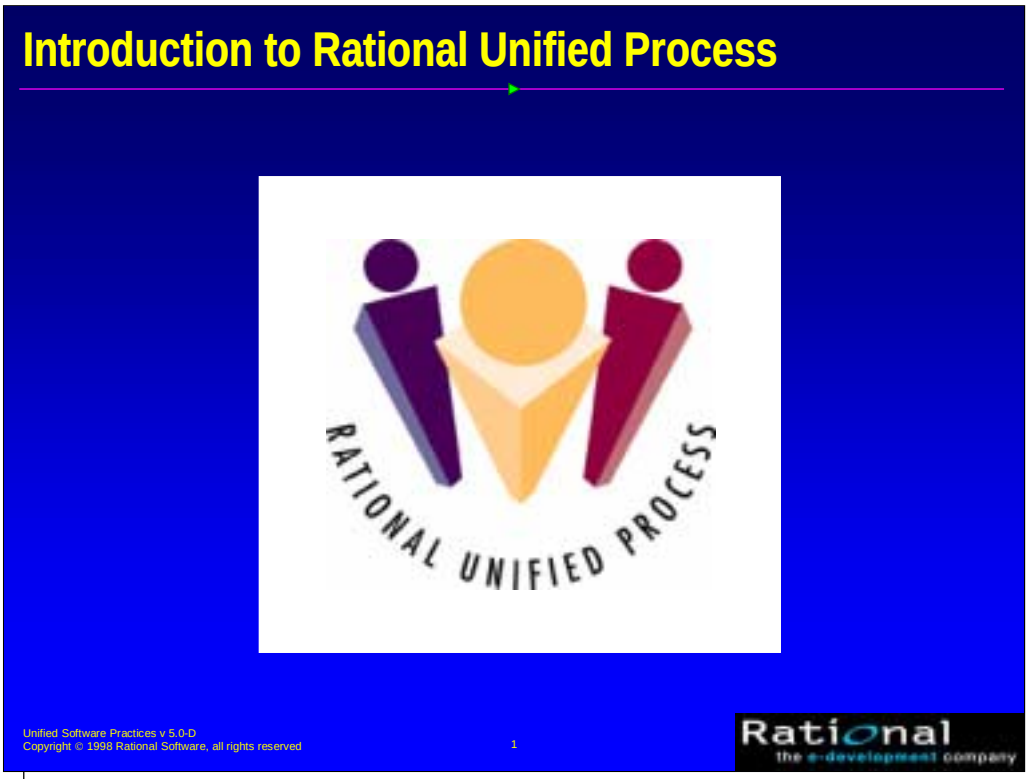
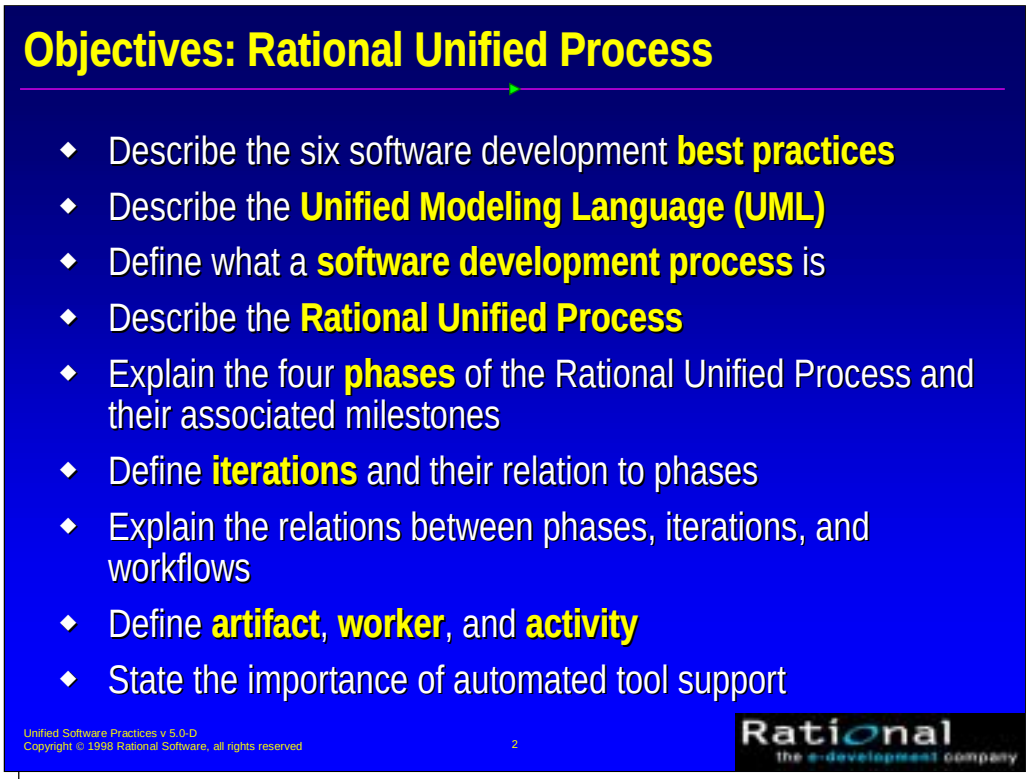
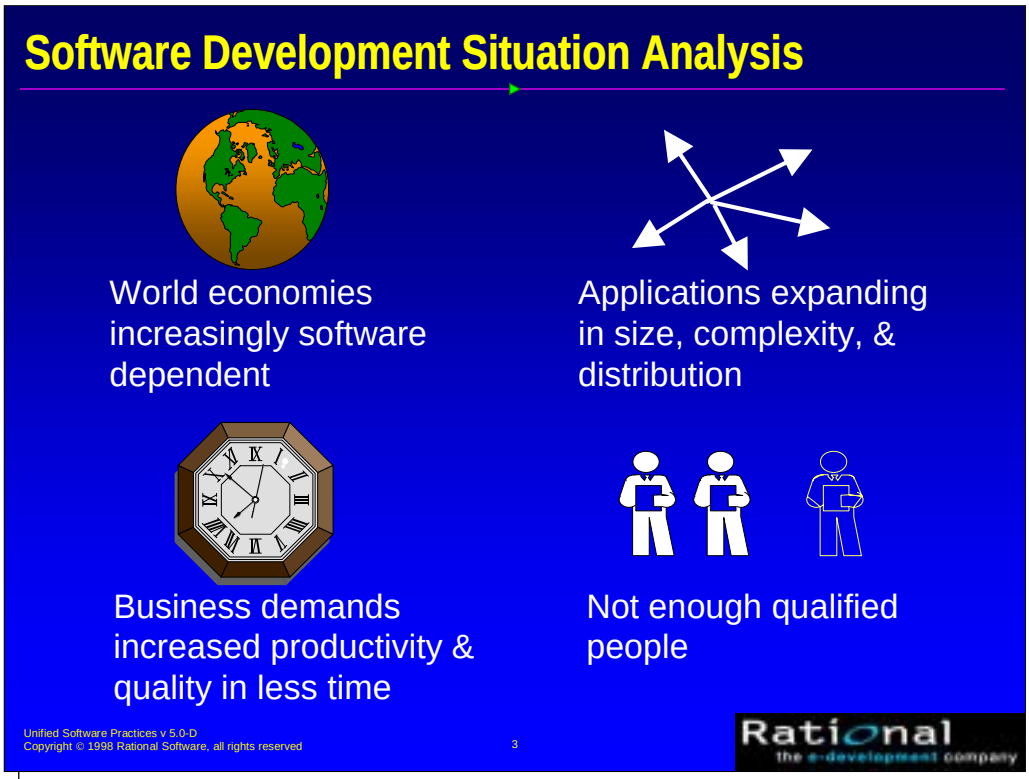


Ing. Giuseppe Calavaro
Corso Informatica Industriale
Università degli studi di Roma "Tor Vergata"
AA 1999/2000



In this module, we take a high-level look at the Rational Unified Process, our process for the UML that supports and encourages the best practices we discussed earlier.

Software development is the process of developing a software system from requirements. A software process provides a disciplined approach to assigning tasks and responsibilities within a development organization, to ensure production of a high-quality software product that meets the needs of its end users within a predictable schedule and budget. The Rational Unified Software Process is a software process which incorporates the six principles for software development we have just examined. It codifies them into a formal, written set of procedures and practices that are complete, thorough, and self-consistent.



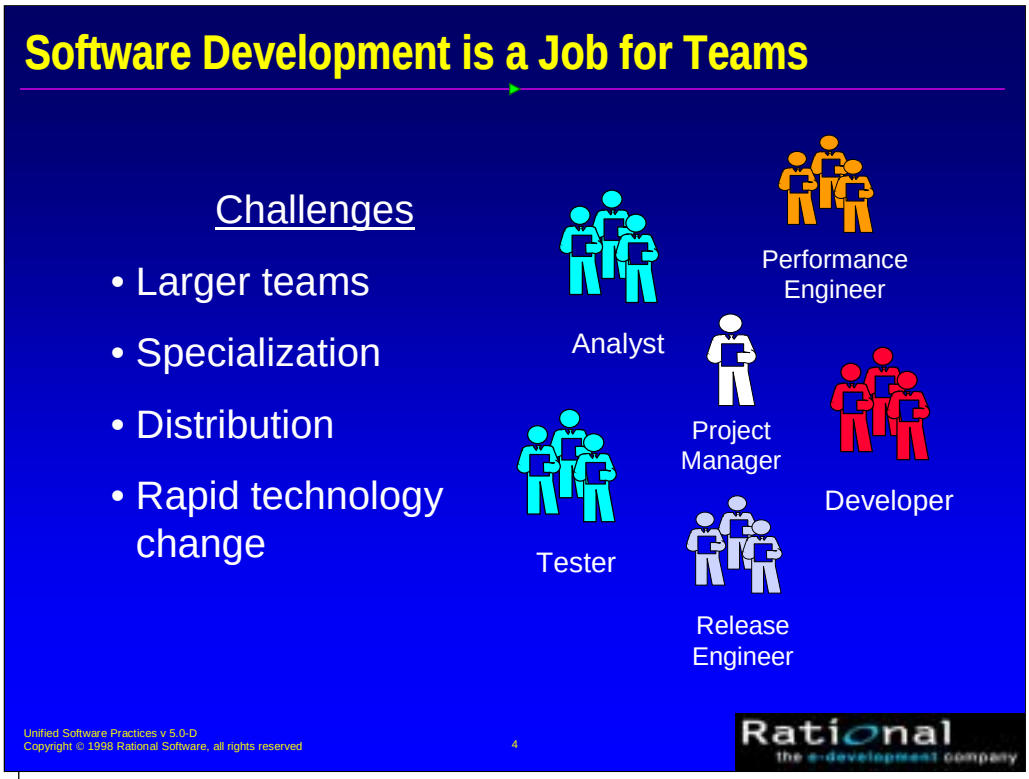
The good news for software professionals is that worldwide economies are becoming increasingly dependent on software. The kinds of software-intensive systems that improvements in technology make possible, and that society demands, are expanding in size, complexity, distribution, and importance. In the US alone, we spend more than $250 billion each year on IT application development of approximately 175,000 projects (Source: Chaos Report, http://www.standishgroup.com).

The bad news is that these systems are expanding in size, complexity, distribution, and importance, for it pushes the limits of what we in the software industry know how to do. Poor quality software is becoming more visible. We often read about it in the newspaper! Some significant examples include:

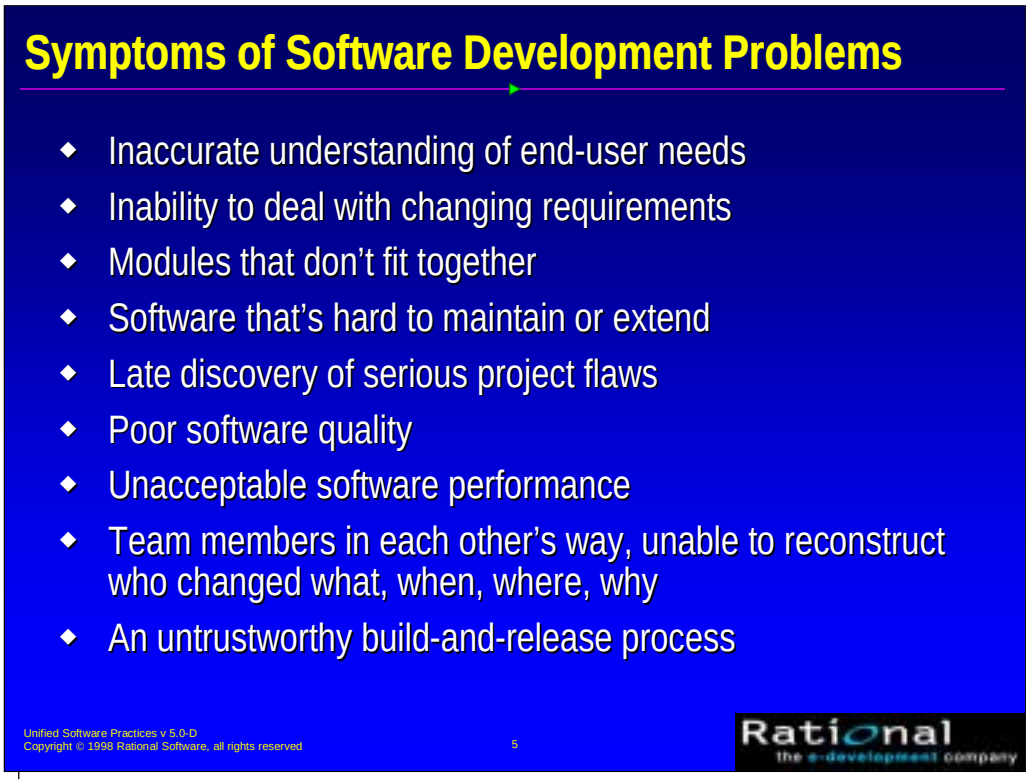
AT&T. A software switching system failed and hindered long distance communication in the US for almost 24 hours. The resolution required a change to a single line of code.

Denver Airport. Software defects delayed the airport's opening almost 9 months, at an estimated taxpayer cost of about 1/2 million dollars / day.

Further compounding the problem, businesses continue to demand increased productivity and quality with faster development and deployment. Additionally, the supply of qualified development personnel is not keeping pace with the demand. 1998 estimates of open software jobs range from 200,000 to 400,000. Economists predict that another million new programmers will be needed within the next nine years.



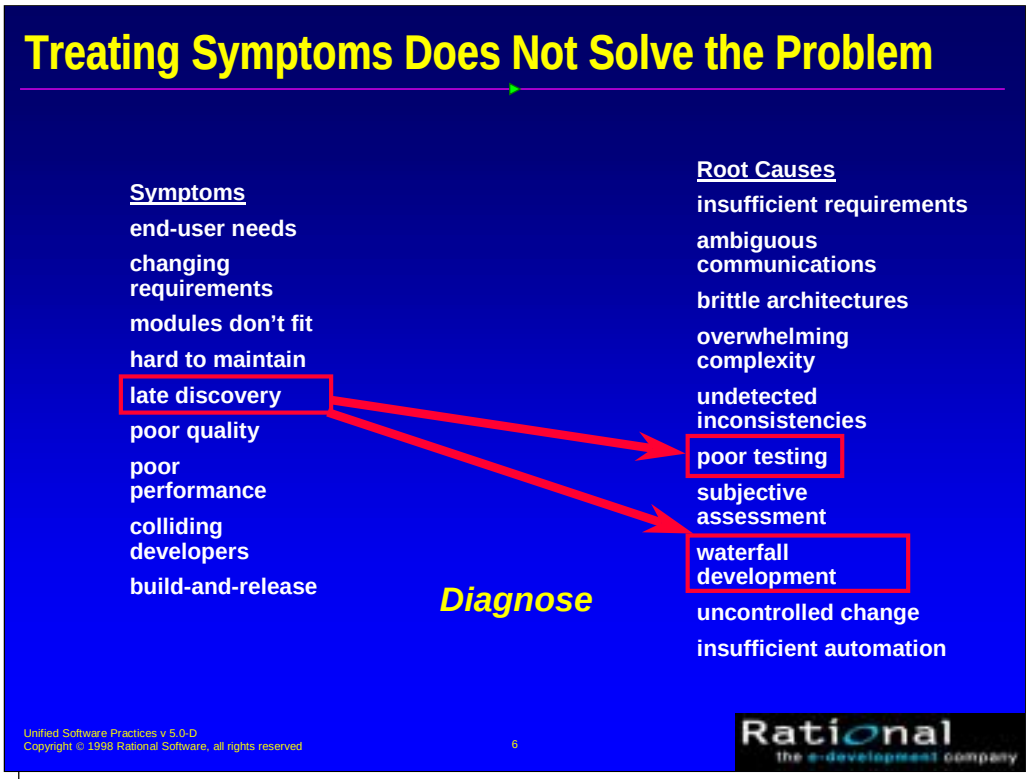
Because of the size and complexity of modern software systems, software development is a team endeavor. Larger teams present communication challenges which are aggravated if the team is distributed. Complex technologies require experts who specialize in only one area. A team may include many such specialists. It is a challenge to ensure they communicate effectively with the rest of the team and that all technologies are integrated to produce a successful system. The pace of technological change continues to accelerate and many technologies are used before they have been proven.



