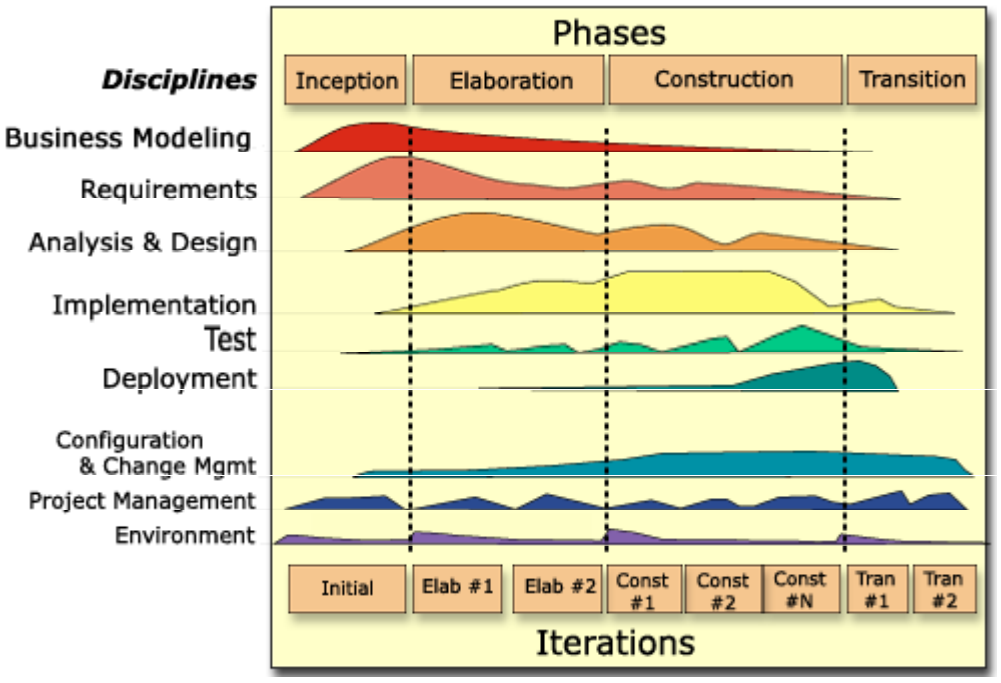
Rational Unified Process: Overview

The Rational Unified Process® or RUP® product is a software engineering process. 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 has two dimensions:

- the horizontal axis represents time and shows the lifecycle aspects of the process as it unfolds
- the vertical axis represents disciplines, which group activities logically by nature.

The first dimension represents the dynamic aspect of the process as it is enacted, and it is expressed in terms of phases, iterations, and milestones.

The second dimension represents the static aspect of the process: how it is described in terms of process components, disciplines, activities, workflows, artifacts, and roles.

The graph shows how the emphasis varies over time. For example, in early iterations, we spend more time on requirements, and in later iterations we spend more time on implementation.

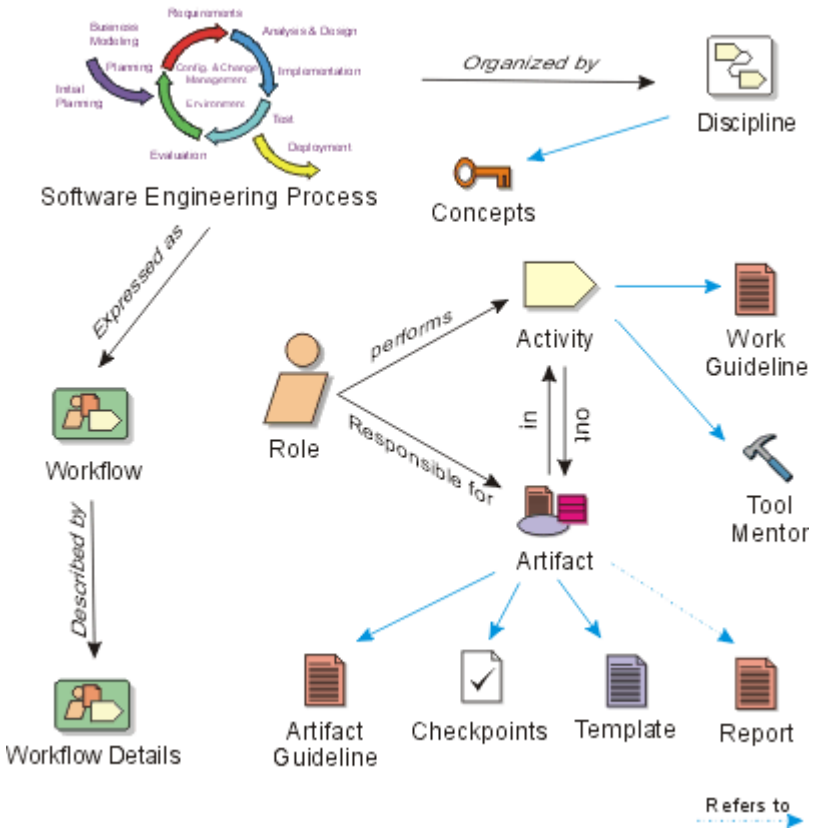
The RUP is now taken as the standard Unified Process (UP) by most of the software industries.

1. Key Concepts

1.1. Discipline

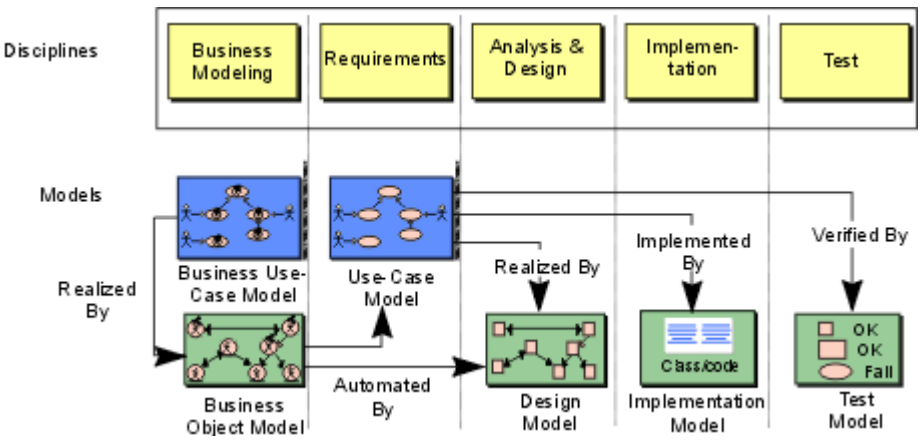
A discipline is a collection of related activities that are related to a major 'area of concern' within the overall project. The grouping activities into disciplines is mainly an aid to understanding the project from a 'traditional' waterfall perspective - typically, for example, it is more common to perform certain requirements activities in close coordination with analysis and design activities. Separating these activities into separate disciplines makes the activities easier to comprehend but more difficult to schedule.

Like other workflows, a discipline's workflow is a semi-ordered sequence of activities which are performed to achieve a particular result. The "semi-ordered" nature of discipline workflows emphasizes that the discipline workflows cannot present the real nuances of scheduling "real work", for they cannot depict the optionality of activities or iterative nature of real projects. Yet they still have value as a way for us to understand the process by breaking it into smaller 'areas of concern'.



Each 'area of concern' or discipline has associated with it one or more 'models', which are in turn composed of associated artifacts. The most important artifacts are the models that each discipline yields: use-case model, design model, implementation model and test suite.

For each discipline, an **activity overview** is also presented. The activity overview shows all activities in the discipline along with the role that performs the activity. An **artifact overview** diagram is also presented. This diagram shows all artifacts and roles involved in the discipline.



Each discipline is associated with a particular set of models.

It is useful to note that the 'discipline-centric' organization of artifacts is sometimes, though not always, slightly different from the **artifact set** organization of artifacts. The reason for this is simple: some artifacts are used across disciplines; a strict discipline-centric grouping makes it more difficult to present an integrated process. If you are using only a part of the process, however, the discipline-centric artifact overviews may prove more useful.

1.2. Workflow

A mere enumeration of all roles, activities and artifacts does not constitute a process; we need a way to describe meaningful sequences of activities that produce some valuable result, and to show interactions between roles. A *workflow* is a sequence of activities that produces a result of observable value.

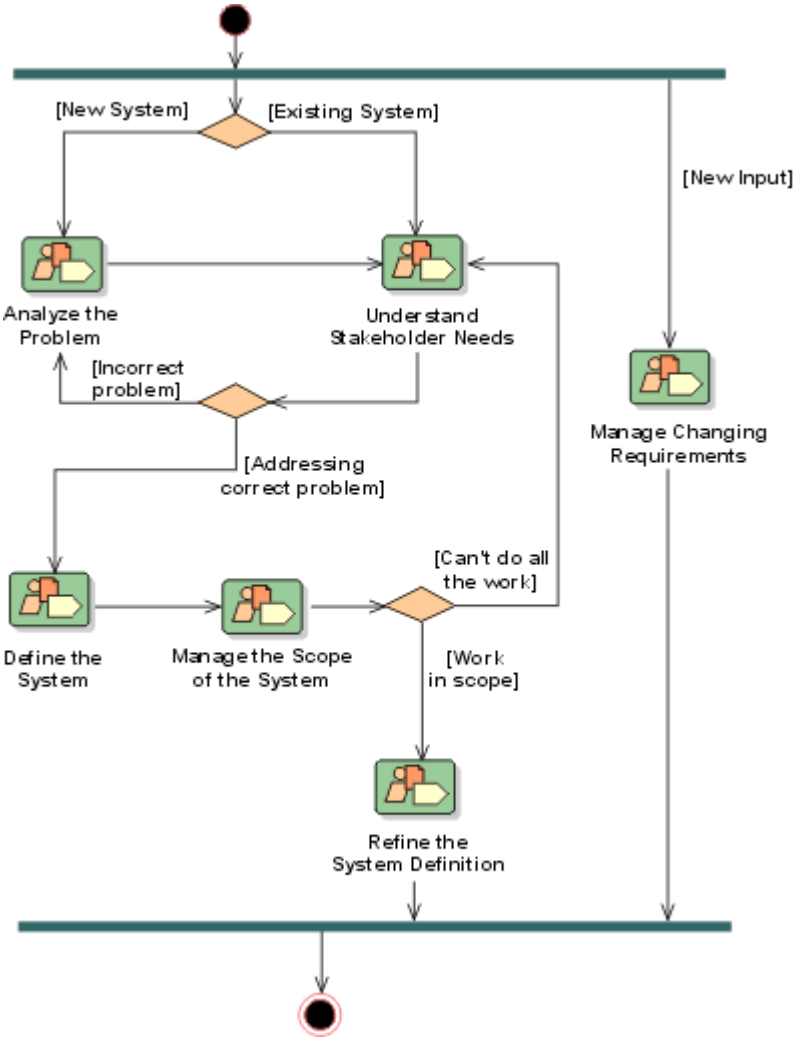
In UML terms, a workflow can be expressed as a sequence diagram, a collaboration diagram, or an activity diagram. We use a form of activity diagrams in the RUP. For each discipline, an **activity** diagram is presented. This diagram shows the workflow, expressed in terms of workflow details.

