Overview of Software Development Processes

Objectives

Upon completion of this module, you should be able to:

- Explain the best practices for OOSD methodologies
- Describe the features of several common methodologies
- Choose a methodology that best suits your project
- Develop an iteration plan

Additional Resources

Additional resources – The following references provide additional information on the topics described in this module:

- Arlow, Jim, Ila Neustadt. *UML and the Unified Process*. Reading: Addison Wesley Longman, Inc., 2002.
- Beck, Kent. *eXtreme Programming eXplained*. Reading: Addison Wesley Longman, Inc., 2000.
- Cockburn, Alistair. *Agile Software Development*. Reading: Addison Wesley Longman, Inc., 2001.
- Folwer, Martin. *Refactoring (Improving the Design of Existing Code)*. Reading: Addison Wesley Longman, Inc., 2000.
- Jacobson, Ivar, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley. Reading. 1999.
- Knoernschild, Kirk. *Java Design (Objects, UML, and Process)*. Reading: Addison Wesley Longman, Inc., 2002.

Reviewing Software Methodology

As described in "Describing Software Methodology" on page 2-4, methodology describes the top-level organization of a software development project. This structure usually includes phases, workflows, activities, and artifacts that transform the user requirements into a deployed software system.

Note – There are some disagreements in the industry about terminology. Some people use the term *macro-phase* instead of *phase* and *micro-phase* instead of *workflow*.

This process is illustrated in Figure 14-1.

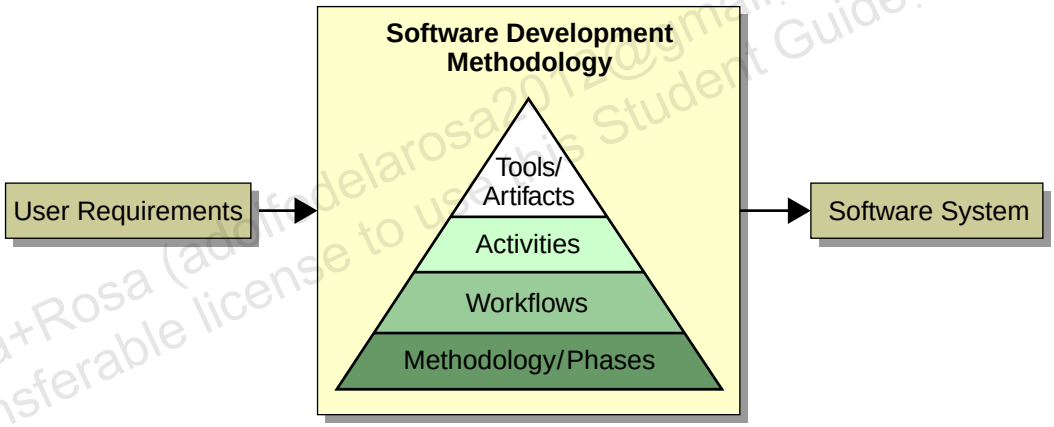


Figure 14-1 Software Development Methodology

This module provides an overview of the best practices in software methodology, an overview of several mainstream and modern methodologies, and an approach on how to decide which methodology is appropriate for your development team.

Exploring Methodology Best Practices

This section provides an overview of the best practices in software methodology. These include:

- Use-case-driven
- Systemic-quality-driven
- Architecture-centric
- Iterative and incremental
- Model-based
- Design best practices

Use-Case-Driven

> "A software system is brought into existence to serve its users." (Jacobson USDP page 5)

To know understand how a software system is to be built, you must first understand what functions the system needs to perform for the users of the system. This means that the whole development process is Use-case-driven because use cases capture the functional requirements of the system.

Here are the main issues:

- All software has users (human or machine).

  These are called *actors*. The terms *user* and *actor* are used interchangeably in this course. An actor is some entity outside of the system, such as a human user or an external system (for example, a credit card authorization system).

- Users use software to perform activities or accomplish goals.

  These are called *use cases*. Other methodologies might call this *user stories*, but the idea is the same.

- A software development methodology supports the creation of software that facilitates use cases.

- Use cases drive the design of the system.

The design workflow takes the use cases and creates a set of software components that implement the functionality specified in the use case. The designed Solution model is then used to implement the software system.

Systemic-Quality-Driven

The complete requirements of a software system include functional and non-functional requirements. *Systemic qualities* is the name given to non-functional requirements by SunTone^{SM} Architecture Methodology. A related term is *quality of service*, which refers to the qualitative characteristics of the system.

Examples include:

- Performance – such as responsiveness and latency
- Reliability – the mitigation of component failure
- Scalability – the ability to support additional load, such as more users

Systemic qualities drive the Architecture baseline of the software. After the Architecture baseline has been developed, the software designers can craft the remaining functional requirements into this baseline.

Architecture-Centric

> "Architecture is all about capturing the strategic aspects of the high-level structure of a system." (Arlow and Neustadt page 18)

Many areas of risk in software projects involve quality of service issues; such as scalability, availability, and extensibility. The Architecture workflow deals with these issues; therefore, it is a best practice to perform the Architecture workflow as early in the development process as possible. This practice mitigates risk earlier in the process rather than later.

Strategic aspects are:

- Systemic qualities drive the architectural components and patterns.

The Architecture baseline includes all components and architectural patterns that satisfy the NFRs. For example, scalability can be achieved with redundant services distributed across multiple server hosts.

- Use cases must fit into the architecture.