Producing quality software on time is very challenging; a large number obstacles and problems must be overcome. Problems are often first recognized by their symptoms. These are some all-too-common ones.

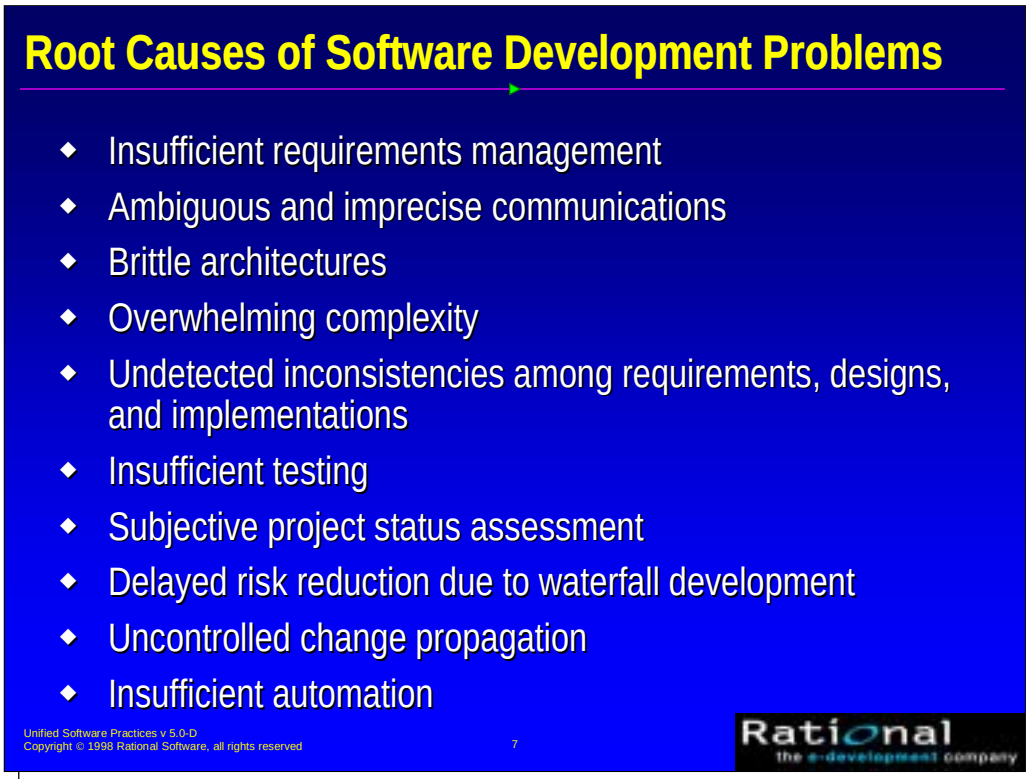
The reality is that we can't stop change. We used to try to baseline requirements at the start of a project and then resist all changes to them. A better approach is to develop practices that allow us to manage change when it occurs, with minimal impact to the development process.

An example associated with software that's hard to maintain or extend is the Year 2000 (Y2K) problem. The software itself has been unexpectedly successful, lasting well beyond its estimated end of life projections. But the software is very hard to modify since the Y2K problem is pervasive. It occurs throughout the software and each occurrence must be found and fixed individually.



Unfortunately, treating these symptoms does not treat the disease. For example, late discovery of serious project flaws is only a symptom of larger problems, namely, inadequate testing. Waterfall development compounds the problem since testing is done very late in the development schedule when the flaws discovered cannot be corrected without a significant delay in delivery.

From Rational's many collective years working with customers, we have identified a set of root causes that underlie these symptoms.

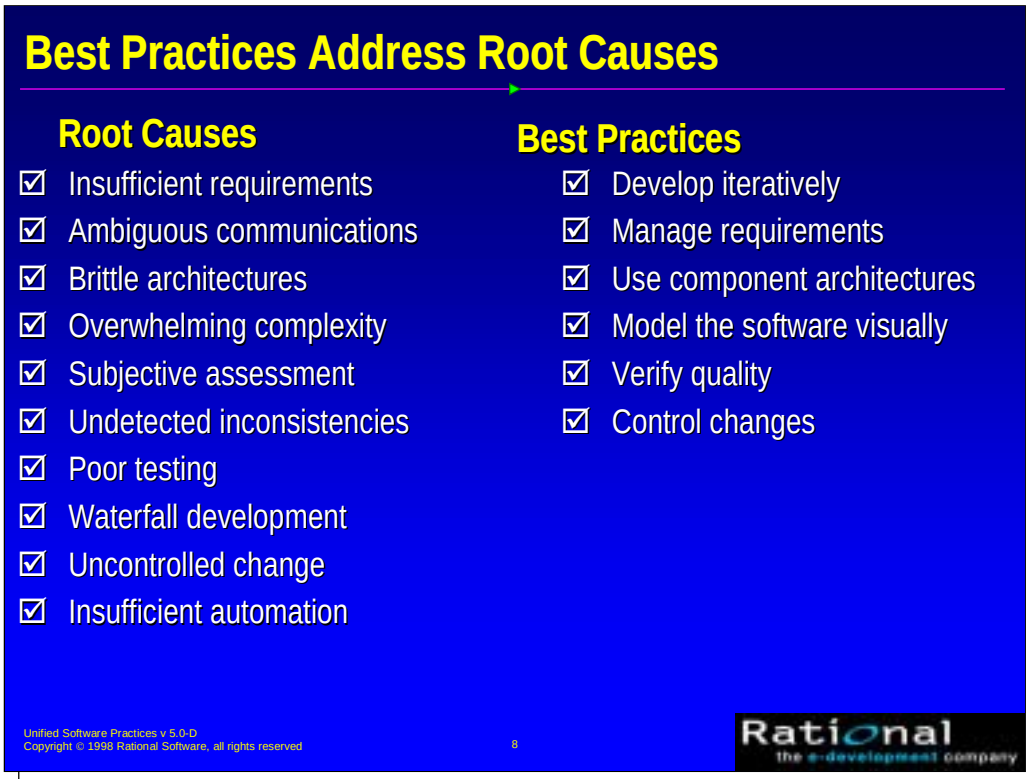


These root causes have been determined by analyzing Rational's experience in developing its software products together with the experience of Rational's many customers. Common patterns and themes have been detected and are documented as these root causes and the best practices that address them.

Insufficient requirements management makes scope creep inevitable and seriously jeopardizes the development team's ability to bring the project to a conclusion on time.

Brittle architectures are architectures that break easily in response to stress or requirements/technology changes. They typically provide no reusable components nor are they based on a reusable framework.

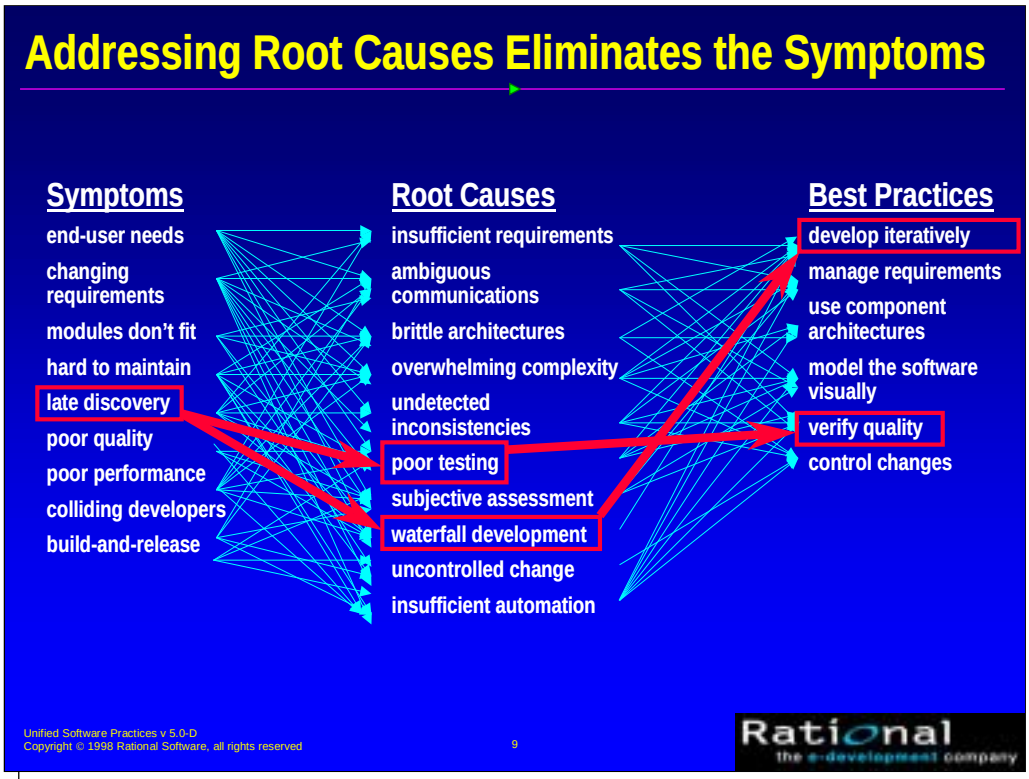
Subjective project status assessment depends on subjective estimates of the level of "doneness". This is often referred to as the "90% done, 90% remaining" phenomenon. In other words, the developers estimate that they are 90% done, but it takes 90% of project resources to get the rest of the way to 100% done.

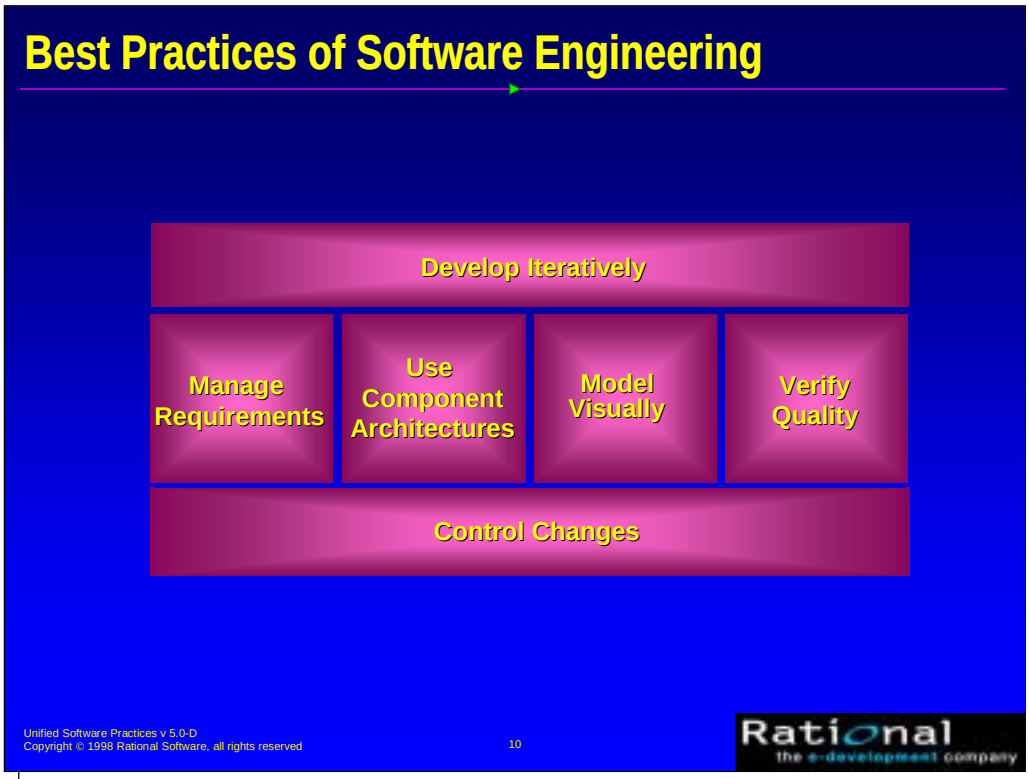


Treat these root causes and you'll eliminate the symptoms. Eliminate the symptoms, and you'll be in a much better position to develop quality software in a repeatable and predictable fashion.

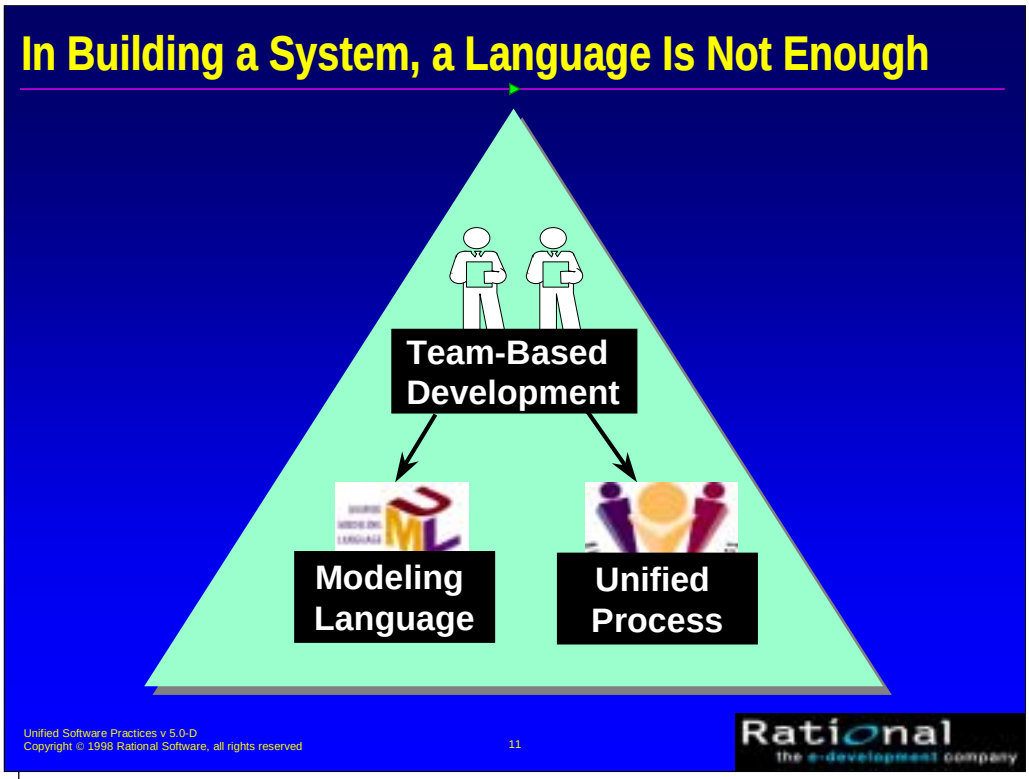
Best practices are a set of commercially proven approaches to software development which, when used in combination, strike at the root causes of software development problems. These are so called "best practices" not so such much because we can precisely quantify their value, but rather, because they are observed to be commonly used in industry by successful organizations.

These six best practices form the foundation of the Rational Unified Process. They are documented in The Rational Unified Process - An Introduction, by Philippe Kruchten (Addison-Wesley-Longman, 1998).


Addressing Root Causes Eliminates the Symptoms
Symptoms
end-user needs
changing requirements
modules don't fit
hard to maintain
late discovery
poor quality
poor performance
colliding developers
build-and-release
Root Causes
insufficient requirements
ambiguous communications
brittle architectures
overwhelming complexity
undetected inconsistencies
poor testing
subjective assessment
waterfall development
uncontrolled change
insufficient automation
Best Practices
develop iteratively
manage requirements
use component architectures
model the software visually
verify quality
control changes
Unified Software Practices v 5.0-D
Copyright © 1998 Rational Software, all rights reserved
9
Rational
the e-development company



In the remainder of this module, we describe recommended software development practices and give the reasons for these recommendations. We will use this key graphic again and again in presenting the 6 best practices.



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 *process* of development.

Despite all of the value that a common modeling language brings, successful development of today's complex systems cannot be achieved solely by the use of the UML. Successful development also requires the employment of an equally robust development process.

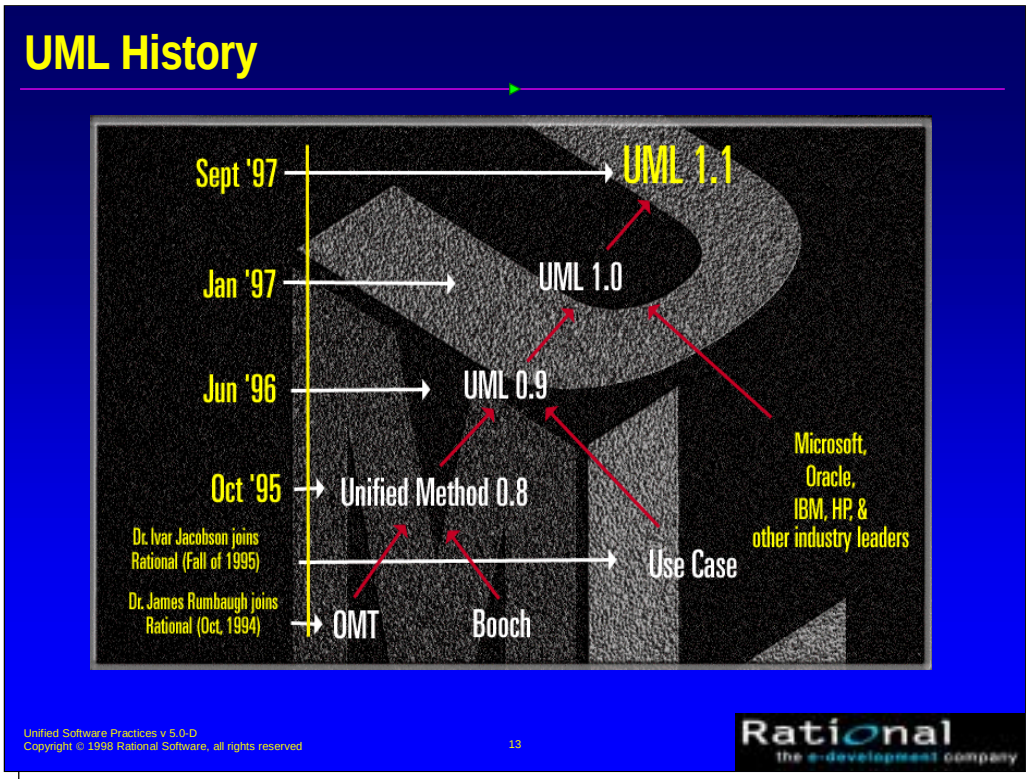
What Is the UML?