One of the great difficulties of describing the process is that there are many ways to organize the set of activities into workflows. RUP is organized using:

- Disciplines
- Workflow details

1.3. Workflow Detail

For most of the disciplines, you will also find **workflow detail** diagrams, which show groupings of activities that often are performed "together". These diagrams show roles involved, input and output artifacts, and activities performed. The workflow detail diagrams are there for the following reasons:

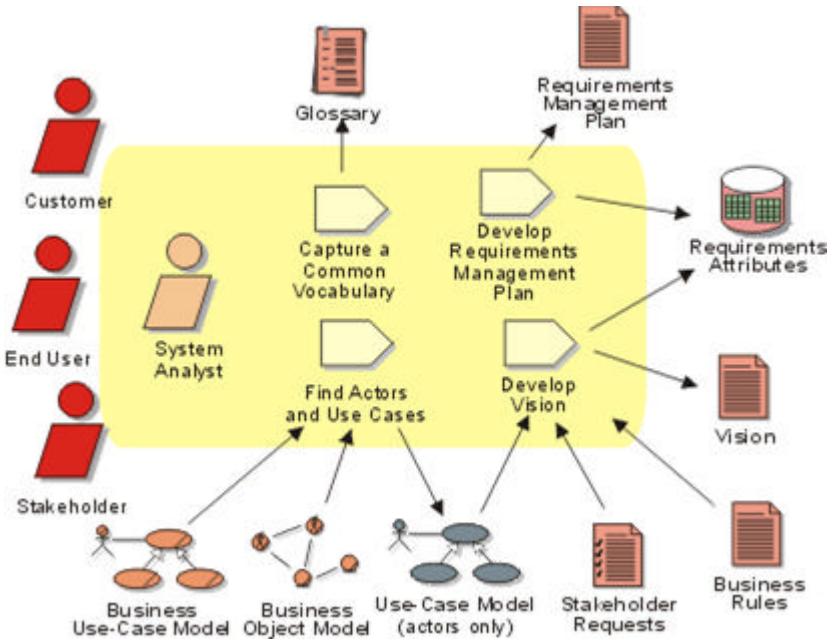


Sample activity diagram, from the requirements discipline, showing workflow details and transitions.

- The activities of a workflow are neither performed in sequence, nor done all at once. The workflow detail diagram shows how you often will work in workshops or team meetings while performing a workflow. You typically work in parallel on more than one activity, and look at more than one artifact while doing that. There are several workflow detail diagrams for a discipline.
- It becomes too complex to show input and output artifacts for all activities of a discipline in one diagram. The workflow detail diagram allows us to show you activities and artifacts together, for one part of a workflow at a time.
- The disciplines are not completely independent of one another. For example, integration occurs in both the implementation and test disciplines, and in reality you never really do one without the other. The workflow detail diagram can show a group of activities and artifacts in the discipline, together with closely related activities in another discipline.

1.4. Role

The most central concept in the Process is that of a role. A **role** defines the behavior and responsibilities of an individual, or a set of individuals working together as a team, within the context of a software engineering organization. The Roles Overview provides additional information on roles.



Sample workflow detail diagram, from the requirements discipline.

Note that **roles are not individuals**; instead, they describe how individuals should behave in the business and the responsibilities of an individual. Individual members of the software development organization will wear different hats, or perform different roles. The mapping from individual to role, performed by the project manager when planning and staffing the project, allows different individuals to act as several different roles, and for a role to be played by several individuals.

1.5. Activity

Roles have activities that define the work they perform. An **activity** is something that a role does that provides a meaningful result in the context of the project.

An activity is a unit of work that an individual playing the described role may be asked to perform. The activity has a **clear purpose**, usually expressed in terms of creating or updating some artifacts, such as a model, a class, a plan. Every activity is assigned to a specific role. The granularity of an activity is generally a few hours to a few days, it usually involves one role, and affects one or only a small number of artifacts. An activity should be usable as an element of planning and progress; if it is too small, it will be neglected, and if it is too large, progress would have to be expressed in terms of an activity's parts.

Activities may be repeated several times on the same artifact, especially when going from one iteration to another, refining and expanding the system, by the same role, but not necessarily the same individual.

Steps

Activities are broken down into steps. Steps fall into three main categories:

- **Thinking** steps: where the individual performing the role understands the nature of the task, gathers and examines the input artifacts, and formulates the outcome.
- **Performing** steps: where the individual performing the role creates or updates some artifacts.

- **Reviewing** steps: where the individual performing the role inspects the results against some criteria.

Not all steps are necessarily performed each time an activity is invoked, so they can be expressed in the form of alternate flows.

Example of steps:

The *Activity: Find use cases and actors* decomposes into the steps:

1. Find actors
2. Find use cases
3. Describe how actors and use cases interact
4. Package use-cases and actors
5. Present the use-case model in use-case diagrams
6. Develop a survey of the use-case model
7. Evaluate your results

The finding part [steps 1 to 3] requires some thinking; the performing part [steps 4 to 6] involves capturing the result in the use-case model; the reviewing part [step 7] is where the individual performing the role evaluates the result to assess completeness, robustness, intelligibility, or other qualities.

Work Guideline

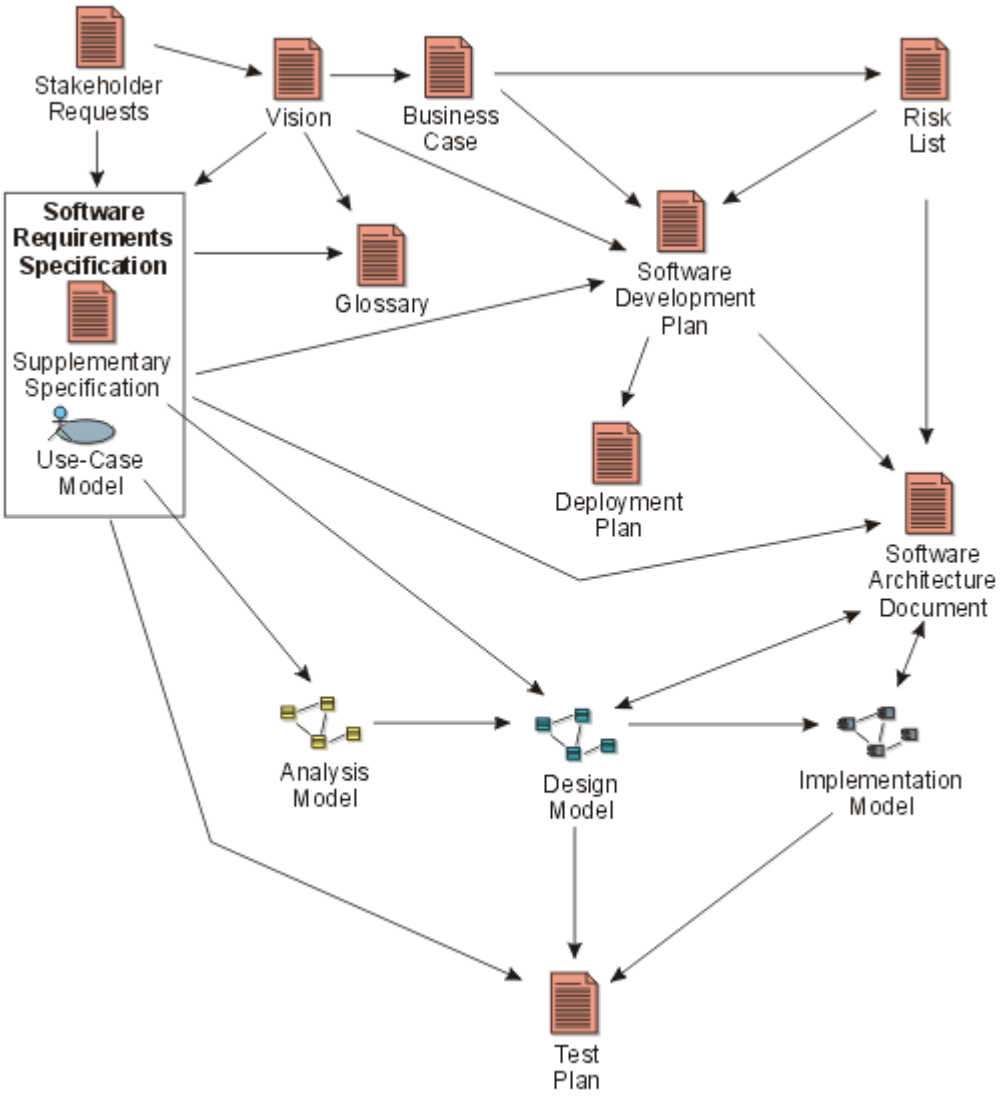
Activities may have associated Work Guidelines, which present techniques and practical advice that is useful to the role performing the activity.

1.6. Artifact

Activities have input and output artifacts. An **artifact** is a work product of the process: roles use artifacts to perform activities, and produce artifacts in the course of performing activities. Artifacts are the responsibility of a single role and promote the idea that every piece of information in the process must be the responsibility of a specific person. Even though one person may "own" the artifact, many other people may use the artifact, perhaps even updating it if they have been given permission.

The diagram below shows how information flows through the project, via the artifacts; the arrows show how changes in one artifact ripple through other artifacts along the arrows. For clarity, many artifacts are omitted (e.g. the many artifacts in the design model are omitted, being represented by the Artifact: Design Model).

To simplify the organization of artifacts, they are organized into "information sets", or **artifact sets**. An artifact set is a grouping of related artifacts that tend to be used for a similar purpose. The Artifact Overview presents more information on artifacts and artifact sets.



Major artifacts in the process, and the approximate flow of information between them.

Artifacts may take various shapes or forms:

- A **model**, such as the Use-Case Model or the Design Model, which contains other artifacts.
- A **model element**, i.e. an element within a model, such as a Design Class, a Use Case or a Design Subsystem
- A **document**, such as Business Case or Software Architecture Document
- Source code and executables (kinds of Components)
- Executables

Note that "**artifact**" is the term used in the RUP. Other processes use terms such as **work product**, **work unit**, and so on, to denote the same thing. Deliverables are only the subset of all artifacts that end up in the hands of the customers and end-users.

Artifacts are most likely to be subject to version control and configuration management. This is sometimes only achieved by versioning the container artifact, when it is not possible to do it for the elementary, contained artifacts. For example, you may control the versions of a whole design model, or design package, and not of individual classes they contain.

