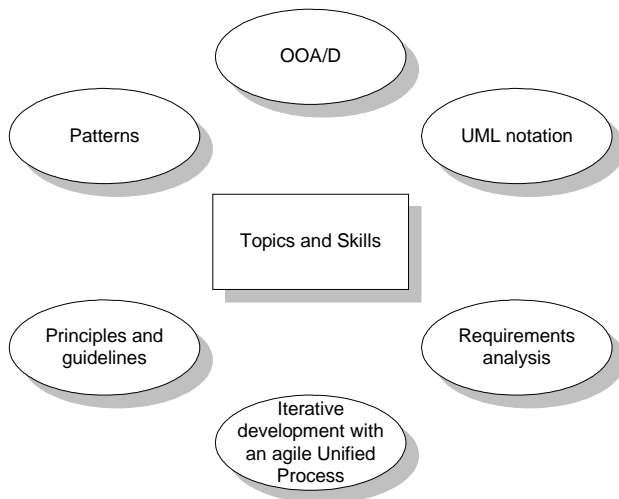


# Chapter 1



## Important learning goal

- A critical ability in OO development is to skillfully assign responsibilities to software object.

## Assign responsibilities to software object. Why?

- Strongly influences the robustness, maintainability and reusability of software components.
- **GRASP** → emphasize principles of responsibility assignment. Such as -> Information Expert and Creator

**Analysis** → emphasizes an investigation of the problem and requirements rather than a solution. It emphasizes the question “what” rather than “how”. Analysis is abroad term. In **requirements analysis**, it means an investigation of the requirements. In **object oriented analysis** it means an investigation of the domain objects.

**Design** → emphasizes a conceptual solution (in hardware or software) that fulfills the requirements rather than its implementation. It emphasizes the question “how” the system will work. Ultimately, designs can be implemented, and the implementation (hardware or software) expresses complete and true realized design. e.g. a description of a database schema and software object.

Analysis → **Do the right things**  
 Design → **Do the things right**

**OO** → a paradigm that uses objects and their interactions to design a system.

**OO Analysis** → there is an emphasis in finding and describing the objects or concepts in the problem domain. e.g. Flight IS has concepts such as plane, flight and pilot.

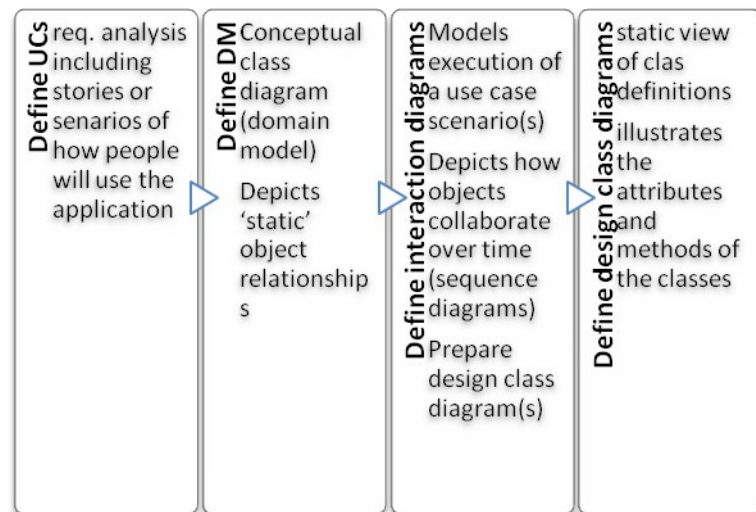
**OO Design** → or simply object design → there is an emphasis on defining software objects and how they collaborate to fulfill the requirements. e.g. Plane software object may have tailNo att. And getFlightHistory method. During implementation or object oriented programming, design objects are implemented, such as Plane class in Java.

**OOA/D** → Object oriented Analysis and Design → for creation of well-designed, robust, maintainable software using OO technologies and languages such as Java or C#.

**OO Programming** → A programming language that supports the concepts of encapsulation, inheritance and polymorphism.

**Pattern** → a named description of a problem, solution, when to apply the solution, and how to apply the solution in new context.

## Basic SW Process



**UML** → the Unified Modeling Language is a visual language for specifying, constructing and documenting the artifacts of systems.

- [1] UML is just diagramming notation
- [2] UML is not OOA/D or a method
- [3] UML is de facto standard diagramming notation for drawing or presenting pictures (primarily OO software)

UML define various **UML profiles** that specialize subset of the notation for common subject areas.

