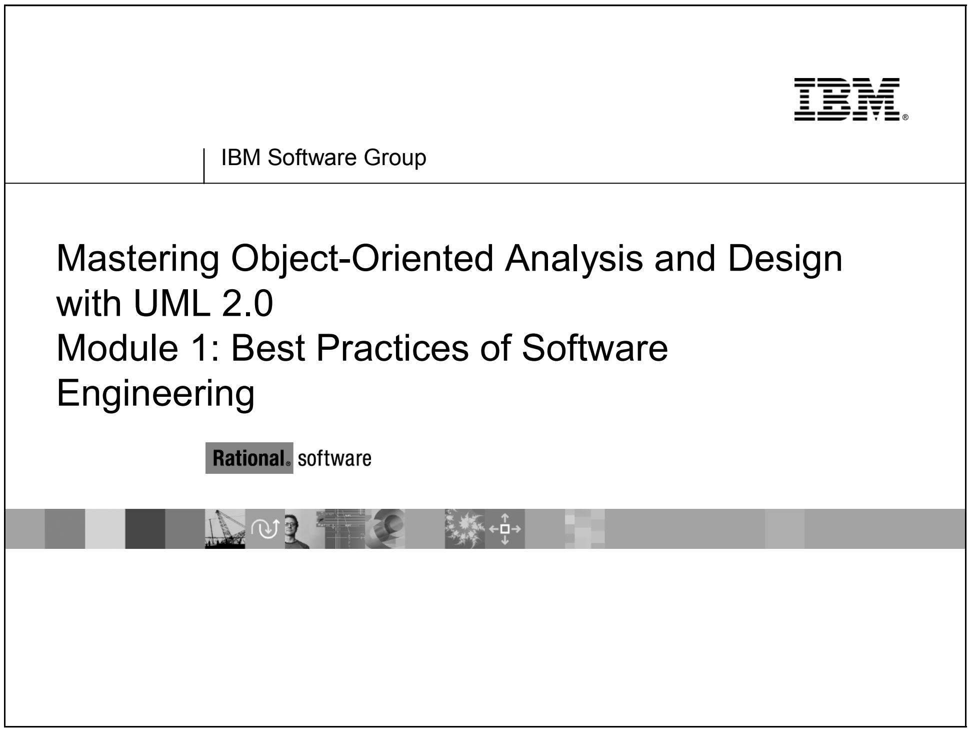
▸ ▸ ▸ Module 1 Best Practices of Software Engineering



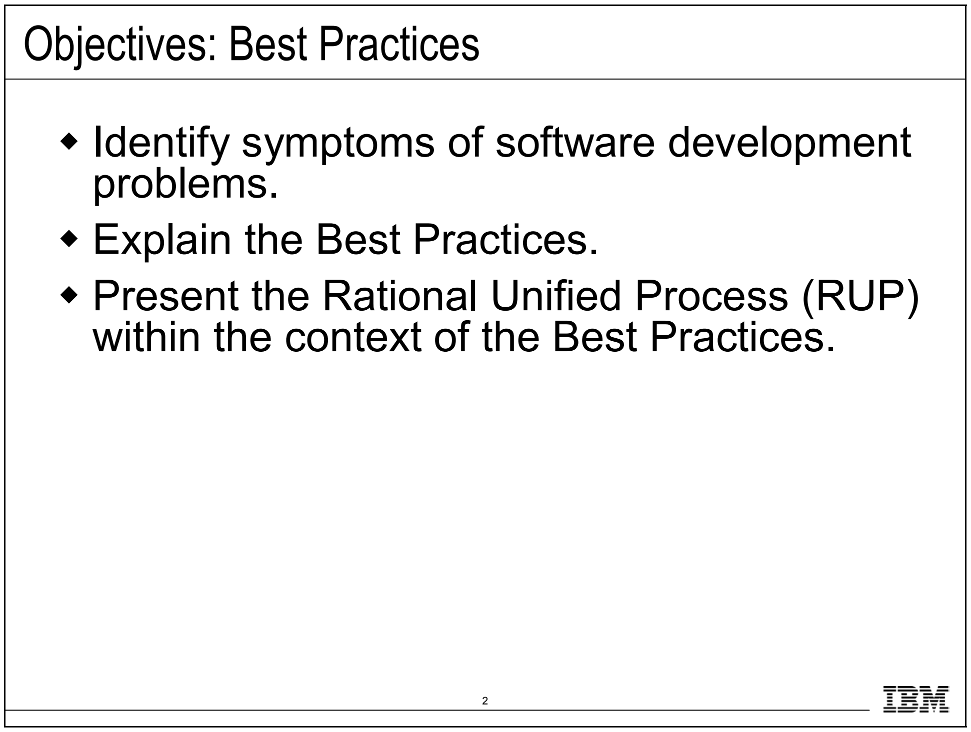
IBM Software Group

Mastering Object-Oriented Analysis and Design with UML 2.0
Module 1: Best Practices of Software Engineering

Rational. software

Topics

Objectives: Best Practices



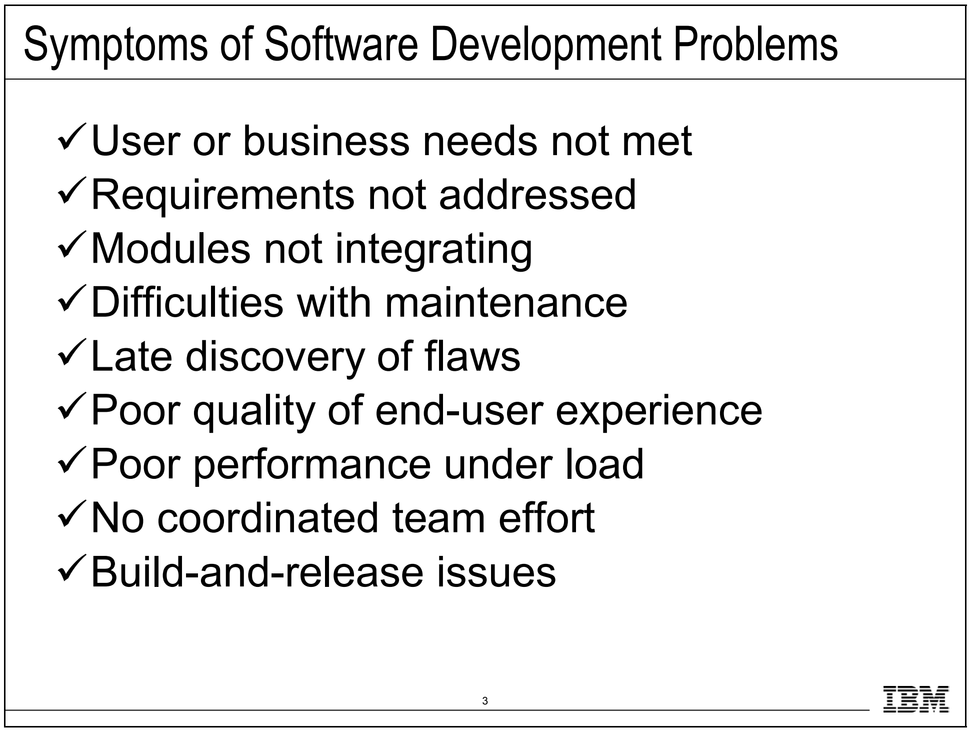
Objectives: Best Practices

- Identify symptoms of software development problems.
- Explain the Best Practices.
- Present the Rational Unified Process (RUP) within the context of the Best Practices.

2

In this module, you will learn about recommended software development Best Practices and the reasons for these recommendations. Then you will see how the Rational Unified Process (RUP) is designed to help you implement the Best Practices.