Artifacts are typically *not* documents. Many processes have an excessive focus on documents, and in particular on *paper documents.* The RUP discourages the systematic production of paper documents. The most efficient and pragmatic approach to managing project artifacts is to maintain the artifacts *within* the appropriate tool used to create and manage them. When necessary, you may generate documents (snapshots) from these tools, on a just-in-time basis. You should also consider delivering artifacts to the interested parties inside and together with the tool, rather than on paper. This approach ensures that the information is always up-to-date and based on actual project work, and it shouldn't require any additional effort to produce.

Examples of artifacts:

- A design model stored in Rational Rose.
- A project plan stored in Microsoft Project.
- A defect stored in Rational ClearQuest.
- A project requirements database in Rational RequisitePro.

However, there are still artifacts which have to be plain text documents, as in the case of external input to the project, or in some cases where it is simply the best means of presenting descriptive information.

Artifact Guidelines and Checkpoints

Artifacts typically have associated guidelines and checkpoints which present information on how to develop, evaluate and use the artifacts. Much of the substance of the Process is contained in the artifact guidelines; the activity descriptions try to capture the essence of what is done, while the artifact guidelines capture the essence of doing the work. The checkpoints provide a quick reference to help you assess the quality of the artifact.

Both guidelines and checkpoints are useful in a number of contexts: they help you decide what to do, they help you to do it, and they help you to decide if you've done a good job when you're done.

Template

Templates are "models," or prototypes, of artifacts. Associated with the artifact description are one or more templates that can be used to create the corresponding artifacts. Templates are linked to the tool that is to be used.

For example:

- Microsoft Word templates would be used for artifacts that are documents, and for some reports.
- Rational SoDA templates for Microsoft Word or FrameMaker would extract information from tools such as Rational Rose, Rational RequisitePro, or Rational TeamTest.
- Microsoft FrontPage templates for the various elements of the process.
- Microsoft Project template for the project plan.

As with guidelines, organizations may want to customize the templates prior to using them by adding the company logo, some project identification, or information specific to the type of project.

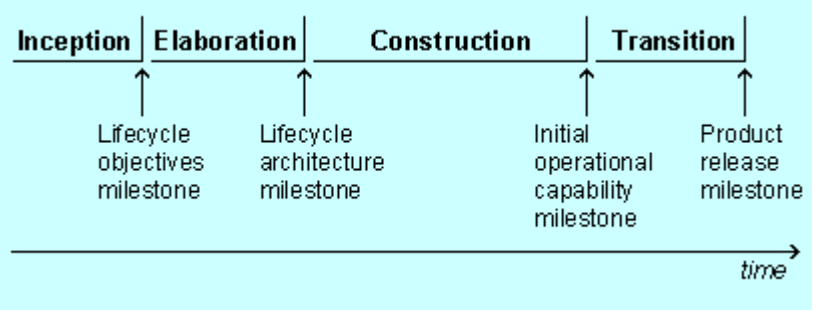
Report

Models and model elements, may have reports associated with them. A report extracts information about models and model elements from a tool. For example, a report presents an artifact or a set of artifacts for a review. Unlike regular artifacts, reports are not subject to version control. They can be reproduced at any time by going back to the artifacts that generated them.

1.7. Tool Mentor

Activities, steps, and associated guidelines provide general guidance to the practitioner. To go one step further, *tool mentors* are an additional means of providing guidance by showing how to perform the steps using a specific software tool. Tool mentors are provided in the RUP, linking its activities with tools such as Rational Rose, Rational RequisitePro, Rational ClearCase, Rational ClearQuest, Rational Suite TestStudio. The tool mentors almost completely encapsulate the dependencies of the process on the tool set, keeping the activities free from tool details. An organization can extend the concept of tool mentor to provide guidance for other tools.

2. Rational Unified Process: Phases



The phases and milestones of a project

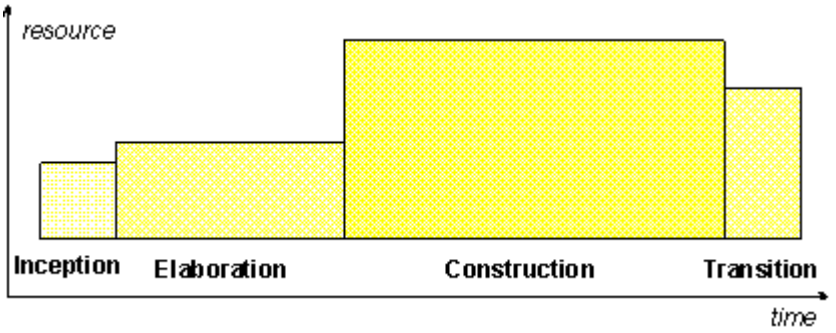
From a management perspective, the software lifecycle of the Rational Unified Process (RUP) is decomposed over time into four sequential phases, each concluded by a major milestone; each phase is essentially a span of time between two major milestones. At each phase-end an assessment is performed to determine whether the objectives of the phase have been met. A satisfactory assessment allows the project to move to the next phase.

Planning Phases

All phases are not identical in terms of schedule and effort. Although this varies considerably depending on the project, a typical initial development cycle for a medium-sized project should anticipate the following distribution between effort and schedule:

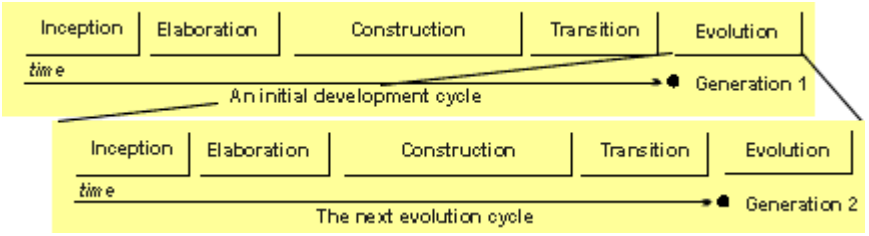
	Inception	Elaboration	Construction	Transition
Effort	~5 %	20 %	65 %	10%
Schedule	10 %	30 %	50 %	10%

which can be depicted graphically as



For an evolution cycle, the inception and elaboration phases would be considerably smaller. Tools which can automate some portion of the Construction effort can mitigate this, making the construction phase much smaller than the inception and elaboration phases together.

One pass through the four phases is a **development cycle**; each pass through the four phases produces a **generation** of the software. Unless the product "dies," it will evolve into its next generation by repeating the same sequence of inception, elaboration, construction and transition phases, but this time with a different emphasis on the various phases. These subsequent cycles are called **evolution cycles.** As the product goes through several cycles, new generations are produced.



Evolution cycles may be triggered by user-suggested enhancements, changes in the user context, changes in the underlying technology, reaction to the competition, and so on. Evolution cycles typically have much shorter Inception and Elaboration phases, since the basic product definition and architecture are determined by prior development cycles. Exceptions to this rule are evolution cycles in which a significant product or architectural redefinition occurs.

2.1. Phase: Inception

Objectives

The overriding goal of the inception phase is to achieve concurrence among all stakeholders on the lifecycle objectives for the project. The inception phase is of significance primarily for new development efforts, in which there are significant business and requirements risks which must be addressed before the project can proceed. For projects focused on enhancements to an existing system, the inception phase is more brief, but is still focused on ensuring that the project is both worth doing and possible to do.

The primary objectives of the inception phase include:

- Establishing the project's software scope and boundary conditions, including an operational vision, acceptance criteria and what is intended to be in the product and what is not.
- Discriminating the critical use cases of the system, the primary scenarios of operation that will drive the major design trade-offs.

- Exhibiting, and maybe demonstrating, at least one candidate architecture against some of the primary scenarios
- Estimating the overall cost and schedule for the entire project (and more detailed estimates for the elaboration phase that will immediately follow)
- Estimating potential risks (the sources of unpredictability)
- Preparing the supporting environment for the project.

Essential activities

- **Formulating the scope of the project**. This involves capturing the context and the most important requirements and constraints to such an extent that you can derive acceptance criteria for the end product.
- **Planning and preparing a business case**. Evaluating alternatives for risk management, staffing, project plan, and cost/schedule/profitability trade-offs.
- **Synthesizing a candidate architecture**, evaluating trade-offs in design, and in make/buy/reuse, so that cost, schedule and resources can be estimated. The aim here is to demonstrate feasibility through some kind of proof of concept. This may take the form of a model which simulates what is required, or an initial prototype which explores what are considered to be the areas of high risk. The prototyping effort during inception should be limited to gaining confidence that a solution is possible - the solution is realized during elaboration and construction.
- **Preparing the environment for the project**, assessing the project and the organization, selecting tools, deciding which parts of the process to improve.

2.2. Phase: Elaboration

Objectives

The goal of the elaboration phase is to baseline the architecture of the system to provide a stable basis for the bulk of the design and implementation effort in the construction phase. The architecture evolves out of a consideration of the most significant requirements (those that have a great impact on the architecture of the system) and an assessment of risk. The stability of the architecture is evaluated through one or more architectural prototypes.

The primary objectives of the elaboration phase include:

- To ensure that the architecture, requirements and plans are stable enough, and the risks sufficiently mitigated to be able to predictably determine the cost and schedule for the completion of the development. For most projects, passing this milestone also corresponds to the transition from a light-and-fast, low-risk operation to a high cost, high risk operation with substantial organizational inertia.
- To address all architecturally significant risks of the project
- To establish a baselined architecture derived from addressing the architecturally significant scenarios, which typically expose the top technical risks of the project.
- To produce an evolutionary prototype of production-quality components, as well as possibly one or more exploratory, throw-away prototypes to mitigate specific risks such as:
  - design/requirements trade-offs
  - component reuse
  - product feasibility or demonstrations to investors, customers, and end-users.
- To demonstrate that the baselined architecture will support the requirements of the system at a reasonable cost and in a reasonable time.

- To establish a supporting environment.

In order to achieve this primary objectives, it is equally important to set up the supporting environment for the project. This includes creating a development case, create templates, guidelines, and setting up tools.

Essential activities

- **Defining, validating and baselining the architecture** as rapidly as practical.
- **Refining the Vision**, based on new information obtained during the phase, establishing a solid understanding of the most critical use cases that drive the architectural and planning decisions.
- **Creating and baselining detailed iteration plans for the construction phase**.
- **Refining the development case and putting in place the development environment**, including the process, tools and automation support required to support the construction team.
- **Refining the architecture and selecting components**. Potential components are evaluated and the make/buy/reuse decisions sufficiently understood to determine the construction phase cost and schedule with confidence. The selected architectural components are integrated and assessed against the primary scenarios. Lessons learned from these activities may well result in a redesign of the architecture, taking into consideration alternative designs or reconsideration of the requirements.

2.3. Phase: Construction

Objectives

The goal of the construction phase is on clarifying the remaining requirements and completing the development of the system based upon the baselined architecture. The construction phase is in some sense a manufacturing process, where emphasis is placed on managing resources and controlling operations to optimize costs, schedules, and quality. In this sense the management mindset undergoes a transition from the development of intellectual property during inception and elaboration, to the development of deployable products during construction and transition.

