Applying UML and Patterns

An Introduction to Object-oriented Analysis and Design and Iterative Development

Chapter 1
Object-oriented Analysis and Design
What Will You Learn

• What does it mean to have a good object design
  - Owning a hammer doesn’t make one an architect
• UML vs. Thinking in Objects
  - The important thing to learn: how to think in objects
  - UML: language for OOA/D and “software blue prints”, both as a tool of thought and as a form of communication
• OOD: principle and pattern
  - Design metaphor: responsibility-driven design
  - Best practice principle, heuristics, or pattern
• Iterative development, Agile modeling, and Agile UP (Unified Process)

Topics and Skills

- OOA/D
- Patterns
- UML notation
- Principles and guidelines
- Requirements analysis
- Iterative development with an agile Unified Process
- Topics and Skills
Analysis and Design

• **Analysis**
  - emphasizes an *investigation* of the problem and requirements, rather than a solution. For example, if a new online trading system is desired, how will it be used? What are its functions?

• **Design**
  - emphasizes a *conceptual solution* (in software and hardware) that fulfills the requirements, rather than its implementation. For example, a description of a database schema and software objects.

---

Analysis and Design

• **Analysis**
  - Discover the key abstractions that form the vocabulary of the problem domain.
  - Remove programming language concepts and emphasize the language of the domain.
  - Abstractions, their behavior, and interactions that define the conceptual model of the problem (not software) domain

• **Design**
  - Structure the system within an architectural framework
  - Map analysis abstractions into a software design class hierarchy.
  - Assemble objects (class instances) and their behaviors into collaborations.
  - Discover and invent software abstractions not in the problem domain but needed for implementation
  - Organize classes in hierarchies
Object-oriented Analysis and Design

- **Object-oriented analysis**
  - Emphasis on finding and describing the objects—or concepts—in the problem domain. For example, in the case of the flight information system, some of the concepts include *Plane*, *Flight*, and *Pilot*.

- **Object-oriented design**
  - Emphasis on defining software objects and how they collaborate to fulfill the requirements. For example, a *Plane* software object may have a *tailNumber* attribute and a *getFlightHistory* method.

```java
public class Plane {
    private String tailNumber;
    public List getFlightHistory() {...}
}
```
Object-oriented Analysis and Design

• **Object-oriented analysis**
  - Defines the problem domain according to the requirements
  - Sets the basic “vocabulary” of the problem domain for the design and coding activities
  - Surveys the possible solutions and discusses tradeoffs
  - Models the problem from the object perspective

• **Advantage of object-oriented analysis**
  - The analysts don’t have to be “language experts”
    • The experts in the problem domain and the implementation-level experts can communicate using a common notation

Object-oriented Analysis and Design

• **Object-oriented design**
  - Takes the products produced by analysis, then details and designs the solution in terms of some target environment
  - Concerned with real-world concerns like, reliability, performance ..
  - Deals with “assignment of functionality to different processes or tasks”
  - Deals with database issues and “distributed object environments”

• Object-oriented analysis and design use the same kinds of modeling notations – the main difference is “problem” vs. “solution” modeling
Object-oriented Analysis and Design 5

- Examples of object oriented models
  - Requirements and analysis:
    - Use case diagram
    - Interface model
    - Business/Domain Object model
    - Application Object model
    - Object Interaction model
    - Dynamic model
  - Design
    - Design Object model
    - Design Object Interaction model
    - Design Dynamic model
  - Implementation: Source code
  - Testing: Test cases

Object-oriented Analysis and Design 6

<table>
<thead>
<tr>
<th>Analysis - Investigate the Problem</th>
<th>Design - Create Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Analogy</td>
<td></td>
</tr>
<tr>
<td>Object-oriented Analysis &amp; Design</td>
<td></td>
</tr>
<tr>
<td>Associated Documents</td>
<td></td>
</tr>
<tr>
<td>What are the business processes?</td>
<td>Requirements analysis</td>
</tr>
<tr>
<td></td>
<td>Use cases</td>
</tr>
<tr>
<td>What are the roles?</td>
<td>Domain analysis</td>
</tr>
<tr>
<td></td>
<td>Conceptual model</td>
</tr>
<tr>
<td>Who is responsible for what?</td>
<td>Responsibility assignment, interaction design</td>
</tr>
<tr>
<td>How do they interact?</td>
<td>Design class diagrams, Collaboration diagrams</td>
</tr>
</tbody>
</table>
A Short Example 1

- **Define Use Cases**
  - Use cases are not an object-oriented artifact—they are simply written stories. They are a popular tool in requirements analysis.
  - *Play a Dice Game* use case:
    - Player requests to roll the dice. System presents results: If the dice face value totals seven, player wins; otherwise, player loses.

