

Module 1

Software Engineering Practices



Principles of Software Testing for Testers

Module 1: Software Engineering Practices

(Some things Testers should know about them)

Topics

| | |
|--|------|
| Objectives | 1-2 |
| Software Development Problems | 1-3 |
| Six Software Engineering Practices | 1-6 |
| Software Engineering Process and Practices | 1-28 |
| Review | 1-32 |

Objectives

Objectives

- ◆ Identify some common software development problems.
- ◆ Identify six software engineering practices for addressing common software development problems.
- ◆ Discuss how a software engineering process provides supporting context for software engineering practices.

In this module, we explore a number of software engineering practices and explain why these are considered to be good practices to follow. We will also look at how a software engineering process helps you to implement these and many other engineering practices.

Software Development Problems

Module 1 - Content Outline (Agenda)

- ➔ Software development problems
- ◆ Six software engineering practices
- ◆ Supporting software engineering practices with process

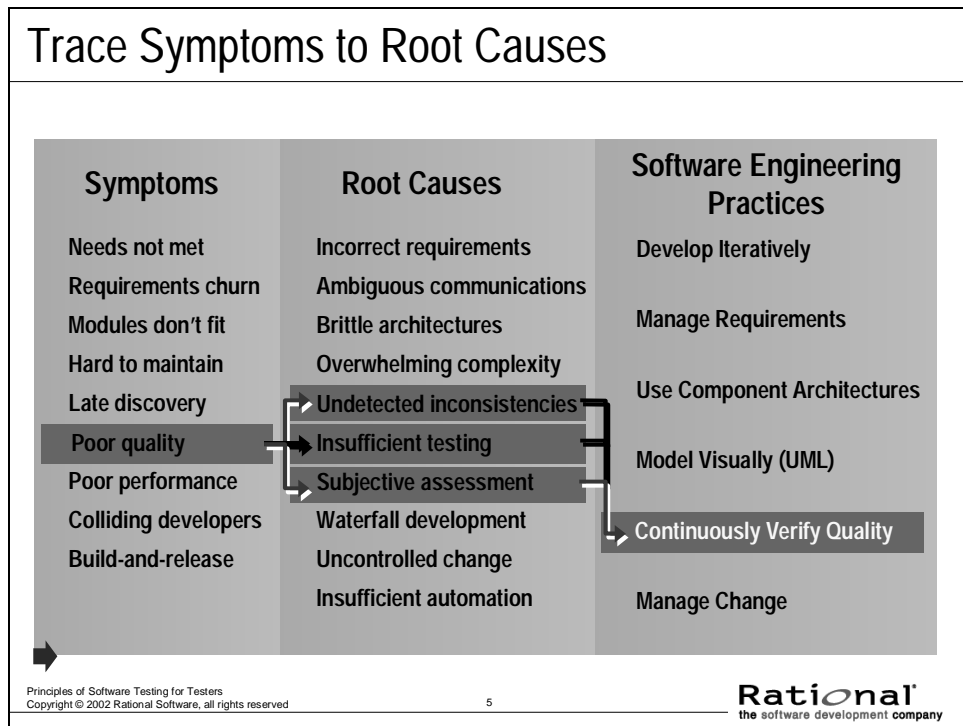
In this section, we describe some common software development problems and their root causes.

Symptoms of Software Development Problems

Symptoms of Software Development Problems

- × **User or business needs not met**
- × **Requirements churn**
- × **Modules don't integrate**
- × **Hard to maintain**
- × **Late discovery of flaws**
- × **Poor quality or poor user experience**
- × **Poor performance under load**
- × **No coordinated team effort**
- × **Build-and-release issues**

Trace Symptoms to Root Causes



Treat these root causes, and you'll eliminate the symptoms. Eliminate the symptoms, and you'll be in a much better position to develop quality software in a repeatable and predictable fashion.

The software engineering practices listed here are approaches to developing software that have been commercially-proven. When used in combination, they strike at many of the common root causes of software development problems. These are also referred to as "best practices," not so much because we can precisely quantify their value, but rather because they are observed to be the common practices adopted by successful organizations.

These software engineering practices have been identified by observing thousands of customers on thousands of projects and they align with similar observations made by independent industry experts*.

*(CHAOS Report ©1999, The Standish Group International).

Six Software Engineering Practices

Module 1 - Content Outline (Agenda)

- ◆ Software development problems
- ➔ Six software engineering practices
- ◆ Supporting software engineering practices with process

In this section, we describe some commonly recommended software engineering practices.

Practice 1: Develop Iteratively

Practice 1: Develop Iteratively

Software Engineering Practices

Develop Iteratively
Manage Requirements
Use Component Architectures
Model Visually (UML)
Continuously Verify Quality
Manage Change

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved

7

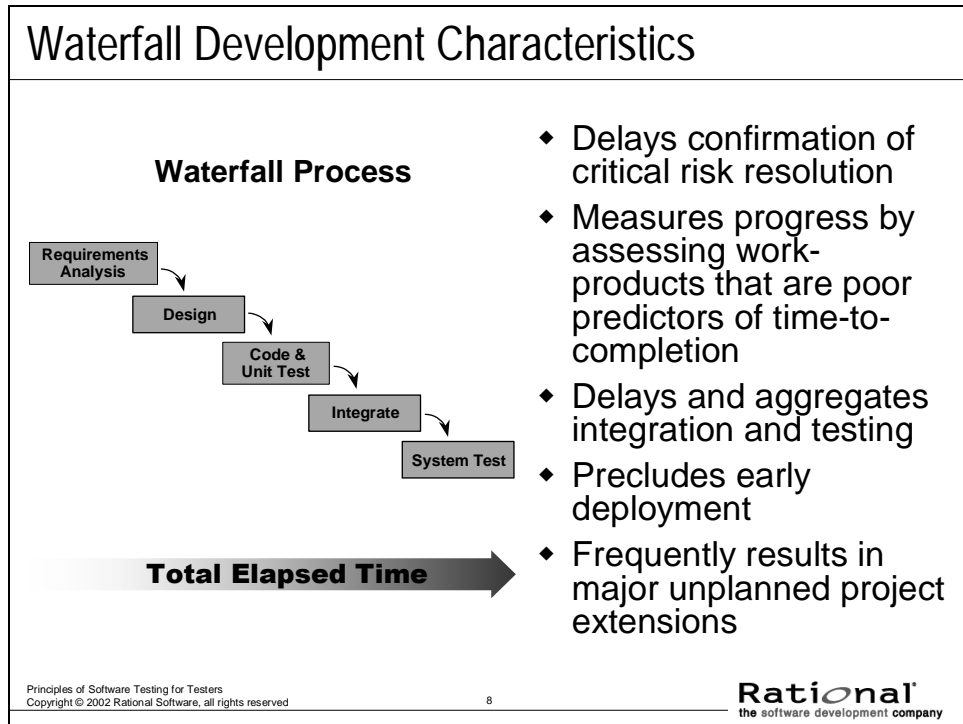
Rational
the software development company

Developing iteratively is a technique that is used to deliver the functionality of a system in a successive series of releases of increasing completeness. Each release is developed in a specific, fixed time period called an “iteration.”

