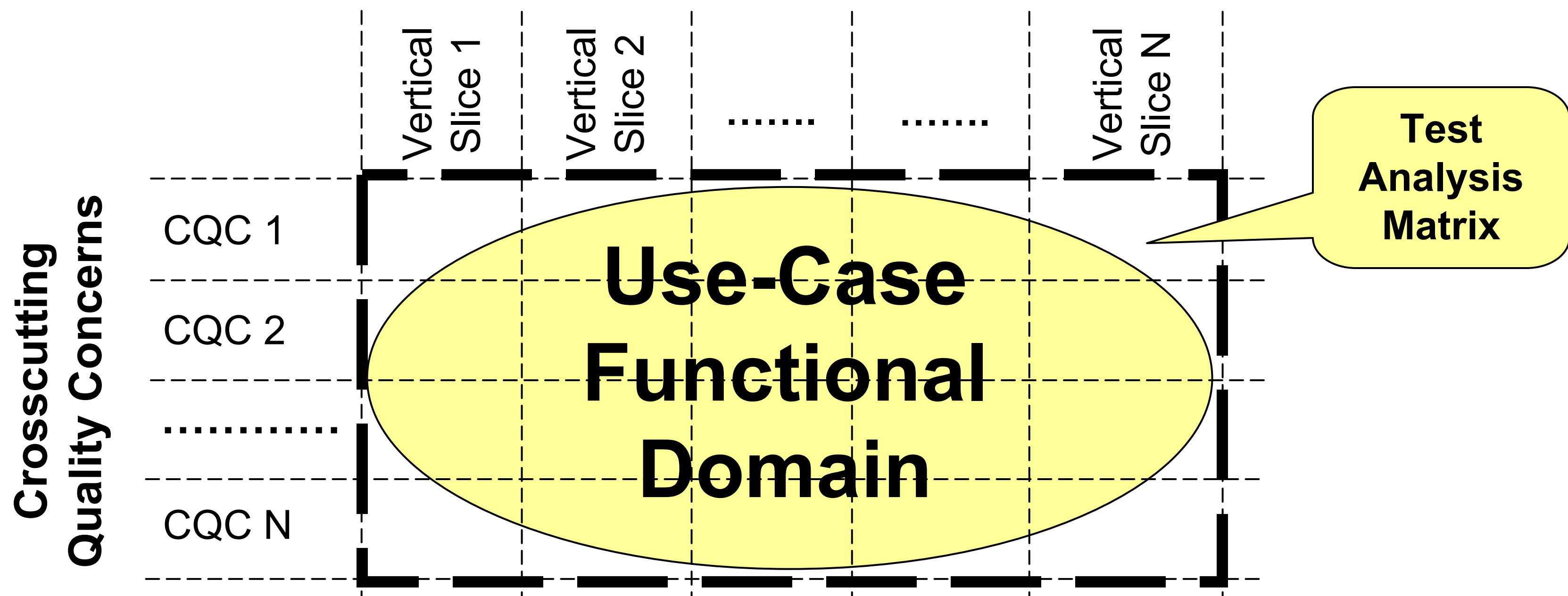
- CQCs commonly correspond to supplemental requirements, for example, GUI, concurrency, or data dictionary requirements that are scattered across the application and crosscut multiple use cases.
- Each CQC can have its own test design. Being encapsulated, these test designs can then be reused across the applicable use cases. For example, the way we test such features as data-entry field edits, GUI objects, or concurrency is the same across the application.

Separation of Quality Concerns in System Testing

- The proposed test analysis technique is based on the old and proven software engineering principle – ***Separation of Concerns***.
- E.W. Dijsktra proposed and discussed this principle in his book “A Discipline of Programming” he published in 1976.
- I. Jacobson applied the same principle to use-case-driven development in his latest book “Aspect-Oriented Software Development with Use Cases”.
- In this presentation we discuss the test analysis technique that is based on separation of crosscutting quality concerns from use-case-specific quality concerns.