A Short Example 2

- **Define a Domain Model (aka conceptual object model)**
  - Creating a description of the domain from the perspective of objects. There is an identification of the concepts, attributes, and associations that are considered noteworthy.
A Short Example

- Assign object responsibilities and draw interaction diagrams
  - To illustrate these collaborations is the sequence diagram. It shows the flow of messages between software objects, and the invocation of methods.

```
DiceGame
play()

Die
faceValue : int
getFaceValue() : int
roll()

Die
fv1 := getFaceValue()

Die
fv2 := getFaceValue()
```

A Short Example

- Define Design Class Diagrams
  - A static view of the class definitions is usefully shown with a design class diagram. This illustrates the attributes and methods of the classes.

```
DiceGame
  die1 : Die
  die2 : Die
  play()

Die
  faceValue : int
  getFaceValue() : int
  roll()
```
Unified Modeling Language 1

- The UML is standard diagramming language to visualize the results of analysis and design.
- Notation (the UML) is a simple, relatively trivial thing.
- Much more important: Skill in designing with objects.
  - Learning UML notation does not help
- The UML is **not**
  - A process or methodology
  - Object-Oriented analysis and design
  - Guidelines for design

Unified Modeling Language 2

- **Three Ways to Apply UML**
  - **UML as sketch**—Informal and incomplete diagrams (often hand sketched on whiteboards) created to explore difficult parts of the problem or solution space, exploiting the power of visual languages.
  - **UML as blueprint**
    - Relatively detailed design diagrams used either for
      - reverse engineering to visualize and better understanding existing code in UML diagrams,
      - code generation (forward engineering). If reverse engineering, a UML tool reads the source or binaries and generates (typically) UML package, class, and sequence diagrams.
    - Help the reader understand the big picture elements, structure, and collaborations.
    - Before programming, some detailed diagrams can provide guidance for code generation (e.g., Java), either manually or automatically with a tool.
Unified Modeling Language

- **UML as programming language**
  - Complete executable specification of a software system in UML. Executable code will be automatically generated, but is not normally seen or modified by developers; one works only in the UML “programming language.” This use of UML requires a practical way to diagram all behavior or logic (probably using interaction or state diagrams), and is still under development in terms of theory, tool robustness and usability.

- **UML and “Silver Bullet” Thinking**
  - Tool don’t compensate for design ignorance
  - A person not having good OO design and programming skills who draws UML is just drawing bad design

---

Unified Modeling Language

- **Three Perspectives to Apply UML**
  - **Conceptual perspective**—the diagrams are interpreted as describing things in a situation of the real world or domain of interest.
  - **Specification (software) perspective**—the diagrams (using the same notation as in the conceptual perspective) describe software abstractions or components with specifications and interfaces, but no commitment to a particular implementation (e.g., not C# or Java).
  - **Implementation (software) perspective**—the diagrams describe software implementations in a particular technology (such as Java).
Unified Modeling Language 4

Diagram:

- DiceGame
  - die1 : Die
  - die2 : Die
  - play()

- Die
  - faceValue : int
  - getFaceValue() : int
  - roll()

Conceptual Perspective (domain model)
Raw UML class diagram notation used to visualize real-world concepts.

Specification or Implementation Perspective (design class diagram)
Raw UML class diagram notation used to visualize software elements.

Unified Modeling Language 5

- Class-related terms consistent with the UML and the UP,
  - Conceptual class—real-world concept or thing. A conceptual or essential perspective. The UP Domain Model contains conceptual classes.
  - Software class—a class representing a specification or implementation perspective of a software component, regardless of the process or method.
  - Implementation class—a class implemented in a specific OO language such as Java

- UML history
  - Started in 1994 by Booch and Rumbaugh (Unified Method)
  - Joined at Rational by Ivar Jacobson, became industry standard in 1997 (OMG UML 1.0)
  - UML 2.0 in the end of 2004
Chapter 2
Iterative, Evolutionary, and Agile

Iterative and Evolutionary Software Process

- **Software process**
  - Describes an approach to building, deploying, and possibly maintaining software