Symptoms of Software Development Problems



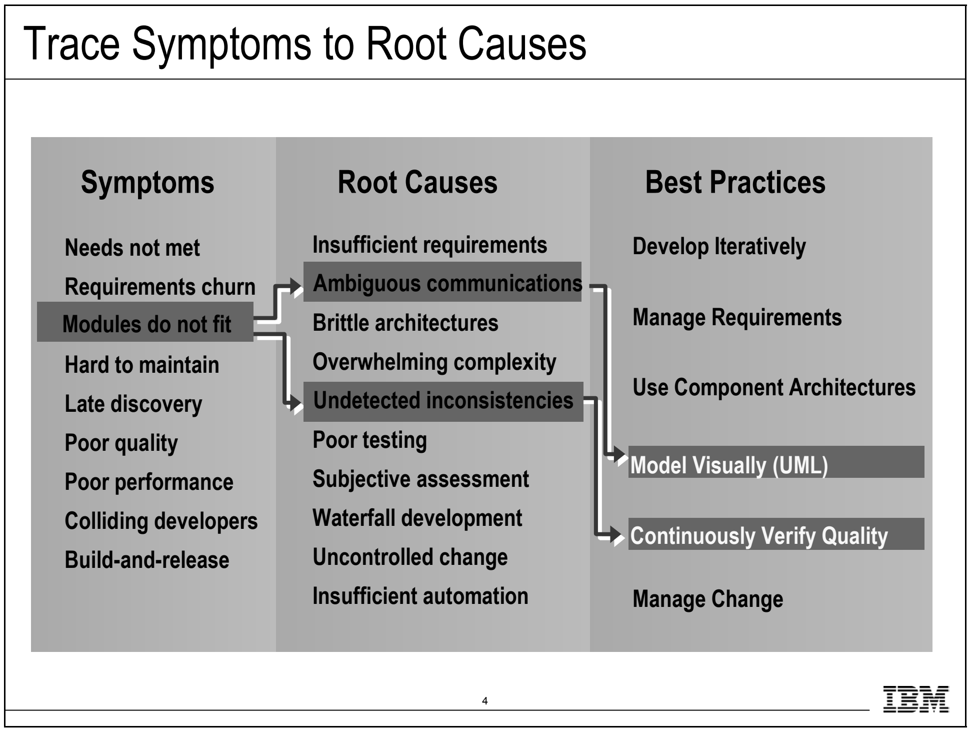
Symptoms of Software Development Problems

- ✓User or business needs not met
- ✓Requirements not addressed
- ✓Modules not integrating
- ✓Difficulties with maintenance
- ✓Late discovery of flaws
- ✓Poor quality of end-user experience
- ✓Poor performance under load
- ✓No coordinated team effort
- ✓Build-and-release issues

3

IBM

Trace Symptoms to Root Causes



Trace Symptoms to Root Causes

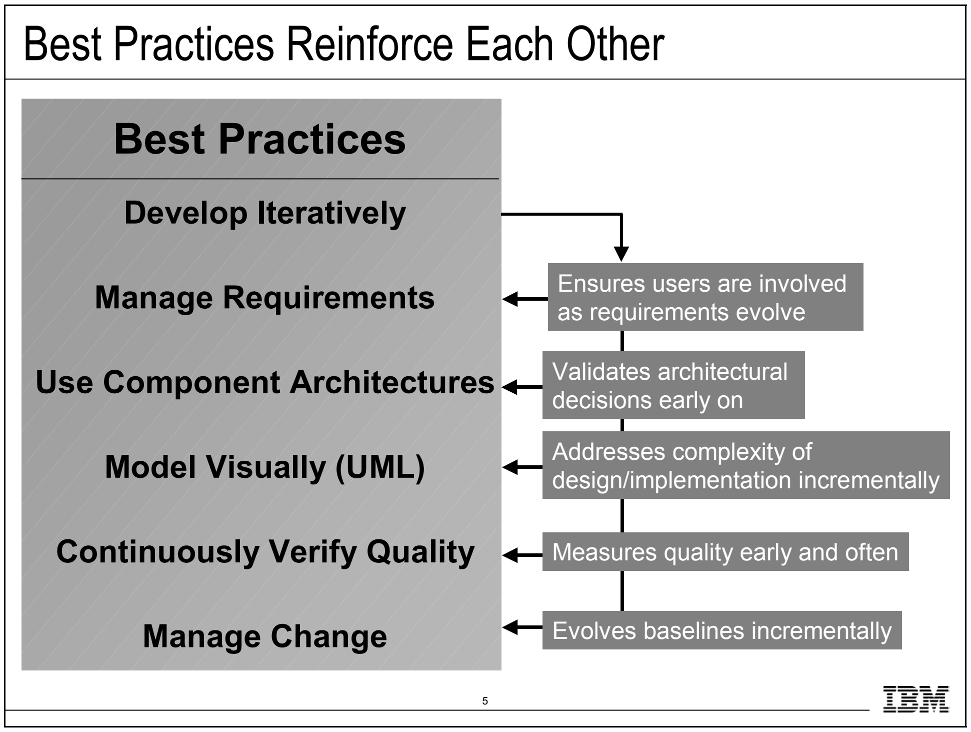
Symptoms	Root Causes	Best Practices
Needs not met	Insufficient requirements	Develop Iteratively
Requirements churn	Ambiguous communications	Manage Requirements
Modules do not fit	Brittle architectures	Use Component Architectures
Hard to maintain	Overwhelming complexity	Model Visually (UML)
Late discovery	Undetected inconsistencies	Continuously Verify Quality
Poor quality	Poor testing	Manage Change
Poor performance	Subjective assessment	
Colliding developers	Waterfall development	
Build-and-release	Uncontrolled change	
	Insufficient automation	

4

By treating these root causes, you will eliminate the symptoms. By eliminating the symptoms, you'll be in a much better position to develop high-quality software in a repeatable and predictable fashion.

Best Practices are a set of commercially proven approaches to software development, which, when used in combination, strike at the root causes of software development problems. They are called "Best Practices," not because we can precisely quantify their value, but because they have been observed to be commonly used in the industry by successful organizations. The Best Practices have been harvested from thousands of customers on thousands of projects and from industry experts.

Best Practices Reinforce Each Other



Best Practices Reinforce Each Other

Best Practices
Develop Iteratively
Manage Requirements
Use Component Architectures
Model Visually (UML)
Continuously Verify Quality
Manage Change
Ensures users are involved as requirements evolve
Validates architectural decisions early on
Addresses complexity of design/implementation incrementally
Measures quality early and often
Evolves baselines incrementally
5
IBM

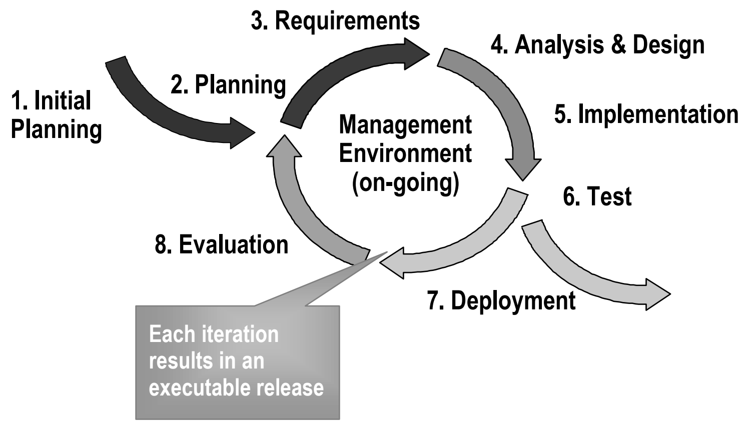
In the case of our Best Practices, the whole is much greater than the sum of the parts. Each of the Best Practices reinforces and, in some cases, enables the others. This slide shows just one example: how iterative development supports the other five Best Practices. However, each of the other five practices also enhances iterative development. For example, iterative development done without adequate requirements management can easily fail to converge on a solution. Requirements can change at will, which can cause users not to agree and the iterations to go on forever.