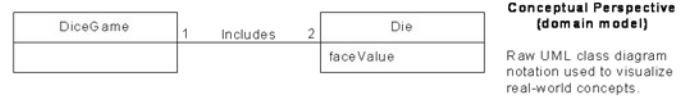
Model Driven Architecture (MDA) → underlying the UML notation is the **UML meta-model** that describes the semantics of the modeling elements.

### 3 ways to use UML

- **Sketch**
  - Informal and incomplete diagrams
  - Very quick, requires no fancy tools
  - Preferred by Agile modelers
  - Often good enough to explore difficult parts of the problem or solution space, exploiting the power of visual languages.
- **Blueprint**
  - Detailed model drawn by tool
  - Used either for reverse engineering to visualize and better understanding existing code in UML diagrams
  - Or for code generation
- **Executable programming language**
  - Prepare complete suite of models for system
  - Compile models automatically into executable code but normally seen or modified by developers (Not ready for ‘primetime’)
  - Eliminates costly coding activity

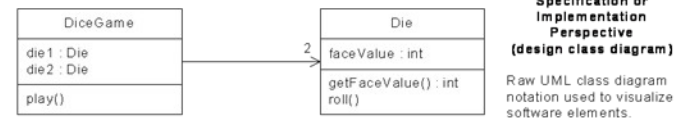
### 3 perspectives to apply UML

#### Conceptual class



#### 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.

#### SW or Implementation class

Implementation perspective uses Language-specific syntax

- **Conceptual Perspective**
  - Diagrams are interpreted as describing things in a situation of the real world or domain of interest.
- **Specification (SW) Perspective**
  - Diagrams, using the same notation as in the conceptual perspective, describe sw abstractions or components with specifications and interfaces , but no commitment to a particular implementation such as Java or C#.
- **Implementation (SW) Perspective**
  - Diagrams describe sw implementations in a particular technology such as Java.

### The meaning of class in different perspectives

- **Domain Concept or 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 sw component, regardless of the process or method. UML **class diagram** contains sw classes.
- **Implementation Class** → a class implemented in a specific OO language such as Java. UML **class diagram** contains implementation classes.

## Chapter 2

**SW development process** → describes an approach to building, deploying and possibly maintaining software.

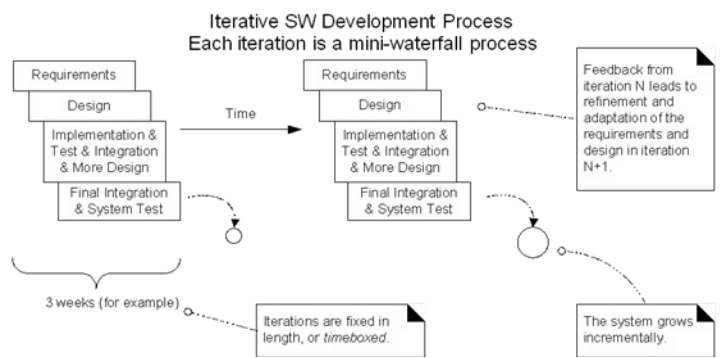
**Iterative development** → contrasted with a sequential or “waterfall” lifecycle → involves early programming and testing of a partial system in repeating cycles. It also normally assumes development starts before all the requirements are defined in detail; feedback is used to clarify and improve the evolving specifications.

In this lifecycle **UP** approach, development is organized into a series of short, fixed-length mini-projects called **iterations**. The outcome of each iteration is a tested, integrated and executable partial system. Each iteration include its own requirements analysis, design, implementation and testing activities.

Also called:

**(1) Iterative and evolutionary development** → because feedback and adaptation evolve the specification and design after each iteration.

**(2) Iterative and incremental development** → because of successive enlargement and refinement of a system through multiple iterations.



### Key benefits of iterative process

1. Higher project success rate
  - [1] Better management of complexity
  - [2] Early mitigation of high risks (technical, requirements, objectives, usability..etc)
  - [3] Easier adaptation to changing reqs.
2. Higher productivity & lower defect rate
3. Client visibility into project status.. early feedback, user engagement and adaptation.
4. Early lessons learned applied to later iterations

### How long an iteration be? Iteration Timeboxing..

Recommended length between 2 and 6 weeks. Small steps, rapid feedback and adaptation are central ideas in iterative development. Long iterations subvert the core motivation for iterative development and increase risks.