Approach to Performing Test analysis for Use Cases

- Two perspectives to decompose a use-case functionality:
  - **Horizontal slicing** – identifies applicable crosscutting quality concerns and their realizations in the given use case context (e.g., GUI, field edits, etc.);
  - **Vertical slicing** – identifies use-case-specific quality concerns (i.e., use-case instantiation scenarios).
- The last step of test analysis is mapping the applicable CQCs to the vertical slices. The end-result can be presented as a **Test Analysis Matrix**.



Documenting the Test Analysis Results

IEEE Std.829 "Standard for Test Documentation"

- Test Plan Document
- Test Design Specification
- Test Case Specification
- Test Procedure Specification

The document type – ***Test Analysis Specification*** – is not defined in Std.829.

Example of a Test Analysis Specification Template

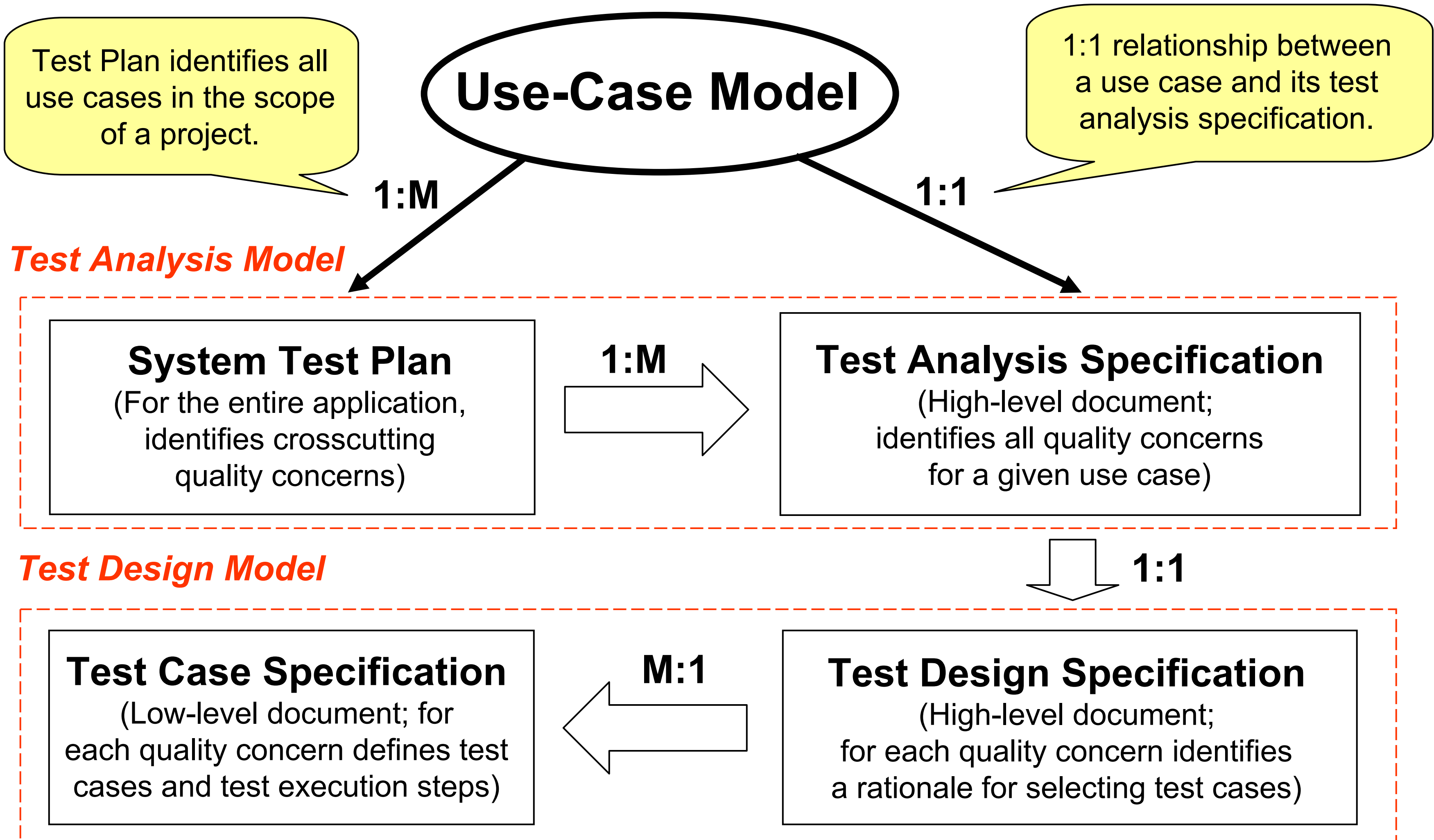
- Title Page
- Use Case Short Description
- Use Case Relationships
- Realizations of Crosscutting Quality Concerns
- Use Case Decomposition into Vertical Slices
- Quality Risks not to be Tested
- Test Analysis Matrix