  The Architecture baseline defines the structure into which all of the designed components (those that satisfy the use cases) must fit. For example, in a web application, user interface component might be implemented using servlets and JavaServer Pages™ (JSP™) technology pages using a Model 2 architecture.

High-level structure is:

- Tiers, such as client, business, and resource

  Module 11, "Introducing Architectural Concepts and Diagrams," suggests that a software system be separated into components in several logical tiers. The Client tier includes all user interface components, the Business tier includes all business logic and domain entity components, and the Resource tier manages the persistence of business entities.

- Tier components and their communication protocols

  Components in adjacent tiers must be able to communicate to get their work done. Identifying or creating these protocols is a critical element of the Architecture baseline.

- Layers, such as application, platform, hardware

  In a component-based system, Application layer components are built upon specific APIs. For example, in a Java technology web application, components are built upon the servlet and JSP specifications and APIs.

  Therefore, the Application layer components depend on the APIs and the platform upon which they built. These system layers are another important consideration in the Architecture baseline.

Iterative and Incremental

"Iterative development focuses on growing the system in small, incremental, and planned steps." (Knoernschild page 77)

Requirements change. This is the unwritten law of software development. Except for very rare situations, you must assume that you will not know the complete set of requirements at the start of a software project. Iterative development makes this assumption a built-in feature of the software methodology.

Here are the key ingredients for iterative development:

- Each iteration includes a complete OOSD life cycle, including analysis, design, implementation, and test.

  Within an iteration, you will perform all of the OOSD workflows: requirements gathering, requirements analysis, architecture, design, implementation, testing, and deployment.

  Depending on the stage of development, you might spend more time on analysis and architecture than on implementation and testing. However, at the end of each iteration you should have an integrated and working (albeit incomplete) system.

  Seeing a working system early in the OOSD provides momentum to the development team and gives the client a real indication that progress is being made.

- Models and software are built incrementally over multiple iterations.

  Because you never have a complete set of requirements, you will never have a complete model of the system. That constraint has important implications:

  - First, you should only build models (and the corresponding artifacts) that are required for team communication. Building too many models or diagrams will put high demands on the team to keep them up-to-date as the project moves through successive iterations.
  - Second, the client should have their expectations set appropriately. If the client sees a prototype of the user interface, they might get the impression that more of the underlying system exists when in fact it does not.

- Maintenance is simply another iteration (or series of iterations).

  As mentioned earlier, maintenance is not a workflow, but rather a new revision of the system. Simple maintenance (such as bug fix releases) can be accomplished in a single iteration. More complex maintenance (requiring significant enhancements) might require multiple iterations.

Model-Based

Models are the primary means of communication between all stakeholders in the software project. Artifacts are physical representations of our mental models. Modeling is a fundamental property of how humans think. Whether explicit or not, all humans have mental models of reality and the systems that they develop. The choice to give these models physical representation through an artifact is a decision of the project manager.

There are several types of model artifacts:

- Textual documents

  Requirements specifications and use case scenarios require both formal and informal text. Documents, such as the Vision and System Requirements Specification, are examples of requirements-level artifacts.

- UML diagrams

  The UML provides a rich set of diagrams to represent visual views of your mental models of the proposed software system.

- Prototypes

  Prototypes are small-scale software systems built to model a certain aspect of the proposed software system. There are two main types of prototypes: technology and user interface. Technology prototypes explore a new, unknown, or risky area of technology. User interface prototypes explore the visual representation of the proposed system and enables the client and UI designer to explore various usability concerns.

Models can server many purposes:

- Communication

  Model artifacts communicate your mental models to other project stakeholders.

- Problem solving

  Some models can be constructed (either singularly or in a team) to solve a particularly difficult problem. These models tend to be *throw-away* because they serve a short-term need.

- Proof-of-concept

Some models (especially prototypes) are created to demonstrate the feasibility of a concept. These also tend to be throw away in nature. If proof-of-concept prototypes are shown to the client, it is important to set the expectation correctly that this prototype will *not* be part of the final system.

Note – Some methodologies (such as XP) do not put a high value on creating artifacts of models. However, XP's emphasis on close ties between users and developers reduces the risk of not understanding the use cases (and other requirements) of the system.

Design Best Practices

Understanding and applying design-level best practices can improve the flexibility and extensibility of a software solution. You will be introduced to these best practices throughout the course.

These best practices include the following:

- Design principles

  Design principles are rules about how software components (including classes and packages) interact or are combined to provide flexibility, extensibility, and decoupling. These principles provide the rules from which patterns arise.

  Example design principles include: Separation of Concerns, Dependency Inversion Principle, and Stable Dependency Principle. These are described in other modules of the course.

- Software patterns

  A software pattern is a *repeatable solution to a reoccurring problem within a context*. Patterns describe expert solutions to everyday problems.

  Module 10, "Applying Design Patterns to the Design Model," provides more information about design patterns.

- Refactoring

  Refactoring is "the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure." (Fowler page xvi)

  Refactoring is done all of the time during software development and especially during maintenance. Refactoring is recognized as a legitimate and significant practice. Programmers should be encouraged to develop the skill of refactoring. Fowler's seminal book, *Refactoring*, provides a detailed exploration of this topic.

- Sun Blueprints

  Sun Microsystems publishes documents that bring together best practices in a wide variety of technical areas including J2EE development, cluster configurations, and so on. For more information visit the web page at `http://www.sun.com/blueprints/`.