The primary objectives of the construction phase include:

- Minimizing development costs by optimizing resources and avoiding unnecessary scrap and rework.
- Achieving adequate quality as rapidly as practical
- Achieving useful versions (alpha, beta, and other test releases) as rapidly as practical
- Completing the analysis, design, development and testing of all required functionality.
- To iteratively and incrementally develop a complete product that is ready to transition to its user community. This implies describing the remaining use cases and other requirements, fleshing out the design, completing the implementation, and testing the software.
- To decide if the software, the sites, and the users are all ready for the application to be deployed.
- To achieve some degree of parallelism in the work of development teams. Even on smaller projects, there are typically components that can be developed independently of one another, allowing for natural parallelism between teams (resources permitting). This parallelism can accelerate the development activities significantly; but it also increases the complexity of resource management and workflow synchronization. A robust architecture is essential if any significant parallelism is to be achieved.

Essential Activities

- Resource management, control and process optimization
- Complete component development and testing against the defined evaluation criteria
- Assessment of product releases against acceptance criteria for the vision.

2.4. Phase: Transition

Objectives

The focus of the Transition Phase is to ensure that software is available for its end users. The Transition Phase can span several iterations, and includes testing the product in preparation for release, and making minor adjustments based on user feedback. At this point in the lifecycle, user feedback should focus mainly on fine tuning the product, configuring, installing and usability issues, all the major structural issues should have been worked out much earlier in the project lifecycle.

By the end of the Transition Phase lifecycle objectives should have been met and the project should be in a position to be closed out. In some cases, the end of the current life cycle may coincide with the start of another lifecycle on the same product, leading to the next generation or version of the product. For other projects, the end of Transition may coincide with a complete delivery of the artifacts to a third party who may be responsible for operations, maintenance and enhancements of the delivered system.

This Transition Phase ranges from being very straightforward to extremely complex, depending on the kind of product. A new release of an existing desktop product may be very simple, whereas the replacement of a nation's air-traffic control system may be exceedingly complex.

Activities performed during an iteration in the Transition Phase depend on the goal. For example, when fixing bugs, implementation and test are usually enough. If, however, new features have to be added, the iteration is similar to one in the construction phase requiring analysis & design, etc.

The Transition Phase is entered when a baseline is mature enough to be deployed in the end-user domain. This typically requires that some usable subset of the system has been completed with acceptable quality level and user documentation so that transitioning to the user provides positive results for all parties.

The primary objectives of the Transition Phase are:

- beta testing to validate the new system against user expectations
- beta testing and parallel operation relative to a legacy system that it's replacing
- converting operational databases
- training of users and maintainers
- roll-out to the marketing, distribution and sales forces
- deployment-specific engineering such as cutover, commercial packaging and production, sales roll-out, field personnel training
- tuning activities such as bug fixing, enhancement for performance and usability
- assessment of the deployment baselines against the complete vision and the acceptance criteria for the product
- achieving user self-supportability
- achieving stakeholder concurrence that deployment baselines are complete

- achieving stakeholder concurrence that deployment baselines are consistent with the evaluation criteria of the vision

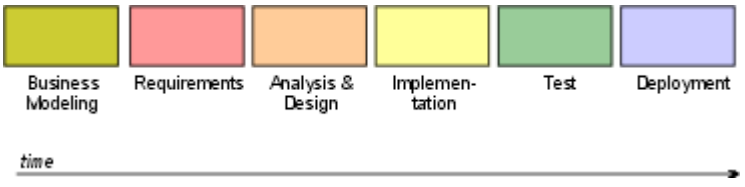
Essential activities

- executing deployment plans
- finalizing end-user support material
- testing the deliverable product at the development site
- creating a product release
- getting user feedback
- fine-tuning the product based on feedback
- making the product available to end users

3. Rational Unified Process: Iteration

3.1. Why Iterate?

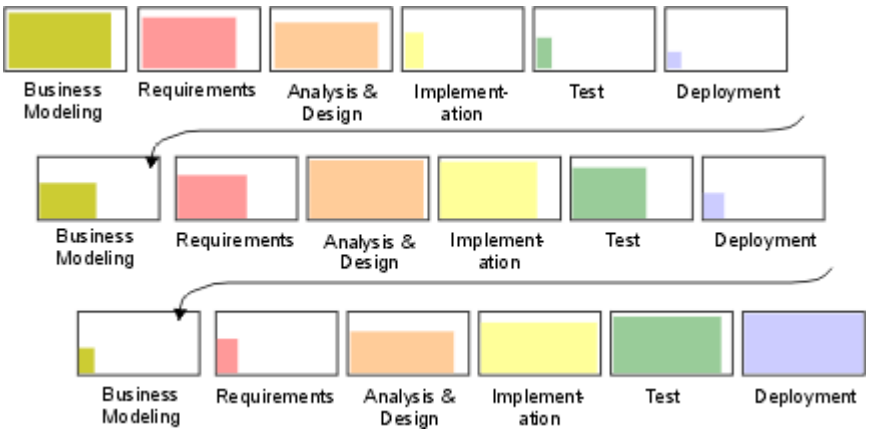
Traditionally, projects have been organized to go through each discipline in sequence, once and only once. This leads to the **waterfall** lifecycle:



This often results in an integration 'pile-up' late in implementation when, for the first time, the product is built and testing begins. Problems which have remained hidden throughout Analysis, Design and Implementation come boiling to the surface, and the project grinds to a halt as a lengthy bug-fix cycle begins.

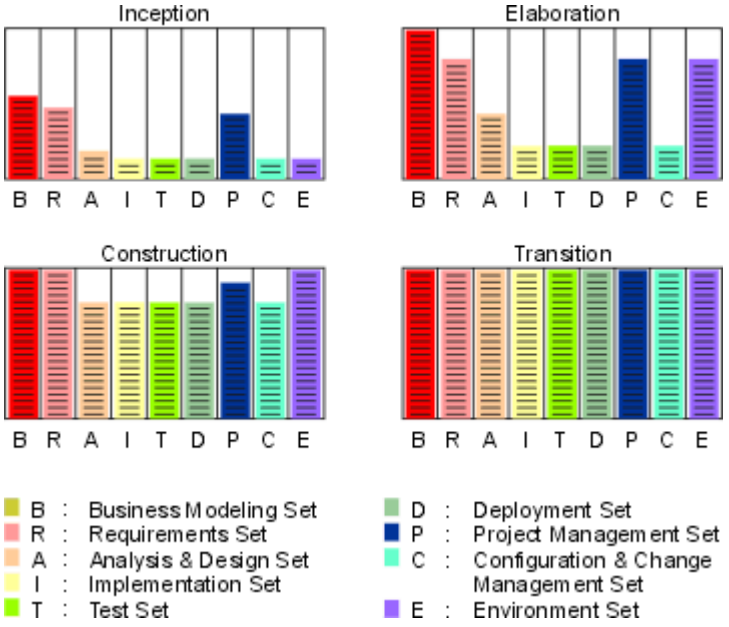
A more flexible (and less risky) way to proceed is to go several times through the various development disciplines, building a better understanding of the requirements, engineering a robust architecture, ramping up the development organization, and eventually delivering a series of implementations that are gradually more complete. This is called an **iterative** lifecycle. Each pass through the sequence of process disciplines is called an **iteration**.

Therefore, from a development perspective the software lifecycle is a succession of **iterations**, through which the software develops incrementally. Each iteration concludes with the **release** of an executable product. This product may be a subset of the complete vision, but useful from some engineering or user perspective. Each release is accompanied by supporting artifacts: release description, user documentation, plans, and so on, and updated models of the system.



The main consequence of this iterative approach is that the sets of artifacts, described earlier, grow and mature over time, as shown in the following diagram.

3.2. What is an Iteration?



Information set evolution over the development phases.

An iteration encompasses the development activities that lead to a product **release**—a stable, executable versions of product, together with any other peripheral elements necessary to use this release. So a development iteration is in some sense one complete pass through all the disciplines: Requirements, Analysis & Design, Implementation, and Test, at least. It is like a small waterfall project in itself. Note that evaluation criteria are established when each iteration is planned. The release will have planned capability which is demonstrable. The duration of an iteration will vary depending on the size and nature of the project, but it is likely that **multiple** builds will be constructed in **each** iteration, as specified in the Integration Build Plan for the iteration. This is a consequence of the continuous integration approach recommended in the Rational Unified Process (RUP): as unit-tested components become available, they are integrated, then a build is produced and subjected to integration testing. In this way, the capability of the integrated software grows as the iteration proceeds, towards the goals set when the iteration was planned. It could be argued that each build itself represents a mini-iteration, the difference is in the planning required and the formality of the assessment performed. It may be appropriate and convenient in some projects to construct builds on a daily basis, but these would not represent iterations as the RUP defines them—except perhaps for a very small, single person project. Even for small multi-person projects (for example, involving five people building 10,000 lines of code), it would be very difficult to achieve an iteration duration of less than a week.

3.3. Release

A release can be internal or external. An internal release is used only by the development organization, as part of a milestone, or for a demonstration to users or customers. An external release (or delivery) is delivered to end users. A release is not necessarily a complete product, but can just be one step along the way, with its usefulness measured only from an engineering perspective. Releases act as a forcing function that drives the development team to get closure at regular intervals, avoiding the "90% done, 90% remaining" syndrome.

Iterations and releases allow a better usage over time of the various specialties in the team: designers, testers, writers, and so forth. Regular releases let you break down the integration and test issues and spread them across the development cycle. These issues have often been the downfall of large projects because all problems were discovered at once during the single massive integration step, which occurred very late in the cycle, and where a single problem halts the whole team.

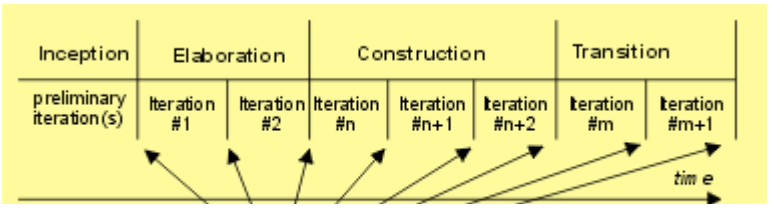
At each iteration, artifacts are updated. It is said that this is a bit like "growing" software. Instead of developing artifacts one after another, in a pipeline fashion, they are evolving across the cycle, although at different rates.

3.4. Minor milestone

Each iteration is concluded by a minor milestone, where the result of the iteration is assessed relative to the objective success criteria of that particular iteration.

3.5. Iteration and Phases

Each phase in the RUP can be further broken down into iterations. An iteration is a complete development loop resulting in a release (internal or external) of an executable product, a subset of the final product under development, which grows incrementally from iteration to iteration to become the final system.



Each iteration within a phase results in an executable release of the system.