Benefits of the Test Analysis Specification

- Helps us perform test effort estimation;
- Provides a road-map for detailed test design;
- Helps us perform impact analysis when requirements have changed.

© 2005 by Valley Forge Consulting, Inc.

Test Documentation Structure for UCDT Projects



Specific #3: Dealing with the Iterative Evolution of the Use-Case Model

- The use-case model can frequently change in the course of a project for the following common reasons:
  - new use cases can be incrementally added to the existing use-case model in a new project iteration;
  - more details can be added to the existing use cases, for example, as alternative or exception scenarios; and
  - developers can periodically refactor the use-case model to make its structure easier to understand and maintain.
- Testers should analyze each change to the use-case model, understand how it affects their own work products, and evolve and maintain their test models to keep them consistent with the use-case model.
- When anticipating inevitable use-case model changes, testers can follow some guidelines that can make the change impact on the test process less expensive.

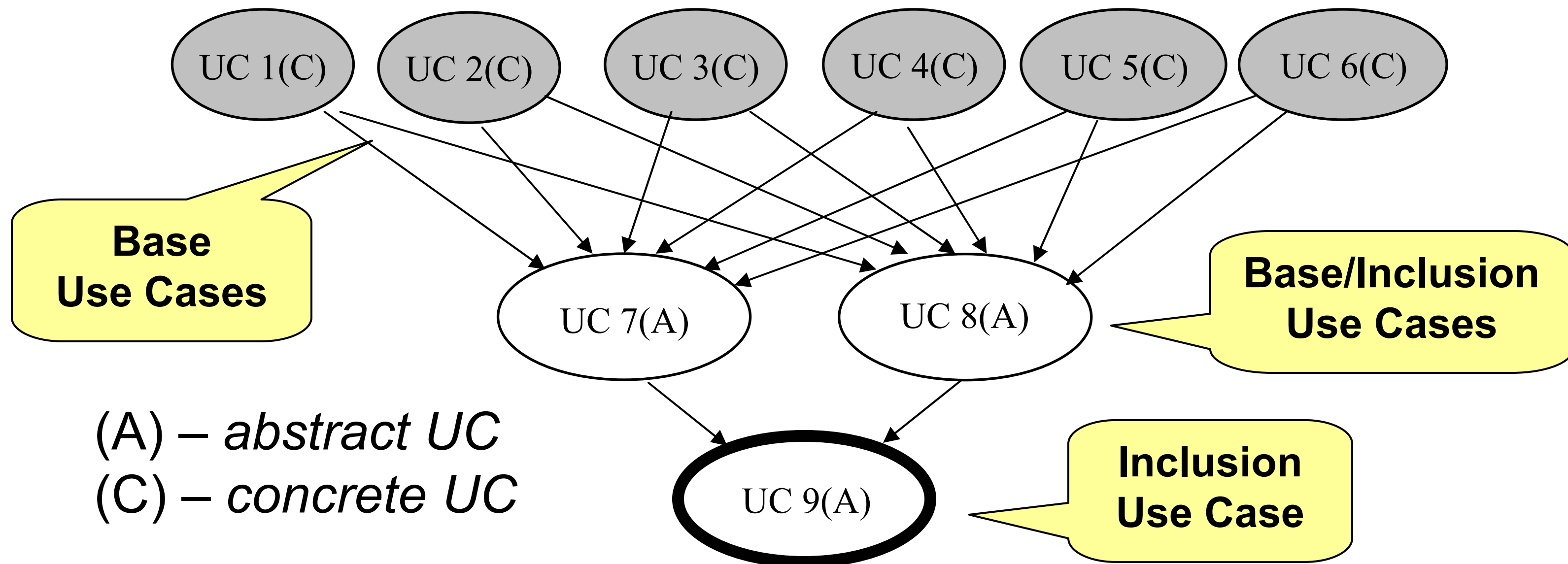
Specific #3: Tips & Guidelines

- **Test Planning.** Follow the use-case-driven concept and structure testers' tasks in the test project plan and provide their effort estimation per use case. Plan testing each use case as a separate mini-project;