Surveying Several Methodologies

In this section you will be introduced to several mainstream and new OOSD methodologies:

- Waterfall
- Unified Software Development Process (USDP or just UP)
- Rational Unified Process (RUP)
- Scrum
- eXtreme Programming (XP)

Waterfall

The Waterfall methodology is one of the oldest development processes. It is presented here as a *baseline* to evaluate other methodologies. The Waterfall methodology has the following characteristics:

- Waterfall uses a single phase in which all workflows proceed in a linear fashion.

  Requirements gathering precedes analysis, which precedes architecture, and so on. Variations of the Waterfall methodology permit backing up to the previous workflow.

- This methodology does not support iterative development.

  The Waterfall methodology proceeds as a single, large iteration.

- This methodology works best for a project in which all requirements are known at the start of the project and requirements are not likely to change.

  This is the fundamental flaw of the Waterfall methodology because it is very unlikely that the system requirements can be fully specified at the beginning of the development process. It is even more unlikely that requirements changes will not occur during development.

- Some government contracts might require this type of methodology.
- Some consulting firms use this methodology in which each workflow is contracted with a fixed-price bid.

  Some consulting firms are required (by the client) to produce a system on a fixed-price bid. Changes to requirements can be managed by the use of an "engineering change order" agreement between the consultant and the client.

The Waterfall methodology is illustrated in Figure 14-2.

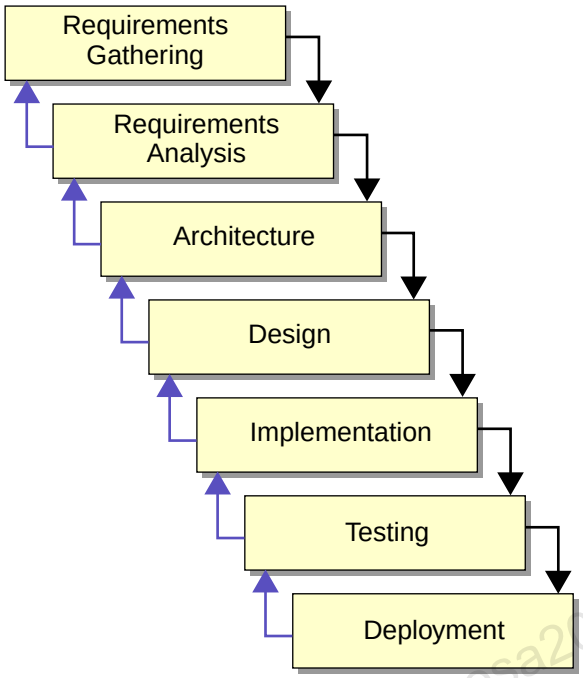


Figure 14-2 Waterfall Methodology

Note – Figure 14-2 shows arrows pointing up to a previous workflow. This feature might not be considered part of a formal definition of the Waterfall methodology.

Unified Software Development Process

The Unified Software Development Process (USDP) is the non-proprietary version of Rational's methodology created by Grady Booch, Ivar Jacobson, and James Rumbaugh. The USDP is also called the Unified Process (UP).

The UP methodology organizes the development process into four phases:

- *Inception* – Creates a vision of the software

  This phase focuses on understanding the business case for the proposed system. Also, a preliminary architecture is proposed, important project risks are identified, and the elaboration phase is planned in detail.

- *Elaboration* – Most use cases are defined plus the system architecture

  The purpose of this phase is to reduce project risk by creating an architecture baseline upon which the rest of the software system will be constructed. At the end of this phase, the project manager should have enough information to plan the construction and transition phases.

- *Construction* – The software is built

  Through multiple iterations the software system is built incrementally by adding additional use cases to the architecture baseline. At the end of this phase, the system is ready for Beta release.

- *Transition* – Software moves from Beta to production

  This phase prepares the system for production release. Activities include acceptance testing, debugging, training, and building the production environment.

The UP methodology is illustrated in Figure 14-3. What is not shown in this diagram is that multiple iterations can be constructed in each phase.