- The Unified Modeling Language (UML) is a language for
  - Specifying
  - Visualizing
  - Constructing
  - Documenting

  the artifacts of a software-intensive system

12
Rational
the e-development company

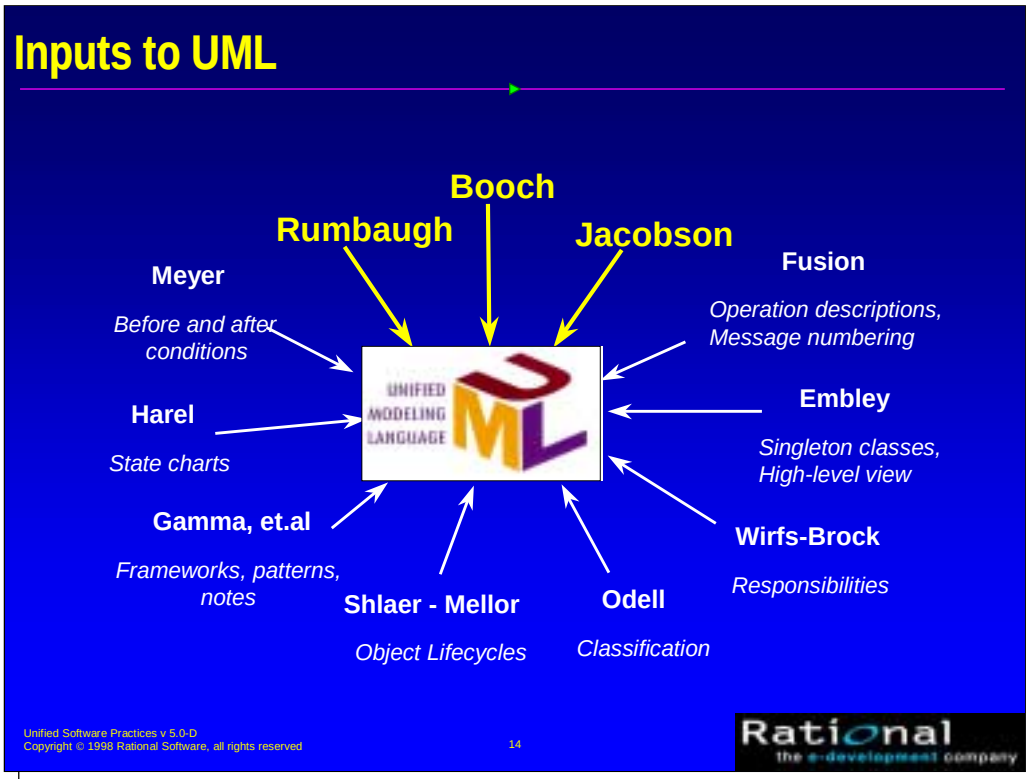
The software systems we develop today are much more complex than the human mind can comprehend in their entirety. This is why we model systems. The choice of what models to create has a profound influence upon how a problem is attacked and how a solution is shaped. No single model is sufficient; every complex system is best approached through a small set of nearly independent models. To increase comprehension, a common language like the Unified Modeling Language is used to express models.



The UML is now the industry standard modeling language. Its development was spearheaded by three leaders in the object-oriented industry: Grady Booch, Ivar Jacobson, and Jim Rumbaugh. Each of them had unique approaches in the early 1990s that were converging independently. Rational Software brought these three industry leaders together to accelerate convergence to a single modeling approach. The UML is the outcome of that effort.

The UML has been under development since 1990. Early versions have gradually converged into the UML standard 1.1 that was adopted unanimously by the OMG in November 1997. Numerous books, articles, and papers have been written about or are using the UML today and many more are to come. And finally, most tool vendors today are supporting the UML.

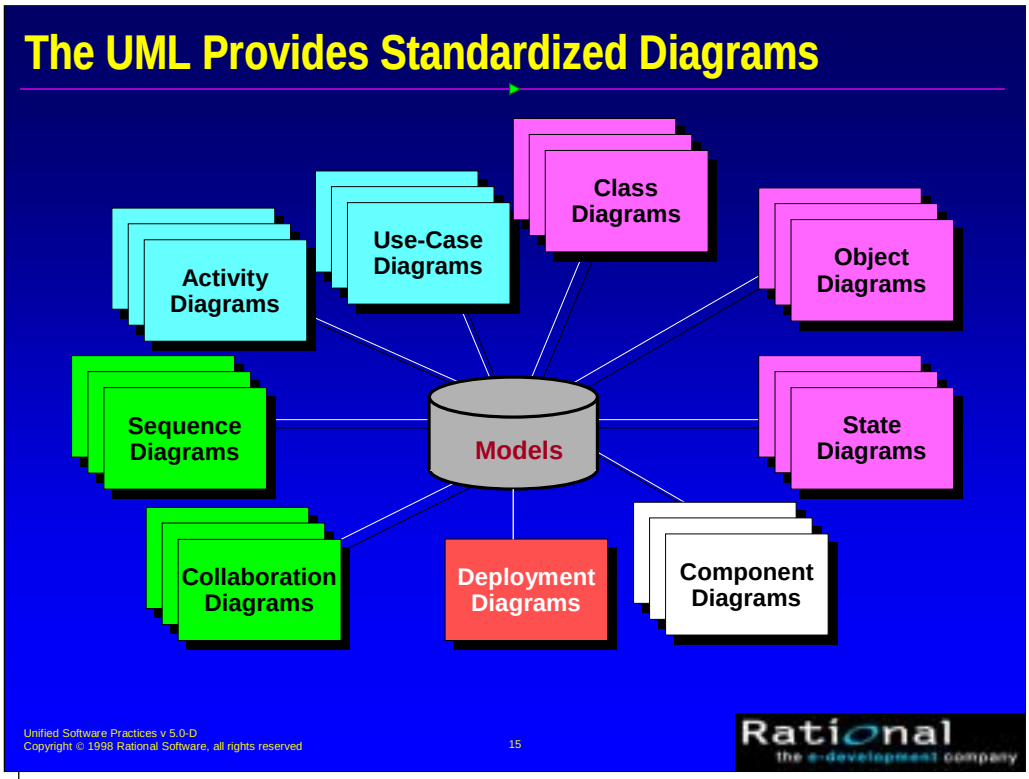
Version 1.3 of the UML is under development at the present time.



UML development included incorporating ideas from numerous other methodologists. The main challenge was constructing an approach that was simple yet allowed the modeling of a broad range of systems. The conceptual framework was established quickly but the notational semantics took more time.

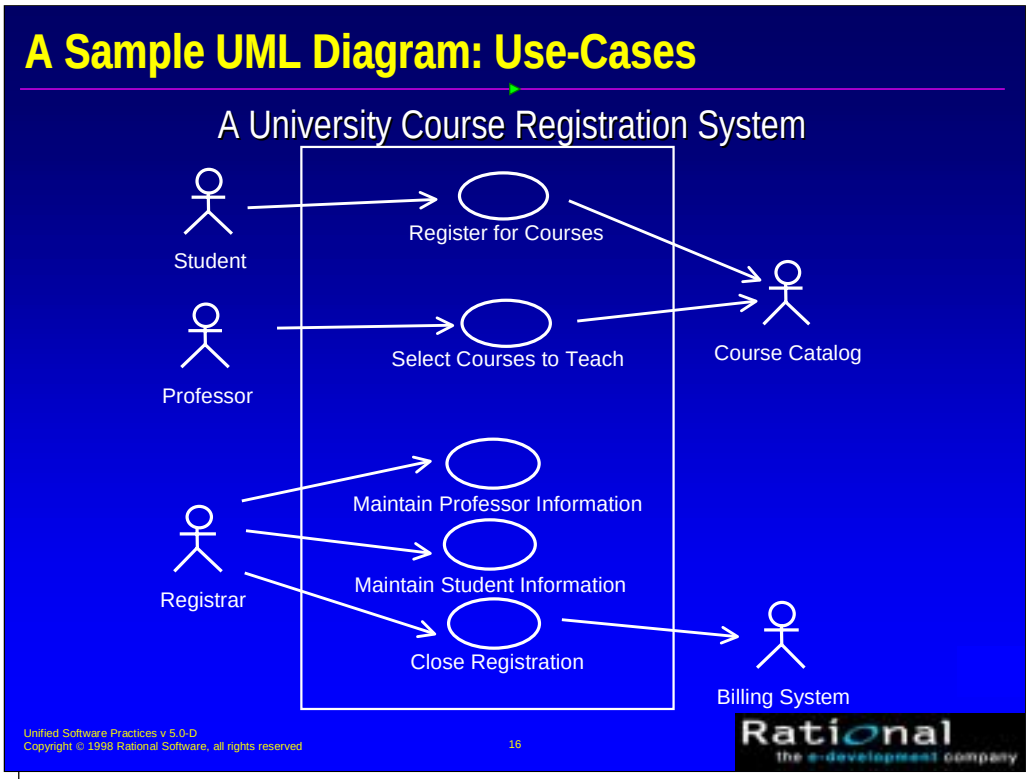
Active collaboration with other industry leaders has brought unique expertise and experience into the UML effort. The UML effort was supported by a large cross-section of the industry. Partners in the UML effort included HP, ICON Computing, IBM, I-Logix, Intellicorp, MCI Systemhouse, Microsoft, ObjecTime, Oracle, Platinum Technology, Ptech, Reich Technologies, Softeam, Sterling Software, Taskon, and Unisys. These partners provided contributors, reviewers, and advocates for the standardization efforts.

In the end, a modeling language was created that has already stood up to the test of widespread use in the industry and to the scrutiny of international standardization efforts.



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 our 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.
- State diagrams to illustrate behavior.
- Component diagrams to illustrate physical structure of the software.
- Deployment diagrams to show the mapping of software to hardware configurations.
- Interaction diagrams (i.e., collaboration and sequence diagrams) to illustrate behavior.
- Activity diagrams to illustrate the flow of events in a Use-Case.

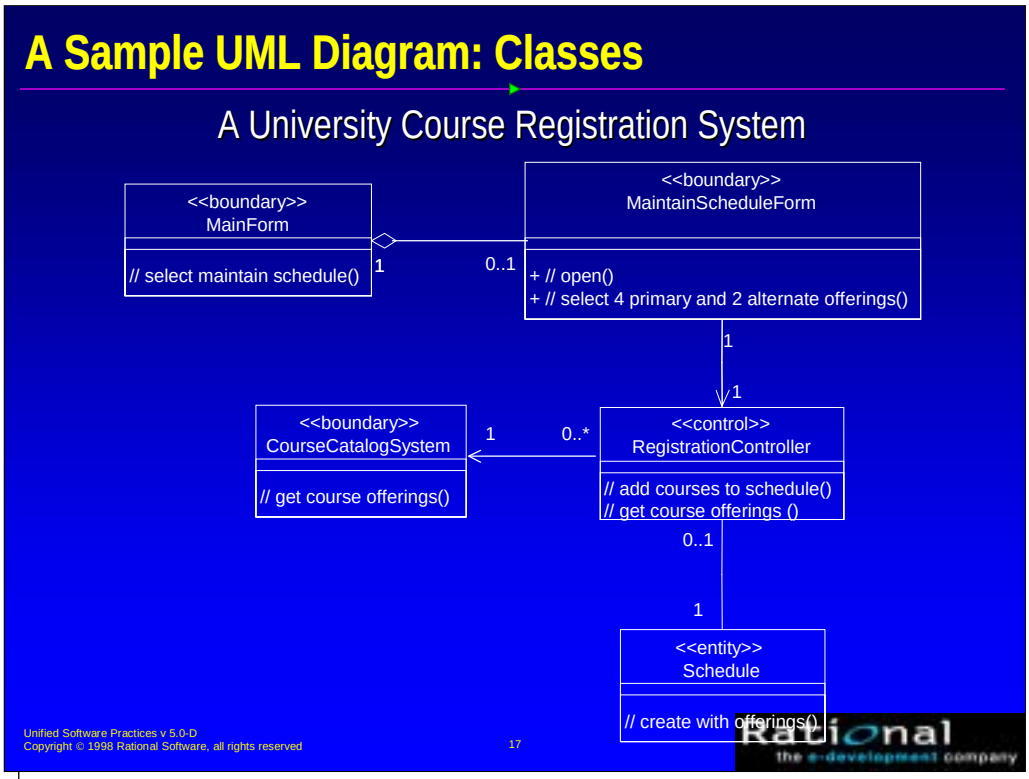


Use-Case diagrams are used to show the existence of Use-Cases and their relationships, both to each other and to actors. An actor is something external to the system that has an interface with the system, such as users. A Use-Case models a dialogue between actors and the system. A Use-Case is initiated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

The arrow (which is optional) indicates the direction of initiation of the interaction. For example, the Student actor initiates the Register for Courses Use-Case. However, the Close Registration Use-Case initiates interaction with the Billing System.

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.

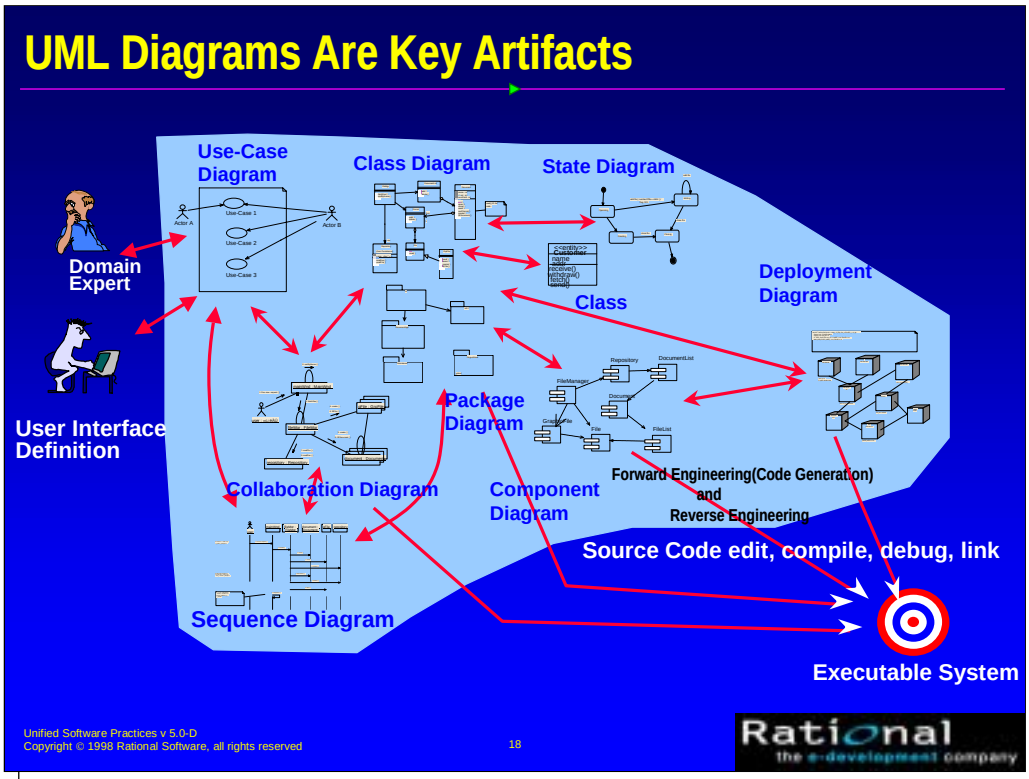
Taken together, all Use-Cases constitute all possible ways of using the system.