3.5.1. Iteration pattern: Incremental Lifecycle

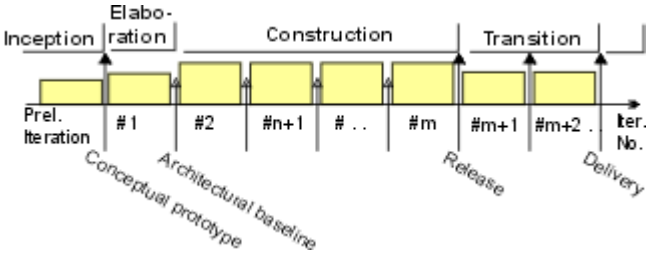
"The incremental strategy determines user needs, and defines the system requirements, and then performs the rest of the development in a sequence of builds. The first build incorporates parts of the planned capabilities, the next build adds more capabilities, and so on until the system is complete."

The following iterations are characteristic:

- a short Inception iteration to establish scope and vision, and to define the business case
- a single Elaboration iteration, during which requirements are defined, and the architecture established
- several Construction iterations during which the use cases are realized and the architecture fleshed-out
- several Transition iterations to migrate the product into the user community

This strategy is appropriate when:

- The problem domain is familiar.
- Risks are well-understood.
- The project team is experienced.



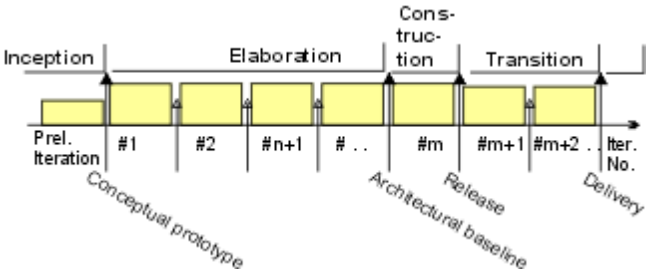
3.5.2. Iteration pattern: Evolutionary Lifecycle

"The evolutionary strategy differs from the incremental in acknowledging that user needs are not fully understood, and all requirements cannot be defined up front, they are refined in each successive build."

The following iterations are characteristic:

- a short Inception iteration to establish scope and vision, and to define the business case
- several Elaboration iterations, during which requirements are refined at each iteration

- a single Construction iteration, during which the use cases are realized and the architecture is expanded upon
- several Transition iterations to migrate the product into the user community



This strategy is appropriate when:

- The problem domain is new or unfamiliar.
- The team is inexperienced.

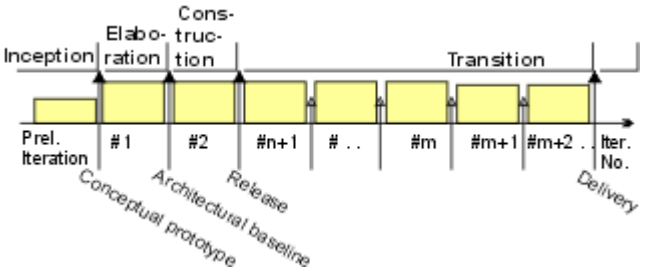
3.5.3. Iteration pattern: Incremental Delivery Lifecycle

Some authors have also phased deliveries of incremental functionality to the customer. This may be required where there are tight time-to-market pressures, where delivery of certain key features early can yield significant business benefits.

In terms of the phase-iteration approach, the transition phase begins early on and has the most iterations. This strategy requires a very stable architecture, which is hard to achieve in an initial development cycle, for an "unprecedented" system.

The following iterations are characteristic:

- a short Inception iteration to establish scope and vision, and to define the business case
- a single Elaboration iteration, during which a stable architecture is baselined
- a single Construction iteration, during which the use cases are realized and the architecture fleshed-out
- several Transition iterations to migrate the product into the user community



This strategy is appropriate when:

- The problem domain is familiar:
  - the architecture and requirements can be stabilized early in the development cycle
  - there is a low degree of novelty in the problem
- The team is experienced.
- Incremental releases of functionality have high value to the customer.

3.5.4. Iteration pattern: "Grand Design" Lifecycle

The traditional waterfall approach can be seen as a degenerated case in which there is only one iteration in the construction phase. It is called "grand design" in. In practice, it is hard to avoid additional iterations in the transition phase.

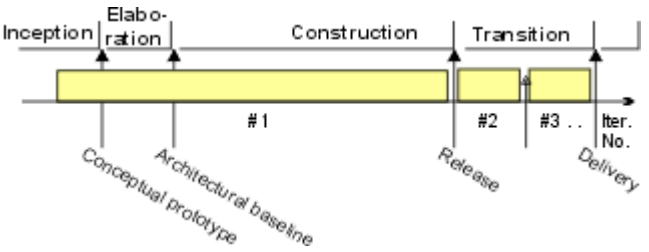
The following iterations are characteristic:

- a short Inception iteration to establish scope and vision, and to define the business case

- a single very long Construction iteration, during which the use cases are realized and the architecture fleshed-out
- several Transition iterations to migrate the product into the user community

This strategy is appropriate when:

- a small increment of well-defined functionality is being added to a very stable product
- the new functionality is well-defined and well-understood
- The team is experienced, both in the problem domain and with the existing product



3.5.5. Iteration pattern: Hybrid Strategies

In practice few projects strictly follow one strategy. You often end up with a **hybrid,** some evolution at the beginning, some incremental building, and multiple deliveries. Among the advantages of the phase-iteration model is that it lets you accommodate a hybrid approach, simply by increasing the length and number of iterations in particular phases:

- For complex or unfamiliar problem domains, where there is a high degree of exploration: increase the number of iterations in the elaboration phase and its length.
- For more complex development problems, where there is complexity translating the design into code: increase the number of iterations in the construction phase and its length.
- To deliver software in a series of incremental releases: increase the number of iterations in the transition phase and its length.

4. Rational Unified Process: Disciplines

4.1. Introduction to Disciplines

A **discipline** shows all activities you may go through to produce a particular set of artifacts. We describe these disciplines at an overview level—a summary of all roles, activities, and artifacts that are involved. We also show, at a more detailed level, how roles collaborate, and how they use and produce artifacts. The steps at this detailed level are called "workflow details".

4.2. Descriptions of Disciplines

4.2.1. Business Modeling: Overview

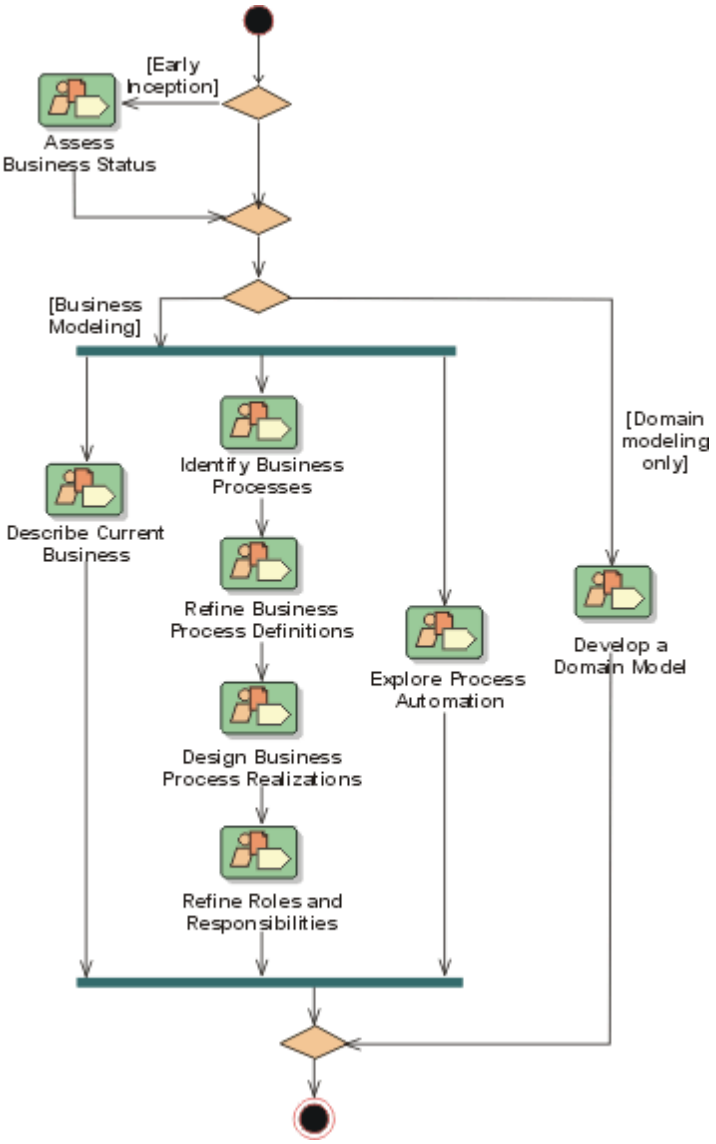
Purpose

The purposes of business modeling are:

- To understand the structure and the dynamics of the organization in which a system is to be deployed (the target organization).
- To understand current problems in the target organization and identify improvement potentials.

- To ensure that customers, end users, and developers have a common understanding of the target organization.
- To derive the system requirements needed to support the target organization.

To achieve these goals, the business modeling discipline describes how to develop a vision of the new target organization, and based on this vision define the processes, roles, and responsibilities of that organization in a business use-case model and a business object model.



Complementary to these models, the following artifacts are developed:

- Supplementary Business Specification
- Glossary

Relation to Other Disciplines

The business modeling discipline is related to other disciplines, as follows:

- The **Requirements** discipline uses business models as an important input to understanding requirements on the system.
- The **Analysis & Design** discipline uses business entities as an input to identifying entity classes in the design model.
- The **Environment** discipline develops and maintains supporting artifacts, such as the Business-Modeling Guidelines.