Each iteration is focused on identifying, defining and analyzing some set of requirements, and designing, building and testing software based on the understanding of those requirements.

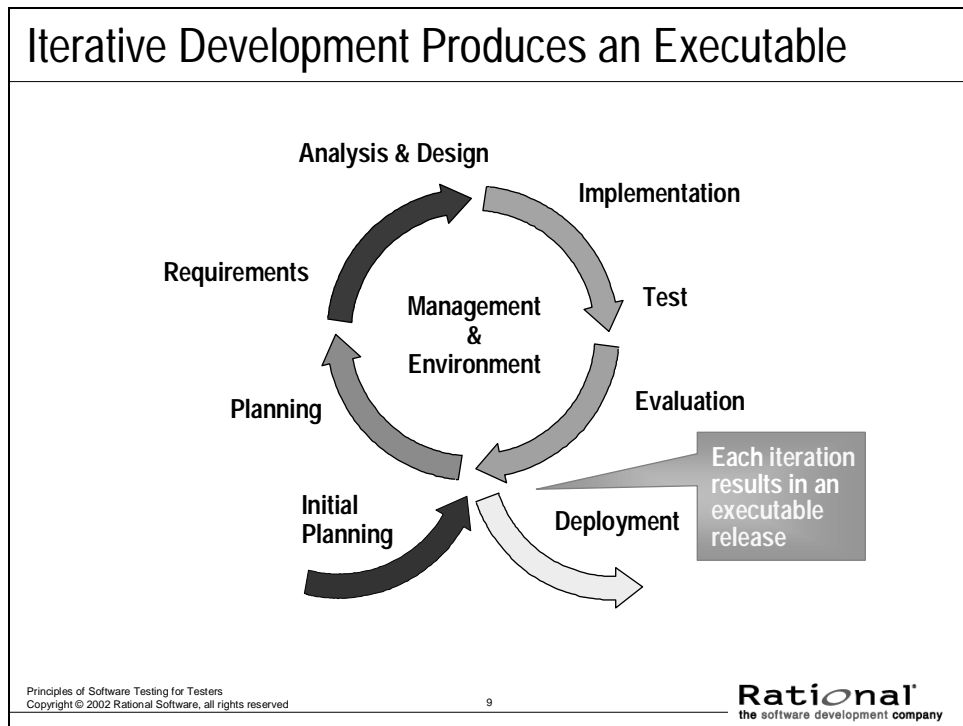
If you want to learn more about how software can be developed iteratively, you can take the *Rational Unified Process Fundamentals* course.

Waterfall Development Characteristics



Waterfall is conceptually straightforward because it produces a single deliverable. The fundamental problem of this approach is that it pushes risk forward in time, where it's costly to undo mistakes from earlier phases. An initial design will likely be flawed with respect to its key requirements, and furthermore, the late discovery of design defects tends to result in costly overruns and/or project cancellation. The waterfall approach tends to mask the real risks to a project until it is too late to do anything meaningful about them.

Iterative Development Produces an Executable



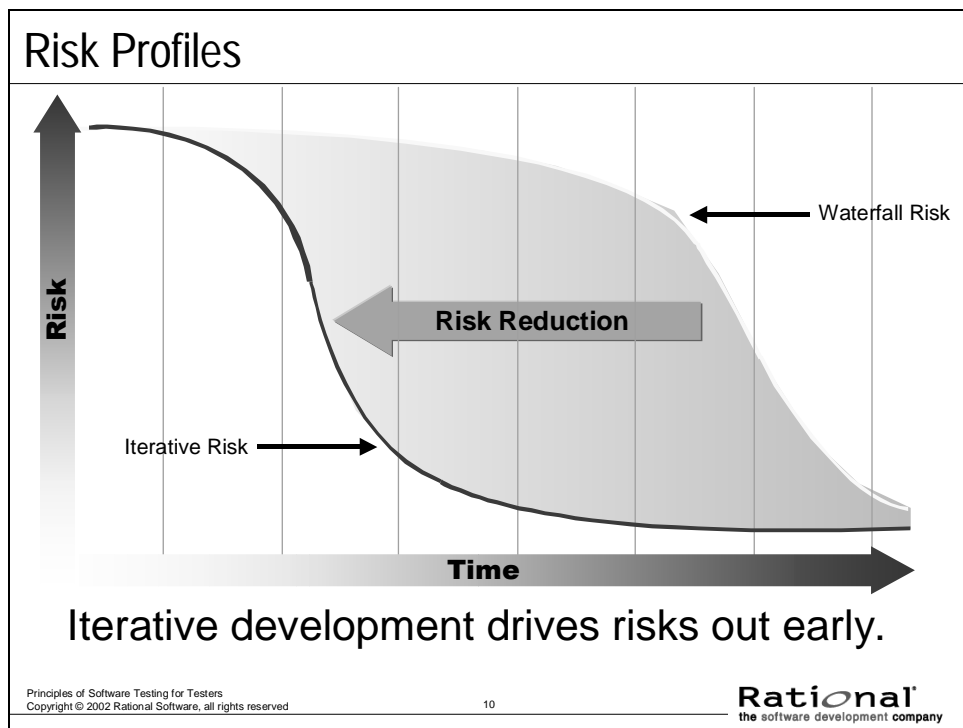
The earliest iterations address greatest risks. Each iteration produces an executable release. Each iteration includes integration and test. Iterations help to:

- resolve major risks before making large investments
- enable early objective feedback
- make testing and integration continuous
- focus the project on achievable short-term objective milestones
- make it possible to deploy partial implementations of the completed final system

Iterative processes were developed to address the problems with the waterfall discussed on the previous slide. With an iterative process, the phase concerns of the waterfall process are addressed in each iteration (although not typically in sequence, usually somewhat more in parallel). Instead of developing the whole system in lock step, an increment (i.e. a subset of system functionality) is selected and developed, then another increment, etc.