When requirements are managed, this is less likely to happen. Changes to requirements are visible, and the impact on the development process is assessed before the changes are accepted. Convergence on a stable set of requirements is ensured. Similarly, every Best Practices supports each of the other Best Practices. Hence, although it is possible to use one Best Practice without the others, this is not recommended, since the resulting benefits will be significantly decreased.

Practice 1: Develop Iteratively

Develop Iteratively

- Iterative development produces an executable



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**.

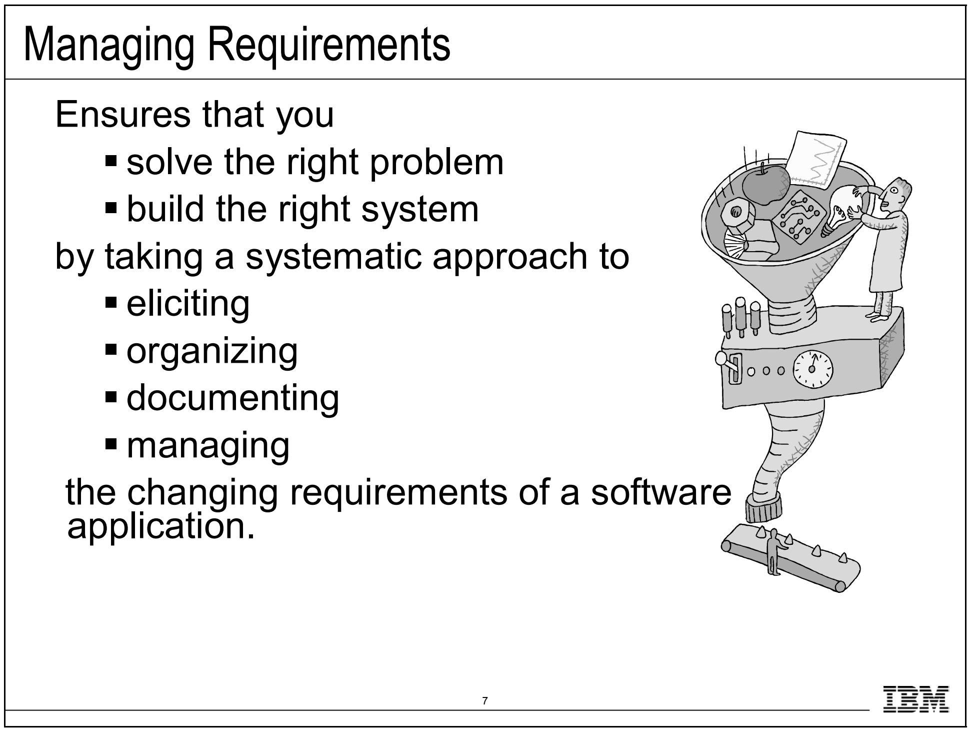
Each iteration is focused on defining, analyzing, designing, building, and testing a set of requirements.

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

- Resolve major risks before making large investments.
- Enable early user feedback.
- Make testing and integration continuous.
- Define a project's short-term objective milestone.
- Make deployment of partial implementations possible.

Instead of developing the whole system in lock step, an increment (for example, a subset of system functionality) is selected and developed, then another increment, and so on. The selection of the first increment to be developed is based on risk, with the highest priority risks first. To address the selected risk(s), choose a subset of use cases. Develop the *minimal* set of use cases that will allow objective verification (that is, through a set of executable tests) of the risks that you have chosen. Then, select the next increment to address the next-highest risk, and so on.

Practice 2: Manage Requirements

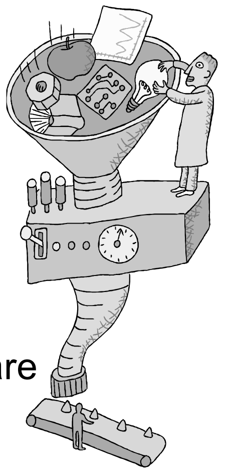


Managing Requirements

Ensures that you
- solve the right problem
- build the right system

by taking a systematic approach to
- eliciting
- organizing
- documenting
- managing

the changing requirements of a software application.



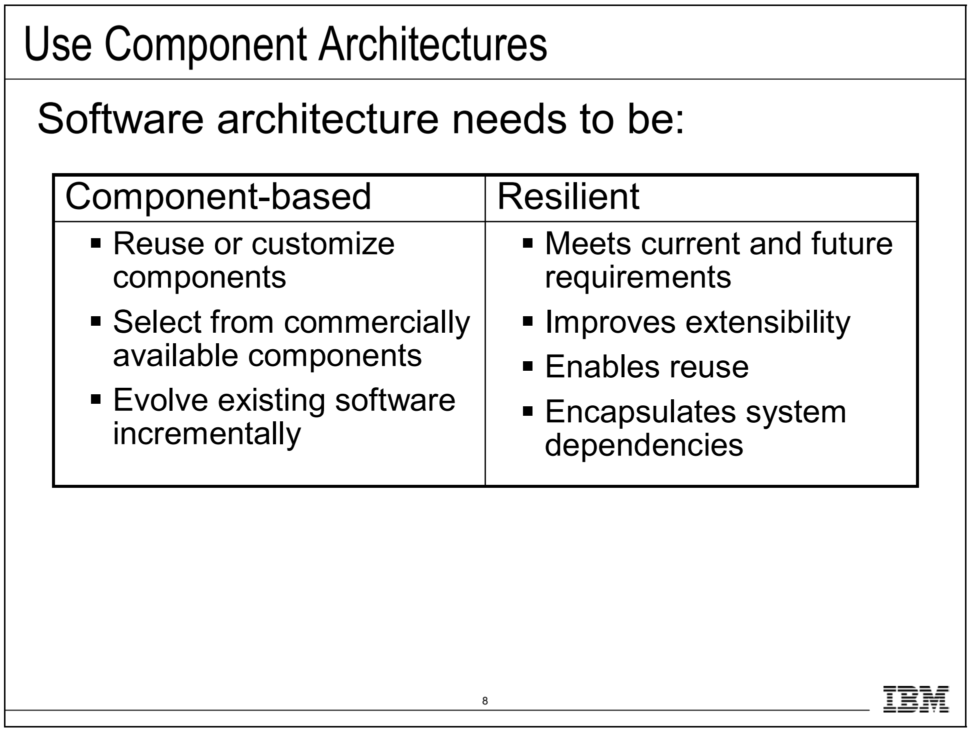
7

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 (canceled) projects accounted for 31.1%. These failures are attributed to incorrect requirements definition from the start of the project and poor requirements management throughout the development lifecycle. (Source: *Chaos Report*, http://www.standishgroup.com)

Aspects of requirements management:

- Analyze the problem
- Understand user needs
- Define the system
- Manage scope
- Refine the system definition
- Manage changing requirements

Practice 3: Use Component Architectures



Use Component Architectures

Software architecture needs to be:

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

8

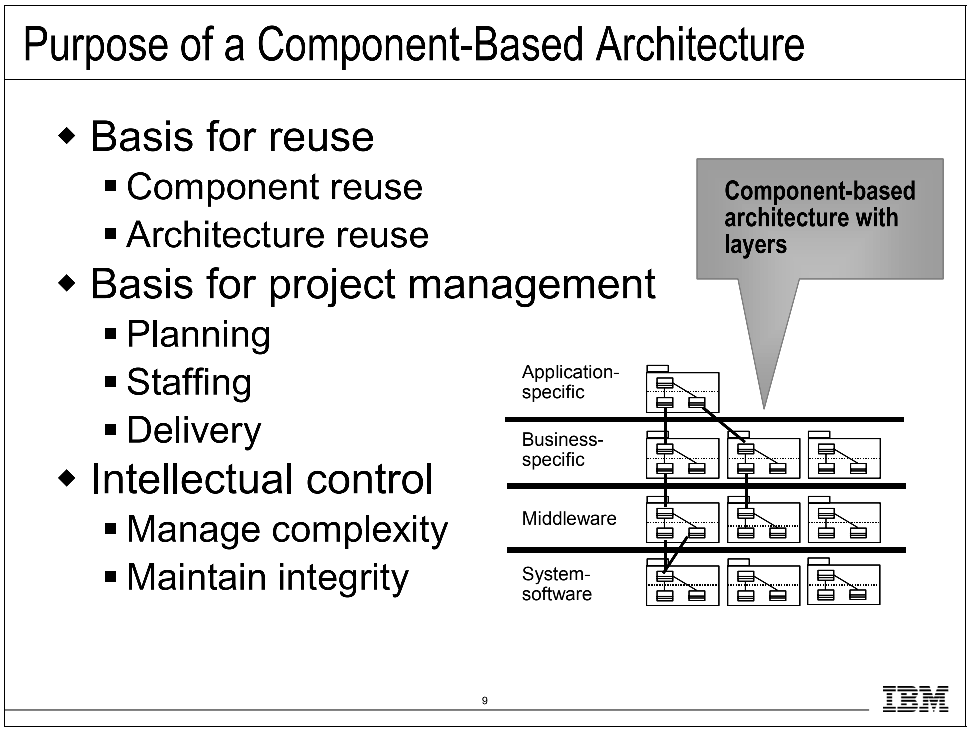
Architecture is a part of Design. It is about making decisions on how the system will be built. But it is not all of the design. It stops at the major abstractions, or, in other words, the elements that have some pervasive and long-lasting effect on system performance and ability to evolve.

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 the 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.

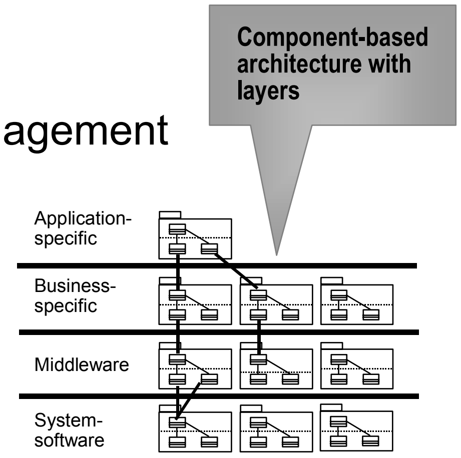
Software architecture is the development product that gives the highest return on investment with respect to quality, schedule, and cost, according to the authors of *Software Architecture in Practice* (Len Bass, Paul Clements, and Rick Kazman [1998] Addison-Wesley). The Software Engineering Institute (SEI) has an effort underway called the Architecture Tradeoff Analysis (ATA) Initiative that focuses 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. As a result of performing architecture evaluations, AT&T reported a 10% productivity increase (from news@sei, Vol. 1, No. 2).

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



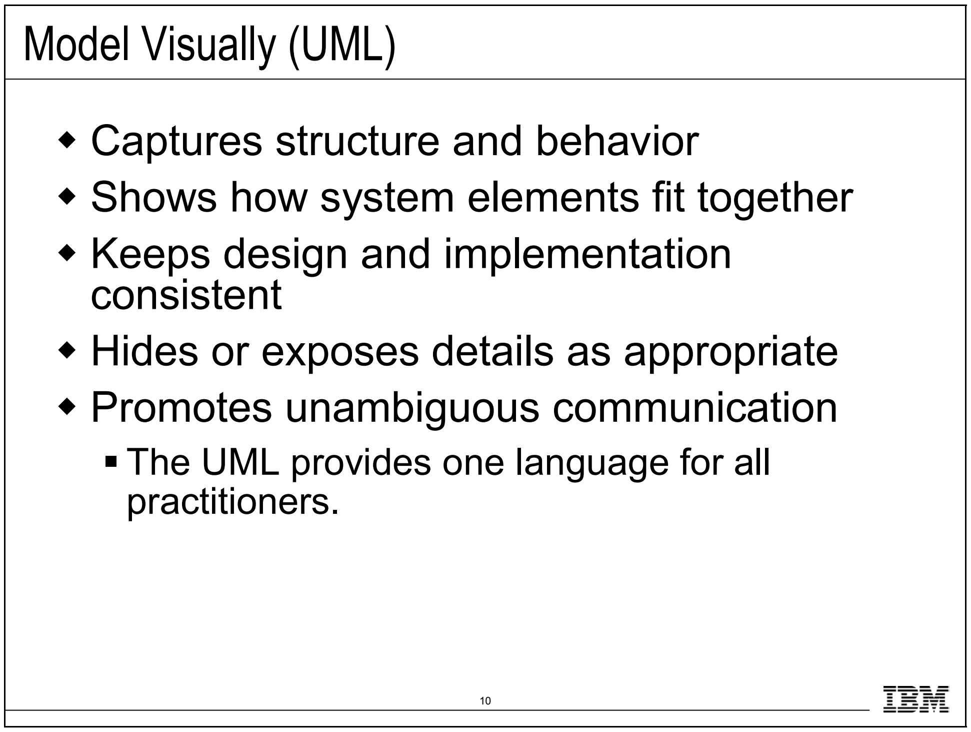
9

Definition of a (software) component:

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

Practice 4: Model Visually



Model Visually (UML)

- Captures structure and behavior
- Shows how system elements fit together
- Keeps design and implementation consistent
- Hides or exposes details as appropriate
- Promotes unambiguous communication
  - The UML provides one language for all practitioners.

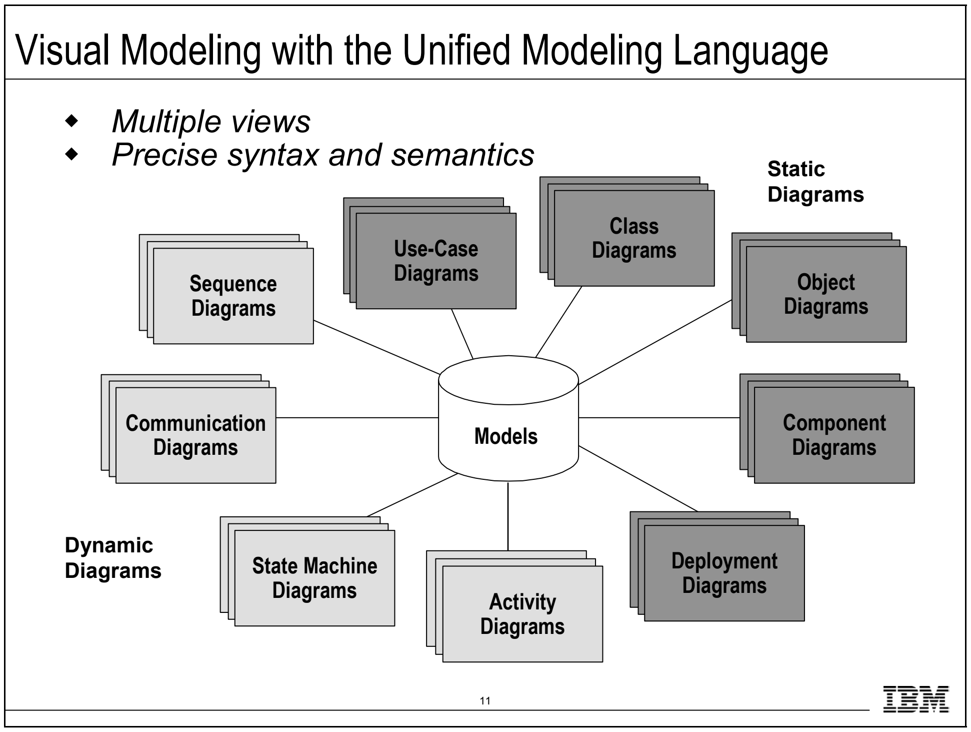
10

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 building. We build models of complex systems because we cannot comprehend any such system in its entirety.