- **Test Design.** Remember, the more test documentation you plan to produce, the more expensive it will be to create and maintain. Focus on producing high-level test analysis and test design specifications, as opposed to low-level test case specifications.
- **Test Design.** Establish a one-to-one relationship between use cases and their related test analysis and test design specifications that provides the following benefits:
  - any changes to use cases can be easily traced to their corresponding test designs;
  - when developers refactor the use-case model structure, again, having the structure identical on both sides can make the impact analysis and test design maintenance much easier;
- **Implementing Version Control.** Periodically create project baselines that include artifacts of both (see Slide 11) the use-case model and the corresponding test models. This way testers can better keep in sync the evolving models on both sides of the software process.

© 2005 by Valley Forge Consulting, Inc.

Specific #4: Analyzing the Use Case Types and Relationships

Unlike traditional functional requirements, use cases can be of different types and can form a complex structure of the use-case model via the <<generalization>>, <<include>>, and <<extend>> types of relationship.



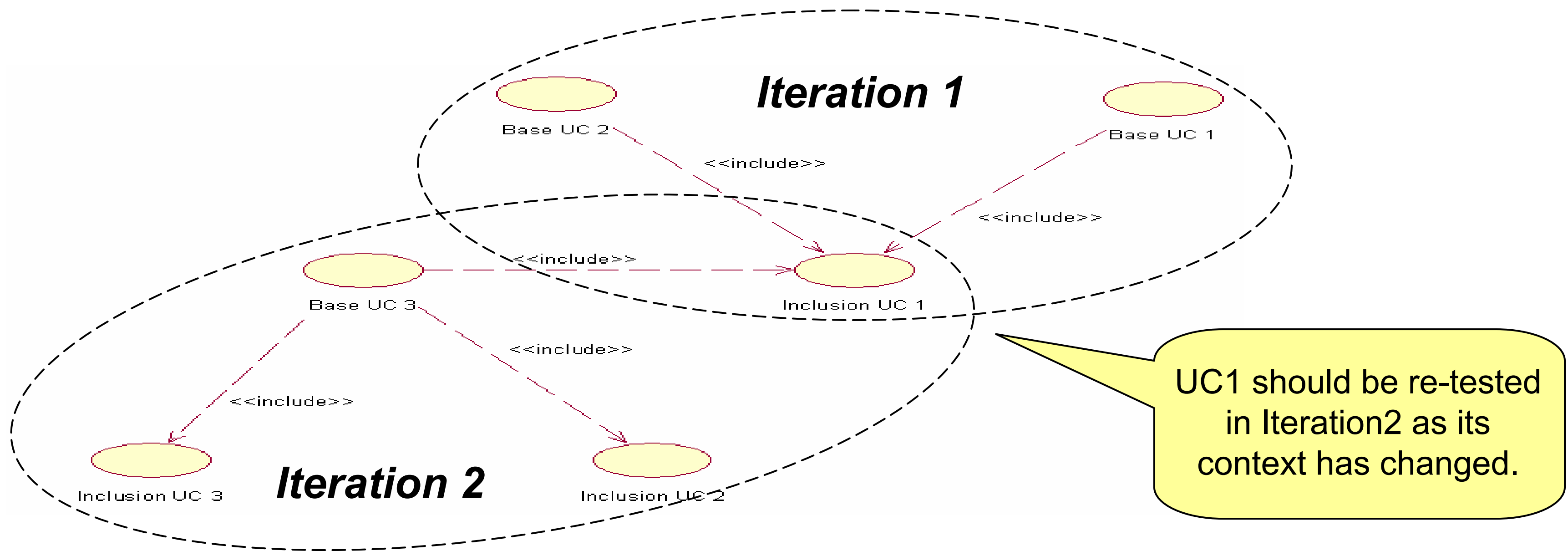
Analyzing the use-case model structure can help testers make better decisions in test planning and test design and it should always be performed before generating test cases.