4.2.2. Requirements: Overview

Purpose

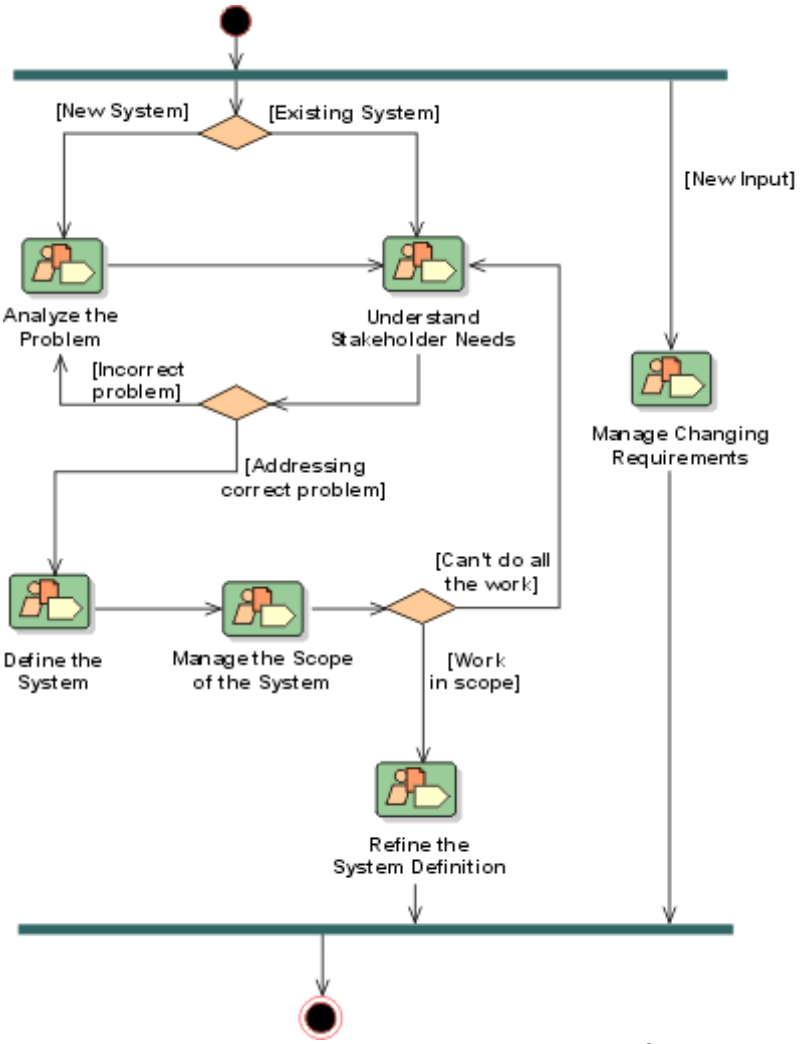
The purpose of the Requirements discipline is:

- To establish and maintain agreement with the customers and other stakeholders on what the system should do.
- To provide system developers with a better understanding of the system requirements.
- To define the boundaries of (delimit) the system.
- To provide a basis for planning the technical contents of iterations.
- To provide a basis for estimating cost and time to develop the system.
- To define a user-interface for the system, focusing on the needs and goals of the users.

To achieve these goals, it is important, first of all, to understand the definition and scope of the problem which we are trying to solve with this system. The Business Rules, Business Use-Case Model and Business Object Model developed during Business Modeling will serve as valuable input to this effort. Stakeholders are identified and Stakeholder Requests are elicited, gathered and analyzed.

A Vision document, a use-case model, use cases and Supplementary Specification are developed to fully describe the system - **what** the system will do - in an effort that views all stakeholders, including customers and potential users, as important sources of information (in addition to system requirements).

Stakeholder Requests are both actively elicited and gathered from existing sources to get a "wish list" of what different stakeholders of the project (customers, users, product champions) expect or desire the system to include, together with information on how each request has been considered by the project.



The Vision document provides a complete vision for the software system under development and supports the contract between the funding authority and the development organization. Every project needs a source for capturing the expectations among stakeholders. The vision document is written from the customers' perspective, focusing on the essential features of the system and acceptable levels of quality. The Vision should include a description of what features will be included as well as those considered but not included. It should also specify operational capacities (volumes, response times, accuracies), user profiles (who will be using the system), and inter-operational interfaces with entities outside the system boundary, where applicable. The Vision document provides the contractual basis for the requirements visible to the stakeholders.

The use-case model should serve as a communication medium and can serve as a contract between the customer, the users, and the system developers on the functionality of the system, which allows:

- Customers and users to validate that the system will become what they expected.
- System developers to build what is expected.

The use-case model consists of use cases and actors. Each use case in the model is described in detail, showing step-by-step how the system interacts with the actors, and what the system does in

the use case. Use cases function as a unifying thread throughout the software lifecycle; the same use-case model is used in system analysis, design, implementation, and testing.

The Supplementary Specifications are an important complement to the use-case model, because together they capture all software requirements (functional and nonfunctional) that need to be described to serve as a complete software requirements specification.

A complete definition of the software requirements described in the use cases and Supplementary Specifications may be packaged together to define a Software Requirements Specification (SRS) for a particular "feature" or other subsystem grouping.

A Requirements Management Plan specifies the information and control mechanisms which will be collected and used for measuring, reporting, and controlling changes to the product requirements.

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

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

The Glossary is important because it defines a common terminology which is used consistently across the project or organization.

The Use-Case Storyboard and User-Interface Prototype are all results of user-interface modeling and prototyping, which are done in parallel with other requirements activities. These artifacts provide important feedback mechanisms in later iterations for discovering unknown or unclear requirements.

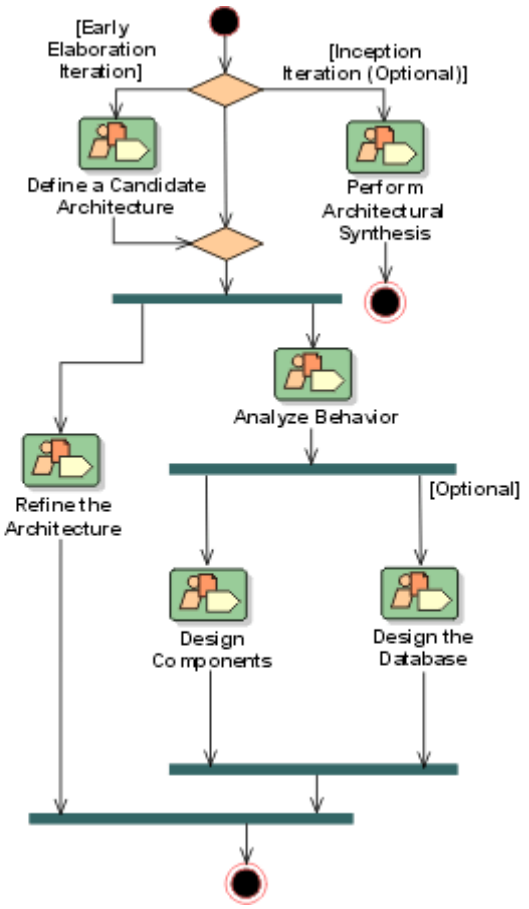
Relation to Other Disciplines

The Requirements discipline is related to other process disciplines.

- The **Business Modeling** discipline provides Business Rules, a Business Use-Case Model and a Business Object Model, including a Domain Model and an organizational context for the system.
- The **Analysis & Design** discipline gets its primary input (the use-case model and the Glossary) from Requirements. Flaws in the use-case model can be discovered during analysis & design; change requests are then generated, and applied to the use-case model.
- The **Test** discipline validates the system against (amongst other things) the Use-Case Model. Use Cases and Supplementary Specifications provide input on requirements used in the definition of the evaluation mission, and in the subsequent test and evaluation activities.
- The **Configuration & Change Management** discipline provides the change control mechanism for requirements. The mechanism for proposing a change is to submit a Change Request, which is reviewed by the Change Control Board.
- The **Project Management** discipline plans the project and each iteration (described in an Iteration Plan). The use-case model and Requirements Management Plan are important inputs to the iteration planning activities.
- The **Environment** discipline develops and maintains the supporting artifacts that are used during requirements management and use-case modeling, such as the Use-Case-Modeling Guidelines and User-Interface Guidelines.

4.2.3. Analysis & Design: Overview

Purpose



The purposes of Analysis & Design 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.

Relation to Other Disciplines

The Analysis & Design discipline is related to other disciplines, as follows:

- The **Business Modeling** discipline provides a organizational context for the system.
- The **Requirements** discipline provides the primary input for Analysis and Design.
- The **Test** discipline tests system designed during Analysis and Design.
- The **Environment** discipline develops and maintains the supporting artifacts that are used during Analysis and Design.
- The **Project Management** discipline plans the project, and each iteration (described in an Iteration Plan).

4.2.4. Implementation: Overview

Purpose

The purpose of implementation is:

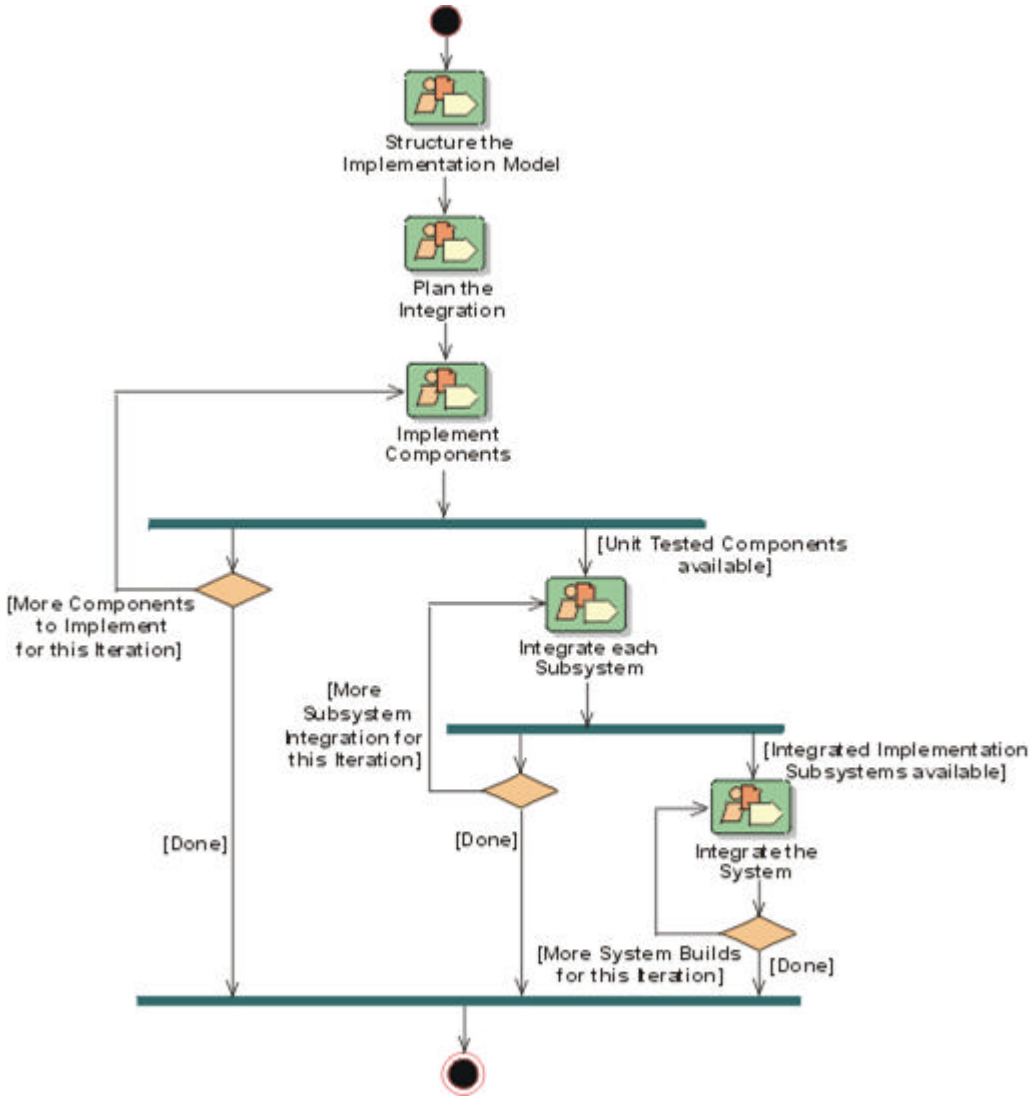
- to define the organization of the code, in terms of implementation subsystems organized in layers
- to implement classes and objects in terms of components (source files, binaries, executables, and others)
- to test the developed components as units
- to integrate the results produced by individual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

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

Relation to Other Disciplines

The implementation is related to other disciplines:

- The **Requirements** discipline describes how to, in a use-case model, capture requirements that the implementation should fulfill.
- The **Analysis & Design** discipline describes how to develop a design model. The design model represents the intent of the implementation, and is the primary input to the Implementation discipline.
- The **Test** discipline describes how to integration test each build during the integration of the system. It also describes how to test the system to verify that all requirements have been met, as well as how defects are detected and submitted.
- The **Environment** discipline describes how to develop and maintain supporting artifacts that are used during implementation, such as the process description, the design guidelines, and the programming guidelines.
- The **Deployment** discipline describes how to use the implementation model to produce and deliver the code to the end-customer.
- The **Project Management** discipline describes how to best plan the project. Important aspects of the planning process are the iteration plan, change management and defect tracking systems.