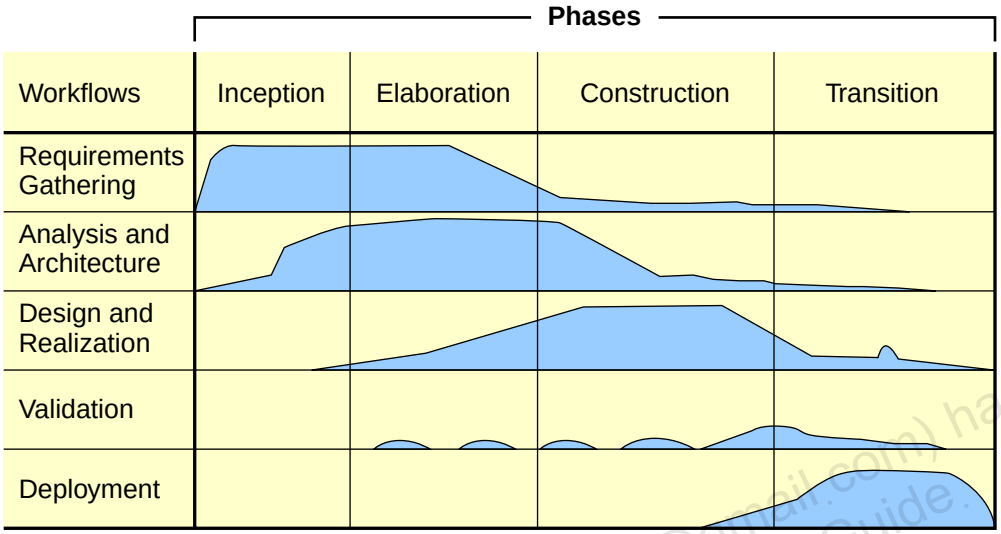
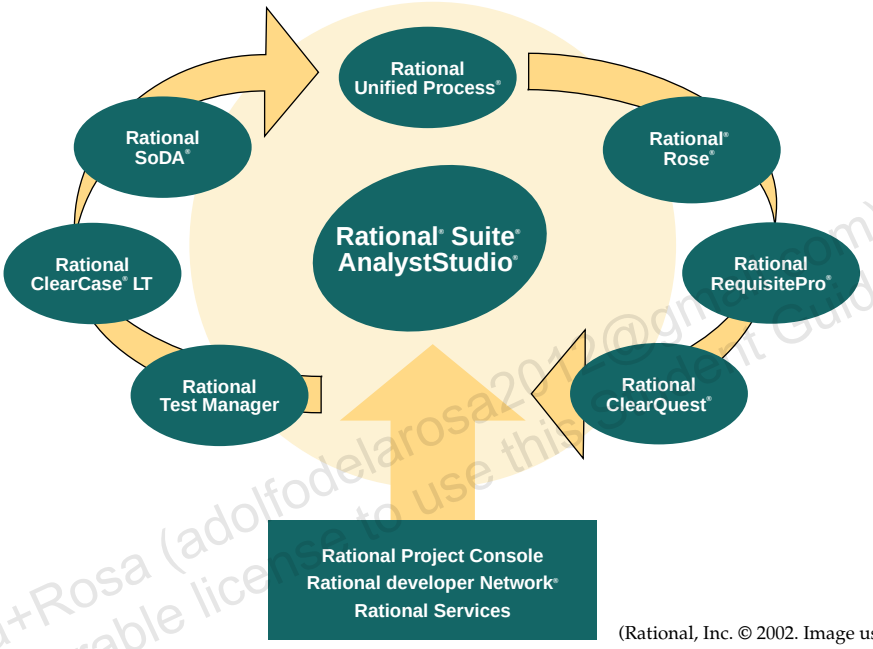


Figure 14-3 UP Methodology

The UP methodology embraces all of the best practices listed in the previous section, with less focus on systemic-qualities-driven. The UP is also focuses on using UML to develop modeling artifacts and on planning the activities of the development team.

Rational Unified Process

RUP is the commercial version of the UP methodology created by Grady Booch, Ivar Jacobson, and James Rumbaugh. RUP is UP with the support of Rational's tool set. These tools manage the phases, workflows, and artifacts throughout the project life cycle. The set of tools that support RUP are shown in Figure 14-4.



(Rational, Inc. © 2002. Image used with permission.)

Figure 14-4 Rational's Tool Set for the RUP Methodology

Scrum

Scrum is an iterative and incremental development framework that is often used in conjunction with agile software development.

The Scrum framework includes a set of practices and roles. The key features of Scrum are:

- Each Sprint is typically 15 to 30 days and produces a deliverable increment of Software.
- A subset of features or requirements are moved from the Product backlog to the Sprint backlog at the beginning of each Sprint.

  The Product backlog is a high-level document containing system requirements, estimates of priorities, and the required development effort.

- The requirements in the Sprint backlog are developed during the Sprint.

  Each Sprint produces a version of the software that the client can use.

- The Sprint progress is reviewed every 24 hours in a daily Scrum meeting.
- The Scrum framework requires team-oriented responsibilities.

Figure 14-6 shows the Scrum development methodology.

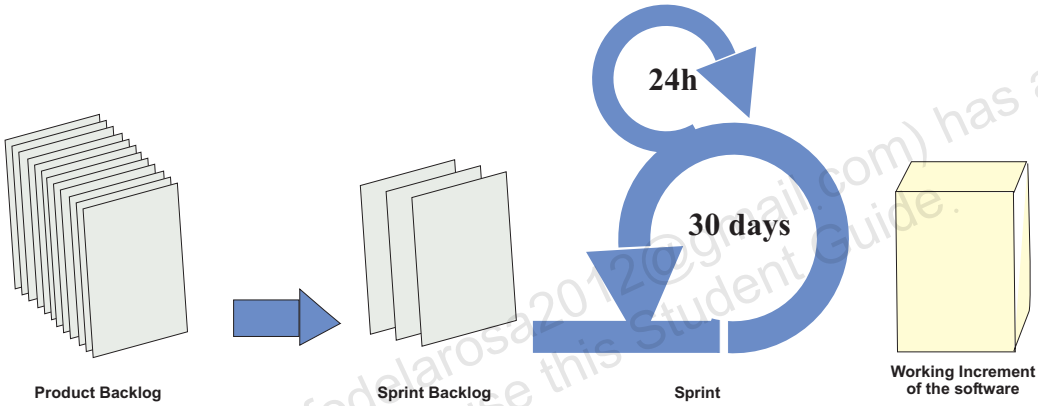


Figure 14-5 Scrum Development Framework

eXtreme Programming

> "XP nominates coding as the key activity throughout a software project." [Erich Gamma, forward to Beck's XP book]

XP is one of several relatively new methodologies, called *Agile methodologies*. In many ways, XP is a dramatic departure from traditional methodologies. XP focuses on facilitation of change and communication within the team and with the clients.