The selection of each increment to be developed is based on its potential to address key risks, the highest priority risks being addressed first. To address the selected risk(s), a subset of use cases or use-case instances are selected. The minimal set of use-case instances are realized (developed) that will allow objective verification (i.e., through a set of executable tests) of the risks that you have chosen to address. The next increment addresses the next highest risks, and so on.

Risk Profiles



Iterative development produces the architecture first, allowing integration to occur “as the verification activity” of the design phase, and allowing design flaws to be detected and resolved earlier in the lifecycle. Continuous integration throughout the project replaces the big bang integration at the end of a project.

Iterative development also provides much better insight into quality, because system characteristics that are largely inherent in the architecture (e.g., performance, fault tolerance, maintainability) are tangible earlier in the process. Thus, issues are still correctable without jeopardizing target costs and schedules.

Practice 2: Manage Requirements

Practice 2: Manage Requirements

**Software Engineering
Practices**

Develop Iteratively
Manage Requirements
**Use Component
Architectures**
Model Visually (UML)
**Continuously Verify
Quality**
Manage Change

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved

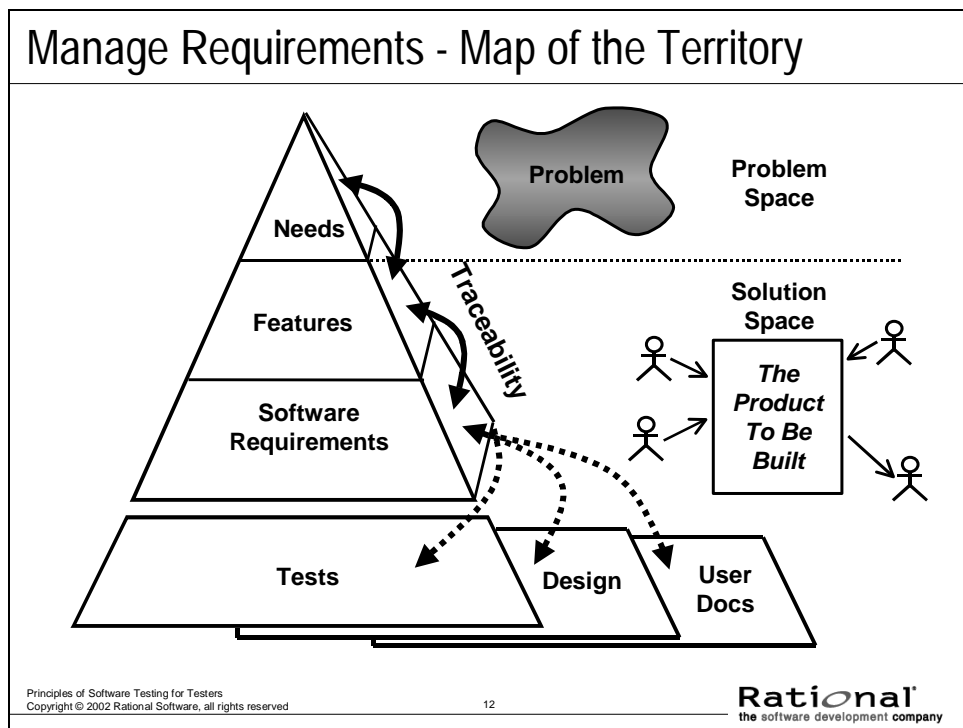
11

Rational
the software development company

A report from the Standish Group confirms that a distinct minority of software development projects is completed on-time and on-budget. In their report, the success rate was only 16.2%, while challenged projects (operational, but late and over-budget) accounted for 52.7%. Impaired projects (canceled) accounted for 31.1%. These failures are attributed to poor requirements management, incorrect definition of requirements from the start of the project, and poor requirements management throughout the development lifecycle. (Source: Chaos Report, <http://www.standishgroup.com>).

If you want to learn more about how to manage requirements, you can take the *Requirements Management with Use Cases* course.

Manage Requirements - Map of the Territory



Managing requirements involves the translation of stakeholder requests into a set of key stakeholder needs and system features. These in turn are detailed into specifications for functional and non-functional requirements. Detailed specifications are translated into a design, user documentation and tests.

The requirements for the software are a key input to testing. You will often find important problems at the boundary between each section of the pyramid – for example, are the needs appropriately reflected in the features? Does the design appropriately reflect the requirements? Later in this course we will discuss testing based on requirements.

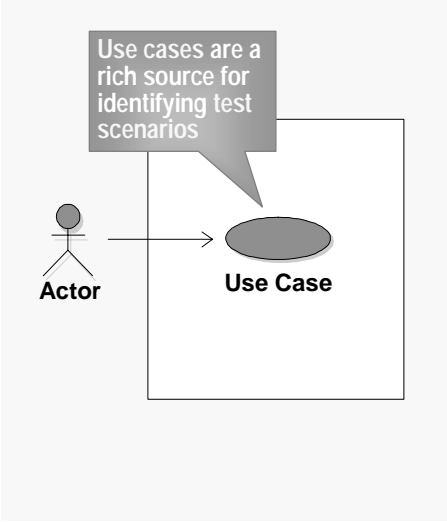
To help manage the relationship between the requirements and the tests derived from those requirements, you can establish traceability relationships between those elements. Traceability assists us to do many things, including:

- Assess the project impact of a change in a requirement
- Assess the impact of a failure of a test on requirements (i.e., if test fails, the requirement may not be satisfied)
- Manage the scope of the project
- Verify that all requirements of the system are fulfilled by the implementation
- Verify that the application does only what it was intended to do
- Manage change

Later in this course we will discuss traceability and assessment needs for testing.

Manage Requirements - Use-Case Concepts

Manage Requirements - Use-Case Concepts



Use cases are a rich source for identifying test scenarios

Actor → Use Case

An **actor** represents a person or another system that interacts with the system.