- **Iterative and evolutionary development**
  - Involves early programming and testing of a partial system, in repeating cycles
  - Development starts before all the requirements are defined in detail
  - Feedback is used to clarify and improve the evolving specifications
Unified Process 1

- Unified process (UP) is a relatively popular iterative process for projects using OOA/D
- Rational unified process (or RUP)
  - A detailed refinement of the unified process that has been widely accepted
- Unified process
  - Very flexible and open
  - Encourages including skillful practices from other iterative methods, such as from extreme programming (XP) and scrum

Unified Process 2

- The Unified Process (UP) represents a mainstream approach for software development across the spectrum of project scales.
- The process is scalable: you need not use the entire framework of the process for every project, only those that are effective.
- The process is effective: it has been successfully employed on a large population of projects.
- Improves productivity through use of practical methods that you’ve probably used already (but didn’t know it).
- Iterative and incremental approach allows start of work with incomplete, imperfect knowledge.
Unified Process Workflows

- Workflows define a set of activities that are performed
- Workflows cut across the phases, but with different levels of emphasis in each phase
- The core workflows
  - Business Modeling
  - Requirements analysis
  - Design
  - Implementation
  - Test and Integration

The Core Workflows

- Business Modeling
  - Develop and refine the problem definition
  - Identify stakeholder needs
  - Define system features to be considered
  - Define the system scope
  - Build the use-case model

- Requirements Analysis
  - Refine use-case model
  - Define the domain model
  - Define a candidate architecture (transitions to design)
  - Refine the architecture (transitions to design)
The Core Workflows

- **Design**
  - Design the physical realizations of the use cases
  - Develop the design model
  - Develop the deployment model

- **Implementation**
  - Plan subsystem implementation
  - Implement components: classes, objects, etc.
  - Perform unit-level testing
  - Perform component and system integration

- **Test and Integration**
  - Build the test model: test cases and expected results
  - Plan, design, implement, execute, and evaluate tests

Use Case Driven

- **Use case**
  - A prose representation of a sequence of actions
  - Actions are performed by one or more **actors** (human or non-human) and the system itself
  - These actions lead to valuable results for one or more of the actors—helping the actors to achieve their goals

- **Use cases are expressed from the perspective of the users, in natural language, and should be understandable by all stakeholders**

- **Use-case-driven** means the development team employs the use cases from requirements gathering through code and test
Architecture Centric

- Software architecture captures decisions about:
  - The overall structure of the software system
  - The structural elements of the system and their interfaces
  - The collaborations among these structural elements and their expected behavior

- Architecture-centric: software architecture provides the central point around which all other development evolves
  - Provides a ‘big picture’ of the system
  - Provides an organizational framework for development, evolving the system by attending to modifiability qualities of the system
  - Guides the prioritization and choice of use cases for development
  - Facilitates reuse

Iterative and Evolutionary Development

- Iterative and Evolutionary Development
  - A key practice in both the UP and most other modern methods
  - Organized into a series of short, fixed-length mini-projects called iterations
  - Each iteration includes its own requirements analysis, design, implementation, and testing activities
  - Allows start of development with incomplete, imperfect knowledge
  - The iterative lifecycle is based on the successive enlargement and refinement of a system through multiple iterations, with cyclic feedback and adaptation
  - The system grows incrementally over time, iteration by iteration
Iterative and Evolutionary Development

- Iterative and Evolutionary Development
- Requirements
- Design
- Implementation & Test & Integration & More Design
- Final Integration & System Test

Time

3 weeks (for example)

Iterations are fixed in length, or timeboxed.

Feedback from iteration N leads to refinement and adaptation of the requirements and design in iteration N+1.

The system grows incrementally.

How to Handle Change on an Iterative Project

- A key attitude of iterative development:
  - Based on an attitude of embracing change and adaptation as unavoidable and indeed essential drivers; rather than fighting the inevitable change that occurs in software development by trying (unsuccessfully) to fully and correctly specify, freeze, and “sign off” on a frozen requirement set and design before implementation

- UP balances the need
  - On the one hand to agree upon and stabilize a set of requirements
  - On the other hand with the reality of changing requirements, as stakeholders clarify their vision or the marketplace changes
Early iterations are farther from the "true path" of the system. Via feedback and adaptation, the system converges towards the most appropriate requirements and design.