It is impossible to give it a rich presentation in this module. However, here are a *few* key ideas:

- Pair programming – If code reviews are good, then review code all the time.

  Developers are paired together during coding sessions. As one person types the other can be reviewing the code and making suggestions to clean up or refactor the code.

- Testing – If testing is good, then test all the time, even the customers.

  Testing is the cornerstone of XP. No system code should be written until a test is developed to verify that the system code is written correctly. Creating the tests helps the developers think through the code solution before they write the code.

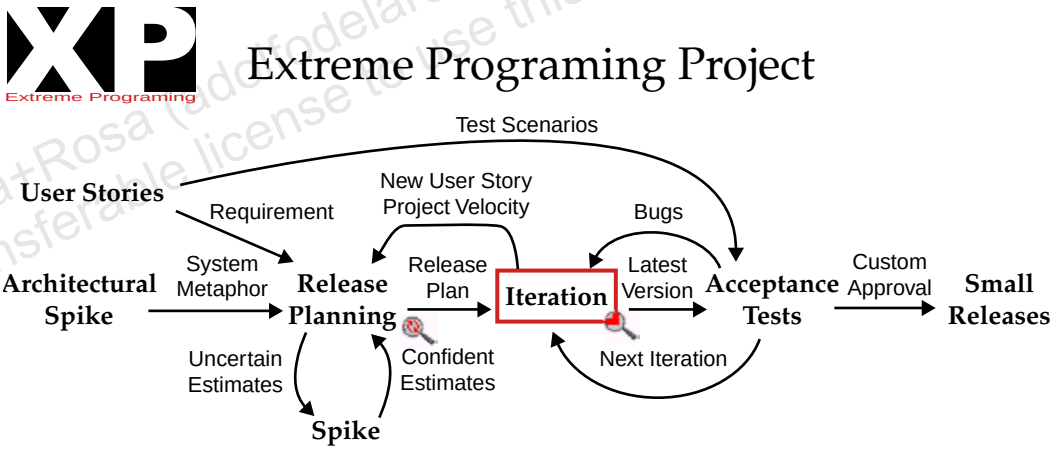
- Refactoring – If design is good, then make it part of everybody's daily business.

  Pair programming encourages continual refactoring. Refactoring code is essentially doing design while you code.

- Simplicity – If simplicity is good, then always leave the system with the simplest design.

  This idea is powerful: The system should always have the simplest design to support the use cases; no more, no less.

Iterative development is essential and is taken to the extreme with XP. Small, almost daily, releases are built from *user stories* which are also called use case scenarios. This methodology is illustrated in Figure 14-6.



(J. Donvan Wells © 2002. Image used with permission.)

Copyright 2000 J. Donvan Wells

Figure 14-6 XP Methodology

Choosing a Methodology

Among the numerous factors that can guide the choice of a methodology for a given product, this section will focus on the following factors:

- Company culture – Process-oriented or product-oriented
- Make up of team – Less experienced developers might need more structure and people have distinct job roles
- Size of project – A larger project might need more documentation (communication between stakeholders)
- Stability of requirements – How often requirements change

Choosing Waterfall

When to use:

- Large teams with distinct role

  Large teams require a great deal of structure to support the communication problems between developers and subteams. Sometimes, teams exist only for a single workflow. For example, a team of business analysts might perform a complete requirements analysis. They then hand off their work to the architecture team, and so on.

- Choose waterfall when the project is not risky

  This is the most important factor in deciding to use Waterfall. A project is not risky when the domain of the requirements and the technologies to be used in the proposed system are well known.

Issues:

- Not resilient to requirements changes

  It is difficult to go back to a previous workflow to make changes to the system. It is usually acceptable to go back to the last workflow, but it is nearly impossible to go from the Implementation workflow back to the Requirements Gathering workflow.

- Tends to be documentation heavy

  The Waterfall methodology depends on well-defined documentation deliveries between workflows. These documents tend to be large from the start because everything in the current workflow must be defined before proceeding to the next workflow.

Choosing UP

When to use:

- Company culture is process-oriented

  There is a continuum along the lines of *process orientation* in which on one side there is too much process (large amount of documentation and formal ceremony) and too little process (what some might characterize as *hacking*). UP tends to favor a process-oriented culture.

- Teams with members that have flexible job roles

  While UP encourages the delineation of job roles, it is not as strict about the separation of workflows and the distinction of job roles that perform each workflow, which is the hallmark of the Waterfall methodology.

- Medium- to large-scale projects

  Projects of a certain size require documentation so that all of the stakeholders of the project can understand the problem and solution domains clearly. UP also supports documentation used to communicate across workflow boundaries.

- Requirements are allowed to change

  Because UP uses an iterative process, requirements changes can be folded into later iterations. Every iteration goes through the complete set of workflows, so new requirements can be supported in another iteration.

Issues:

- Tends to be process and documentation heavy

  UP makes extensive use of the UML to diagram various models developed in the workflows. This close tie to the UML makes UP documentation heavy.

- This is overkill for small projects

  UP is overkill for small-scale projects because UP requires more process and modeling than a small project needs.

Choosing RUP

When to use:

- Same reasons as UP

Because RUP is based on UP, it is applicable whenever UP is applicable.

- Your company owns Rational's tool set

  RUP requires the Rational tools. If your company already uses Rational tools, then you are in a position to use the RUP methodology.

Issues:

- Same issues as UP

  Because RUP is based on UP, it has the same issues as UP.