Modeling is important because it helps the development team visualize, specify, construct, and document the structure and behavior of system 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 system artifacts: its requirements, designs, and implementations. In short, visual modeling helps improve a team's ability to manage software complexity.

Visual Modeling with the Unified Modeling Language



Visual Modeling with the Unified Modeling Language

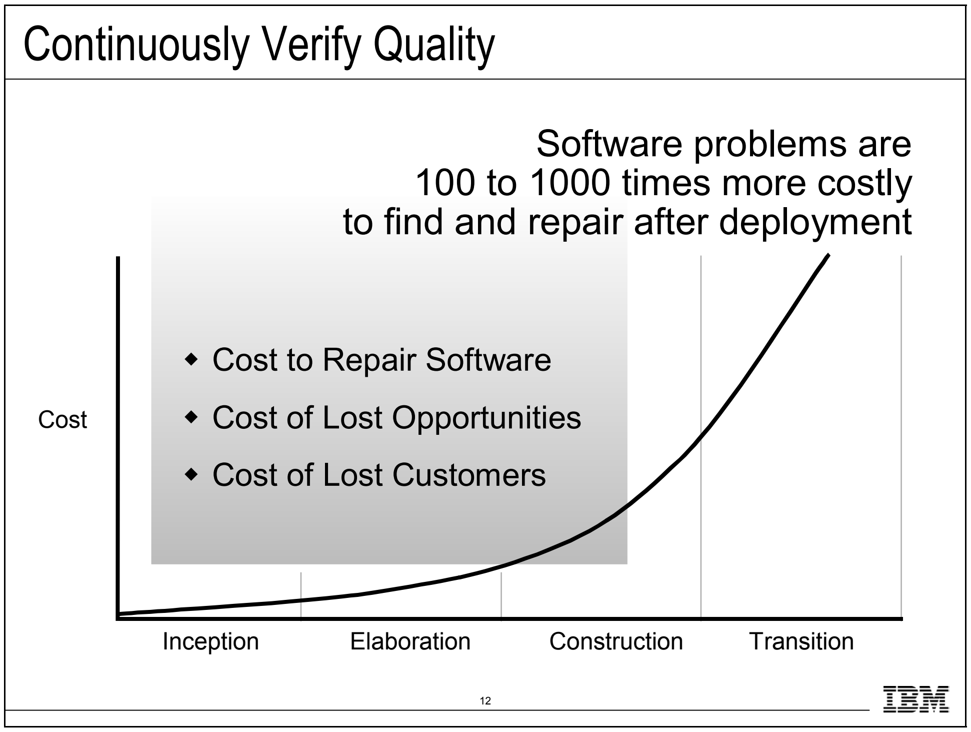
- *Multiple views*
- *Precise syntax and semantics*

In building a visual model of a system, many different diagrams are needed to represent different views of the system. The UML provides a rich notation for visualizing models. This includes the following key diagrams:

- Use-case diagrams to illustrate user interactions with the system
- Class diagrams to illustrate logical structure
- Object diagrams to illustrate objects and links
- Component diagrams to illustrate physical structure of the software
- Deployment diagrams to show the mapping of software to hardware configurations
- Activity diagrams to illustrate flows of events
- State Machine diagrams to illustrate behavior
- Interaction diagrams (that is, Communication and Sequence diagrams) to illustrate behavior

This is not all of the UML diagrams, just a representative sample.

Practice 5: Continuously Verify Quality



Quality, as used within the RUP, is defined as "The characteristic of having demonstrated the achievement of producing a product which meets or exceeds agreed-upon requirements, as measured by agreed-upon measures and criteria, and is produced by an agreed-upon process." Given this definition, achieving quality is not simply "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 the desired degree of quality (and can be repeated and managed).

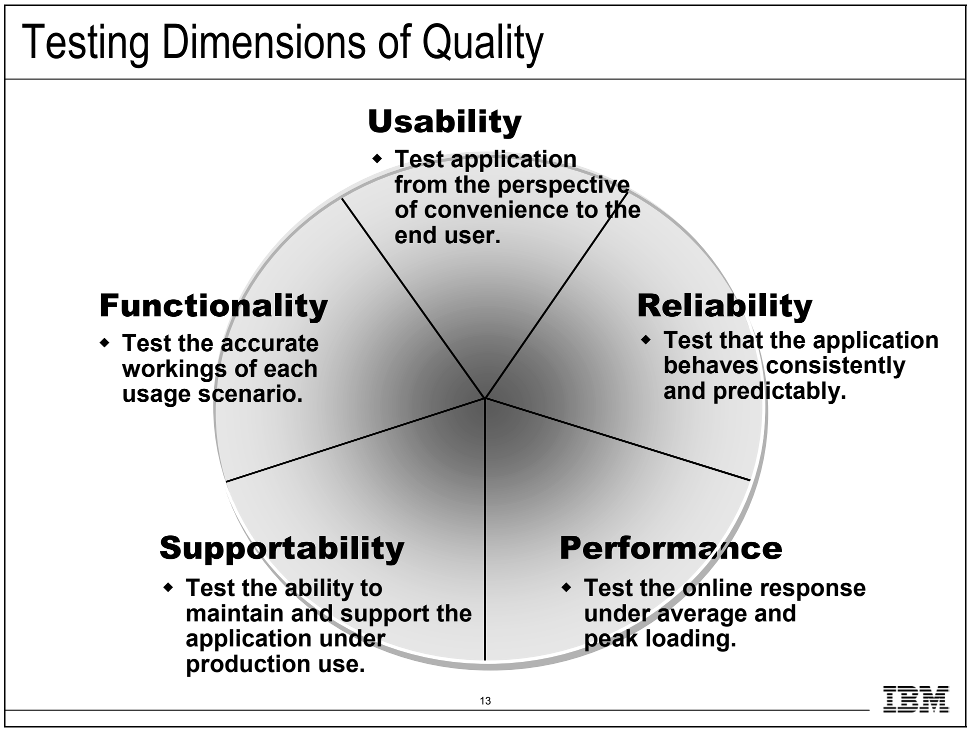
This principle is driven by a fundamental and well-known property of software development: It is a lot less expensive to correct defects during development than to correct them after deployment.

Tests for key scenarios ensure that all requirements are properly implemented.

- Poor application performance hurts as much as poor reliability.
- Verify software reliability by checking for memory leaks and bottlenecks.
- Test every iteration by automating testing.

Inception, Elaboration, Construction, and Transition are all RUP terms that will be discussed shortly.

Testing Dimensions of Quality



Testing Dimensions of Quality

Usability
- Test application from the perspective of convenience to the end user.

Functionality
- Test the accurate workings of each usage scenario.

Reliability
- Test that the application behaves consistently and predictably.

Supportability
- Test the ability to maintain and support the application under production use.

Performance
- Test the online response under average and peak loading.

13

Functional testing verifies that a system executes the required use-case scenarios as intended. Functional tests may include the testing of features, usage scenarios, and security.