In late iterations, a significant change in requirements is rare, but can occur. Such late changes may give an organization a competitive business advantage.

Iterative feedback and evolution leads towards the desired system. The requirements and design instability lowers over time.

**Iterative and Evolutionary Development 3**

- Benefits to Iterative Development:
  - Less project failure, better productivity, and lower defect rates
  - Early rather than late mitigation of high risks (technical, requirements, objectives, usability, ...)
  - Early visible progress
  - Early feedback, user engagement, and adaptation, leading to a refined system that more closely meets the real needs of the stakeholders
  - Managed complexity; the team is not overwhelmed by "analysis paralysis" or very long and complex steps
  - The learning within an iteration can be methodically used to improve the development process itself

**Iterative and Evolutionary Development 4**
Iterative and Evolutionary Development

• How Long Should an Iteration Be? What is Iteration Timeboxing?
  - Most iterative methods recommend an iteration length between 2 and 6 weeks
  - A key idea is that iterations are timeboxed, or fixed in length
  - If it seems that it will be difficult to meet the deadline, the recommended response is to de-scope
    • Remove tasks or requirements from the iteration, and include them in a future iteration, rather than slip the completion date

What About the Waterfall Lifecycle?

• Research indicates that waterfall was ironically a poor practice for most software projects, rather than a skillful approach?
  - high rates of failure, lower productivity, and higher defect rates (than iterative projects)
  - On average, 45% of the features in waterfall requirements are never used, and early waterfall schedules and estimates vary up to 400% from the final actuals

• Guideline: Don’t Let Waterfall Thinking Invade an Iterative or UP Project
Why is the Waterfall so Failure-Prone?

- A key false assumption underlying many failed software projects
  - The specifications are predictable and stable and can be correctly defined at the start, with low change
  - A typical software project experienced a 25% change in requirements
  - with change rates that go even higher 35% to 50% for large projects

The Need for Feedback and Adaptation

- In complex, changing systems (such as most software projects) feedback and adaptation are key ingredients for success
  - Feedback from early development, programmers trying to read specifications, and client demos to refine the requirements
  - Feedback from tests and developers to refine the design or models
  - Feedback from the progress of the team tackling early features to refine the schedule and estimates
  - Feedback from the client and marketplace to re-prioritize the features to tackle in the next iteration
Iterative and Evolutionary Analysis and Design 1

- Assumes there will ultimately be 20 iterations on the project before delivery:
  - Before iteration-1, hold the first timeboxed requirements workshop, such as exactly 2 days.
  - Business and development people (including the chief architect) are present
  - On the morning of day 1, do high-level requirements analysis, such as identifying just the names of the use cases and features, and key non-functional requirements. The analysis will not be perfect.

Iterative and Evolutionary Analysis and Design 2

- Ask the chief architect and business people to pick 10% from this high-level list (such as 10% of the 30 use case names) that have a blending of these three qualities:
  - 1) Architecturally significant (if implemented, we are forced to design, build, and test the core architecture),
  - 2) High business value (features business really cares about), and
  - 3) High risk (such as “be able to handle 500 concurrent transactions”). Perhaps three use cases are thus identified: UC2, UC11, UC14
- For the remaining 1.5 days, do intensive detailed analysis of the functional and non-functional requirements for these three use cases. When finished, 10% are deeply analyzed, and 90% are only high-level
Before iteration-1, hold an iteration planning meeting in which a subset from UC2, UC11, and UC14 are chosen to design, build, and test within a specified time (for example, four-week timeboxed iteration).

- Note that not all of these three use cases can be built in iteration-1, as they will contain too much work.
- After choosing the specific subset goals, break them down into a set of more detailed iteration tasks, with help from the development team.

Do iteration-1 over three or four weeks

- On the first two days, developers and others do modeling and design work in pairs, sketching UML-ish diagrams at many whiteboards (along with sketching other kinds of models) in a common war room, coached and guided by the chief architect
- Then the developers take off their “modeling hats” and put on their “programming hats.” They start programming, testing, and integrating their work continuously over the remaining weeks, using the modeling sketches as a starting point of inspiration, knowing that the models are partial and often vague
Iterative and Evolutionary Analysis and Design

5

- Much testing occurs: unit, acceptance, load, usability, and so forth
- One week before the end, ask the team if the original iteration goals can be met; if not, de-scope the iteration, putting secondary goals back on the "to do" list
- On Tuesday of the last week there's a code freeze; all code must be checked in, integrated, and tested to create the iteration baseline
- On Wednesday morning, demo the partial system to external stakeholders, to show early visible progress. Feedback is requested