- Tool set learning curve

  There are many tools used by RUP. There is a steep learning curve to using these tools properly within the RUP methodology.

- Tools lock the team into a process

  The tool set will lock you into the RUP methodology. However, the RUP tools can be customized. The customization process is difficult.

Choosing Scrum

When to use:

- Priorities of the requirements are constantly changing

  Features to be developed in each Sprint are decided at the beginning of the Sprint and not at the beginning of the project.

- When you need to deliver a working increment of the software every 30 days

  You can deliver a working incremental release of the software at the end of each iteration.

Issues:

- Requires a committed team

  Each Sprint team is responsible for the delivery of that Sprint increment. Scrum refers to their role as Pigs, as their bacon is on the line.

- Primary focus is on the functional requirements for each Sprint

Non-functional requirements might be not be considered adequately. The optimum choice of system architecture is primarily dependent on non-functional requirements. Therefore, a Scrum process might result in a non-optimum system architecture development.

Choosing XP

When to use:

- Company culture permits experimentation

  The XP methodology is a radical departure from traditional development processes. It must be part of the company culture to permit experimentation with the process. If your company wants to experiment with the XP methodology, then a small project that does not have significant schedule constraints would be a good candidate.

- Small, close (proximity) teams with flexible work spaces

  Pair programming and flexible *open* work spaces are fundamental to the XP development process. A project using XP is likely to fail if the teams are distributed geographically (even within the same building).

- Team must have as many experienced developers as inexperienced

  To facilitate pair programming and coaching, the development team must have a good ratio of experienced to inexperienced developers. A team full of inexperienced developers will likely fail using XP.

- Requirements change frequently

  One of the greatest benefits of XP is that the process easily supports changing requirements.

Issues:

- Tends to be documentation light

  User stories and the code are documented by using the XP methodology. XP discourages creating models. Design is done in real-time and documented in the code.

Project Constraints and Risks

Project constraints and risks are often managed by Project Architect.

Project constraints and risks should be assessed during the initial stages of the development process.

Project Constraints

Project development constraints are often confused with NFRs, which are the runtime constraints of the system.

Project constraints primarily focus on constraints during the development process.

Typical project constraints are:

- Project must be developed on a specific platform

  For example, the application and the web server must run on Sun hardware.

- Project requires specific technologies

  For example, the application server must run on Java Platform, Enterprise Edition (Java EE), the Web Services service must use SOAP (Simple Object Access Protocol). The data store must use Oracle RDBMS (Relational Database Management System).

- Project has a fixed deadline

  For example, the project must be ready by the next olympic games.

Note – A detailed discussion of the issues and risks around making a project calendar-constrained are beyond the scope of this course. For more information, read Fred Brooks *The Mythical Man-Month*.

- System interacts with specific external systems

  Integration with specific external systems is an important constraint on the project.

Project Risks

Every project involves some level of risk. Any situation or factor that could lead to an unsuccessful conclusion to a project defines a risk. Each constraint will usually affect several risks factors, some positively and some negatively.

The five main risk areas are:

- Political
- Technological
- Resources
- Skills
- Requirements

Political Risks

A political risk is any situation in which the project is competing with other projects (internal or external) or in which the project might conflict with a governmental law. These are extremely sensitive areas to discuss with the business owner. The best thing to do is to listen closely for warning signs in what the business owner is saying.

Here are a few warning signs:

- When there is a competing project or competitor.

  Sometimes the competition comes from within. In large companies, there might be two or three groups trying to solve the same problem. Usually this situation might not be known.

  Other times the competition comes from a competing company with a similar product. The client company might choose to pursue the project, but if the competitor has a better product or gets to market earlier, then the client company might terminate project.

- When the project manager's boss has revoked funding, equipment, or human resources without warning in the past.

  Financial situations might cancel some projects. Try to ascertain if such issues have cancelled other projects within this organization.

- When there are interpersonal problems or infighting either within the development team or within the management hierarchy.

Some large companies have organizations that are not mutually cooperative. Does the project depend on help from another organization that is hostile to the business owner's organization?

Sometimes interpersonal problems occur within the project manager's organization itself. Are there members of the development team that have had fights over design strategies or architectural decisions? How well were these issues resolved? Were they resolved at all?

- When the project conflicts with governmental laws and regulations.

  Beyond interoffice political risks, there could also be political risks with regard to governmental regulations. For example, if your product is produced in the United States but sold abroad and it uses cipher technologies, then you will have to deal with export considerations.

Technology Risks

A technology risk exists when the project might use a technology that is unproven, cutting edge, or difficult to use. Some of the technologies that you might encounter include: web applications, web services, hand-held devices, voice recognition systems, telephone integration systems, specific computer languages, computer platforms, and so on.

Sometimes even the software development methodology might be a technology risk. For example, if the business owner wants the development team to use XP and the company has never tried XP before.

Here are a few warning signs:

- The business owner uses a lot of technology buzz words, and does not really understand their meaning.

  This is a dangerous situation that needs to be dealt with early. Try to steer the business owner away from any discussion about specific technologies and back toward what the system must do. If you have a good rapport with the business owner, you might attempt to explain some capabilities these technologies and how they might be useful to the project.

- The business owner insists that the project take a specific technological direction.

  If the business owner persists, then make note of the specific technology. You can have a discussion about risks in follow-on interviews.

- The business owner wants to use cutting-edge technologies to solve the business problem.

  This situation is also dangerous. The business owner might be very knowledgeable about certain technologies and the recommendation to use a cutting edge technology might be appropriate. However, it is still a risk if the development team has never worked with that technology.