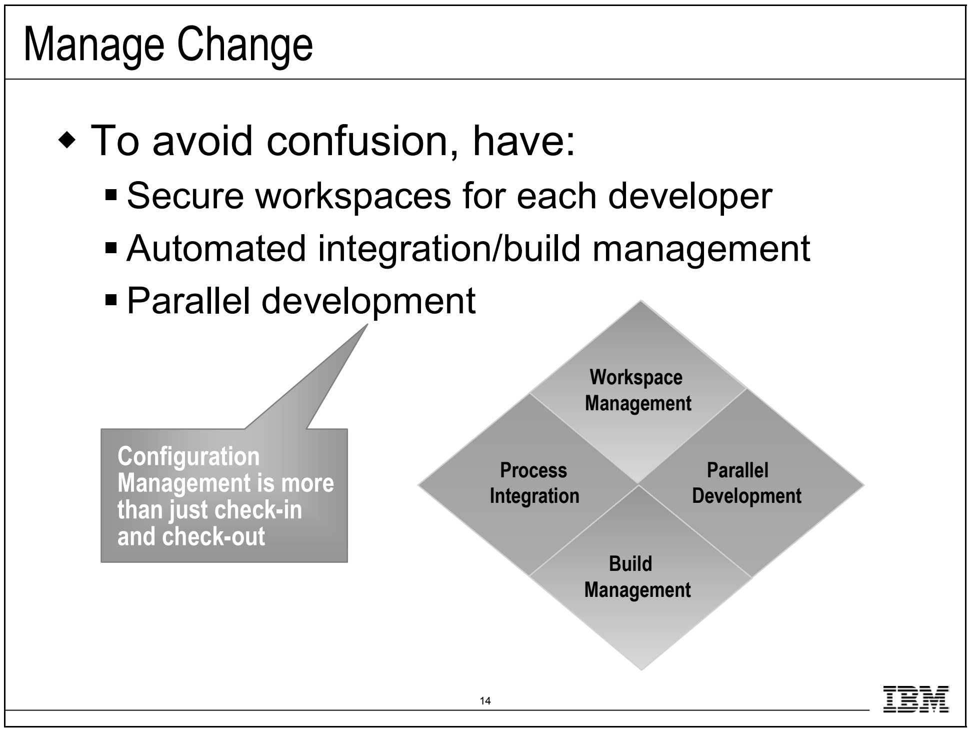
Usability testing evaluates the application from the user's perspective. Usability tests focus on human factors, aesthetics, consistency in the user interface, online and context-sensitive Help, wizards and agents, user documentation, and training materials.

Reliability testing verifies that the application performs reliably and is not prone to failures during execution (crashes, hangs, and memory leaks). Effective reliability testing requires specialized tools. Reliability tests include tests of integrity, structure, stress, contention, and volume.

Performance testing checks that the target system works functionally and reliably under production load. Performance tests include benchmark tests, load tests, and performance profile tests.

Supportability testing verifies that the application can be deployed as intended. Supportability tests include installation and configuration tests.

Practice 6: Manage Change



Manage Change

- To avoid confusion, have:
  - Secure workspaces for each developer
  - Automated integration/build management
  - Parallel development

Configuration Management is more than just check-in and check-out

Workspace Management
Process Integration
Parallel Development
Build Management

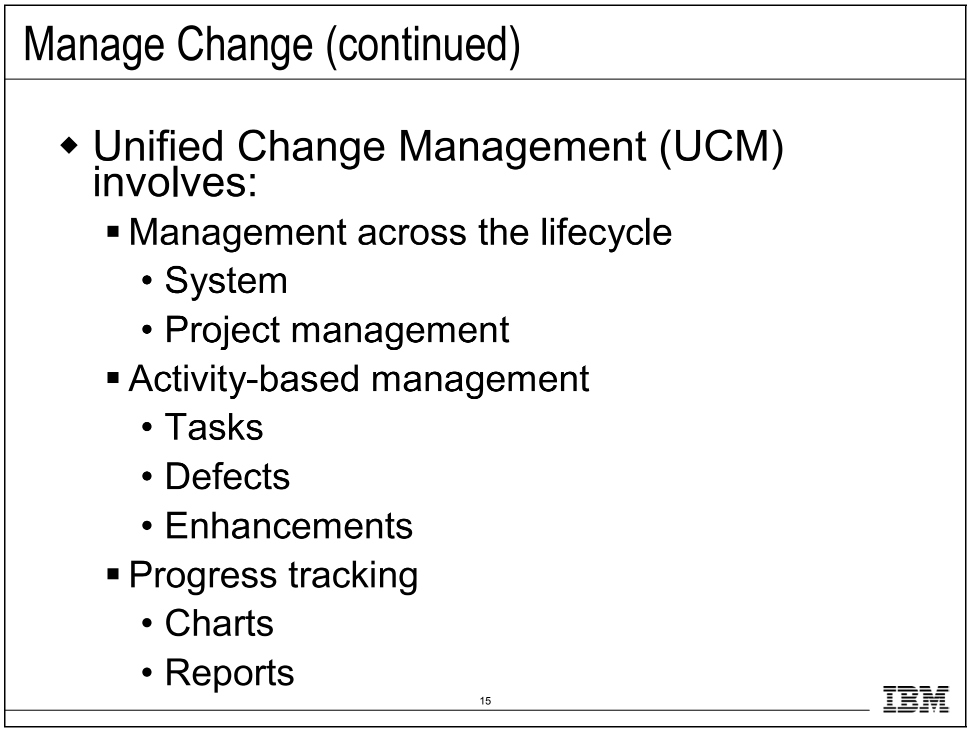
14

Establishing a secure workspace for each developer provides isolation from changes made in other workspaces and control of all software artifacts — models, code, documents and so forth.

A key challenge to developing software-intensive systems is the need to cope with multiple developers, 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 roles separately modify the same artifact, the last one to make changes destroys the work of the others.
- **Limited notification**: When a problem is fixed in shared artifacts, some of the developers are not notified of the change.
- **Multiple versions**: With iterative development, it would not be unusual to have multiple versions of an artifact in different stages of development at the same time. For example, one release is in customer use, one is in test, and one is still in development. If a problem is identified in any one of the versions, the fix must be propagated among all of them.

Manage Change (continued)



Manage Change (continued)

- Unified Change Management (UCM) involves:
  - Management across the lifecycle
    - System
    - Project management
  - Activity-based management
    - Tasks
    - Defects
    - Enhancements
  - Progress tracking
    - Charts
    - Reports

15

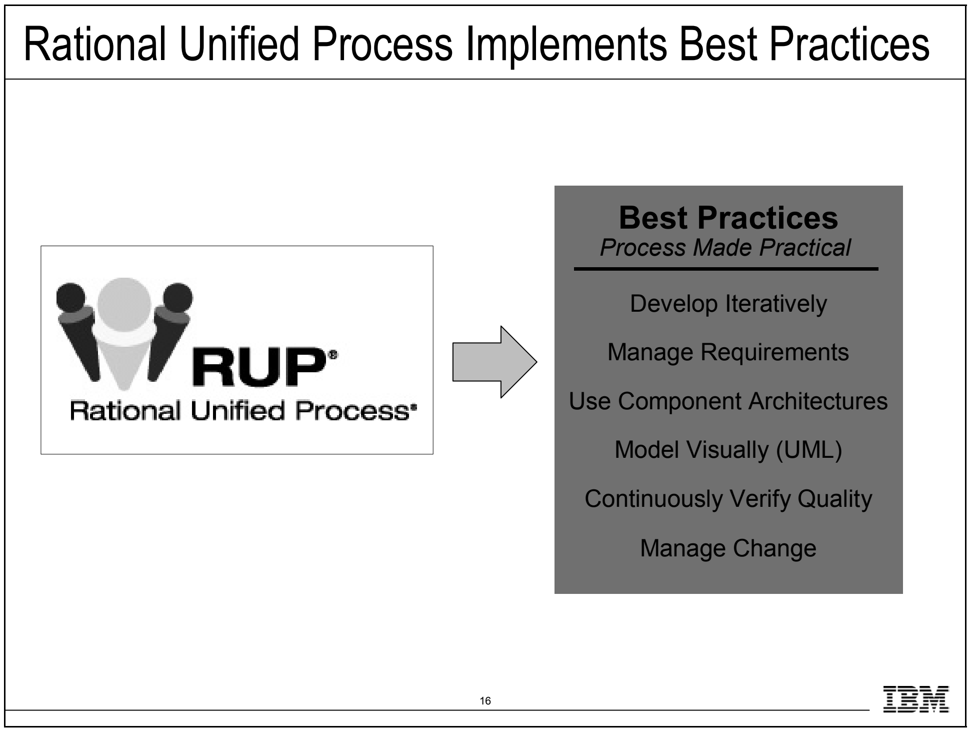
Unified Change Management (UCM) is "the Rational software approach" to managing change in software system development, from requirements to release. UCM spans the development lifecycle, defining how to manage change to requirements, design models, documentation, components, test cases, and source code.