© 2005 by Valley Forge Consulting, Inc.

Specific #4: Test Planning Tips

Planning the Scope of Functional Testing

- Each new or modified use case should be added to the scope of testing.
- When a new base use case (e.g, Base 3) is added in a new project iteration, the scope of testing may also include re-testing any of its inclusions (e.g., Inclusion 1) along with the inclusion's other bases (Base 1, 2). This test might be needed to validate the concurrency issue between the new base (UC3) and the other bases (UC1,2) previously tested.



Specific #4: Test Planning Tips (cont'd)

Planning the Project Resource Allocation

When use cases relate to each other via the <<include>> or <<extend>> relationships, their test designs will also be related. To ensure no gaps exist in the test coverage of related use cases, plan to assign the same test analyst to cover the related use cases.

Planning the Testing Priority

Identify inclusion use cases that have many bases. Schedule their testing early in the cycle as they are on the critical path for testing their bases.

Planning the Smoke Test

Identify base use cases that have many inclusions and capture large chunks of functionality. They could be good candidates for performing the Smoke Test.

Estimating the Test Effort

- When estimating the test effort for an inclusion use case with multiple bases, always analyze the inclusion in the context of its bases. Even a simple inclusion use case could result in a significant testing effort when it has multiple bases.
- Likewise, when estimating the test effort for a base use case with multiple inclusions, analyze the whole cluster of related use cases first.

© 2005 by Valley Forge Consulting, Inc.

Specific #4: Test Design Tips

Test Design for a Base Use Case with Multiple Inclusions

When a base use case has a scenario with multiple inclusions, consider a quality concern validating the end-to-end flow of events for the base use case scenario and its inclusions.

Test Design for an Inclusion Use Case with Multiple Bases

When a given inclusion has multiple bases, consider the following quality concerns:

- ❑ First, validate that each base, one at a time, can execute the inclusion functionality;
- ❑ Second, analyze whether the bases can compete for the same resource manipulated by the inclusion. In this case, plan a concurrency test.

To efficiently analyze the use-case model structure, a list of use cases and their relationships can be maintained in a database, for example, by using tools like DOORS or MS Access.

© 2005 by Valley Forge Consulting, Inc.

Specific #5: Difference in Approaches to Testing Concrete vs. Abstract Use Cases

- While designing or refactoring a use-case model, developers can identify fragments of the system behavior common to multiple use cases and present them as separate inclusion/extension use cases;
- **Drawback** - these fragments of reusable system behavior may not have their own actors who can instantiate their scenarios. Also, such fragments may not have a flow of events; for example, they may only capture a complex system response common to a number of base use cases. In this case, they are called – ***abstract use cases***;
- Ivar Jacobson wrote – “Neither extension use-case fragments, nor inclusion use-case fragments are really use cases, and they should not be treated as use cases” [*“Use cases yesterday, today, and tomorrow”*];
- On complex projects inclusion/extension fragments (abstract use cases) can compose up to 20-30% of the use-case model.

Software testers should understand the difference in testing approaches between concrete and abstract use cases.

Specific #5: Tips & Guidelines

When we discussed the test analysis technique for use cases, we talked about decomposing use cases into ***vertical slices.*** Such a decomposition can be performed differently for concrete and abstract use cases.

Two Formal Techniques to Slice Use Cases

- A concrete use case has a flow of events, hence, the ***control-flow analysis*** technique can be used to analyze and decompose its functionality;
- In contrast, an abstract use case may not have a flow of events; in this case the ***data-flow analysis*** technique can be used to analyze and decompose its functionality;
- Originally, the control-flow and data-flow analysis techniques were developed for the purposes of white-box testing and they were used for designing test cases based on analysis of the program structure.