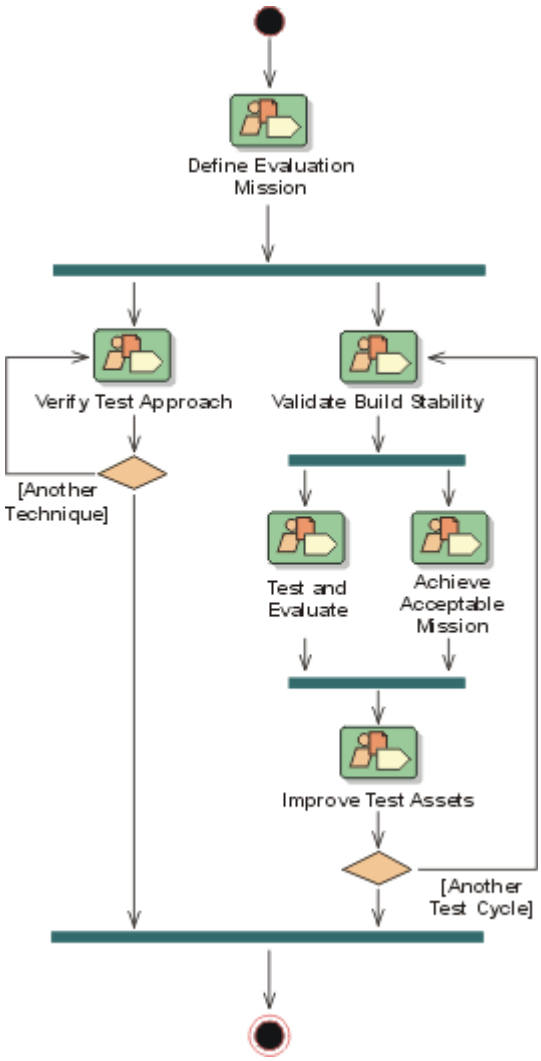
4.2.5. Test: Overview

Purpose

The Test discipline acts in many respects as a service provider to the other disciplines. Testing focuses primarily on the evaluation or assessment of *product quality* realized through a number of core practices:

- Finding and documenting defects in software quality.
- Generally advising about perceived software quality.
- Proving the validity of the assumptions made in design and requirement specifications through concrete demonstration.
- Validating the software product functions as designed.
- Validating that the requirements have been implemented appropriately.

An interesting difference between Test and the other disciplines in RUP is that Test is essentially tasked with finding and exposing weaknesses in the software product. That is interesting in that, to yield the most benefit, it necessitates a different general philosophy to that used in the Requirements, Analysis and Design and Implementation disciplines. The somewhat subtle difference is that while those other disciplines focus on completeness, Test focuses on incompleteness. A good test effort is driven by questions such as "How could this software break?" and "In what possible situations could this software fail to work predictably?". Test challenges the assumptions, risks and uncertainty inherent the work of the other disciplines, addressing those concerns by concrete demonstration and impartial evaluation. The challenge is to avoid two potential extremes: an approach that does not suitably and effectively challenge the software and expose it's inherent problems and weaknesses, and an approach that is inappropriately negative or destructive. Adopting such a negative approach you will likely never find it possible to consider the software product of acceptable quality, and will likely alienate the Test effort from the other disciplines.



Based on information presented in various surveys and essays, software testing is said to account for 30 to 50 percent of total software development costs. It is therefore perhaps surprising to note that most people believe computer software is not well tested before it is delivered. This contradiction is rooted in a few key issues.

First, testing software is enormously difficult. The different ways a given program can behave are unquantifiable. Second, testing is typically done without a clear methodology so results vary from project to project, organization to organization: success is primarily a factor of the quality and skills of the individuals. Third, insufficient use is made of productivity tools, making the laborious aspects

of testing manageable: in addition to the lack of automated test execution, many test efforts are conducted without tools that allow the effect management of extensive Test Data and Test Results. While the flexibility of use and complexity of software makes complete testing an impossible goal, a well-conceived methodology and use of state-of-the-art tools, can help to improve the productivity and effectiveness of the software testing.

For "safety-critical" systems where a failure can harm people (such as air-traffic control, missile guidance, or medical delivery systems), high-quality software is essential for the success of the system. For a typical MIS system, the criticality of the system may not be as immediately obvious, but it's likely that the impact of a defect could cause the business using the software considerable expense in lost revenue or possibly legal costs. In this "information age", with increasing demand on provision of electronically delivered services over the Internet, many MIS systems are now considered "mission-critical"; that is, companies cannot fulfill their functions and experience massive losses when failures occur.

A continuous approach to quality, initiated early in the software lifecycle, can significantly lower the cost of completing and maintaining the software. This greatly reduces the risk associated with deploying poor quality software.

Relation to Other Disciplines

The Test discipline is related to other disciplines.

- The **Requirements** discipline captures requirements for the software product, and those requirements are one of the primary inputs for identifying what tests to perform.
- The **Analysis & Design** discipline determines the appropriate design for the software product; this is the another important input for identifying what tests to perform.
- The **Implementation** discipline produces builds of the software product that are validated by the Test discipline. Within an iteration multiple builds will be tested, typically one per test cycle.
- The **Environment** discipline develops and maintains supporting artifacts that are used during test, such as the Test Guidelines and Test Environment.
- The **Management** discipline plans the project, and the necessary work in each iteration. Described in an Iteration Plan, this artifact is an important input to defining the correct evaluation mission for the test effort.
- The **Configuration & Change Management** discipline controls change within the project team. The test effort verifies that each change has been completed appropriately.

4.2.6. Deployment: Overview

Purpose

The Deployment Discipline describes the activities associated with ensuring that the software product is available for its end users.

The Deployment Discipline describes three modes of product deployment:

- the custom install
- the "shrink wrap" product offering
- access to software over the internet

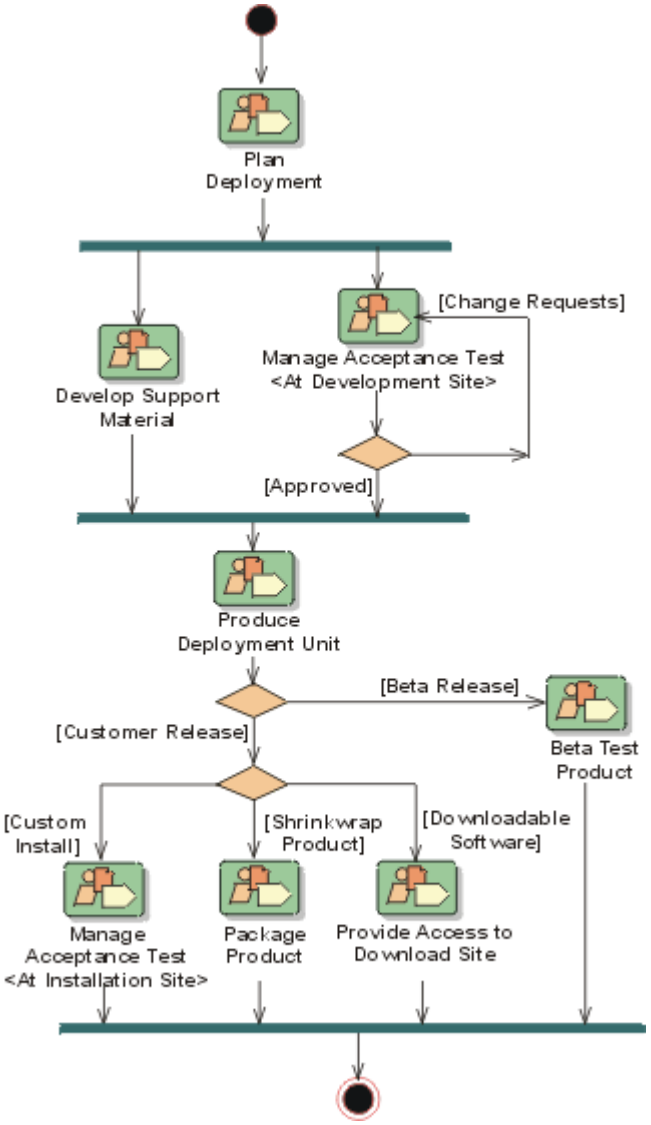
In each instance, there is an emphasis on testing the product at the development site, followed by beta-testing before the product is finally released to the customer.

Although deployment activities peak in the Transition Phase, some of the activities occur in earlier phases to plan and prepare for deployment.

Relation to Other Disciplines

The deployment discipline is related to other disciplines, as follows:

- The **Requirements** discipline produces the Software Requirements Specifications that consists of the use-case model and non-functional requirements. Together with the User-Interface Prototype, the Software Requirements specification is one of the key inputs to developing End-User Support Materials and Training Materials.
- **Testing** is an indispensable part of deployment, and the essential artifacts from the **Test** discipline are the Test Model, Test Results, and activities for managing, executing, and evaluating test results.
- The **Configuration & Change Management** discipline is referenced for providing the baselined build, and releasing the product and mechanisms for handling Change Requests that are generated as result of beta-tests and acceptance tests.
- In the **Project Management** discipline, the activities to develop an Iteration Plan and a Software Development Plan are influential on developing the Deployment Plan. Also, the work to produce a Product Acceptance plan has to be coordinated with how you manage acceptance test in the Deployment discipline.