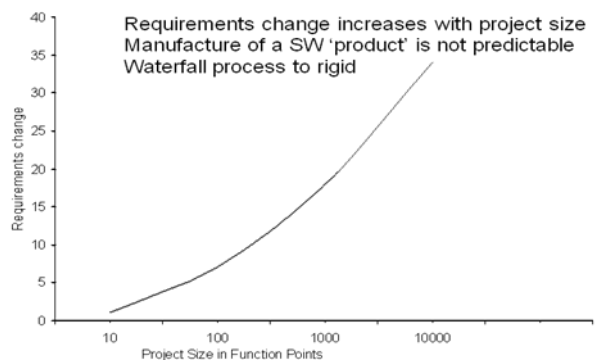
Iterations should be timeboxed, or fixed in length. Partial system must be integrated, tested and stabilized by the scheduled date. Extending the date is illegal.

**Waterfall Lifecycle** → or **sequential lifecycle process** → attempts to define in detail all or most of the requirements before programming. And often, to create a thorough design or set of models before programming.

It promoted big upfront speculative requirements and design steps before programming. It emerges highest failure rates due to belief or hearsay rather than statistically significant evidence of the defined requirements, analysis and designs.

### Why is the waterfall so Failure-Prone?

This is strongly related to a key false assumption that the specifications are predictable and stable, and can be correctly defined at the start of the project with low change rates.



However, sw development is on average a domain of high change and instability → known as the domain of **new product development**.

On the other hand, iterative and evolutionary methods assume and embrace change and adaptation of partial and evolving specifications, models and plans based on feedback.

### The need for feedback and adaptation

In complex, changing systems (such as most sw projects) feedback and adaptation are key ingredients for success.

- [1] Feedback from (i) early development, (ii) programmers trying to read specifications and (iii) client demos → to refine the requirements

- [2] Feedback from (i) test and (ii) developers → to refine the design or models
- [3] Feedback from the (i) progress of the team tackling early features → to refine the schedule and estimates
- [4] Feedback from the (i) client and (ii) marketplace → to re-prioritize the features to tackle in the next iteration

**Risk-Driven & Client-Driven Iterative Planning** → The UP encourage a combination of them. So, the goal 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.

**Architecture-centric iterative development** → the early iterations in risk-driven iterative development focus on building, testing and stabilizing the core architecture.. Why? Because not having a solid architecture is a common high risk.

**Agile** (flexible, light) approach to the well-known **Unified Process(UP)** → is used as the sample iterative development process.

For analysis and design, agile UP validate their applicability to others methods, Scrum, Feature Driven Development, Lean Development, Crystal Method and so on.

**Agile modeling** → emphasizes UML as a sketch; this is a common way to apply UML, often with a high return on the investment of time. The purpose of modeling is primarily to understand not to document.

**It implies a number of practices and values including:**

- [1] Adopting and agile method does not mean avoiding any modeling. Many agile methods such as feature-driven development, DSDM and Scrum normally include significant modeling sessions.
- [2] The purpose of modeling is to support understanding and communication, not documentation .
- [3] Don't model or apply UML to all or most of the sw design. Defer simple or straightforward design problems until programming. Solve them during programming and testing.
- [4] Use the simplest tool possible.

- [5] 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.
- [6] Create models in parallel.
- [7] Use "good enough" simple notation. Exact UML details are not important.
- [8] Know that all models are inaccurate and the final design will be different.

**Agile development** → apply timeboxed iterative and evolutionary development, employ adaptive planning, promote incremental delivery, and include other values and practices that encourage agility – rapid and flexible respond to change.

**Agile methods** → no specific definition for specific practices. However, short, timeboxed iterations with evolutionary refinement of plans, requirements and designs is the best practice the methods share. In addition, they promote practices and principles that reflect an agile sensibility of simplicity, lightness, communication, self-organizing teams and more.

**Agile manifesto** →

Individuals and interactions	over processes and tools
Working SW	over comprehensive documentation
Customer collaboration	over contract negotiation
Responding to change	over following a plan

**Agile Principles** →

- [1] The highest priority is to satisfy the customer reqs.
- [2] Welcome changing reqs even in late development.
- [3] Deliver working sw frequently with a preference to the shorter time scale.
- [4] Business people and developers must work together daily throughout the project.
- [5] Build projects around motivated individuals.

- [6] The most efficient and effective method of conveying info. to and within a development team is face-2-face conversation.
- [7] Working sw is the primary measure of progress.
- [8] Agile processes promote sustainable development.
- [9] Sponsors, developers and users should be able to maintain a constant pace indefinitely.
- [10] Continuous attention to technical excellence and good design enhance agility.
- [11] Simplicity is essential.
- [12] Best architectures, requirements and designs emerge from self-organizing teams.
- [13] At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

**UP → Unified Process →** has emerged as a popular a popular iterative sw development process for building OO systems.

- is very flexible and open,
- encourages including skilful practices from other iterative methods., - such as from Extreme Programming (XP) & Scrum
- UP combines commonly accepted best practices, such as iterative life cycle and risk driven development, into a cohesive and well-documented process description.