> The control-flow and data-flow analysis techniques can also be effective in black-box testing of use cases.

An Overview of the Two Techniques

Control-flow Analysis

Purpose – focuses on analyzing the program structure, presented as a control-flow graph, to select test cases that execute every statement, branch, or path of the program's structure.

The test design **Completeness Criteria** can include:

- Testing all statements *(weak criterion)*
- Testing all branches *(stronger criterion)*
- **Testing all paths** *(strongest criterion)*

Data-flow Analysis

Purpose – focuses on analyzing the variables defined and used in the program to select test cases based on the data *def-use* relations. It considers two types of data use – a predicate use (*p-use*) and a computation use (*c-use*).

The test design **Completeness Criteria** can include:

- Testing all data definitions (All-DEFs, *weak criterion*)
- Testing all predicates (All-P-Uses, *stronger criterion*)
- Testing all calculations (All-C-Uses, *stronger criterion*)
- **Testing all definition-use paths** (All-DEF-Use-paths, *strongest criterion*)

(source – I. Burnstein "Practical Software Testing", Springer, 2003)

© 2005 by Valley Forge Consulting, Inc.

Specific #5: Tips & Guidelines (cont'd)

Applying the Control-flow Analysis to Concrete Use Cases

The following publications discuss the use-case decomposition based on control-flow analysis:

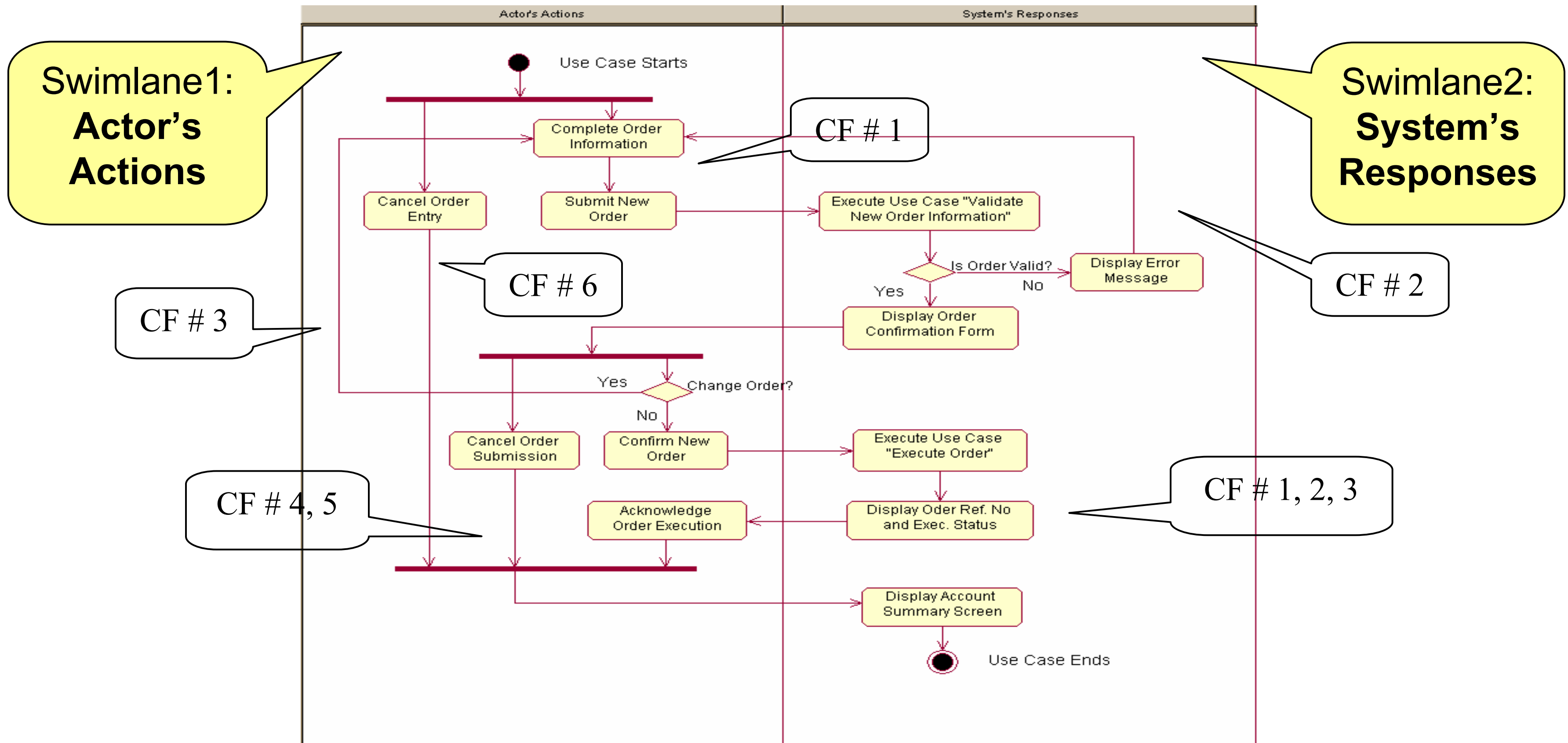
- I. Jacobson, P. Ng "Aspect-Oriented Software Development with Use Cases", Addison-Wesley, 2005
- D. Leffingwell, D. Widrig "Managing Software Requirements", Addison-Wesley, 2003
- R. Collard "Developing Test Cases From Use Cases", *Testing and Quality Engineering magazine*, July, 1999