One of the key aspects of the UCM model is that it unifies the activities used to plan and track project progress and the artifacts undergoing change.

You cannot stop change from being introduced into a project; however, you must control how and when changes are introduced into project artifacts, and who introduces those changes.

You must also synchronize changes across development teams and locations. Unified Change Management (UCM) is the Rational Software approach to managing change in software system development, from requirements to release.

Rational Unified Process Implements Best Practices



Rational Unified Process Implements Best Practices

Why have a process?

- It provides guidelines for efficient development of quality software
- It reduces risk and increases predictability
- It promotes a common vision and culture
- It captures and institutionalizes Best Practices

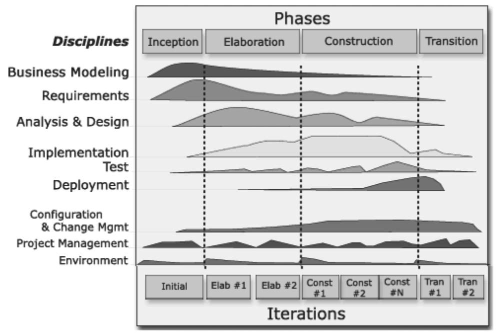
The Rational Unified Process (RUP) is a generic business process for object-oriented software engineering. It describes a family of related software-engineering processes sharing a common structure and a common process architecture. It provides a disciplined approach to assigning tasks and responsibilities within a development organization. Its 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 RUP captures the Best Practices in modern software development in a form that can be adapted for a wide range of projects and organizations.

The UML provides a standard for the artifacts of 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 development also requires employing an equally robust development process, which is where the RUP comes in.

Achieving Best Practices

Achieving Best Practices

- Iterative approach
- Guidance for activities and artifacts
- Process focus on architecture
- Use cases that drive design and implementation
- Models that abstract the system

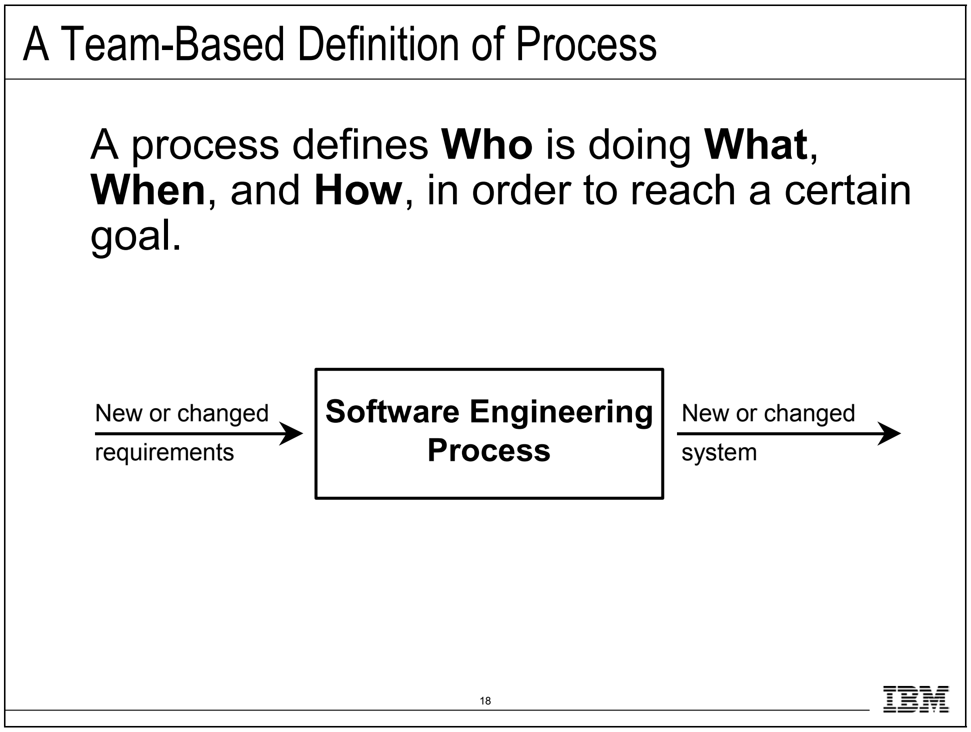


17 IBM

Examples:

- The dynamic structure (phases and iterations) of the Rational Unified Process creates the basis of iterative development.
- The Project Management discipline describes how to set up and execute a project using phases and iterations.
- Within the Requirements discipline, the Use-case model and the risk list determine what functionality you implement in an iteration.
- The workflow details of the Requirements discipline show the activities and artifacts that make requirements management possible.
- The iterative approach allows you to progressively identify components and to decide which one to develop, which one to reuse, and which one to buy.
- The Unified Modeling Language (UML) used in the process represents the basis of visual modeling and has become the de facto modeling language standard.
- The focus on software architecture allows you to articulate the structure: the components, the ways in which they integrate, and the fundamental mechanisms and patterns by which they interact.

A Team-Based Definition of Process



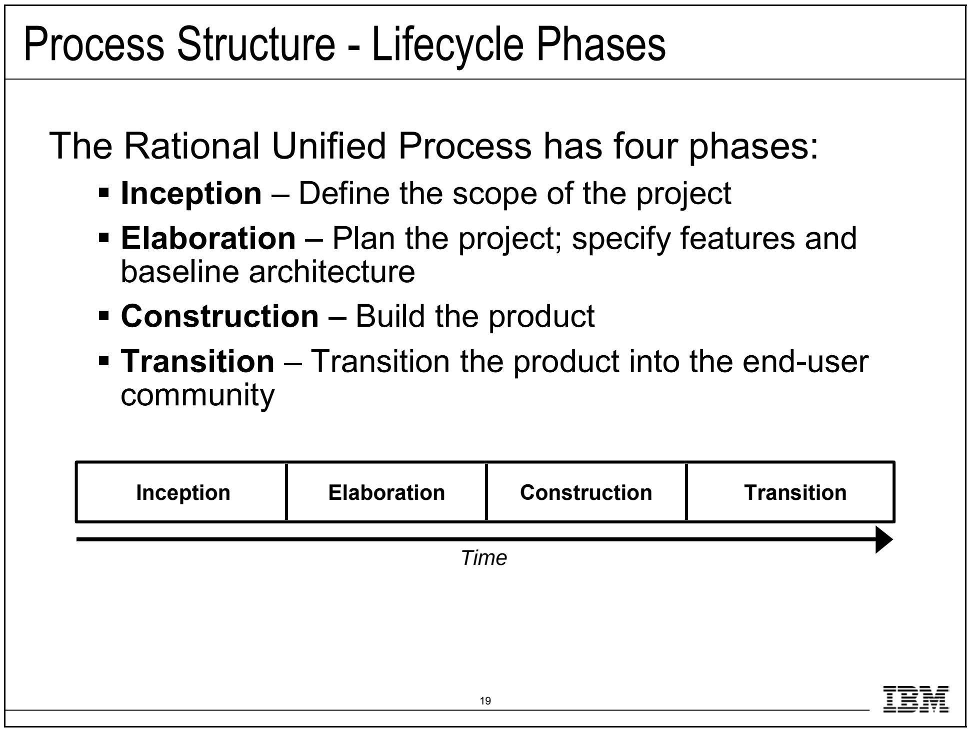
A Team-Based Definition of Process

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

$$\text{New or changed requirements} \rightarrow \boxed{\textbf{Software Engineering Process}} \rightarrow \text{New or changed system}$$

18

Process Structure - Lifecycle Phases



Process Structure - Lifecycle Phases

The Rational Unified Process has four phases:

- **Inception** – Define the scope of the project
- **Elaboration** – Plan the project; specify features and baseline architecture
- **Construction** – Build the product
- **Transition** – Transition the product into the end-user community

Inception
Elaboration
Construction
Transition
Time
19

During Inception, we define the scope of the project: what is included and what is not. We do this by identifying all the actors and use cases, and by drafting the most essential use cases (typically 20% of the complete model). A business plan is developed to determine whether resources should be committed to the project.