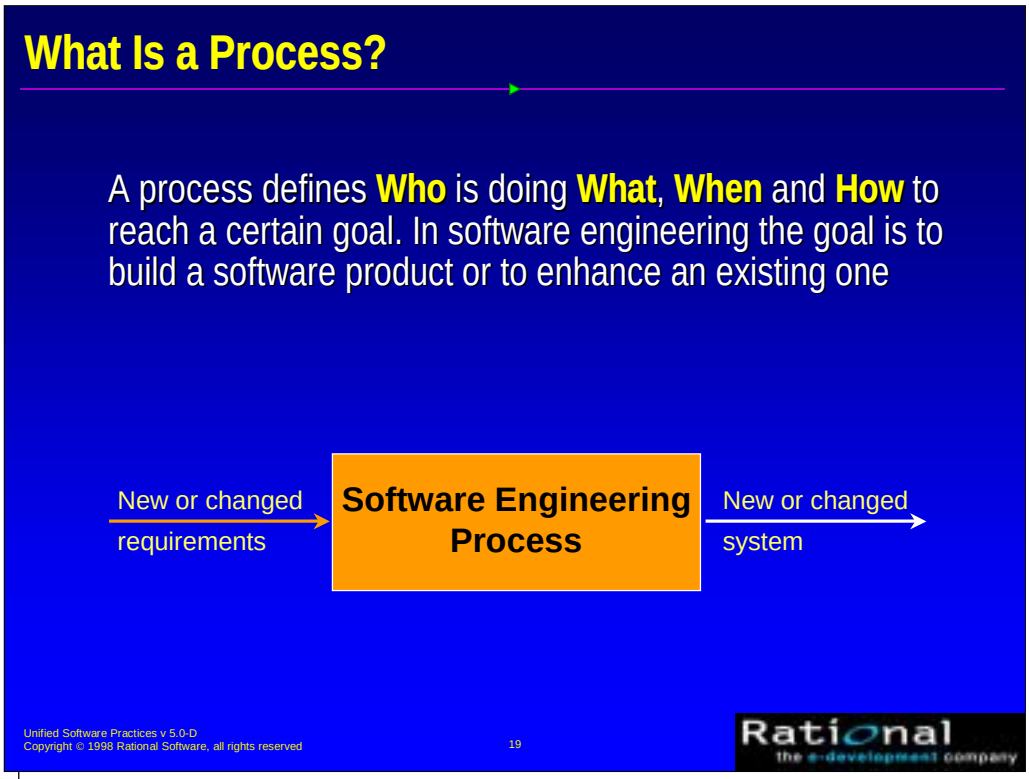
A class diagram is a view of the classes composing the system, as well as their relationships. Classes are represented by rectangles. Relationships are represented by lines. The presence of a relationship between two classes indicates that they collaborate in some way in performing one or more Use-Cases. For example, the relationship between MainForm and MaintainScheduleForm means that a MainForm may or may not contain one MaintainScheduleForm.

The items preceded by "//" in the lower third of the class boxes indicate the responsibilities of the classes. For example, CourseCatalogSystem is responsible for get course offerings. During design of this class, this responsibility will be turned into one or more operations.

The items within angle brackets ("<< >>") indicate the stereotype of the class. For example, CourseCatalogSystem is a boundary class which means that it interfaces with an external entity (I.e., an actor).

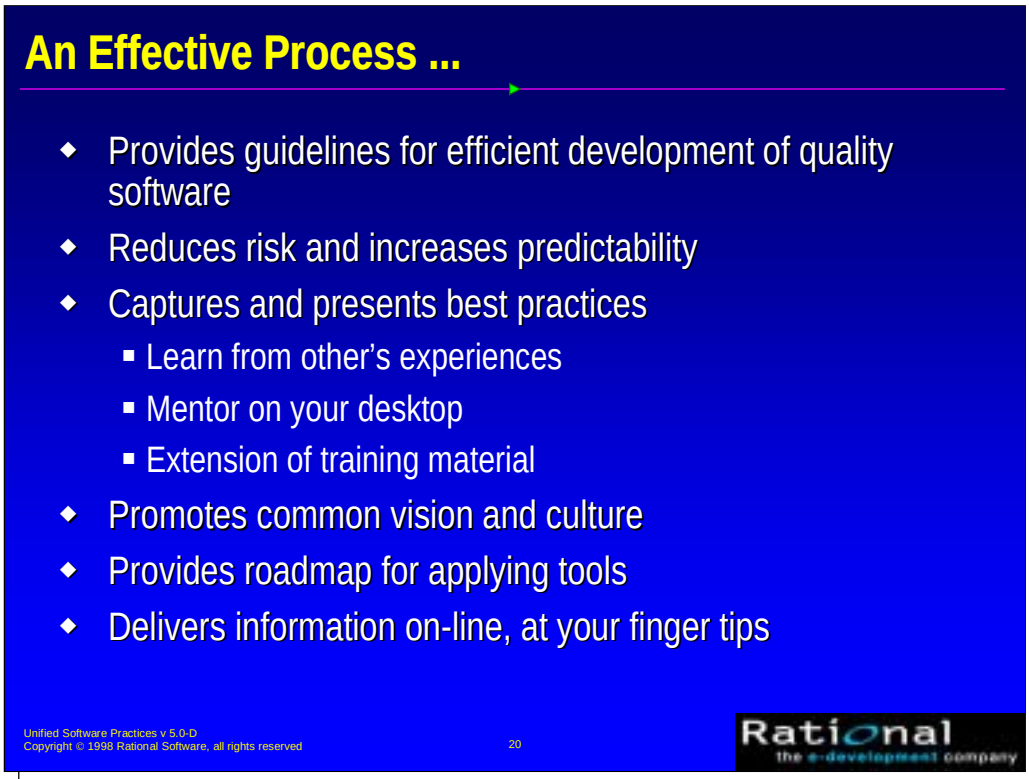


The UML provides a single, common modeling language that is useable across many methods, across the entire lifecycle, and across many different implementation technologies. It maps the artifacts between business engineering, requirements capture and analysis, design, implementation, and test. It defines an expressive and consistent notation that can point out omissions and inconsistencies. It scales to support very small to very large systems development. If facilitates effective communications with all members of the development team.



The UML is a generic modeling language. With UML, you can produce blueprints for any kind of software system.

The Rational Unified Process (RUP) is a generic process that uses UML as a modeling language. RUP can be used for any kind of software system.

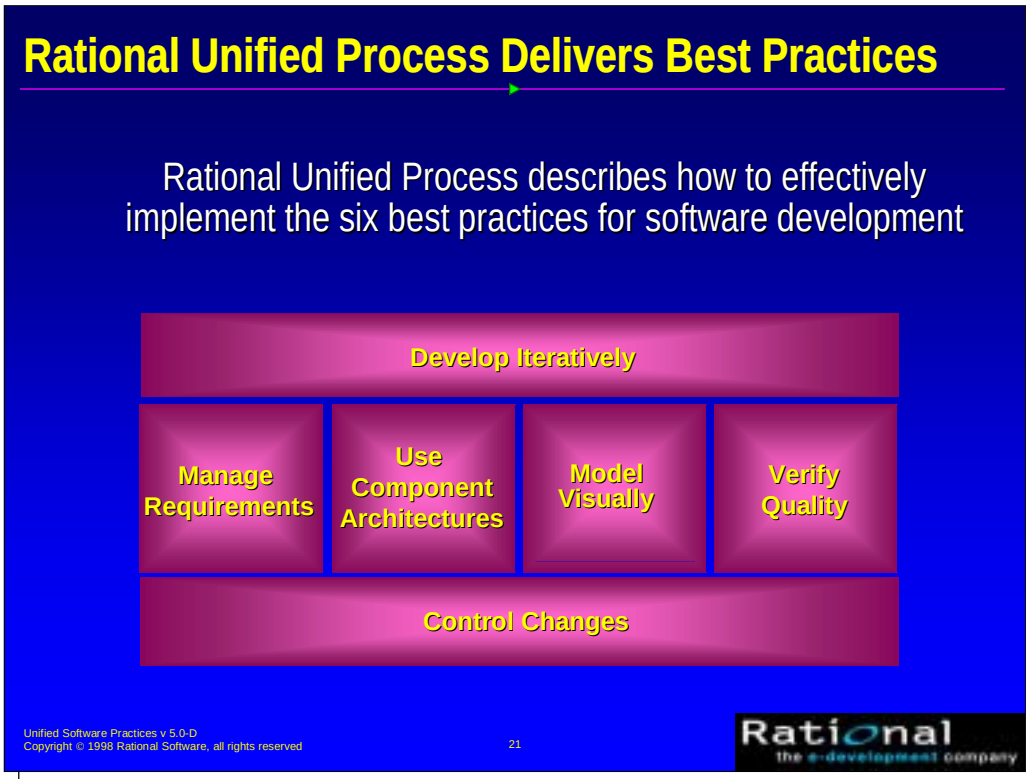


The focus of the process is the production of high-quality executables with a minimum of overhead, rather than what documents to produce.

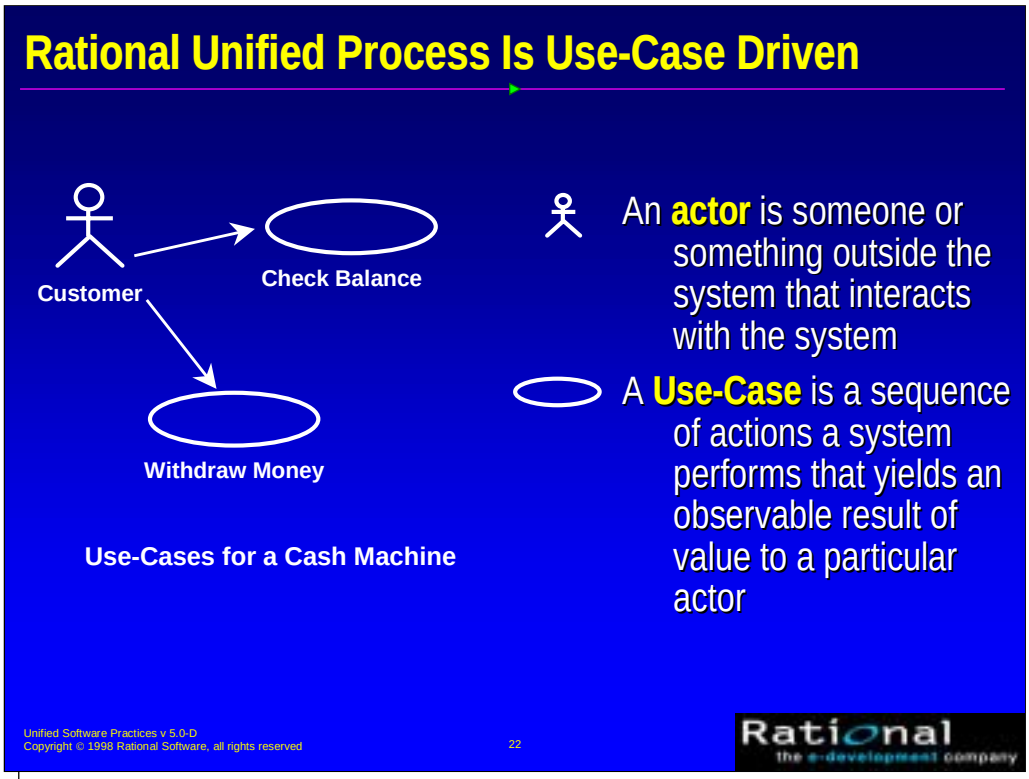
By providing a "written-down", very detailed set of procedures for developing software according to Rational's six best practices, the process is easier to apply and repeatable. This results in software projects that are more predicable and successful.

Another feature of the Rational Unified Process is that it is not just theory. It instructs the developer on how to implement the activities using the tools the developer is using.

Finally, the process is available on-line as a Website. While hardcopy books have their place, most developers prefer to reference the process from their desktops when they need help. Both hardcopy and on-line versions are available.



The six best practices examined in the previous module provide the basis for the Rational Unified Process. However, their effective application requires step-by-step instructions. These instructions are provided in the Rational Unified Process, which looks at all activities that must be performed to build a software product and describes how to conduct those activities consistent with the 6 best practices. Tool mentors are included to explain how to conduct those activities using the Rational tool set.



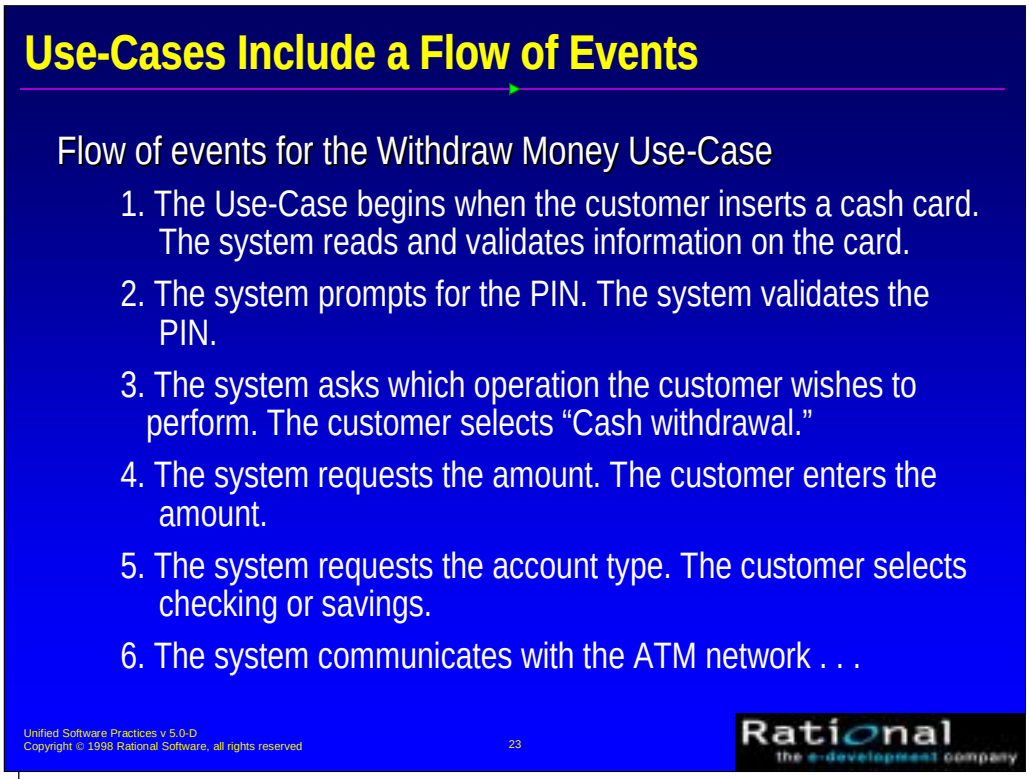
Managing requirements is one of the six best practices already discussed. In particular, the Rational Unified Process captures and manages functional requirements in a Use-Case model. Use-Cases are a key concept within the process. They are used throughout the development cycle as a driver for many activities, flowing information through several models, and encouraging consistency across these models.

An actor can be a human being or another system or a device; anything external with which the system interacts.

A Use-Case describes functionality from the user's point of view. A bank customer can use a cash machine to withdraw money or check the balance of her account. Each Use-Case (oval) represents something that the system does to provide value to the bank's customer, the Customer. The collected Use-Cases constitute the use-case model and represent all the possible ways of using the system. The use-case model is a model of the system's intended functions and its environment, and it serves as a contract between the customer and the developers.

The use-case model is complemented by non-functional specifications covering some of the aspects that cannot be easily conveyed by a Use-Case, or that apply to many or all Use-Cases; for example, high-level product requirements, development constraints, safety or security attributes, and so on.

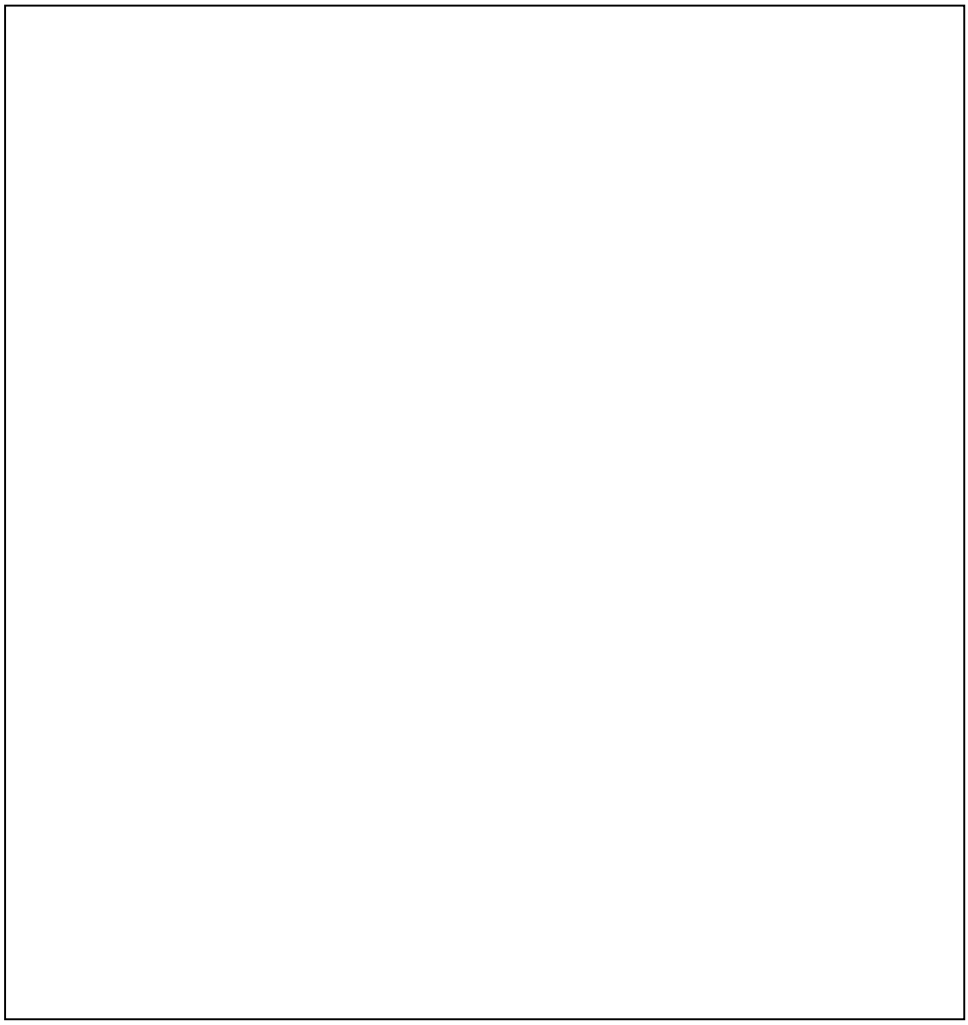
The notation used for representing the use-case model in diagrams is standard notation provided by the UML.