A **use case** defines a sequence of actions a system performs that yields a result of observable value to an actor.

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved
13
Rational
the software development company

Use Cases represent a technique for defining requirements in a way that focuses on the end-user goal. They have been popularized by iterative development processes such as the Rational Unified Process. However, the technique is not specific to iterative development – it can be applied just as well to eliciting and managing requirements in a waterfall development lifecycle.

An **Actor**:

- is not part of the system. It represents a role that users of the system will play when interacting with it.
- can actively interchange information with the system.
- can be a passive recipient of information.
- can be a giver of information.
- can represent a human, a machine or another system.

A **Use Case**:

- specifies a dialogue between an actor and the system.
- is initiated by an actor to invoke certain functionality in the system.
- is a collection of meaningful, related flows of events.
- yields a result of observable value.

Taken together, all use cases provide a high-level, external view of all possible ways of using the system.

Practice 3: Use Component Architectures

Practice 3: Use Component Architectures

**Software Engineering
Practices**

**Develop Iteratively
Manage Requirements
Use Component
Architectures
Model Visually (UML)
Continuously Verify
Quality
Manage Change**

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved

14

Rational
the software development company

Software architecture is the development product that gives the highest return on investment with respect to quality, schedule, and cost, according to the authors of *Software Architecture in Practice* (Len Bass, Paul Clements & Rick Kazman [1998] Addison-Wesley). The Software Engineering Institute (SEI) has an effort underway called the Architecture Tradeoff Analysis (ATA) Initiative to focus on software architecture, a discipline much misunderstood in the software industry. The SEI has been evaluating software architectures for some time and would like to see architecture evaluation in wider use. By performing architecture evaluations, AT&T reports a 10% productivity increase (from news@sei, Vol. 1, No. 2).

If you want to learn more about the use of component architectures in software development, you can take the *Object Oriented Design with UML* or *Principles of Architecting Software* courses.

Resilient Component-Based Architectures

Resilient Component-Based Architectures

- ◆ Resilient
 - Meets current and future requirements
 - Improves extensibility
 - Enables reuse
 - Encapsulates system dependencies
- ◆ Component-based
 - Reuse or customize components
 - Select from commercially available components
 - Evolve existing software incrementally

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved

15

Rational
the software development company

Architecture is an aspect of design. It is about making decisions on how the system will be built. But it is not all of the design. It stops at the major abstractions, or in other words, the elements that have some pervasive and long-lasting effect on the system's performance and ability to evolve.

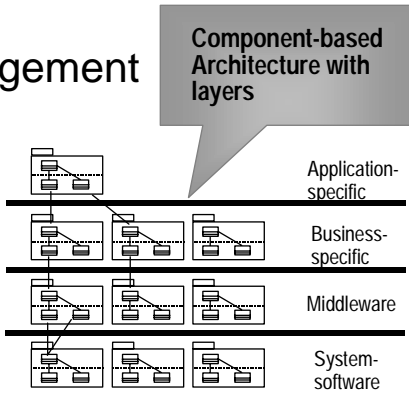
A software system's architecture is perhaps the most important aspect that can be used to control the iterative and incremental development of a system throughout its lifecycle.

The most important property of an architecture is resilience -- flexibility in the face of change. To achieve it, architects must anticipate evolution in both the problem domain and implementation technologies to produce a design that can gracefully accommodate such changes. Key techniques are abstraction, encapsulation, and object-oriented analysis and design. The result is that applications are fundamentally more maintainable and extensible.

Purpose of a Component-Based Architecture

Purpose of a Component-Based Architecture

- ◆ **Basis for reuse**
 - Component reuse
 - Architecture reuse
- ◆ **Basis for project management**
 - Planning
 - Staffing
 - Delivery
- ◆ **Intellectual control**
 - Manage complexity
 - Maintain integrity



The diagram illustrates a four-layer architecture. At the top is the 'Application-specific' layer, followed by 'Business-specific', 'Middleware', and 'System-software' at the bottom. Each layer contains several component icons connected by lines, representing dependencies. A callout box points to the top layer with the text 'Component-based Architecture with layers'.

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved

16

Rational
the software development company

Definition of a (Software) Component:

Process Definition: A non-trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces.

UML Definition: A physical, replaceable part of a system that packages implementation, and conforms to and provides the realization of a set of interfaces. A component represents a physical piece of implementation of a system, including software code (source, binary or executable) or equivalents such as scripts or command files.

Your testing will typically be constrained by and dependent on the delivery and availability of software components. As noted on the slide, planning and staffing of the project will often be based around components, so your test plans will most likely need to reflect this as well.

Components also help to manage complexity by hiding (or encapsulating) unnecessary detail, making it easier to discuss how basic and fundamental interaction occurs between components at different levels of detail (or abstraction). This will assist in gaining an understanding of how the software is designed to work, and will help you to reason about useful tests to conduct.

In some cases components will be acquired from third-party suppliers or reused from other projects within your organization. This poses some interesting potential problems and challenges (as well as opportunities) for testing software systems built using previously developed components.

Practice 4: Model Visually (UML)

Practice 4: Model Visually (UML)

Software Engineering Practices

Develop Iteratively
Manage Requirements
Use Component Architectures
Model Visually (UML)
Continuously Verify Quality
Manage Change

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved

17

Rational
the software development company

A **model** is a simplification of reality that provides a complete description of a system from a particular perspective. We build models so that we can better understand the system we are modeling. We build models of complex systems because we cannot comprehend any such system in its entirety.

If you want to learn more about visual modeling in software development, you can take the *Fundamentals of Visual Modeling with UML* course.

Why Model Visually?

Why Model Visually?