**RUP → Rational Unified Process →** a detailed refinement of the UP that has been widely adopted.

**Agile UP →** The UP was not meant by its creators to be heavy or un-agile. Rather, it was meant to be adopted and applied in the spirit of adaptability and lightness.

Examples of how this applies:

- [1] Prefer a small set of UP activities and artifacts and in general keep it simple. Remember that all UP artifacts are optional, and avoid creating them unless they add value. Focus on early programming not early documenting.
- [2] Since the UP is iterative and evolutionary, requirements and designs are not completed before implementation.

They adaptively emerge through a series of iterations based on feedback.

- [3] Apply UML with agile modeling practices.
- [4] There isn't a detailed plan for the entire project. Detailed plan is done adaptively from iteration to iteration.

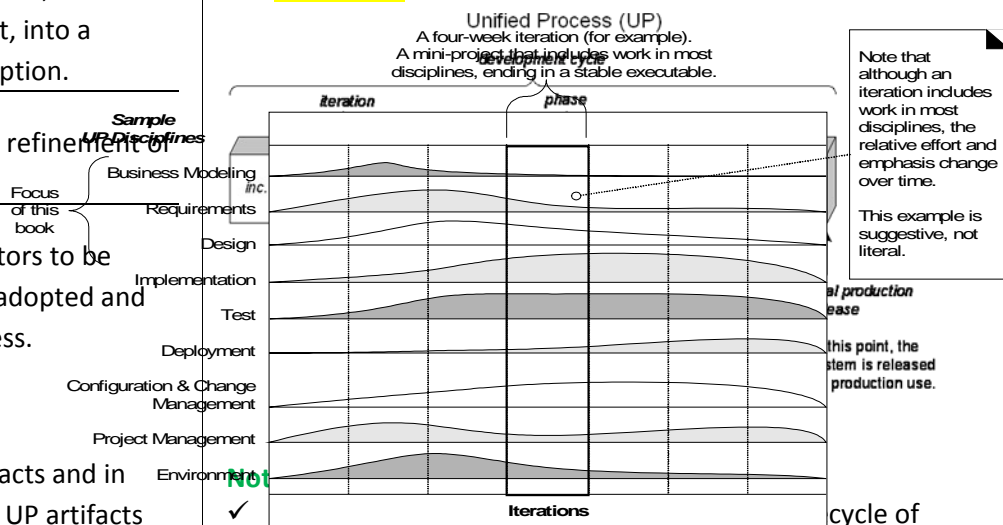
**Important UP practices →**

- [1] Tackle high-risk and high-value in early iterations.
- [2] Continuously engage users for evaluation, feedback and requirements.
- [3] Build core architecture in early iterations.
- [4] Verify quality by testing throughout.
- [5] Focus on essential models using UML
- [6] Manage reqs. with use cases
- [7] Practice change request & configuration management.

**UP phases →**

A UP project organizes the work and iterations across 4 major phases:

- [1] **Inception:** approximate vision, business case, scope, vague estimates.
- [2] **Elaboration:** refined vision, iterative implementation of the core architecture, resolution of high risks, identification of most requirements and scope, more realistic estimates.
- [3] **Construction:** iterative implementation of the remaining lower risk and easier elements and preparation for deployment.
- [4] **Transition:** beta tests, deployment



- ✓ Not cycle of first defining all requirements, and then doing most or all of the design.
- ✓ Inception is not a requirements phase; it is a feasibility phase, where just enough investigation is done to support a decision to continue or stop.

- ✓ Similarly, elaboration, is not the requirements or design phase; it is a phase where the core architecture is iteratively implemented, and high-risk issues are mitigated.

**Discipline** → a set of activities and related artifacts in one subject area, such as the activities within requirements analysis.

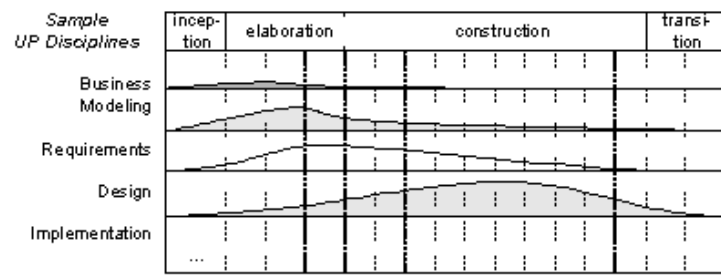
**Artifact** → in UP, is the general term for any work product: code, web graphics, database schema, text docs, diagrams, models, and so on.