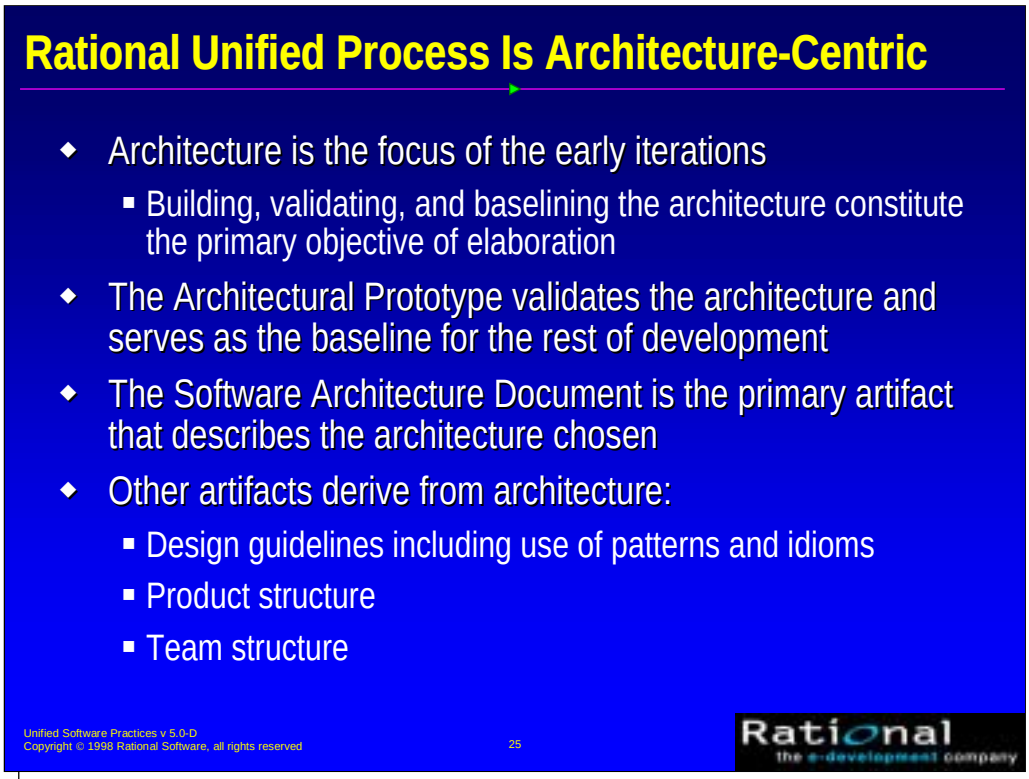


The most important part of the Use-Case is the its flow of events, describing the sequence of actions. It is written in natural language, in a simple, consistent prose, with a precise use of terms, drawing upon a common glossary of the problem domain. For example, the term "ATM" would be an acronym commonly used in the banking industry and defined in the project glossary (as Automated Teller Machine).

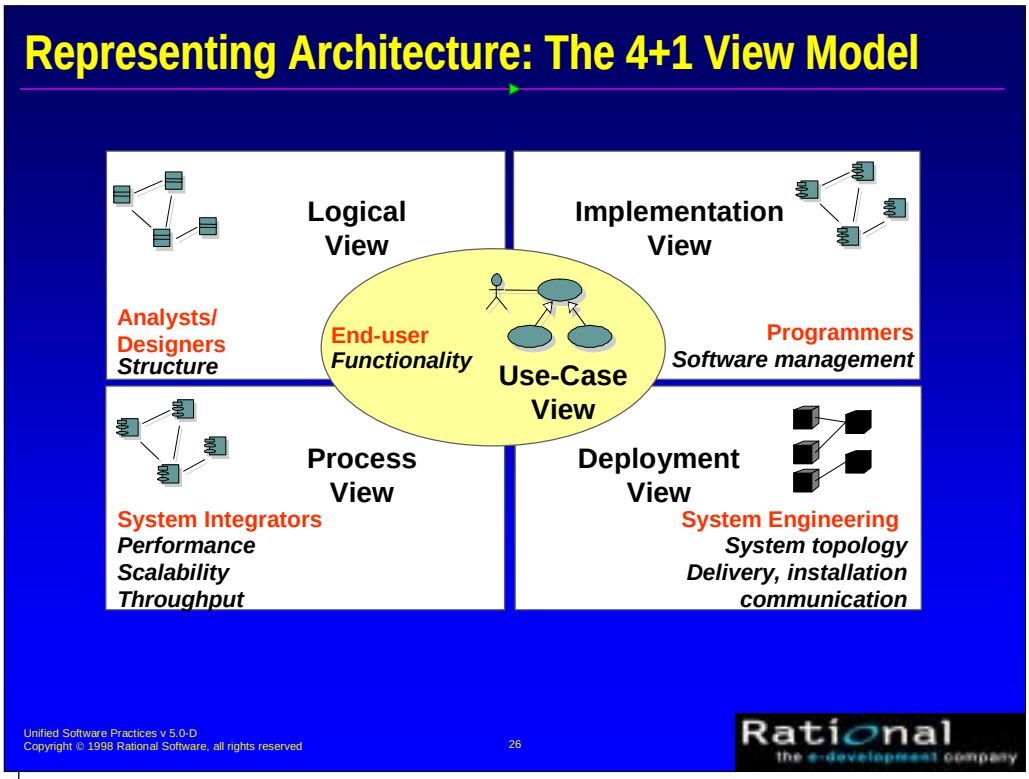
Benefits of a Use-Case Driven Process

- Use-Cases are concise, simple, and understandable by a wide range of stakeholders
  - End users, developers and acquirers understand functional requirements of the system
- Use-Cases drive numerous activities in the process:
  - Creation and validation of the design model
  - Definition of test cases and procedures of the test model
  - Planning of iterations
  - Creation of user documentation
  - System deployment
- Use-Cases help synchronize the content of different models





Use of component-based architectures is one of the six best practices already discussed. Architecture is used in the Rational Unified Process as a primary artifact for conceptualizing, constructing, managing, and evolving the system under development. The Rational Unified Process emphasizes early development and validation of software architecture as a core concept. It defines two primary artifacts related to architecture: the Software Architecture Description (SAD) which describes the architectural views relevant to the project and the Architectural Prototype. The Rational Unified Process also defines a worker, called the Architect, who is responsible for the architecture. The bulk of the activities related to architectural design are described in the analysis and design workflow, but it spills over to the requirements workflow, the implementation workflow, and the project management workflow.



Many different parties are interested in the architecture (e.g., the system analyst, the designers, the end uses, etc.). To allow these parties or stakeholders to communicate, discuss and reason about architecture, we need to have an architectural representation that they understand. Because different stakeholders have different concerns and because architecture is quite complex, multiple views are required to represent architecture adequately. An **architectural view** is a simplified description (an abstraction) of a system from a particular perspective or vantage point, covering particular concerns and omitting entities that are not relevant to this perspective.

While many views of architecture can be useful, the Rational Unified Process identifies 4+1 views as a standard set:

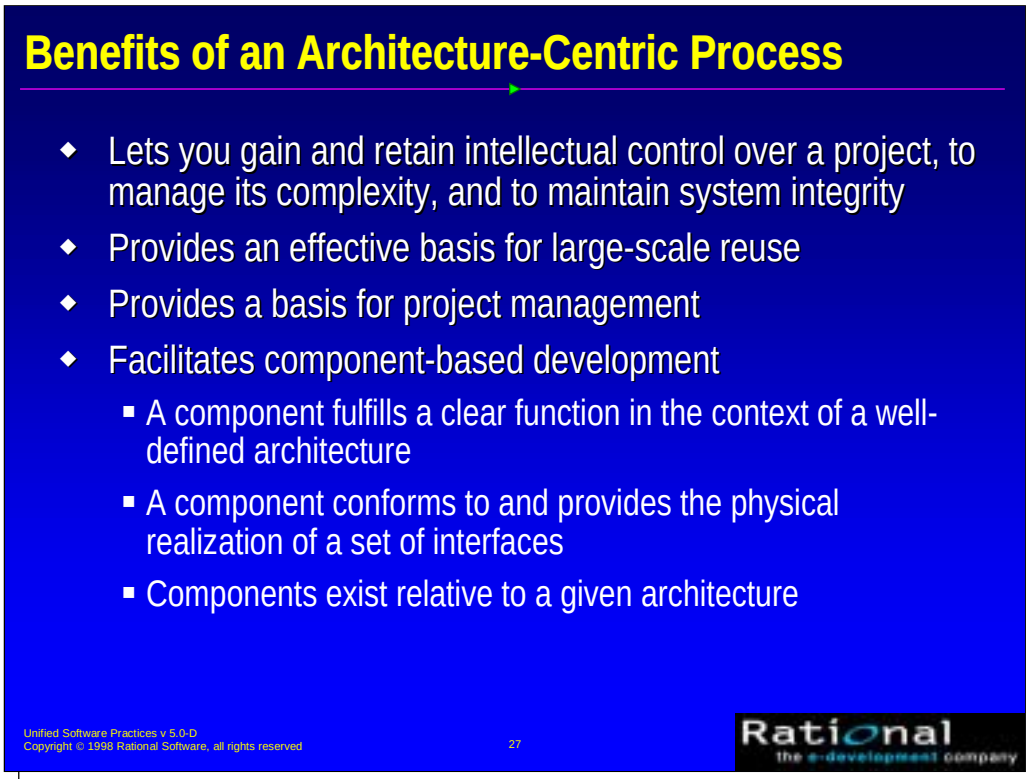
The logical view addresses the functional requirements of the system. It is an abstraction of the design model, identifying major design packages, subsystems and classes.

The implementation view describes the organization of static software modules in the development environment, in terms of packaging, layering, and configuration management.

The process view addresses the concurrent aspect of the system at run-time: tasks, threads or processes, and their interactions.

The deployment view shows how the various executables and other run-time components are mapped onto the underlying platforms or computing nodes.

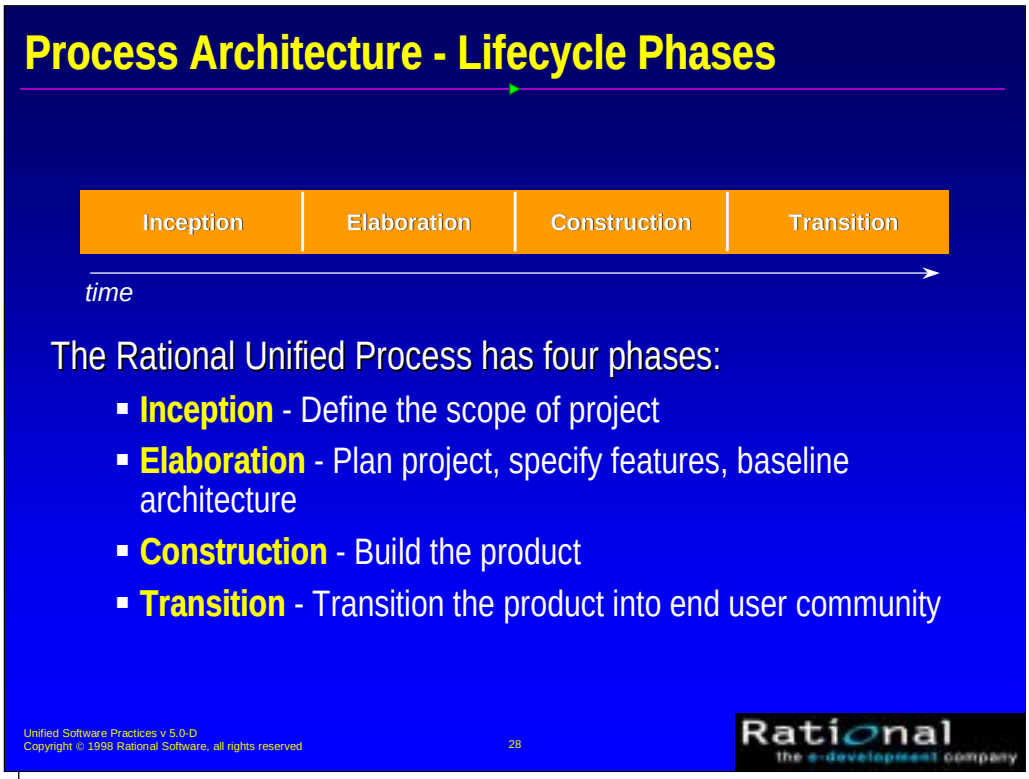
The use-case view contains a few key scenarios or Use-Cases that are used to drive the architecture and to validate it.



A complex system is more than the sum of its parts, more than a succession of small independent tactical decisions. It must have some unifying, coherent structure to organize those parts systematically, and provide precise rules on how to grow the system without having its complexity “explode” beyond human understanding. Architecture provides this structure and these rules.

By clearly articulating the major components and the critical interfaces among them, an architecture lets you reason about reuse, both internal reuse (the identification of common parts), and external reuse (the incorporation of ready-made, off-the-shelf components. Architecture can also facilitate reuse on a larger scale: the reuse of the architecture itself in the context of a product line that addresses different functionality in a common domain.

Planning and staffing are organized along the lines of major components: layers and subsystems.



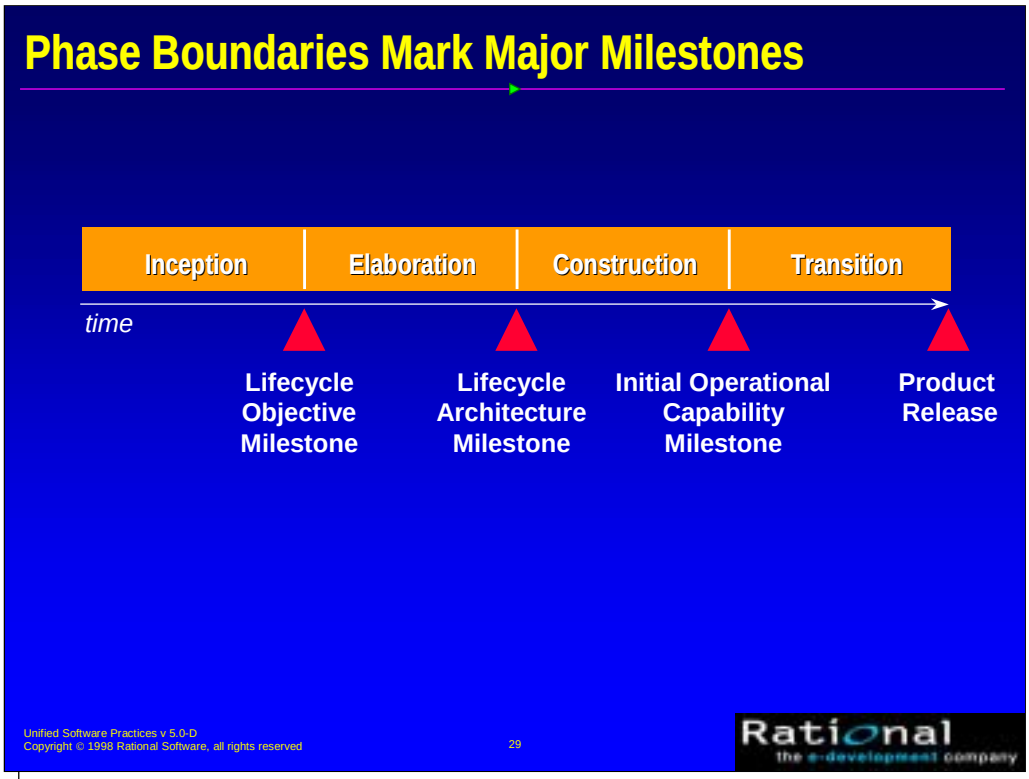
During Inception, we define the scope of the project, what is included, and what is not. This is done by identifying all the actors and Use-Cases, and by drafting the most essential Use-Cases (usually approximately 20% of the complete model). A business plan is developed to determine whether resources should be committed to the project.

During Elaboration, we focus on two things: get a good grasp of the requirements (90% complete) and establish an architectural baseline. If we have a good grasp of the requirements and the architecture, we can eliminate a lot of the risks and will have a good idea what amount of work remains to be done. Detailed cost/resource estimations can be made at the end of Elaboration.