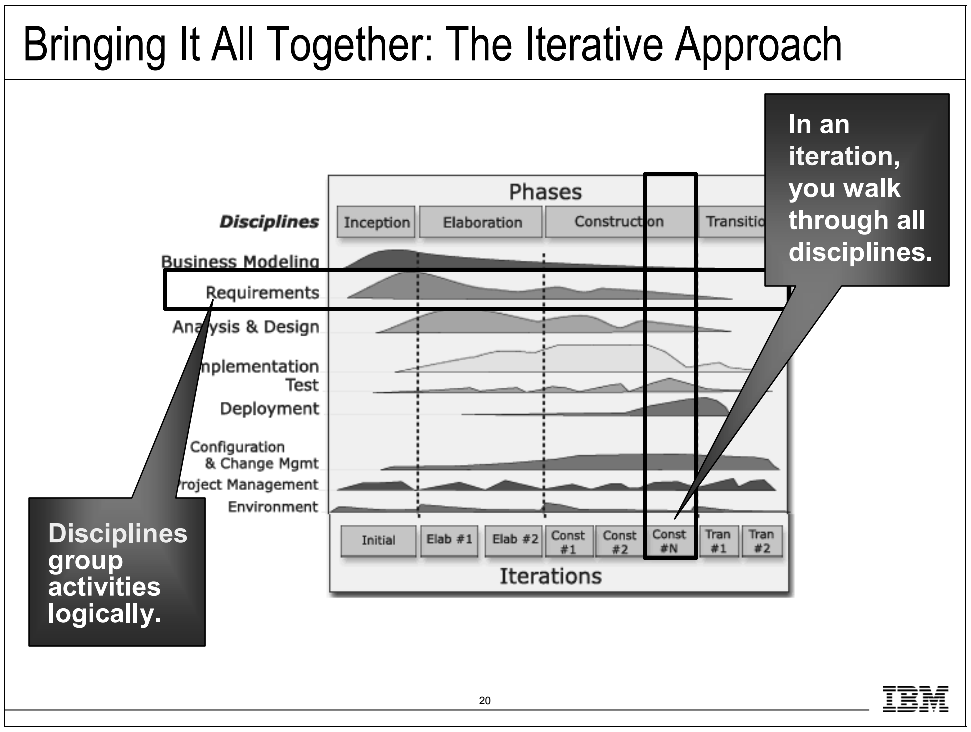
During Elaboration, we focus on two things: getting a good grasp of the requirements (80% complete) and establishing an architectural baseline. If we have a good grasp of the requirements and the architecture, we can eliminate a lot of the risks, and we will have a good idea of how much work remains to be done. We can make detailed cost and resource estimates at the end of Elaboration.

During Construction, we build the product in several iterations up to a beta release.

During Transition, we move the product to the end user and focus on end-user training, installation, and support.

The amount of time spent in each phase varies. For a complex project with many technical unknowns and unclear requirements, Elaboration may include three to five iterations. For a simple project, where requirements are known and the architecture is simple, Elaboration may include only a single iteration.

Bringing It All Together: The Iterative Approach



Bringing It All Together: The Iterative Approach

In an iteration, you walk through all disciplines.

Disciplines group activities logically.

This slide illustrates how phases and iterations (the time dimension) relate to the development activities (the discipline dimension). The relative size of each color area in the graph indicates how much of the activity is performed in each phase or iteration.

Each iteration involves activities from all disciplines. The relative amount of work related to the disciplines changes between iterations. For instance, during late Construction, the main work is related to Implementation and Test, and very little work on Requirements is done.

Note that requirements are not necessarily complete by the end of Elaboration. It is acceptable to delay the analysis and design of well-understood portions of the system until Construction because they are low in risk. This is a brief summary of the RUP disciplines:

Business Modeling – Encompasses all modeling techniques you can use to visually model a business.

Requirements – Defines what the system should do.

Analysis & Design – Shows how the system's *use cases* will be realized in implementation.

Implementation – Implements software components that meet quality standards.

Test – Integrates and tests the system.

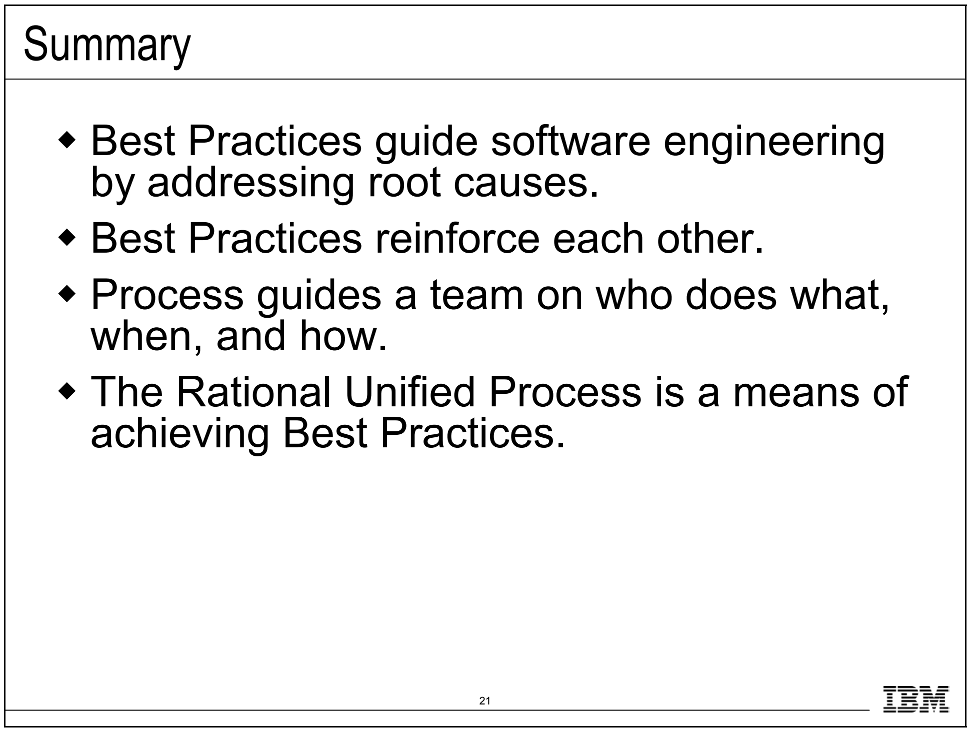
Deployment - Provides the software product to the end-user.

Configuration & Change Management – Controls and tracks changes to *artifacts*.

Project Management – Ensures tasks are scheduled, allocated and completed in accordance with project schedules, budgets and quality requirements.

Environment – Defines and manages the environment in which the system is being developed.

Summary



Summary

- Best Practices guide software engineering by addressing root causes.
- Best Practices reinforce each other.
- Process guides a team on who does what, when, and how.
- The Rational Unified Process is a means of achieving Best Practices.

21

IBM