- ◆ To help manage complexity
 - To capture both structure and behavior
 - To show how system elements fit together
 - To hide or expose details as appropriate
- ◆ To keep design and implementation consistent
- ◆ To promote unambiguous communication
 - UML provides one language for all practitioners

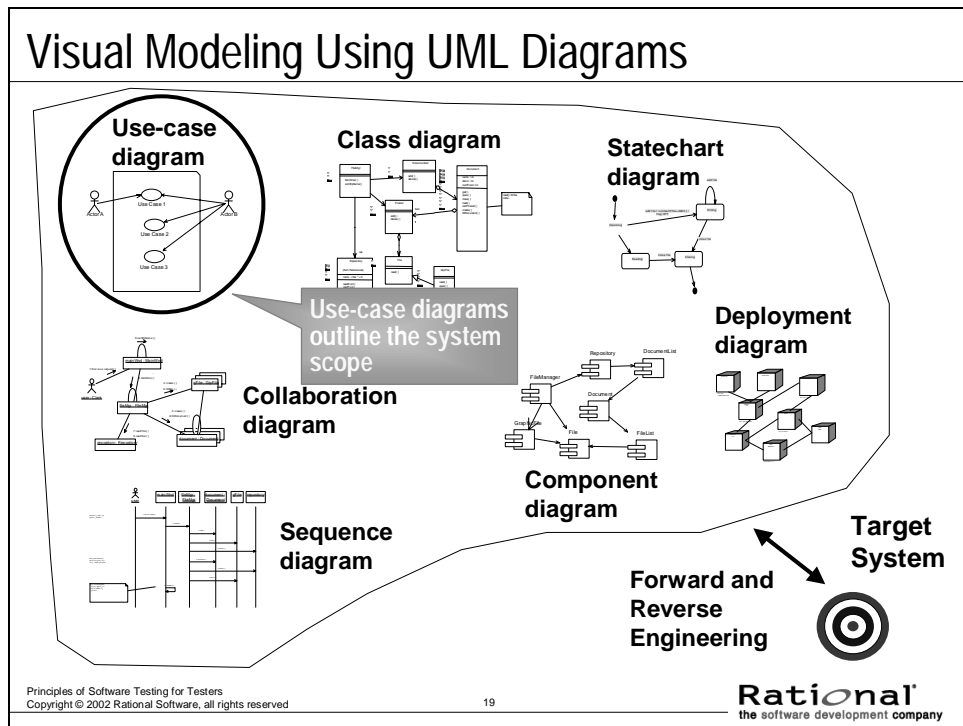
Modeling is important because it helps the development team visualize, specify, construct, and document the structure and behavior of a system's architecture. Using a standard modeling language such as the UML (the Unified Modeling Language), different members of the development team can communicate their decisions unambiguously to one another.

Using visual modeling tools facilitates the management of these models, letting you hide or expose details as necessary. Visual modeling also helps you maintain consistency among a system's artifacts: its requirements, designs, implementations and tests. In short, visual modeling helps improve a team's ability to manage software complexity.

Different techniques can be used to verify aspects of a model prior to the physical implementation of program code associated with the model. The UML itself provides rules to establish whether a model is "well-formed", and various software is commercially available to "walk the model" looking for anomalies. That software can typically be extended with user-defined rules.

Tools are also becoming available that take UML models as input and allow the generation of test assets based on those models. For more information about these tools, you might want to look at the *Rational Quality Architect* product.

Visual Modeling Using UML Diagrams

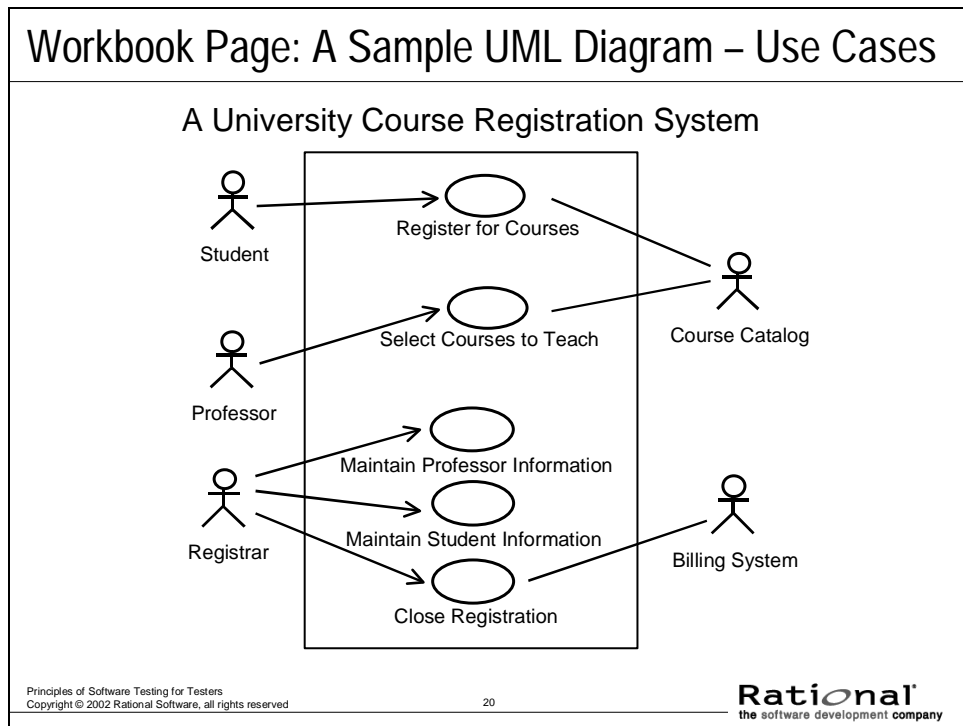


Visual modeling with the UML makes an application’s architecture tangible, permitting us to assess it in multiple dimensions. How portable is it? Can it exploit expected advances in parallel processing? How might we modify it to support a family of applications? We’ve discussed the importance of architectural resilience and quality. The UML enables us to evaluate these key characteristics during early iterations -- at a point when design defects can be corrected before threatening project success.

If your software development team will be making use of visual models, it is worth taking some time to learn how to read, interpret and discuss these models. You will find this assists your communication with the developers, and offers you new and unique insight into what the software is designed to do. In turn, this will help you reason more completely about the appropriate tests that you should conduct.

Advances in forward and reverse engineering techniques permit changes to an application’s model to be automatically reflected in its source code, and changes to its source code to be automatically reflected in its model. This is critical when using an iterative process, where we expect such changes with each iteration.

Workbook Page: A Sample UML Diagram – Use Cases



Often a global use-case diagram will be included in the Use-Case-Model Survey to give a graphical overview of the system.

This should include all use cases, actors, and their relationships that cover the scope of the system being built.