- The new system must integrate with a legacy system.

  Integrating to a legacy system is almost always risky because often the original architect or development team for the legacy system has left the company. This risk requires a great deal of investigation to understand how the legacy system works and what would be involved to enable the new system to communicate with the legacy system.

Resource Risks

A resource risk exists when the project does not have all of the resources (human, equipment, or money) required for successful completion.

Here are a few warning signs:

- The business owner mentions that the project is under a tight budget.

  Budget constraints can be just as difficult to mitigate as schedule constraints because on most projects time equals money. You should provide the client with realistic schedule and cost information. It is then the business owner's decision to proceed with the project or not. Often what happens is that there is a negotiation about what is included in the first and follow-on releases of the system.

  It is important to correctly set expectations with the client. That is exactly why you will ask the business owner to rate each use case as: essential, high-value, and follow-on.

- The IT staff is currently over committed.

  Many organizations with IT departments are over committed; that is, all of the developers are working on other projects. If you are a consultant, then this is only an issue if you are required to have several client IT developers on your team. Moving client IT developers to your team is done to encourage technology transfer from the consulting organization to the client's IT organization.

- The project is calendar-driven.

  This risk is roughly equivalent to the first issue: tight budget.

Skills Risks

A skills risk exists when the project team does not have the necessary knowledge or experience.

Here are a few warning signs:

- When the project is constrained to use a specific technology but the development team has not been trained in that area.

  For example, if the project requires a web application and no one on the team has ever built one.

- When the project will be developed in a language that is not known by the development team.

  This is especially problematic when there is a language *paradigm shift* such as from structured programming to object-oriented programming.

Requirement Risks

A requirement risk exists when a given use case, scenario, or NFR, is not completely known.

Here are a few warning signs:

- When business owner says something like "I will know it when I see it."

  Sometimes it is hard for the business owner to describe in words how a use case functions. In this situation, try to have the business owner show you how it works or draw you a diagram of how it works. Do not spend too much time on this; you can always record this as a risk and move on.

- When the business owner cannot provide a scenario for the use case.

  Sometimes it is hard for the business owner to know how a use case functions. Some use cases require detailed interviews with a domain expert. It is sufficient for the Vision document to simply record this as a risk and to reevaluate the use case with an expert. Make sure that you ask the business owner who is the expert in that area.

- When the business owner does not know that a use case exists.

  This situation is dangerous and might occur if the company is going through several changes or is being merged with another company.

Producing an Iteration Plan

Iteration plans are mainly applicable to iterative and incremental development processes. However, some of the techniques in this topic, can be used to determine the linear order of development in a non-iterative process.

Iteration plans determine the order in which the software components are developed.

If you are using a use-case-driven methodology or any other methodology, consider each use case based on priority, risk, and dependencies.

The iteration plan might be documented in detail early in the project and rigidly followed, or might be loosely planned and open for change as the project progresses.

Prioritizing Use Cases

MuSCoW

Use cases (or user stories, high-level requirements, and product features) are often categorized based on priority. The terms High, Medium, and Low can be used but are open to interpretation. A less ambiguous method that is commonly used is the MuSCoW prioritization technique, where priorities are defined as:

- **Mu**st have this

  Might also be defined as a minimum usable subset.
- **S**hould have this, if at all possible

  Important as Must, but are less time critical or have a workaround
- **Co**uld have this, if it does not affect anything else

  Nice to have. For example, they improve the user experience of the product.
- **W**on't have this time, but would like to include in the future

Assessing Use Case Risk

Consider each use case based on risk. The risk factor considerations should include:

- Complexity of the use case

  A complex use case is usually more risky than a simple use case because of the number of unknown issues that you might encounter during development. However, if a use case is complex, but is similar to the previous work done by the team, the risk factor is reduced.

- Interfaces to external devices and system

  Interfacing to external devices and system often results in unexpected issues. Even the best documented interfaces to external system might miss some critical information.

Architectural Significance

Use cases are architecturally significant if they have project-critical non-functional requirements (NFRs). These NFRs determine the success or failure of the project.

Project-critical NFRs require the architect to choose an architecture that satisfies those NFRs. For example, performance and scalability NFRs can affect the distribution of major components, and the number and type of each server.

Architectural Baseline

An architectural baseline is a subset of system requirements. This baseline can be realized in code early in the project and tested to prove that the chosen architecture satisfies major NFRs.

The baseline code includes the architecturally significant use cases. This process of testing these architecturally significant use cases is often called "Architectural Proof-of-Concept."

In the Unified Process (UP), the architectural baseline should be completed at the end of the elaboration phase.

Timeboxing

Each iteration in an iterative and incremental development process has a fixed time duration. This process is called timeboxing.

The key features of timeboxing are:

- The iteration must complete on schedule.

  Even if the chosen set of use cases is not completely developed, the iteration completes on schedule.

- Unfinished use cases are rescheduled to the next iteration.

  If you have a significant number of deferred use cases, or they are deemed to be more complex than estimated, then you might need to add a new iteration into the schedule, that would extend the project.

Use cases are prioritized. So continuous rescheduling of a specific use case—often called skidding—might indicate that a high risk use case is not developed until almost the end of the project. This rescheduled use case might prove difficult or impossible to build. Therefore, you should avoid skidding.

80/20 Rule

In an iterative and incremental development process, you do not need to fully understand every aspect before moving to the next step.