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

During Transition, we transition 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 very complex project with a lot of technical unknowns and unclear requirements, Elaboration may include 3-5 iterations. For a very simple project where requirements are known and the architecture is simple, Elaboration may include only a single iteration.



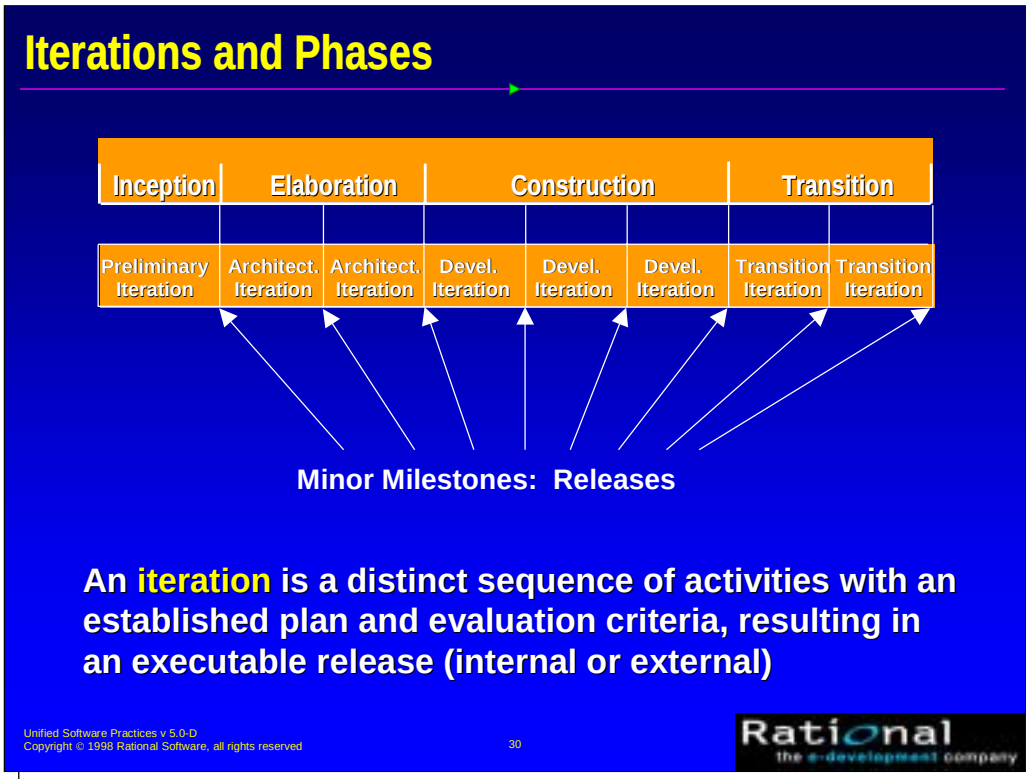
At each of the major milestones,the project is reviewed and a decision made as to whether to proceed with the project as planned, to abort the project, or to revise it. The criteria used to make this decision vary by phase.

The evaluation criteria for the inception phase (LCO) include: stakeholder concurrence on scope definition and cost/schedule estimates; requirements understanding as evidenced by the fidelity of the primary Use-Cases; credibility of cost/schedule estimates, priorities, risks, and development process; depth and breadth of any architectural prototype; actual expenditures versus planned expenditures.

The evaluation criteria for the elaboration phase (LCA) include: stability of the product vision and architecture; resolution of major risk elements; adequate planning and reasonable estimates for project completion; stakeholder acceptance of the product vision and project plan; acceptable expenditure level.

The evaluation criteria for the construction phase (IOC) include: stability and maturity of the product release (I.e., is it ready to be deployed?); readiness of the stakeholders for the transition; acceptable expenditure level.

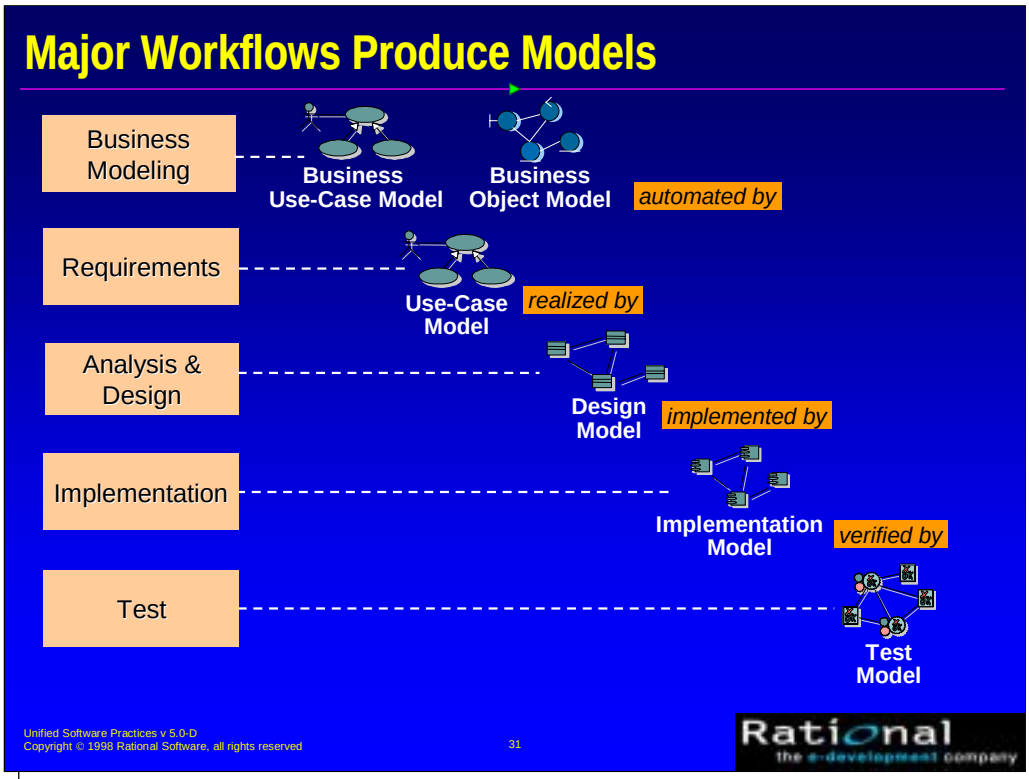
At the end of the transition phase, a decision is made whether to release the product. This will be based primarily on the level of user satisfaction achieved during the transition phase. Often this milestone coincides with the initiation of another development cycle to improve or enhance the product. In many cases, this new development cycle by already be underway.



Within each phase, there is a series of iterations. The number of iterations per phase will vary. Each iteration results in an executable release encompassing larger and larger subsets of the final application.

An internal release is kept within the development environment and (optionally) demonstrated to the stakeholder community. An external release is provided to stakeholders (usually users) for installation in their own environment. External releases are much more expensive (they require user documentation and technical support) and normally occur only during the transition phase.

The end of an iteration marks a minor milestone. It is a point in time when technical results are assessed and future plans revised as necessary.



The Rational Unified Process is a model-driven approach. Several models are needed to fully describe the evolving system. Each major workflow produces one of those models. The models are developed incrementally across iterations.

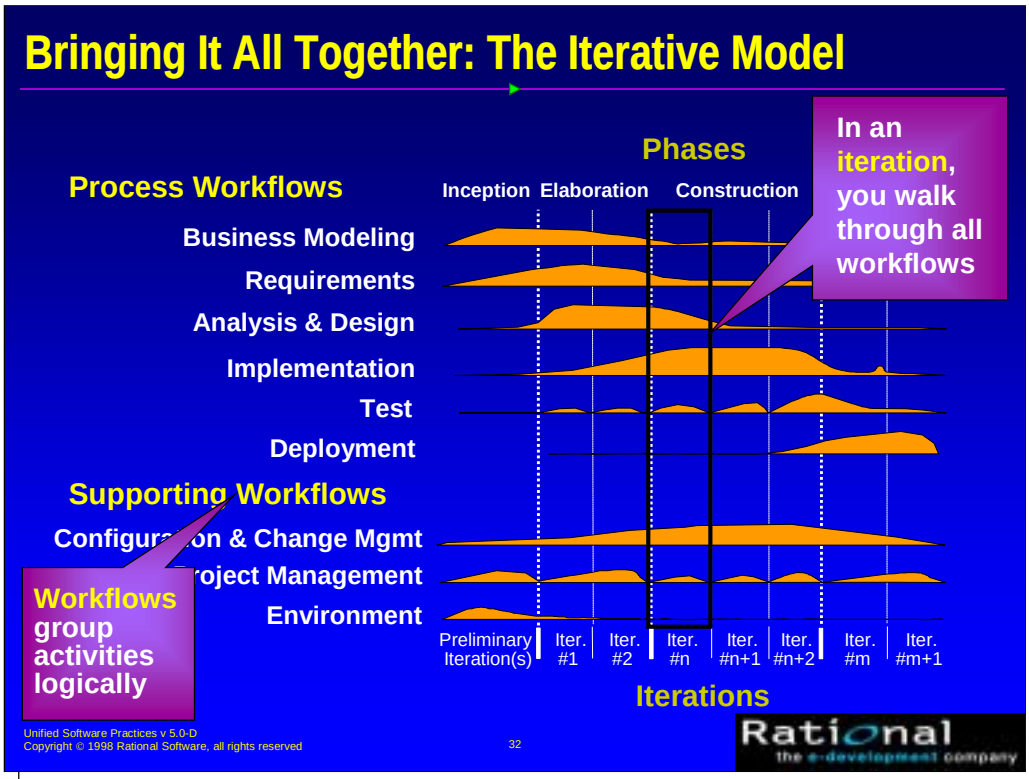
The Business Model is a model of what the business processes are and the business environment. It can be used to generate requirements on supporting information systems.

The Use-Case Model is a model of what the system is supposed to do and the system environment.

The Design Model is an object model describing the realization of Use-Cases. It serves as an abstraction of the implementation model and its source code.

The Implementation Model is a collection of components, and the implementation subsystems that contain them.

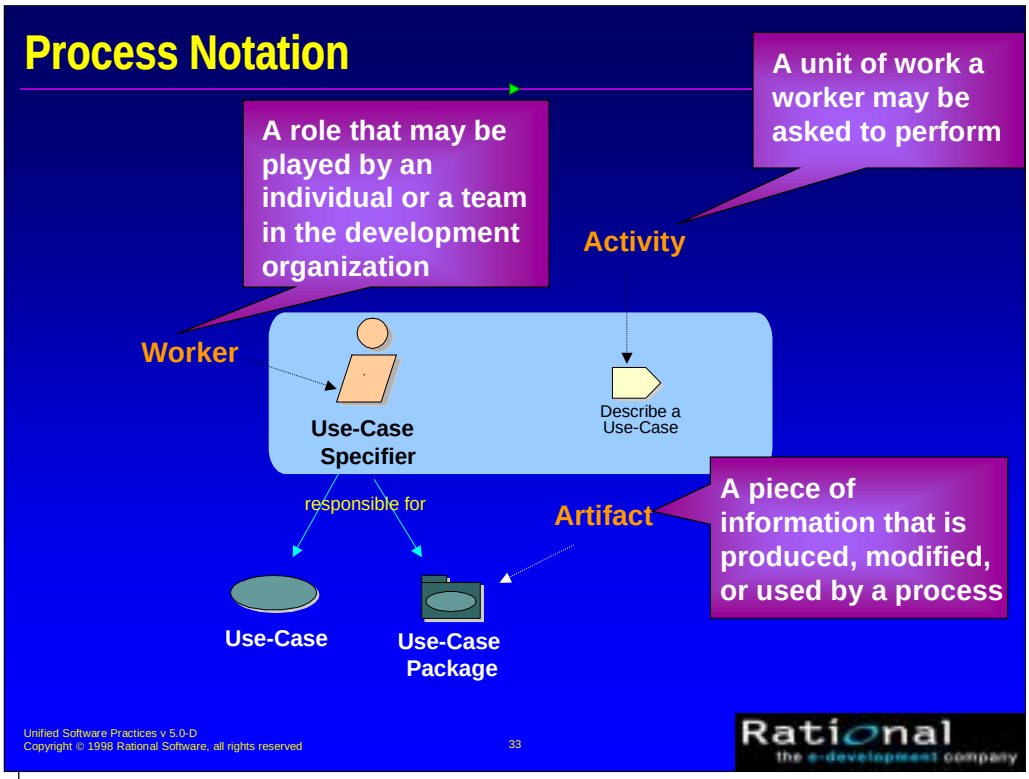
The Test Model encompasses all of the test cases and procedures required to test the system.



This graphic illustrates how phases and iterations, or the time dimension, relates to the development activities performed, or the workflow dimension. The relative size of the color area indicates how much of the activity is performed in each phase/iteration.

Each iteration involves activities from all workflows. The relative amount of work related to the workflows 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.

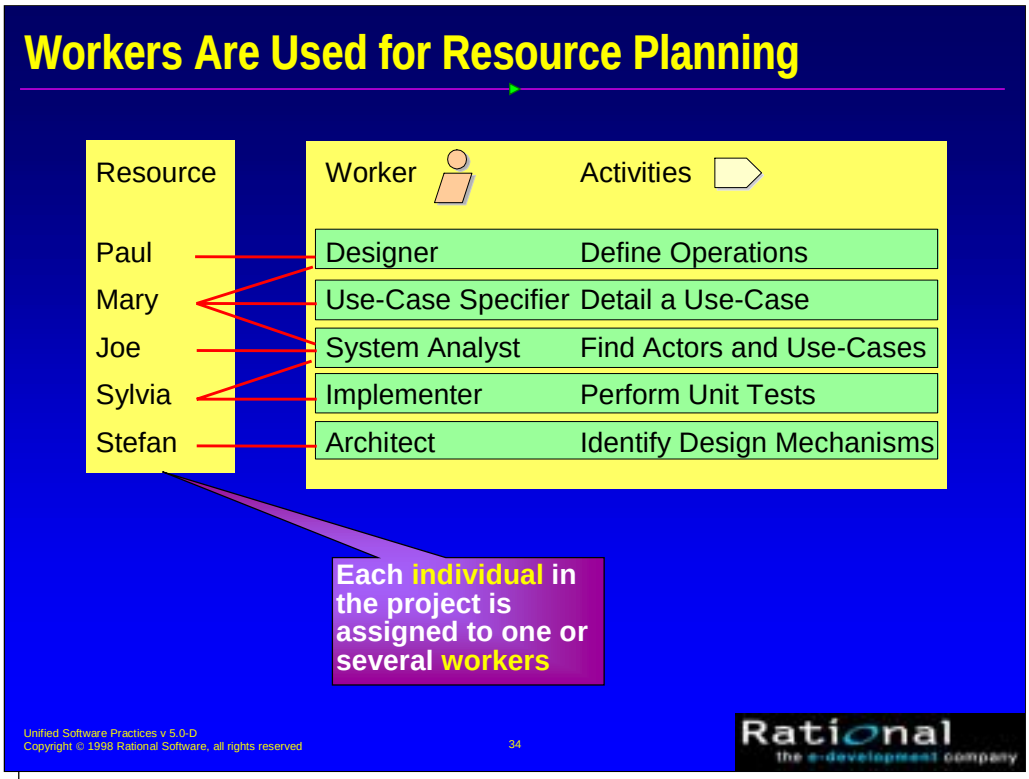
Note that requirements are not necessarily complete by the end of Elaboration. It is acceptable to delay the analysis and design of well-understood portions of the system until Construction because they are low in risk.



An **artifact** is something you produce in the course of developing a software product. It includes the source code itself as well as the models, documents and other products of the life cycle. The UML provides notation for representing many of the artifacts of the development process.

An **activity** is the smallest piece of work that is relevant. It is not reasonable to do only part of an activity. Dividing the work in this manner makes it easier to monitor development. It is better (easier) to know that the project has completed three out of five activities rather than 60% of one activity. Examples of activities are Plan an Iteration, and Review The Design.