- The **Environment** discipline provides the supporting test environment.

4.2.7. Configuration & Change Management: Overview

Introduction

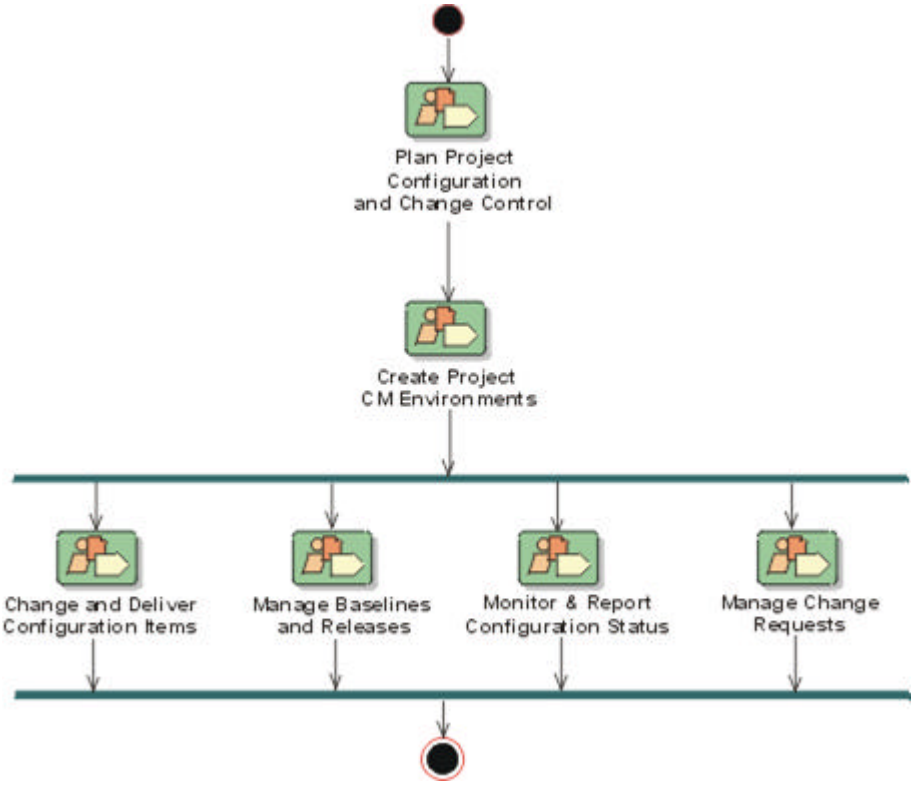
To paraphrase the Software Engineering Institute's Capability Maturity Model (SEI CMM) 'Configuration and Change Request Management controls change to, and maintains the integrity of, a project's artifacts'.

Configuration and Change Request Management (CM and CRM) involves:

- identifying configuration items,
- restricting changes to those items,
- auditing changes made to those items, and
- defining and managing configurations of those items.

The methods, processes, and tools used to provide change and configuration management for an organization can be considered as the organization's CM System.

An organization's Configuration and Change Request Management System (CM System) holds key information about its product development, promotion, deployment and maintenance processes, and retains the asset base of potentially re-usable artifacts resulting from the execution of these processes.



The CM System is an essential and integral part of the overall development processes.

Purpose

A CM System is essential for controlling the numerous artifacts produced by the many people who work on a common project. Control helps avoid costly confusion, and ensures that resultant artifacts are not in conflict due to some of the following kinds of problems:

- Simultaneous Update
- Limited Notification
- Multiple Versions

Simultaneous Update

When two or more team members work separately on the same artifact, the last one to make changes destroys the work of the former. The basic problem is that if a system does not support simultaneous update this leads to serial changes and slows down the development process. However, with simultaneous update, the challenge is to detect that updates have occurred simultaneously and to resolve any integration issues when these changes are incorporated

Limited Notification

When a problem is fixed in artifacts shared by several developers, and some of them are not notified of the change.

Multiple Versions

Most large programs are developed in evolutionary releases. One release could be in customer use, while another is in test, and the third is still in development. If problems are found in any one of the versions, fixes need to be propagated between them. Confusion can arise leading to costly fixes and re-work unless changes are carefully controlled and monitored.

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.

Relation to Other Disciplines

An organization's CM System is used throughout the product's lifecycle, from inception to deployment. As an organization's asset repository, the CM system contains current and historical versions of source files of requirements, design and implementation artifacts that define a particular version of a system or a system component

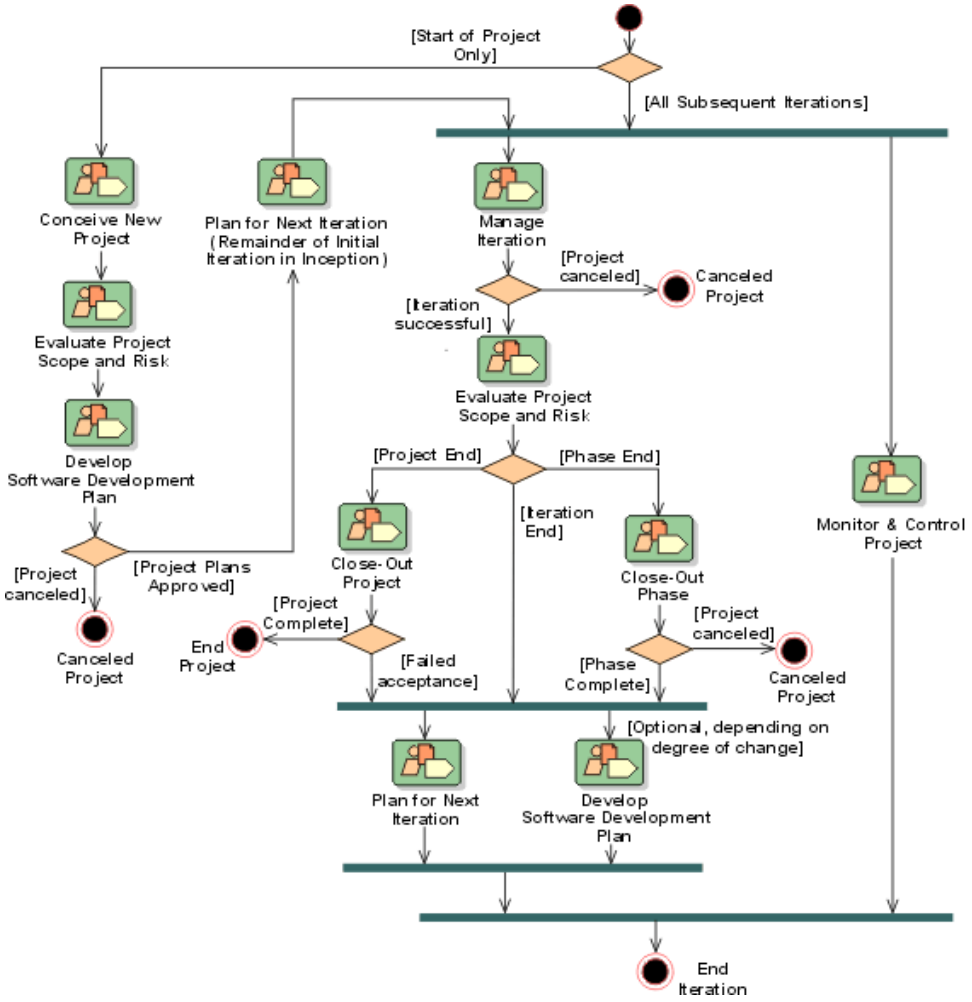
The Product Directory Structure, represented in the CM System, contains all the artifacts required to implement the product. As such, the Configuration & Change Management (CCM) discipline is related to all the other process disciplines as it serves as a repository for their resultant sets of artifacts.

- The **Business Modeling Set**,
- The **Requirements Set**,
- The **Analysis & Design Set**,
- The **Implementation Set**,
- The **Test Set**,
- The **Deployment Set**,
- The **Configuration & Change Management Set**,
- The **Project Management Set**, and
- The **Environment Set.**

4.2.8. Project Management: Overview

Introduction

Software Project Management is the art of balancing competing objectives, managing risk, and overcoming constraints to successfully deliver a product which meets the needs of both customers (the payers of bills) and the users. The fact that so few projects are unarguably successful is comment enough on the difficulty of the task.



Purpose

Our goal with this section is to make the task easier by providing some context for Project Management. It is not a recipe for success, but it presents an approach to managing the project that will markedly improve the odds of delivering successful software.

The purpose of Project Management is:

- 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.

However, this discipline of the Rational Unified Process (RUP) does not attempt to cover all aspects of project management. For example, it does **not** cover issues such as:

- Managing people: hiring, training, coaching
- Managing budget: defining, allocating, and so forth
- Managing contracts, with suppliers and customers

This discipline focuses mainly on the important aspects of an iterative development process:

- Risk management
- Planning an iterative project, through the lifecycle and for a particular iteration
- Monitoring progress of an iterative project, metrics

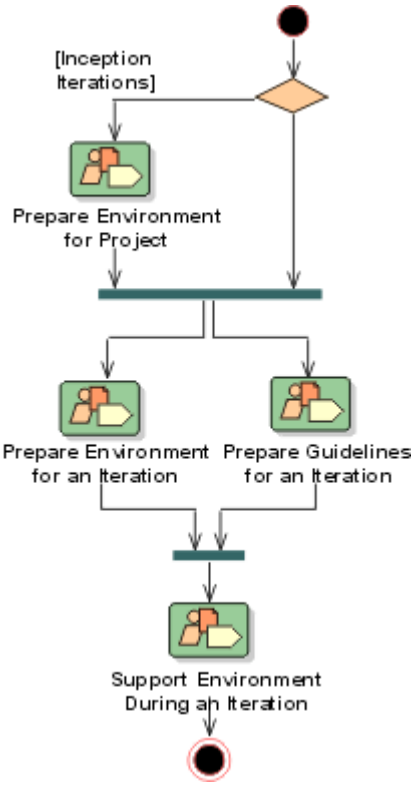
Relation to Other Disciplines

The Project Management Discipline provides the framework whereby a project is created and managed. In doing so, **all** other disciplines are utilized as part of the project work:

- Business Modeling discipline
- Requirement discipline
- Analysis & Design discipline
- Implementation discipline
- Test discipline
- Deployment discipline

The Project Management Discipline is one of the supporting process disciplines, together with:

- Configuration & Change Management discipline
- Environment discipline



4.2.9. Environment: Overview

Purpose

The environment discipline focuses on the activities necessary to configure the process for a project. It describes the activities required to develop the guidelines in support of a project. The purpose of the environment activities is to provide the software development organization with the software development environment—both processes and tools—that will support the development team.

Relation to Other Disciplines

The Environment discipline provides the supporting environment for a project. In doing so, it supports all other disciplines.

Sources:

- Rational Rose Enterprise Suite, Release Version 2002.05.20
- http://www.rational.com