Iterative and Evolutionary Analysis and Design

6

- Do the second requirements workshop near the end of iteration-1, such as on the last Wednesday and Thursday.
  - Review and refine all the material from the last workshop.
  - Then pick another 10% or 15% of the use cases that are architecturally significant and of high business value, and analyze them in detail for one or two days.
  - When finished, perhaps 25% of the use cases and non-functional requirements will be written in detail. They won't be perfect.
Iterative and Evolutionary Analysis and Design

- On Friday morning, hold another iteration planning meeting for the next iteration
- Do iteration-2; similar steps
- Repeat, for four iterations and five requirements workshops,
  - So that at the end of iteration-4, perhaps 80% or 90% of the requirements have been written in detail, but only 10% of the system has been implemented

Iterative and Evolutionary Analysis and Design

- We are perhaps only 20% into the duration of the overall project.
  - In UP terms, this is the end of the elaboration phase.
  - At this point, estimate in detail the effort and time for the refined, high-quality requirements. Because of the significant realistic investigation, feedback, and early programming and testing, the estimates of what can be done and how long it will take are much more reliable
- After this point, requirements workshops are unlikely; the requirements are stabilized – though never completely frozen.
  - Continue in a series of three-week iterations
Iterative and Evolutionary Analysis and Design

Imagine this will ultimately be a 20-iteration project.

In evolutionary iterative development, the requirements evolve over a set of early iterations, through a series of requirements workshops (for example, perhaps 5 iterations and workshops). 90% of the requirements are defined and refined. Nevertheless, only 10% of the software is built.

Kick off meeting: starting iteration goals with the team. 1 hour

Team agile modeling & design, whiteboard sketching. 5 hours

Start coding & testing

Demo & iterative goals of each work

1st check and code freeze for the iteration baseline

1st iteration planning 2 hours

Kick off meeting: planning 1st iteration. 1 hour

Next iteration planning: 2 hours

Week 1

M T W Th F

Week 2

M T W Th F

Week 3

M T W Th F

Week 4

M T W Th F

Week 5

M T W Th F

Week 6

M T W Th F

Week 7

M T W Th F

Week 8

M T W Th F

Week 9

M T W Th F

Week 10

M T W Th F

Week 11

M T W Th F

Week 12

M T W Th F

Week 13

M T W Th F

Week 14

M T W Th F

Week 15

M T W Th F

Week 16

M T W Th F

Week 17

M T W Th F

Week 18

M T W Th F

Week 19

M T W Th F

Week 20

M T W Th F

Week 21

M T W Th F

Week 22

M T W Th F

Week 23

M T W Th F

Week 24

M T W Th F

Week 25

M T W Th F

Week 26

M T W Th F

Week 27

M T W Th F

Week 28

M T W Th F

Week 29

M T W Th F

Week 30

M T W Th F

Week 31

M T W Th F

Week 32

M T W Th F

Week 33

M T W Th F

Week 34

M T W Th F

Week 35

M T W Th F

Week 36

M T W Th F

Week 37

M T W Th F

Week 38

M T W Th F

Week 39

M T W Th F

Week 40

M T W Th F

Week 41

M T W Th F

Week 42

M T W Th F

Week 43

M T W Th F

Week 44

M T W Th F

Week 45

M T W Th F

Week 46

M T W Th F

Week 47

M T W Th F

Week 48

M T W Th F

Week 49

M T W Th F

Week 50

M T W Th F

Week 51

M T W Th F

Week 52

M T W Th F

Risk-Driven and Client-Driven Iterative Planning

- UP encourage a combination of risk-driven and client-driven iterative planning.
  - This means that the goals of the early iterations are chosen to
    - 1) identify and drive down the highest risks, and
    - 2) build visible features that the client cares most about
  - Risk-driven iterative development includes more specifically the practice of architecture-centric iterative development, meaning that early iterations focus on building, testing, and stabilizing the core architecture.
Agile Methods and Attitudes

- Agile development methods
  - Usually apply timeboxed iterative and evolutionary development,
  - Employ adaptive planning,
  - Promote incremental delivery, and
  - Include other values and practices that encourage agility rapid and flexible response to change
  - Example: Extreme Programming (XP), FDD, Scrum, ...