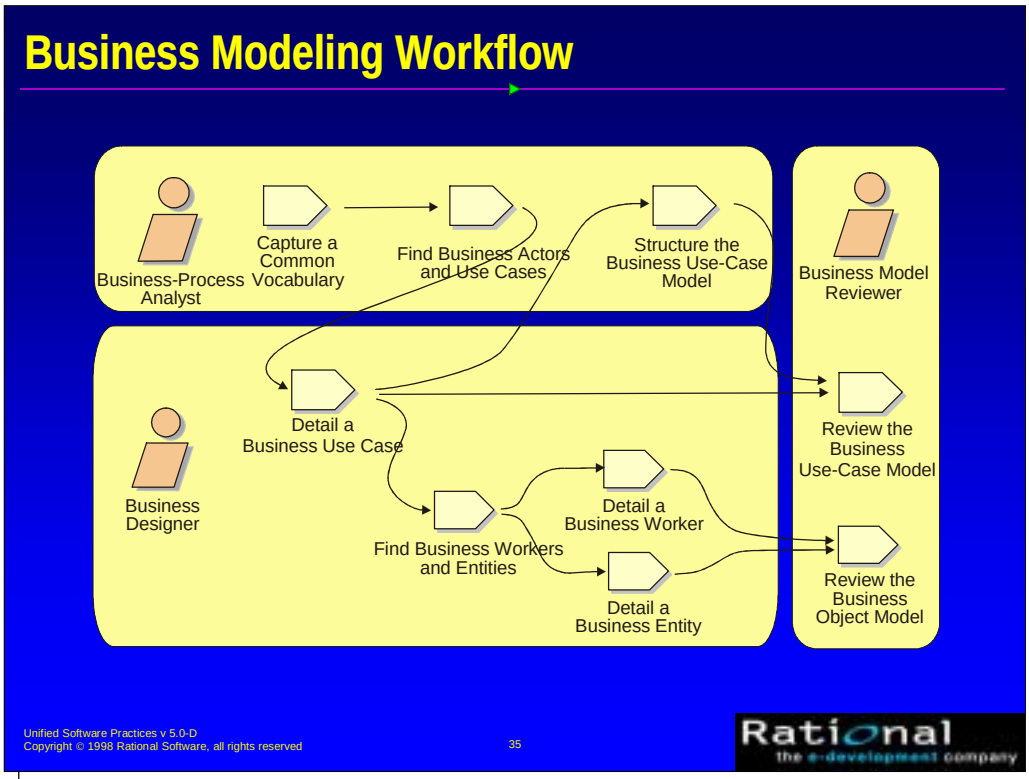
A **worker** defines the behavior and responsibilities of an individual, or a set of individuals working together as a team. This is an important distinction because it is natural to think of a worker as the individual or the team itself. In the Rational Unified Process, the worker is more of a role that defines how the individuals should carry out the work. An example of a worker is a project manager.



In developing a project plan, a project manager assigns the individuals available to workers according to their skills and abilities. Each individual on the project is assigned to one or several workers. The association of individuals to workers is dynamic over time.

An individual may act as several different workers during the same day. We can informally call this “wearing several hats.” Sylvia may be both a design reviewer and a use-case designer.

Several individuals may act as the same worker to perform a certain activity as a team. Paul and Mary may serve as the Use-Case designer for a single Use-Case.

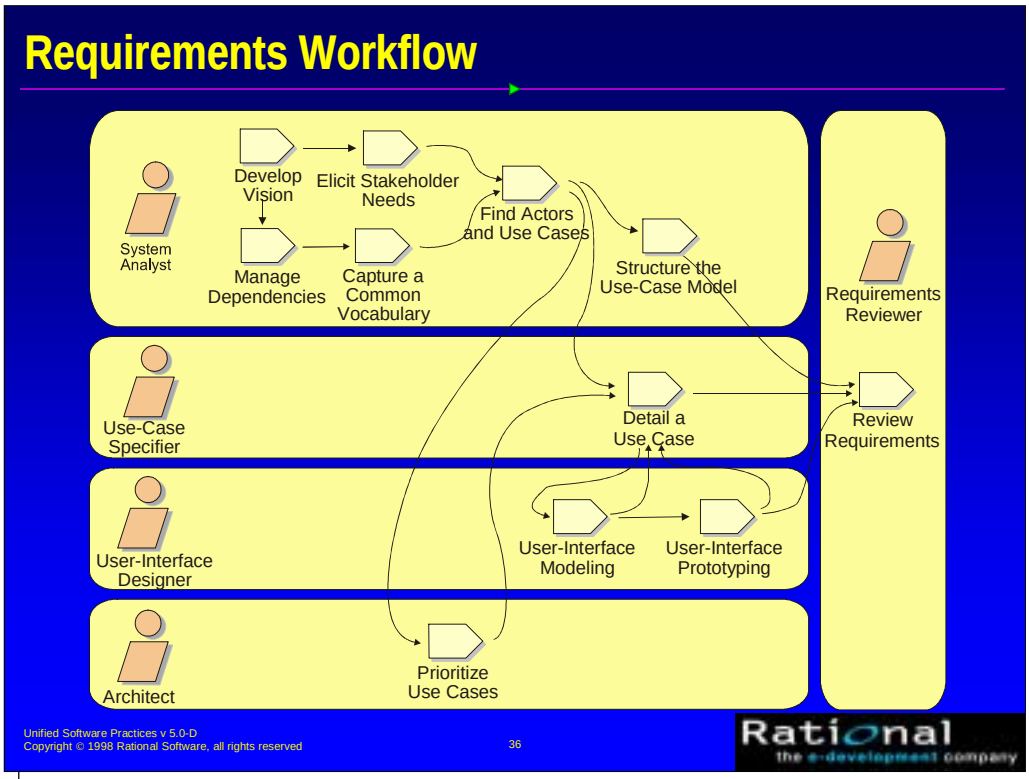


This workflow diagram shows business modeling activities and the workers responsible for them. Not everyone begins system development by doing business modeling. This is an optional step.

The purposes of the Business Modeling workflow are:

- To understand the structure and dynamics of the organization.
- To ensure that customers, end users and developers have a common understanding of the organization.
- To derive requirements on systems to support the organization.

To achieve these goals, a business use-case model and a business object model are developed. Complementary to these models, a Supplementary Business Specification and a Glossary are usually developed.



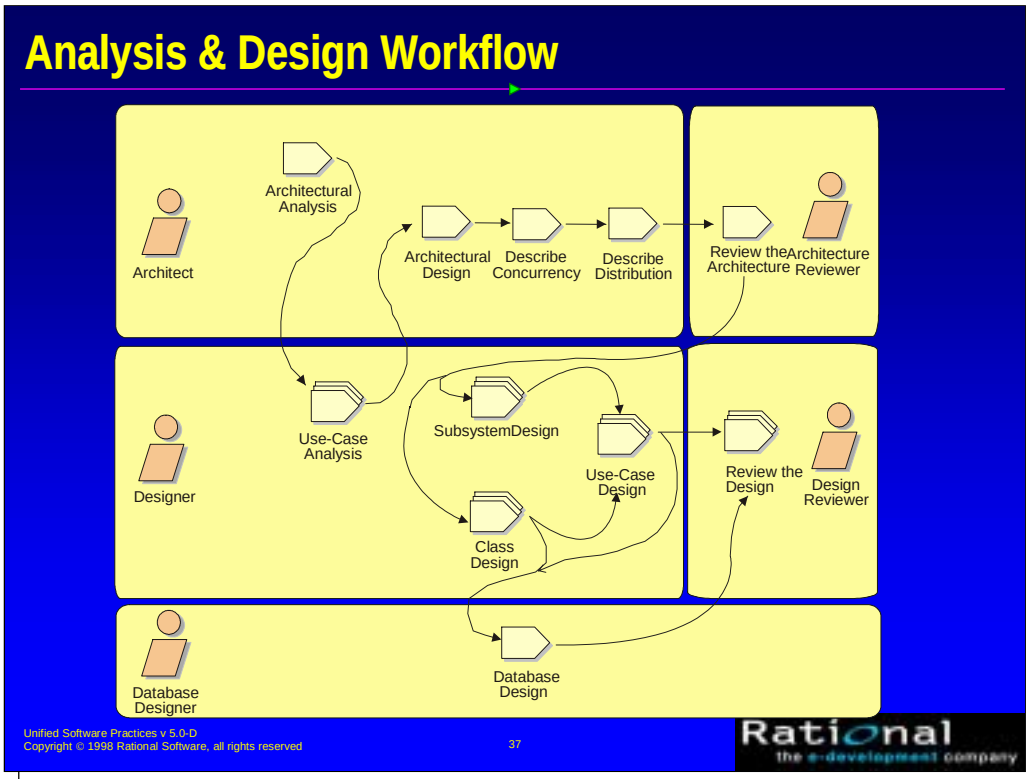
The purposes of the Requirements workflow are:

- To come to an agreement with the customer and the users on what the system should do.
- To give system developers a better understanding of the requirements on the system.
- To delimit the system.
- To provide a basis for planning the technical contents of iterations.
- To define a user-interface for the system.

To achieve these goals, a **Vision** document, a **Stakeholder Needs** document, a **use-case model**, and a **Supplementary Specificatio**n document are developed that describes **what** the system will do - an effort that views customers and potential users as important sources of information (in addition to system requirements).

Complementary to the above mentioned artifacts, the following artifacts are developed:

- Glossary
- Use-Case Storyboard
- Boundary Class
- User-Interface Prototype

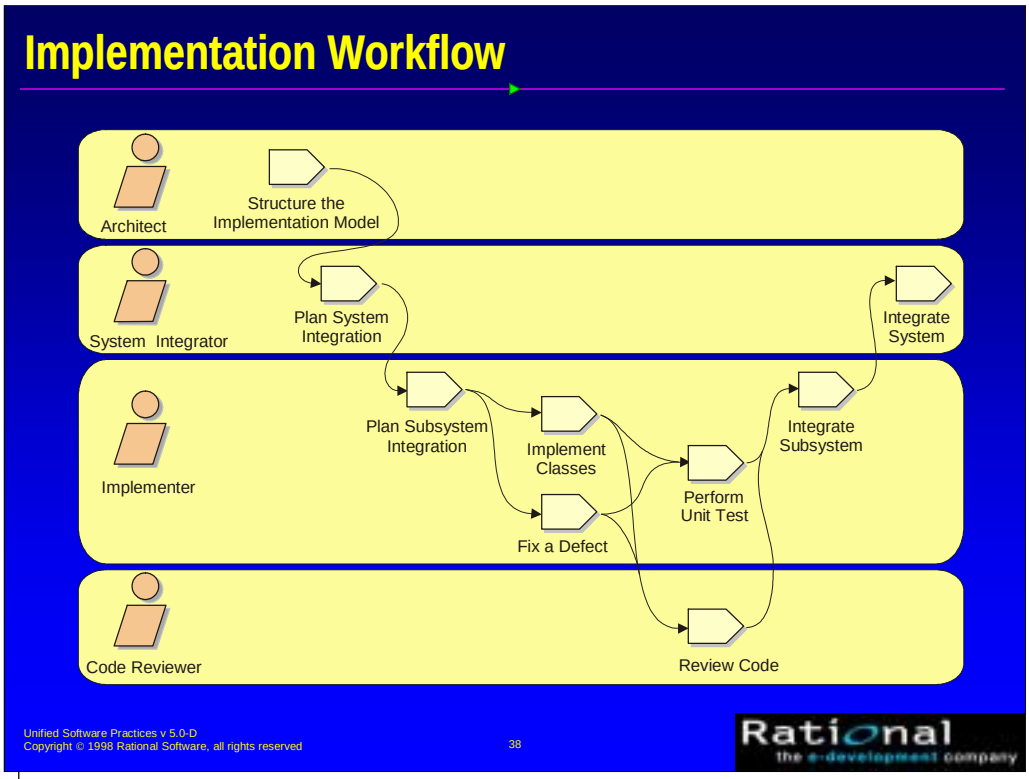


The purposes of the Analysis and Design workflow are:

- To transform the requirements into a design of the system to-be.
- To evolve a robust architecture for the system.
- To adapt the design to match the implementation environment, designing it for performance.

The primary artifacts of the Analysis and Design workflow are:

- Design model, defining Use-Case realizations, classes, and design packages/subsystems
- Software Architecture Document
- Data model



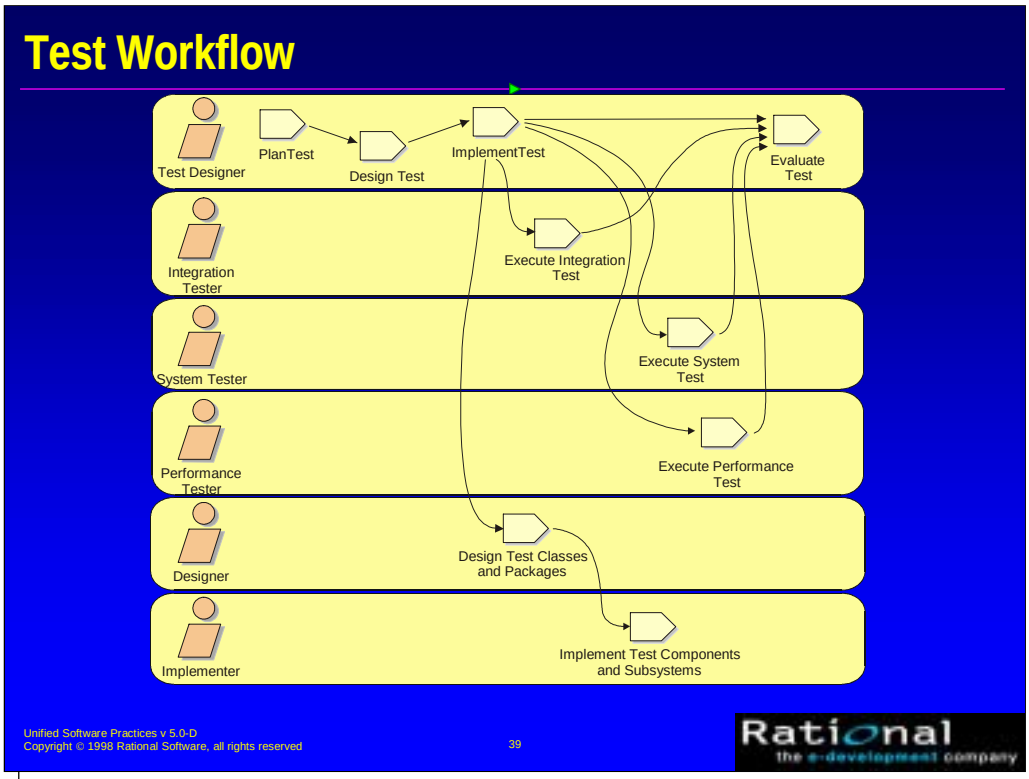
The purposes of the Implementation workflow are:

- To define the organization of the code, in terms of implementation subsystems organized in layers.
- To implement classes and objects in terms of components (source files, binaries, executables, and others).
- To test the developed components as units.
- To integrate the results produced by individual implementers (or teams), into an executable system.

The Implementation workflow limits its scope to how individual classes are to be unit tested. System test and integration test are described in the Test workflow.

The primary artifacts of the Implementation workflow are:

- Implementation model, defining components and implementation subsystems
- Integration Build Plan

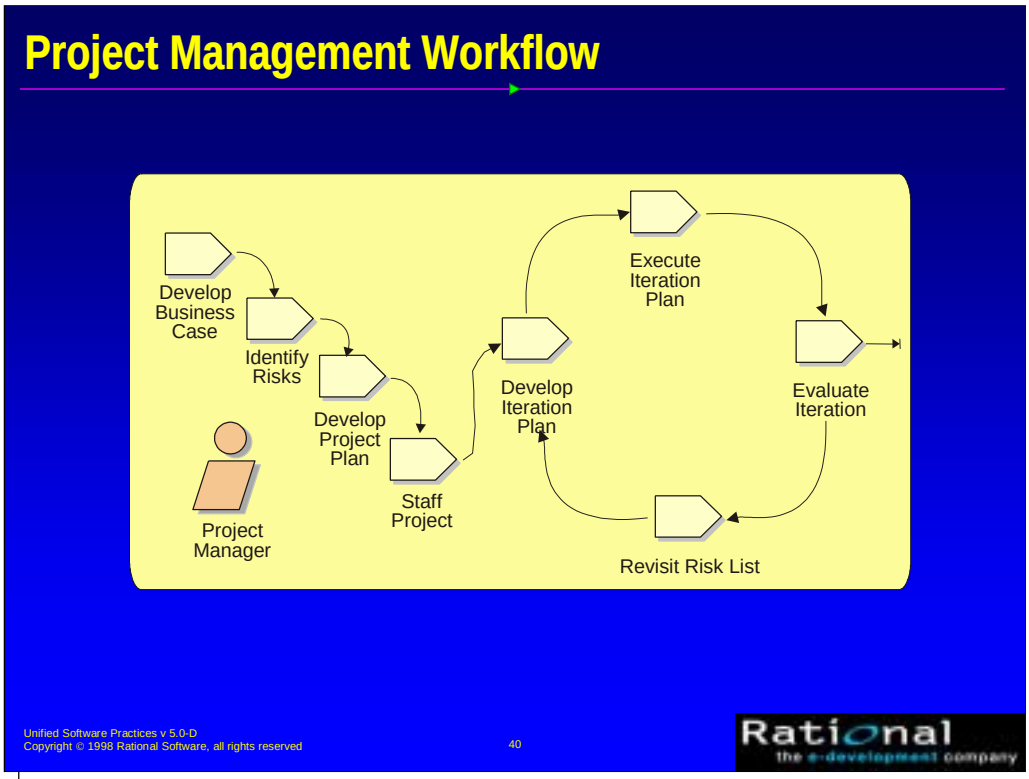


The purposes of the Testing workflow are:

- To verify the interaction between objects.
- To verify the proper integration of all components of the software.
- To verify that all requirements have been correctly implemented.
- To identify and ensure defects are addressed prior to the deployment of the software.