Control-flow Analysis Steps

1. Create a control-flow graph or an activity diagram that presents all use case steps, including the normal, alternative, and exception courses of events;
2. Identify all vertical slices, i.e., end-to-end paths through the use case logic and describe each path as a separate testing scenario;
3. For each vertical slice, identify test cases and test data values.

Specific #5: Tips & Guidelines (cont'd)

Example Use Case "Place an Equity Trade Order"

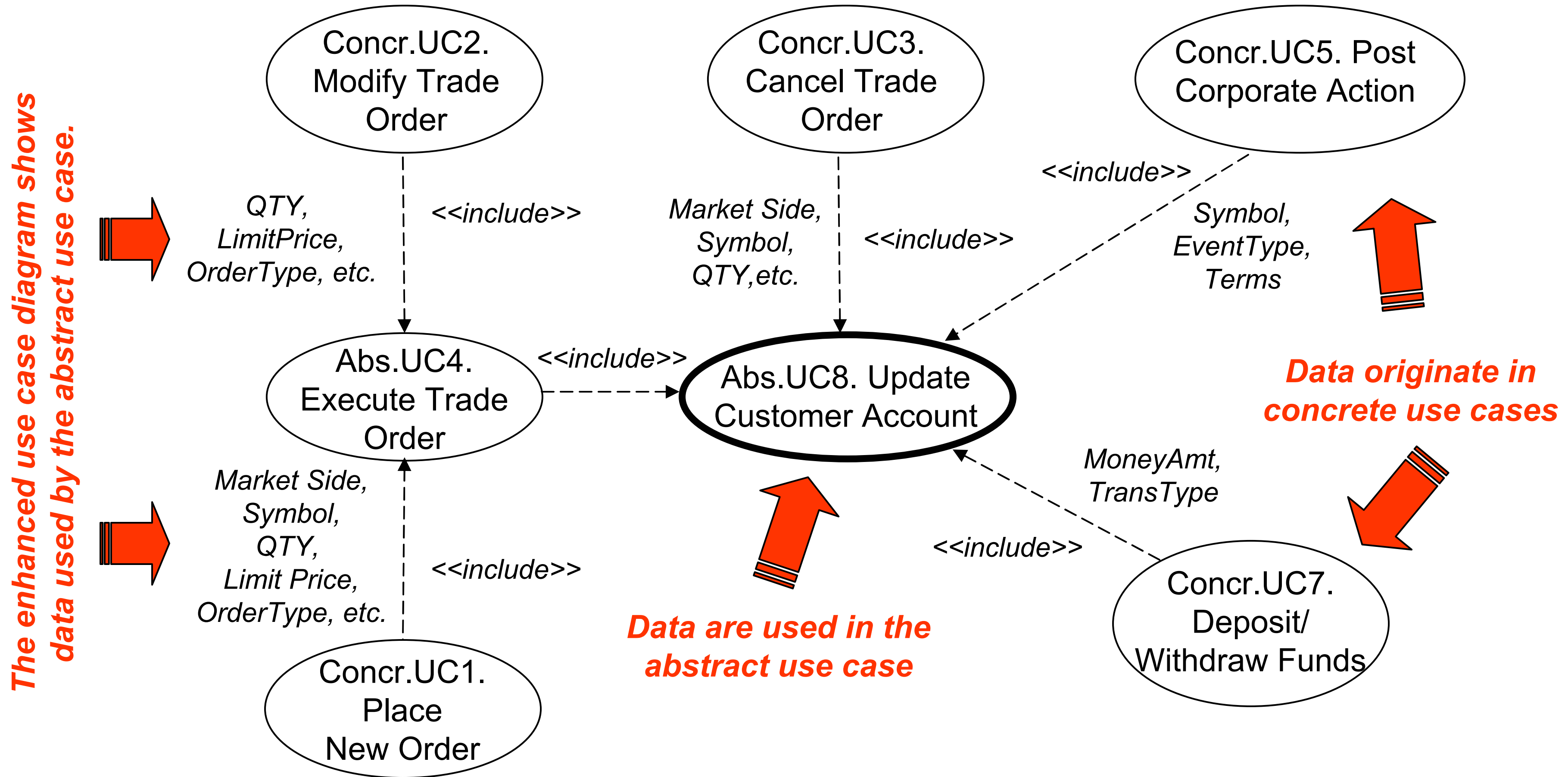


Each identified control flow, i.e., vertical slice presents a quality concern to be tested.

© 2005 by Valley Forge Consulting, Inc.

Specific #5: Tips & Guidelines (cont'd)

Applying Data-flow Analysis to Abstract Use Cases



© 2005 by Valley Forge Consulting, Inc.

Specific #5: Tips & Guidelines (cont'd)

Data-flow Analysis Steps

1. Identify all of the concrete bases of a given abstract use case. Some of the immediate bases can also be abstract (for example UC 4, Slide 32). In this case, look at the next level up in the use case hierarchy until you find all concrete bases;
2. Analyze the data types that are used in a given abstract use case and where these types originate in concrete use cases (DEF points);
3. Analyze how these data are used by the abstract use case. There could be two possibilities:
   a) the received data can be used as part of conditions in the use-case description (P-USE); and
   b) the received data can be used in use-case calculations (C-USE);
4. Identify vertical slices as all paths between the points of origin of the data in the base use cases and all of the points where the data are used in the given abstract use case (All DEF-P-Use-C-Use coverage);
5. For each vertical slice, identify test cases and test data values.

© 2005 by Valley Forge Consulting, Inc.

Presentation Summary

- Use-case-driven testing has important specifics compared with conventional requirements-based testing. In this presentation we discussed some of these specifics from the ***System Test*** perspective.
- The main point of this presentation was that performing ***test analysis*** becomes a critical activity when testers deal with complex use-case models; and test analysis should always be performed before generating test cases from use cases.
- We discussed how the test process can be decomposed into and managed as a collection of continuous workflows. These workflows deliver three test models - Test Analysis, Test Design, and Test Implementation – that can help testers better handle the complexity of use-case models.
- In the presentation we discussed the steps to perform test analysis for individual use cases and the document type – ***Test Analysis Specification*** – that can be used to capture the test analysis results. This document, developed for each use case, becomes a basis for test effort estimation and a road-map for detailed test design.
- We discussed the difference in testing approaches between concrete and abstract use cases. When decomposing a use case into its vertical slices, the control-flow analysis technique can be used for concrete use cases and the data-flow analysis technique can be used for abstract use cases.