- Any iterative method, including the UP, can be applied in an agile spirit

The Agile Manifesto and Principles

- The Agile Manifesto
  - Individuals and interactions over processes and tools
  - Working software over comprehensive documentation
  - Customer collaboration over contract negotiation
  - Responding to change over following a plan
The Agile Manifesto and Principles

- The Agile Principles
  - Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
  - Welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage.
  - Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale.
  - Business people and developers must work together daily throughout the project.
  - Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
  - The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

- Working software is the primary measure of progress.
- Agile processes promote sustainable development.
- The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity — the art of maximizing the amount of work not done is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.
Agile Modeling 1

- Secret of modeling
  - The purpose of modeling (sketching UML, ...) is primarily to understand, not to document

- The views of agile modeling
  - Adopting an agile method does not mean avoiding any modeling
  - The purpose of modeling and models is primarily to support understanding and communication, not documentation
  - Don't model or apply the UML to all or most of the software design

Agile Modeling 2

- Use the simplest tool possible. Prefer "low energy" creativity-enhancing simple tools that support rapid input and change. Also, choose tools that support large visual spaces.
- Don't model alone, model in pairs (or triads) at the whiteboard, in the awareness that the purpose of modeling is to discover, understand, and share that understanding
- Create models in parallel
Agile Modeling

- Use "good enough" simple notation while sketching with a pen on whiteboards. Exact UML details aren't important, as long as the modelers understand each other.
- Know that all models will be inaccurate, and the final code or design different sometimes dramatically different than the model
- Developers themselves should do the OO design modeling, for themselves
The Agile Unified Process

- The Unified Process has been designed from the outset as:
  - Lightweight: ‘Pay as you go.’ Use only the parts that are essential and effective for your project. When in doubt, leave it out.
  - Non-predictive: Requirements and design build gradually as development proceeds rather than being completed before any work can begin.
  - Adaptable: Planning and risk analysis/assessment are on-going and process can be adapted accordingly.

How to Apply UP in Agile Sprit

- Prefer a small set of UP activities and artifacts
  - All UP artifacts are optional, and avoid creating them unless they add value
- Apply the UML with agile modeling practices
- Requirements and designs are not completed before implementation. They adaptively emerge through a series of iterations, based on feedback
- There isn’t a detailed plan for the entire project.
  - There is a high-level plan (called the Phase Plan) that estimates the project end date and other major milestones, but it does not detail the fine-grained steps to those milestones.
  - A detailed plan (called the Iteration Plan) only plans with greater detail one iteration in advance. Detailed planning is done adaptively from iteration to iteration.
Other Critical UP Practices

- Tackle high-risk and high-value issues in early iterations
- Continuously engage users for evaluation, feedback, and requirements
- Build a cohesive, core architecture in early iterations
- Continuously verify quality; test early, often, and realistically
- Apply use cases where appropriate
- Do some visual modeling (with the UML)
- Carefully manage requirements
- Practice change request and configuration management

What are the UP Phases?

- A UP project organizes the work and iterations across four major phases
  - **Inception** — approximate vision, business case, scope, vague estimates
    - Not a requirement phase; rather, it is a feasibility phase
  - **Elaboration** — refined vision, iterative implementation of the core architecture, resolution of high risks, identification of most requirements and scope, more realistic estimates
    - Not the requirements or design phase; rather, it is a phase where the core architecture is iteratively implemented, and high-risk issues are mitigated
  - **Construction** — iterative implementation of the remaining lower risk and easier elements, and preparation for deployment
  - **Transition** — beta tests, deployment
Unified Process Phases

1. **Inception**
   - Feasibility study
   - Business case analysis
   - Scope definition
   - Estimation
   - Risk assessment
   - Candidate architecture

2. **Elaboration**
   - Functional Requirements
   - Use cases
   - Domain model
   - Address Risks
   - Plan/Plan
   - Design model
   - Architectural baseline

3. **Construction**
   - New code
   - Refactoring
   - Added/used use cases
   - Plan/Plan

4. **Transition**
   - Optimization
   - Bug fix
   - Production release

5. **Release**
   - A stable executable subset of the final product. The end of each iteration is a minor release.
   - The difference (delta) between the releases of 2 subsequent iterations.