The 80/20 rule can be applied. For example:

- For 20% of the effort, you can understand 80% of the detail.
- For 20% of the effort, your specification can achieve 80% accuracy.

Missing details and accuracy are found in the subsequent iterations for a fraction of the effort.

This rule works only if the software that you build is flexible. The OO paradigm when applied correctly is flexible with changes being predominantly additive or subtractive.

Producing an Iteration Plan

An iteration plan contains use cases that you intend to develop in each iteration.

In UP, this plan partly occurs during the inception phase and is completely defined by the end of elaboration.

The assessment criteria includes the following items:

- Use case priority
- Use case risk
- Architectural significance
- Estimated time to develop a use case
- Dependency on other use cases

  You can build a simulation of the use case or classes to allow building a dependent use case.

Figure 14-7 shows a subset of the use cases for a library system. There are at least 40 use cases even for the simplest library system.

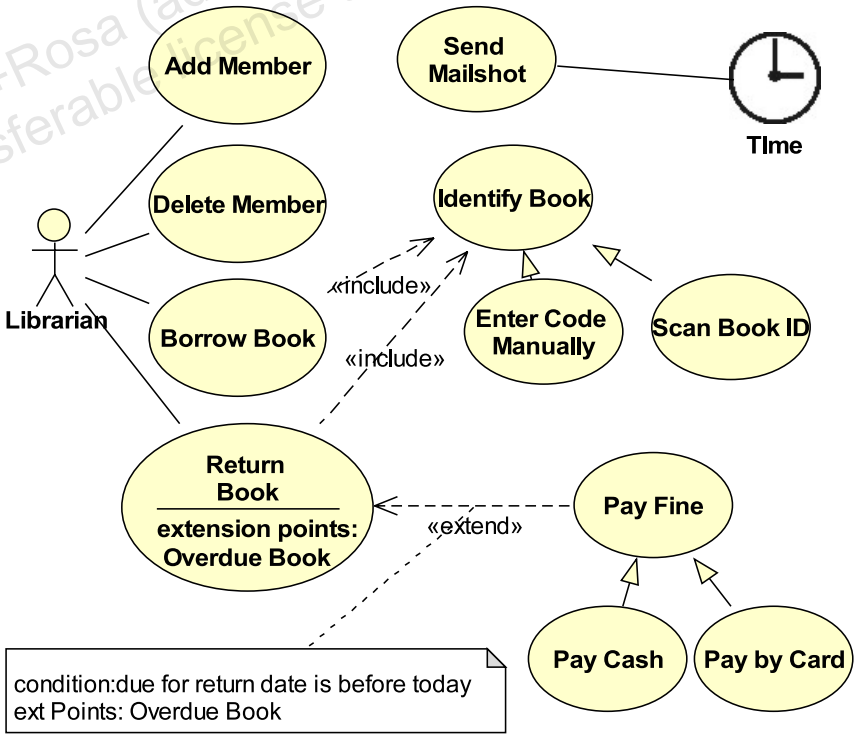


Figure 14-7 Sample Library Use Case Diagram

Table 14-1 shows an example of the information that you might use to determine the iteration plan. In practice, you might not need to build a table as all the information should exist in each Use Case form and Supplementary Specification Document.

Table 14-1 Sample Iteration Plan Details

Use Case Name	Priority	Risk	Architectural Significance	Dependency
Return Book	Must have	Medium	Must be complete within 60 seconds, and easy to use	Borrow Book
Pay Fine	Should have	Medium	None	Return Book
Pay Fine by Cash	Should have	Low	None	Pay Fine
Pay Fine by Card	Could have	High	Transaction with external system must be complete in 30 seconds	Pay Fine
Borrow Book	Must have	Medium	Must be complete within 60 seconds, and easy to use	Add Book Add Member, Identify Book
Identify Book	Must have	Medium	Must complete within 5 seconds	Add Book
Enter Code Manually	Must have	Low	Must be fast	Identify Book
Scan Book ID	Should have	Medium	Interaction with a bar code scanner	Identify Book
Delete Member	Should have	Low	None	Add Member
Send Mailshot	Could have	Low	None	Add Member

Note – The entries in Table 14-1 are highly subjective, and often based on the writer's previous experiences. Therefore, you might disagree with some of the entries. In practice, obtain a consensus of option from all the stakeholders.

Table 14-2 shows a suggested example iteration plan based on the details shown in Table 14-1. This example uses the Add Book and Add Member Book simulations to concentrate on the Borrow Book and Return Book use cases. The first iteration is ignored, which is the inception phase in UP. However, you can build a few use cases in inception to prove the viability of the project. The subsequent iterations of the construction and transition phases are ignored.

Table 14-2 Sample Iteration Plan

Elaboration Iteration 2	Elaboration Iteration 3	Construction Iteration 4	Construction Iteration 5	Construction Iteration 6
Borrow Book	Return Book	Add Book	Add member	Send Mailshot
Identify Book	Pay Fine by Card	Pay Fine by Cash	Modify Member	Delete Member
Enter Code Manually	Scan Book ID			
Simulation of Add Member	Pay Fine			
Simulation of Add Book				

Summary

In this module, you were introduced to several OOSD methodologies. Here are a few important concepts:

- Iterative and incremental development
- Architecture-centric development
- Project risks and constraints
- Developing an iteration plan based on a use case's:
  - Priority
  - Risk
  - Architectural significance

No one methodology fits every organization or project. You can create your own methodology by adapting an existing methodology to suit your requirements and by following the best practices.