Use case diagrams are used to show the existence of use cases and their relationships, both to each other and to actors. An actor is something external to the system that has an interface with the system, such as end users. A use case models a dialogue between actors and the system. A use case is initiated by an actor to invoke a certain functionality in the system. For example, in the diagram above, one class of user of the system is student. In this system, students have a goal to use the system to register for courses. Hence, Register for Courses is a use case.

The arrow (which is optional) indicates the direction in which messages are invoked in the interaction. Here, the Student actor sends messages to the Register for Courses use case.

A use case is a complete and meaningful flow of events. The flow of events supplements the use case diagram and is usually provided in text format.

Taken together, all use cases constitute all possible ways of using the system.

Practice 5: Continuously Verify Quality

Practice 5: Continuously Verify Quality

Software Engineering Practices

Develop Iteratively
 Manage Requirements
 Use Component Architectures
 Model Visually (UML)
Continuously Verify Quality
 Manage Change

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved

21

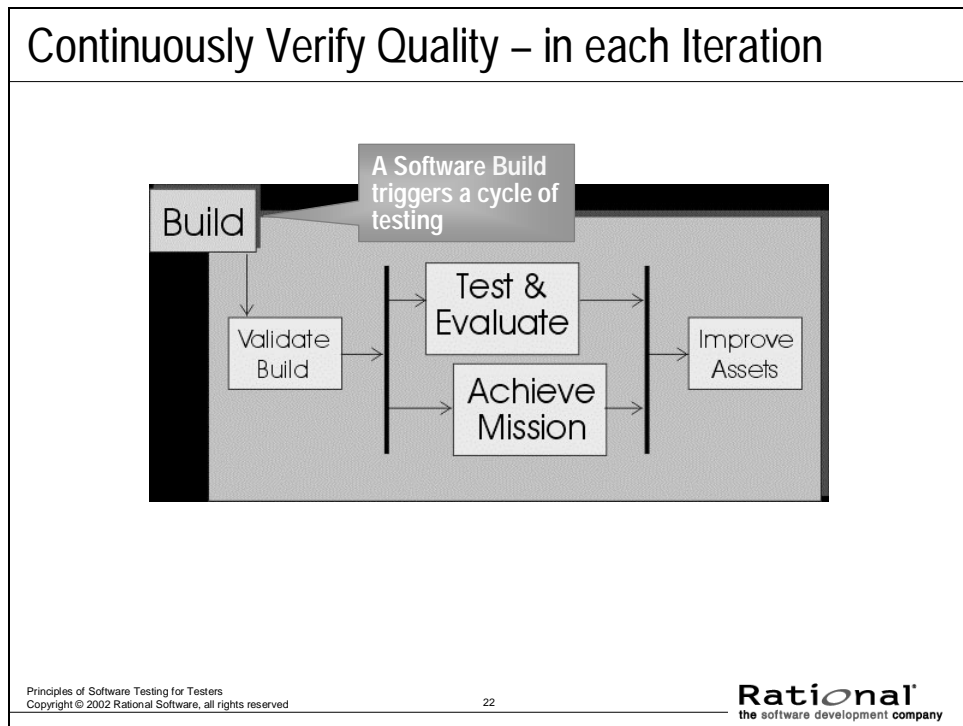
Rational

the software development company

Quality has many definitions. We'll discuss some of the more commonly held opinions about quality in a subsequent module. However, it is fair to state that achieving quality is not simply about "meeting requirements" or producing a product that meets user needs and expectations. Quality also includes identifying the measures and criteria (to demonstrate the achievement of quality), and the implementation of a process to ensure that the resulting product has achieved an appropriate degree of quality.

Software testing is an important aspect of Software Quality process. Software testing accounts for 30% to 50% of software development costs in many organizations, yet most people believe that software is not well-tested before it is delivered. This contradiction is arguably rooted in two interesting observations. First, testing software is enormously difficult. The different ways a given program can behave are almost infinite. Second, testing is typically done without a clear methodology and without adequate supporting tools. While the complexity of software makes "complete" testing an impossible goal, an appropriate methodology for the project context, and use of appropriate supporting tools, can help to improve the productivity and effectiveness of the software testing effort.

Continuously Verify Quality – in each Iteration

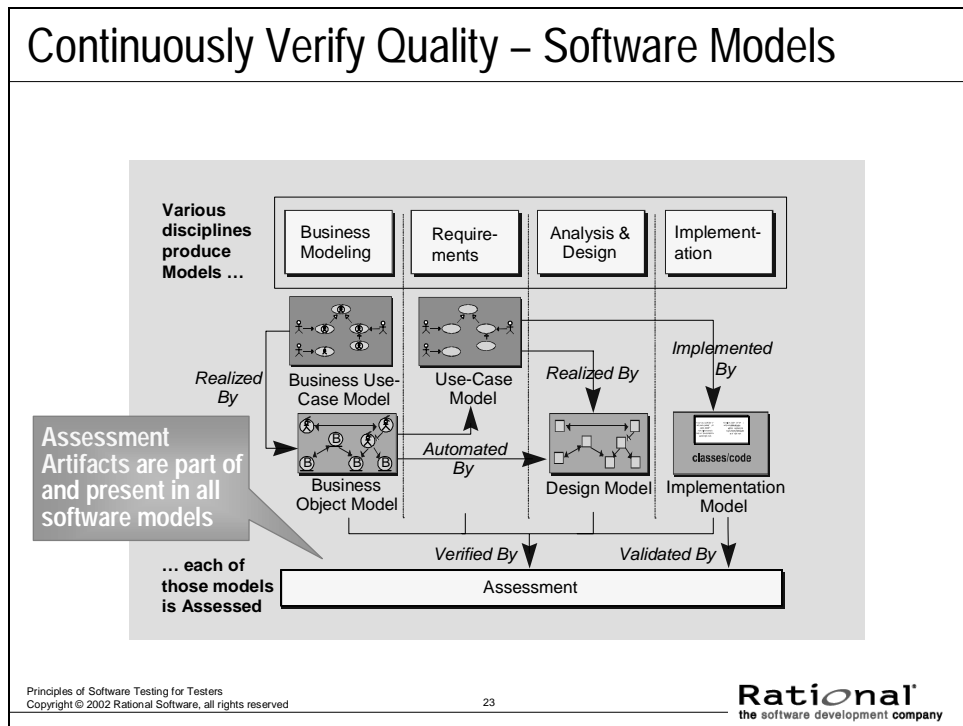


In traditional software development, there is a tendency to delay certain types of assessment—such as black-box testing—until late in the development cycle. In the case of black-box testing, delaying these tests will delay the discovery of potentially important problems until late in the development cycle. In many cases this will mean the problems are too expensive to correct.