6. **Final Production Release**
   - At this point, the system is released for production use.
The UP Disciplines

- The UP disciplines is a set of activities (and related artifacts) in one subject area, such as the activities within requirements analysis.
  - An artifact is the general term for any work product: code, Web graphics, database schema, text documents, diagrams, models, and so on.
- There are several disciplines in the UP; this book focuses on some artifacts in the following three:
  - Business Modeling — The Domain Model artifact, to visualize noteworthy concepts in the application domain.
  - Requirements — The Use-Case Model and Supplementary Specification artifacts to capture functional and non-functional requirements.
  - Design — The Design Model artifact, to design the software objects.

UP Disciplines and Phases

A four-week iteration (for example), a mini-project that includes work in most disciplines, ending in a stable executable.

Note that although an iteration includes work in most disciplines, the relative effort and emphasis change over time. This example is suggestive, not literal.
Changing Relative Effort With Respect to the Phases

Early iterations naturally tend to apply greater relative emphasis to requirements and design, and later ones less so.

The relative effort in disciplines shifts across the phases.

This example is suggestive, not literal.

Book Structure Influenced by UP Phases and Disciplines

Topics such as OO analysis and OO design are incrementally introduced in Iteration 1, 2, and 3.
Customize the UP Process

- A key insight into the UP is that all activities and artifacts (models, diagrams, documents, ...) are optional
- The choice of practices and UP artifacts for a project may be written up in a short document called the Development Case (an artifact in the Environment discipline).

Sample Development Case

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Modeling</td>
<td>agile modeling req. workshop</td>
<td>Domain Model</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requirements</td>
<td>req. workshop vision box</td>
<td>Use-Case Model</td>
<td>s</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>exercise dot voting</td>
<td>Vision</td>
<td>s</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Supplementary</td>
<td>s</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Specification</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Glossary</td>
<td>s</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design</td>
<td>agile modeling test-driven dev.</td>
<td>Design Model</td>
<td>s</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SW Architecture</td>
<td>s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Document</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data Model</td>
<td>s</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td>test-driven dev. pair programming</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>continuous integration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>coding standards</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project Management</td>
<td>agile PM daily Scrum meeting</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

s — start; r — refine
The Case Study Focus

- Generally, applications include UI elements, core application logic, database access, and collaboration with external software or hardware components.
- Although OO technology can be applied at all levels, the focus of OOA/D is on the core application logic layer since:
  - Other layers are usually very technology/platform dependent.
  - The OO design of the core logic layer is similar across technologies.
  - The essential OO design skills learned in the context of the application logic layer are applicable to all other layers or components.
  - The design approach/patterns for the other layers tend to change quickly as new frameworks or technologies emerge.
The Case Study Focus

- User Interface
- Application logic layer
- Other layers or components

- minor focus: explore how to connect to other layers
- primary focus: explore how to design objects
- secondary focus

Case Study Strategy

- Case Study Strategy
  - Iterative Development + Iterative Learning
- Learning path follows iterations

Iteration 1
Introduces just those analysis and design skills related to iteration one.

Iteration 2
Additional analysis and design skills introduced.

Iteration 3
Likewise.
Case One: The NextGen POS System

1. The Point-of-Sale terminal is a computerized system used to record sales and handle payments; it is typically used in a retail store. It includes hardware components such as a computer and bar code scanner, and software to run the system.

Case One: The NextGen POS System

2. It interfaces to various service applications, such as a third-party tax calculator and inventory control. These systems must be relatively fault-tolerant; that is, even if remote services are temporarily unavailable (such as the inventory system), they must still be capable of capturing sales and handling at least cash payments (so that the business is not crippled).

A POS system increasingly must support multiple and varied client-side terminals and interfaces. These include a thin-client Web browser terminal, a regular personal computer with something like a Java Swing graphical user interface, touch screen input, wireless PDAs, and so forth.
Case One: The NextGen POS System

- Furthermore, we are creating a commercial POS system that we will sell to different clients with disparate needs in terms of business rule processing. Each client will desire a unique set of logic to execute at certain predictable points in scenarios of using the system, such as when a new sale is initiated or when a new line item is added.
- Therefore, we will need a mechanism to provide this flexibility and customization. Using an iterative development strategy, we are going to proceed through requirements, object-oriented analysis, design, and implementation.

Case Two: The Monopoly Game System

- The software version of the game Monopoly will run as a simulation. One person will start the game and indicate the number of simulated players, and then watch while the game runs to completion, presenting a trace of the activity during the simulated player turns.