The primary artifacts of the Testing workflow are:

- Test model, defining test cases, procedures, and scripts
- Test Plans
- Defects
- Test packages, classes, subsystems, and components

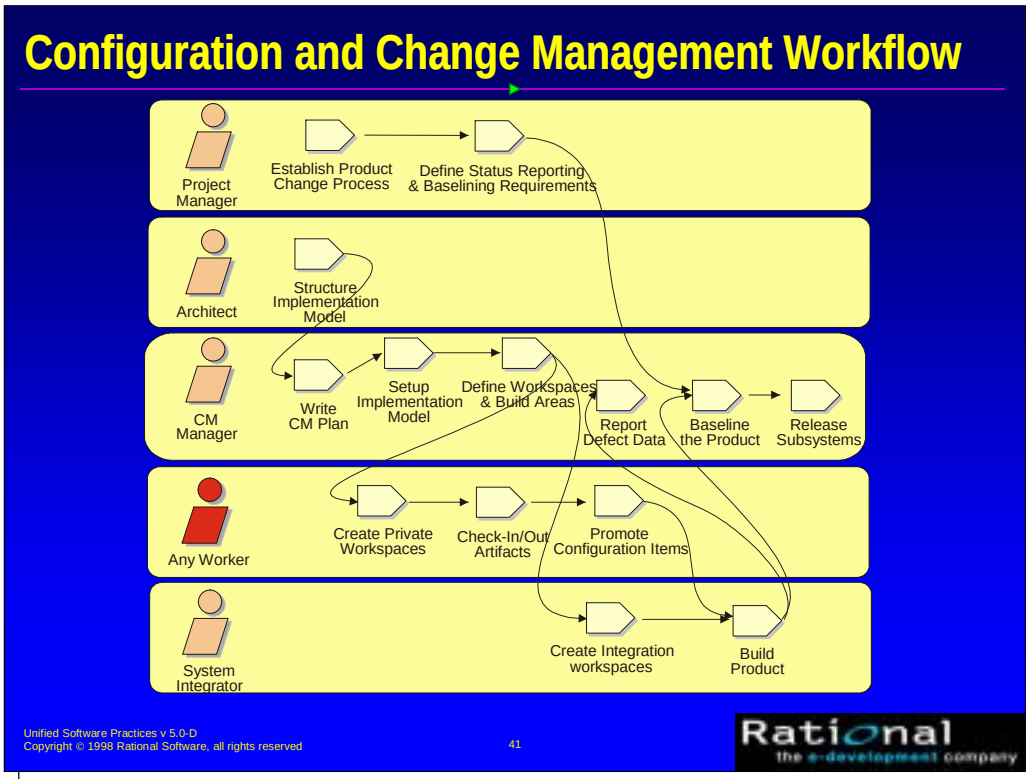


The purposes of the Project Management workflow are:

- To provide a framework for managing software-intensive projects.
- To provide practical guidelines for planning, staffing, executing, and monitoring projects.
- To provide a framework for managing risk.

The Project Manager is responsible for the following artifacts:

- The Software Development Plan, including the Risk List, Project Plan, and Measurement Plan.
- The Business Case.
- The Iteration Plan.
- The Iteration Assessment.
- Status Assessments.

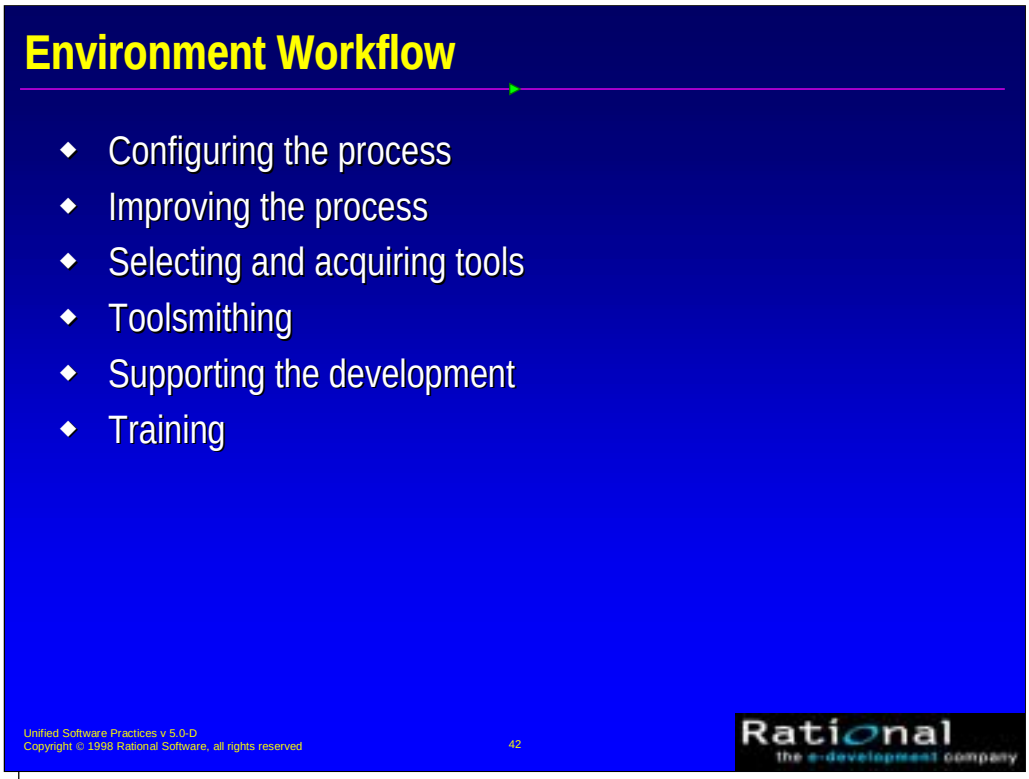


A CM System is useful for managing multiple variants of evolving software systems, tracking which versions are used in given software builds, performing builds of individual programs or entire releases according to user-defined version specifications, and enforcing site-specific development policies.

Some of the direct benefits provided by a CM System are that it:

- supports development methods,
- maintains product integrity,
- ensures completeness and correctness of the configured product,
- provides a stable environment within which to develop the product,
- restricts changes to artifacts based on project policies, and
- provides an audit trail on why, when and by whom any artifact was changed.

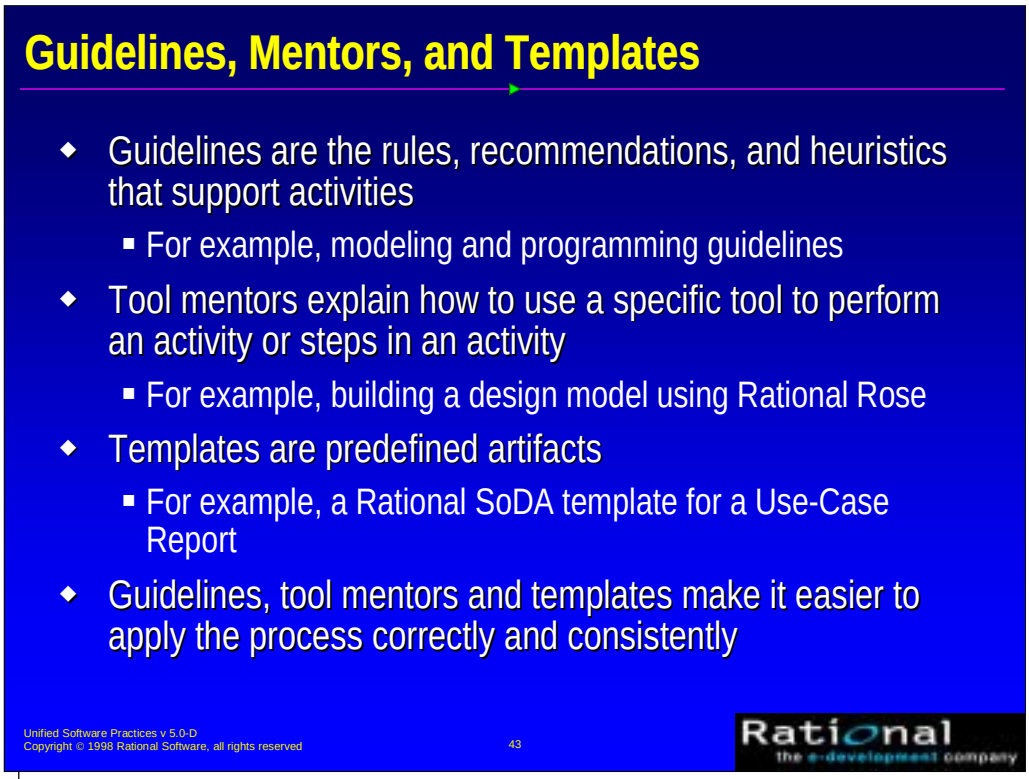
In addition, a CM System stores detailed ‘accounting’ data on the development process itself: who created a particular version (and when, and why), what versions of sources went into a particular build, and other relevant information.



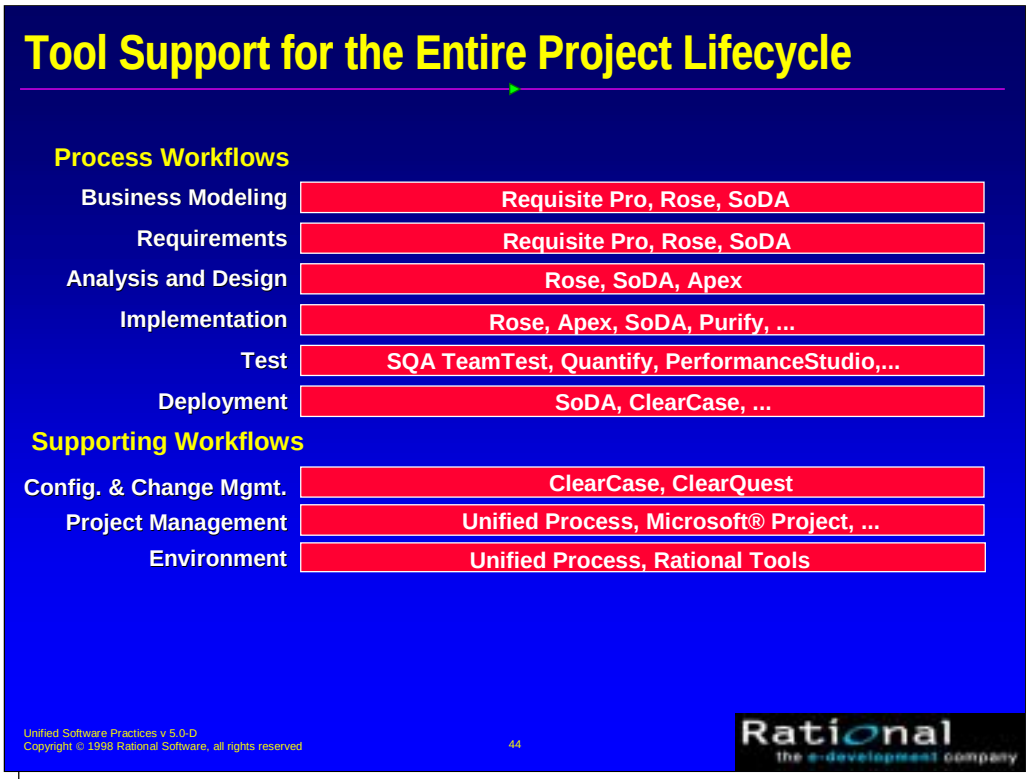
Configuring the process consists of adapting and tailoring the Rational Unified Process to suit the needs of an organization or project. Improving the process allows the process to evolve by capturing lessons learned as a project progresses or is completed.

A toolsmith develops tools to support special needs, provides additional automation of tedious or error-prone tasks, and provides better integration between tools.

Supporting the development includes maintaining the development environment, both hardware and software, system administration, telecommunications, and document creation and reproduction.



While we have focused mainly on the overall framework of the Rational Unified Process, i.e., the phases and workflows, the process provides much more technical and detailed guidance for the developer. Guidelines, tool mentors and templates are used by developers on a daily basis to accomplish their technical tasks.



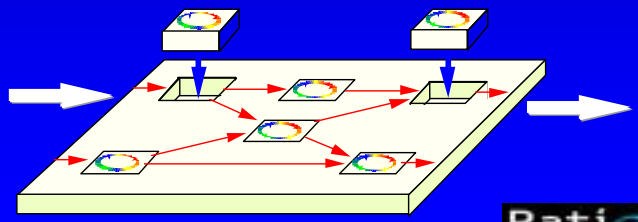
As might be expected, many tools are required to fully support a software development process, including tools for:

- Modeling
- Requirements management
- Code development (editors, compilers, and debuggers)
- Configuration management and change management
- Testing
- Planning and tracking
- Documentation

The graph above is not meant to be an exhaustive list of tools. In general, the mapping from process workflows to tools is one-to-many. For example, textual requirements are managed in RequisitePro, but Use-Cases are typically modeled in Rose. So requirements capture is supported by both RequisitePro and Rose. Further, SoDA can combine the requirements from both Requisite Pro and Rose into a well-formed and complete requirements specification. Hence, for each component, the mapping is one to many.

Adopting a Process

- Process adoption includes configuring and implementing the process
- In **configuring** the process, the process framework is adapted to the needs and constraints of the adopting organization
  - The result is documented in a “Development Case”
- In **implementing** the process, the organization’s practice is changed to effectively use the process



Rational the e-development company

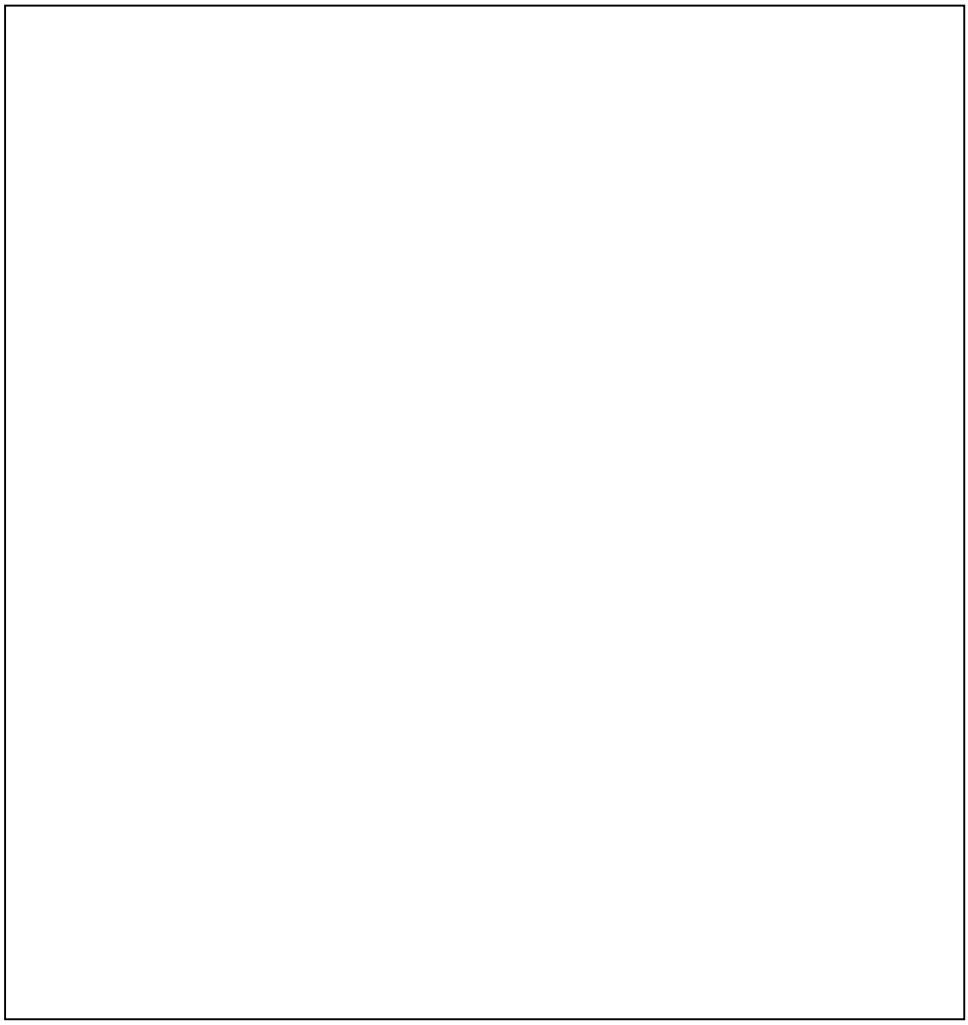
The term "development case" is used to refer to tailoring the Rational Unified Process to a specific organization and/or project. The factors most likely to affect the shape of the process include:

- The business context (contract work versus commercial development)
- The size of the effort (very large projects need more formality and more checks and balances)
- The degree of novelty (a green-field development versus a maintenance cycle)
- The type of application (safety-critical versus time to market driven)

The process itself contains detailed information on how to develop a development case, the decisions to be made, and a sample development case.

Summary: Rational Unified Process

- The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of a software-intensive system
- A software development process defines **Who** is doing **What, When** and **How** in building a software product
- The Rational Unified Process has four phases: **Inception, Elaboration, Construction and Transition**
- Each phase ends at a major milestone and contains one or more iterations
- An **iteration** is a distinct sequence of activities with an established plan and evaluation criteria, resulting in an executable release



Summary (cont.): Rational Unified Process

- A **workflow** groups related activities together
- Each workflow is exercised during an **iteration** and results in a model that is incrementally produced
- An **artifact** is a piece of information that is produced, modified, or used by a process
- A **worker** is a role that may be played by an individual or a team in the development organization
- An **activity** is a unit of work a worker may be asked to perform