**UP disciplines** → UP describes work activities within disciplines. Some UP disciplines include:

- [1] Business Modeling: The Domain Model artifact to visualize noteworthy concepts in the application domain.
- [2] Requirements: The Use-Case Model and Supplementary Specification artifacts to capture functional and non-functional requirements.
- [3] Design: The Design Model artifact to design the software objects.

More!

- A. Implementation discipline in UP means programming building the system not deploying it.
- B. Environment discipline means establishing the tools and customizing the process for the project. i.e. setting up the tool and process environment.



## Chapter 4

**Inception** → An initial step with the following types of questions are explored:

- [1] What is the vision and business case for this project?

- [2] Feasible?
- [3] Buy and/or Build?
- [4] Rough unreliable range of cost?
- [5] Should we proceed or stop?

The **purpose** of the **inception phase** is not to define all the requirements, or to generate believable estimate or project plan. Most of the requirements analysis occurs during the **elaboration phase**.

Inception in 1 sentence:

Envision the product scope, vision and business case

The main problem solved in 1 statement:

Do the stakeholders have basic agreement on the vision of the project, and is it worth investing in serious investigation?

**Inception Artifacts** → a popular iterative sw development process for building OO systems.

Artifact	Comment
Vision and Business Case	Describes the high-level goals and constraints, the business case, and provides an executive summary.
Use-Case Model	Describes the functional requirements. During inception, the names of most use cases will be identified, and perhaps 10% of the use cases will be analyzed in detail.
Supplementary Specification	Describes other requirement, mostly non-functional. During inception, it is useful to have some idea of the key non-functional requirement that have will have a major impact on the architecture.
Glossary	Key domain terminology, and data dictionary.
Risk list & Risk Management Plan	Describes the risks (business, technical, resource, schedules) and ideas for their mitigation or response.
Prototypes and proof-of-concepts	To clarify the vision, and validate technical ideas.
Iteration Plan	Describes what to do in the first elaboration iteration.
Phase Plan & Software Development	Low-precision guess for elaboration phase duration and effort. Tools, people, education, and other resources.

Plan	
Development Case	A description of the customized UP steps and artifacts for this project. In the UP, one always customizes it for the project.

## Chapter 5

**Requirements** → capabilities and conditions to which the system, and more broadly the project, must conform.

The UP promotes a set of best practices and one of them is manage reqs. This does not mean the waterfall attitude of fully define and stabilize the requirements in the first stage of the project before programming, but rather in **UP best practice** is a systematic approach to finding, documenting, organizing and tracking the changing reqs of a system.

### Challenges!

1. Discover
2. Verify
3. Document
4. Prioritize
5. Business value
6. Technical difficulty
7. Revise/update/delete
8. Ensure they're implemented correctly (traceability)

**Deferred/rejected requirements** → Just as important as accepted reqs. We need to document why reqs. deferred/rejected. Otherwise, you'll revisit the same issues. But it is likely to be neglected in many agile projects.

### Interesting stats on reqs gathered in waterfall projects...

- 25% of reqs change during project
- 65% of reqs never/rarely used!.

### Types/Categories of Requirements →

#### FURPS+

- A. **Functional (behaviour)**: ability to perform user's task → features, capabilities, & security
- B. **Non-functional (URPS+) /quality attributes/quality requirements (anything else)**
  - [1] **U**sability → Human factors, help, documentation
  - [2] **R**eliability → Failure rate, recoverability
  - [3] **P**erformance → Responsiveness (response time), throughput, accuracy, availability, & resource usage.

- [4] **S**upportability → Maintainability, configurability, customizability & internationalization.

The "+" in FURPS indicates sub-factors such as:

- [5] Implementation → Languages, tools, hardware
- [6] Interface → Interfaces with 3rd party (external) systems
- [7] Operations → System management & its operating environment
- [8] Packaging → Physical form factor
- [9] Legal → Licensing & regulatory.

### Documenting Reqs →

#### How are reqs organized in UP artifacts?

- Functional reqs go in Use Case Model
- URPS+ usually go in Supplementary Specification
- Glossary may contain validation rules
- Vision may contain high-level reqs
- Business Rules
  - Domain expertise incorporated in system
  - Arguably most valuable company asset
  - Usually cross application boundaries
  - Should be in separate artifact