In iterative development, assessment activities are an integral part of the effort in each iteration: they are needed to provide objective proof that the goals of the iteration have been met. Without iteration-based assessment, it isn't possible to objectively evaluate whether an iteration achieved its goals: Were the key risks addressed satisfactorily? Were the planned features delivered? Did the software exhibit the required quality attributes?

The design and development of tests can be as complex and arduous as developing the software product itself. You can mitigate the risk of expensive problems derailing the testing process by starting early. In general it is best to start testing activities in the same iteration as the first executable software release is planned.

Continuously Verify Quality – Software Models



The UML can be used to produce a number of models that represent various perspectives or views on a software system as it evolves. Several models are useful to fully describe the evolving system, with different software disciplines producing those models. Each model is developed incrementally over multiple iterations.

- The Business Model is a model of what the business processes are and of the business environment. It is primarily used to gain a better understanding of the software requirements in the business context.
- The Use-Case Model is a model of the value the system represents to the external users of the system environment. It describes the “external services” that the system provides.
- The Design Model is a model that describes how the software will “realize” the services described in the use cases. It serves as a conceptual model (or abstraction) of the implementation model and its source code.
- The Implementation Model represents the physical software elements and the implementation subsystems that contain them.

Assessment involves both Verification and Validation activities: verifying that the software product is being built right, and validating that the right software product is being built. This distinction refers to assessing both the appropriateness of the process by which the software product is built (verification) and the appropriateness of the resulting software product that will be delivered to the customer (validation).

Practice 6: Manage Change

Practice 6: Manage Change

**Software Engineering
Practices**

**Develop Iteratively
Manage Requirements
Use Component
Architectures
Model Visually (UML)
Continuously Verify
Quality
Manage Change**

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved

24

Rational
the software development company

As we indicated earlier, we cannot stop change from being introduced into our project. However, we must control how and when changes are introduced into project artifacts, and who introduces the changes. We also must synchronize change across development teams and locations.

Unified Change Management (UCM) is Rational Software's approach to managing change in software system development, from requirements to release.

What Do You Want to Control?

What Do You Want to Control?

- ◆ Changes to enable iterative development
 - Secure workspaces for each worker
 - Parallel development possible
- ◆ Automated integration/build management

The diagram illustrates how Good CM practices help prevent software errors by enabling four key areas: Workspace Management, Parallel Development, Build Management, and Process Integration. A central diamond contains icons for a database, a server, and a report/alert icon. A speech bubble on the left points to the diamond.

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved

25

Rational
the software development company

Establishing secure workspaces for each worker on the project provides isolation from changes made in other workspaces and control of all software artifacts -- models, code, docs, tests etc.

A key challenge to developing software-intensive systems is the need to cope with multiple workers, organized into different teams, possibly at different sites, all working together on multiple iterations, releases, products, and platforms. In the absence of disciplined control, the development process rapidly degrades into chaos. Progress can come to a stop.

Three common problems that result are:

- Simultaneous update -- When two or more workers separately modify the same artifact, the last one to make changes destroys the work of the former.
- Limited notification -- When a problem is fixed in shared artifacts, some of the workers are not notified of the change.
- Multiple versions -- It is feasible to have multiple versions of an artifact in different stages of development at the same time. For example, one software release is in use by the customer, one is actively being developed and tested, and yet another one is undergoing early prototyping of future features. If a problem is identified in any one of the versions, the fix may need to be propagated among all of them and change control can lead to chaos and halt progress.

Workbook Page: Aspects of a CM System

Workbook Page: Aspects of a CM System

- ◆ Change Request Management (CRM)
- ◆ Configuration Status Reporting
- ◆ Configuration Management (CM)
- ◆ Change Tracking
- ◆ Version Selection
- ◆ Software Manufacture

Change Request Management (CRM) addresses the organizational infrastructure required to assess the cost and schedule impacts of a requested change to the existing product. CRM addresses the workings of a Change Review Team or Change Control Board.

Configuration Status Accounting (Measurement) is used to describe the “state” of the product based on the type, number, rate and severity of defects found, and fixed, during the course of product development. Metrics derived under this aspect, either through audits or raw data, are useful in determining the overall completeness status of the project.

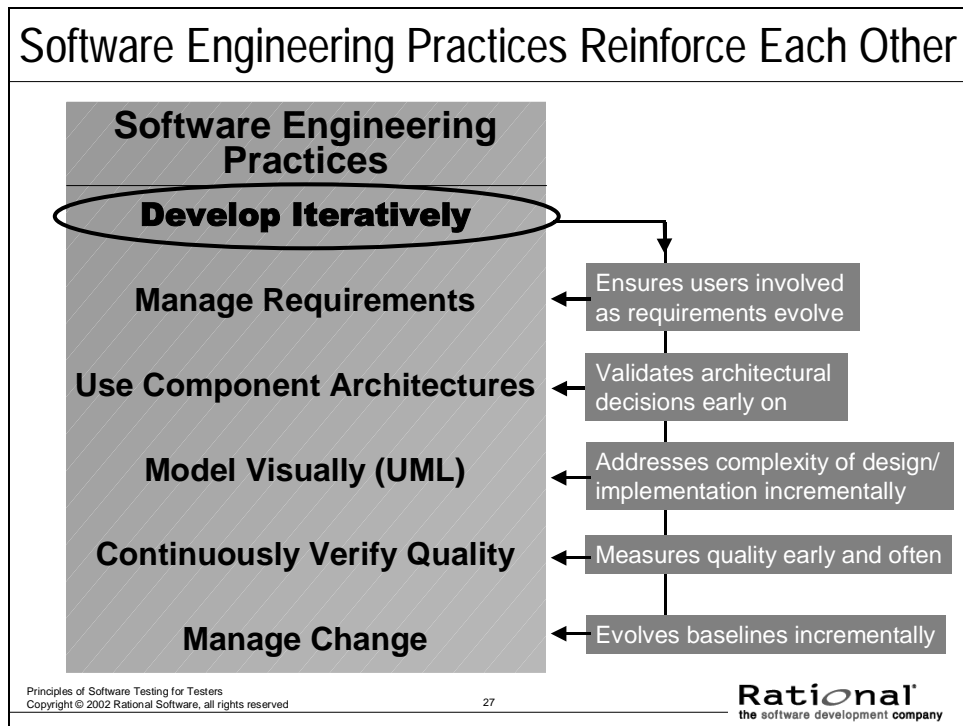
Configuration Management (CM) describes the product structure and identifies its constituent configuration items that are treated as single versionable entities in the configuration management process. CM deals with defining configurations, building and labeling, and collecting versioned artifacts into constituent sets and maintaining traceability between these versions.

Change Tracking describes what is done to components for what reason and at what time. It serves as history and rationale of changes. It is quite separate from assessing the impact of proposed changes as described under “Change Request Management.”

Version Selection ensures that the right versions of configuration items are selected for change or implementation. Version selection relies on a solid foundation of “configuration identification.”

Software Manufacture covers the need to automate the steps to compile, test and package software for distribution.

Software Engineering Practices Reinforce Each Other



In the case of our six software engineering practices, the whole is much greater than the sum of the parts. Each of the practices reinforces and, in some cases, enables the others. The slide shows just one example: how iterative development leverages the other five software engineering practices. However, each of the other five practices also enhances iterative development.

For example, iterative development done without adequate requirements management typically fails to converge on a solution: requirements change at will, users can't agree, and the iterations never reach closure. When requirements are managed, this is less likely to happen. Changes to requirements are visible, and the impact to the development process assessed before they are accepted. Convergence on a stable set of requirements is assured. Similarly, each pair of practices provides mutual support. Hence, although it is possible to use one practice without the others, additional benefits are realized by combining them.

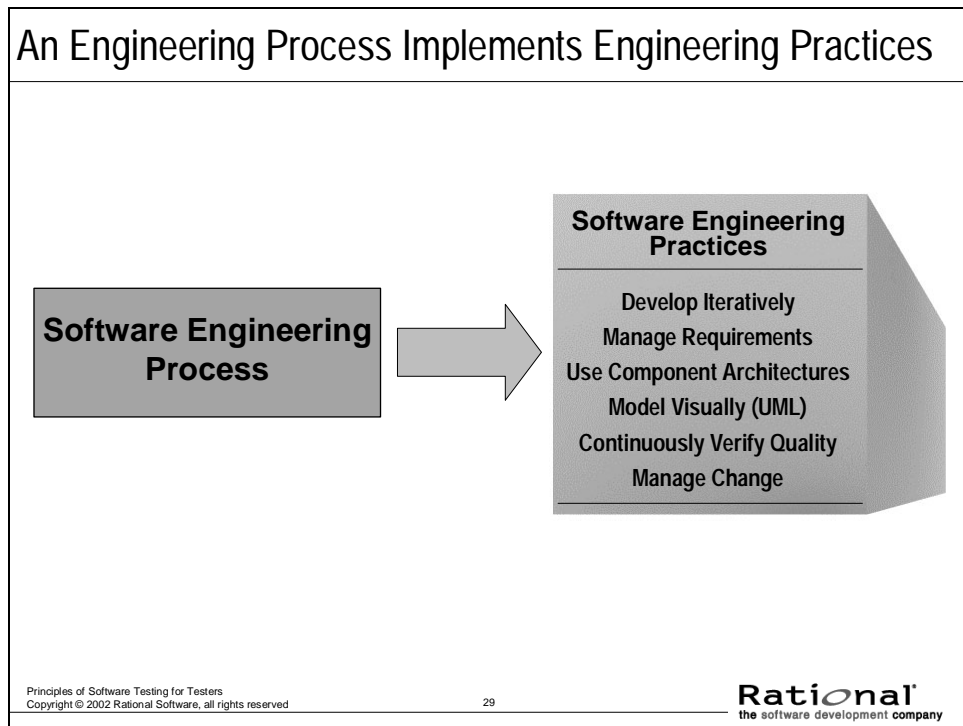
Software Engineering Process and Practices

Module 1 - Content Outline (Agenda)

- ◆ Software development problems
- ◆ Six software engineering practices
- ➔ Software engineering process and software engineering practices

In this section, we briefly discuss how a defined process – such as the Rational Unified Process (RUP) – helps you to implement software engineering practices.

An Engineering Process Implements Engineering Practices



Why have a process?

- Provides guidelines for efficient development of quality software
- Reduces risk and increases predictability
- Promotes a common vision and culture
- Harvests and institutionalizes software engineering practices

A software engineering process should provide a disciplined yet flexible approach to assigning tasks and responsibilities within a software development organization. The goal is to ensure the production of high-quality software that meets the needs of its end users within a predictable schedule and budget.

The UML provides a standard for many of the artifacts of software development (semantic models, syntactic notation, and diagrams): the things that must be controlled and exchanged. But the UML is not a standard for the development process.

Despite all of the value that a common modeling language brings, you cannot achieve successful development of today's complex systems solely by the use of the UML. Successful iterative development also requires employing a repeatable engineering process.

A Team-Based Definition of Process

A Team-Based Definition of Process

A process defines **Who** is doing **What**
When, and **How**, in order to reach a certain
goal.

```
graph LR; A[New or changed requirements] --> B[Software Engineering Process]; B --> C[New or changed system]
```

**This course is about the What, When and
How of Testers' activities in the process.**

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved

Rational
the software development company

Workbook Page: Implementing Software Engineering Practices

Workbook Page: Implementing Software Engineering Practices

A modern engineering process ideally:

- ◆ Supports a controlled, iterative approach
- ◆ Supports the use of user-focused requirements to coordinate and drive the work in requirements, design, implementation and test
- ◆ Enables architectural concerns to be addressed early
- ◆ Allows the process to be configured to suit the context of the individual project
- ◆ Provides guidance for conducting work (activities) and producing work products (artifacts)

Review

Module 1 - Review

- ◆ Software engineering practices guide software development by addressing root causes of problems.
- ◆ Software engineering practices reinforce each other.
- ◆ Process guides a team on who does what when and how.
- ◆ A software engineering process provides context and support for implementing software engineering practices.