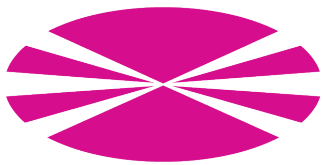
Unit 1: Introduction

Software Design (614G01015)

David Alonso Ríos
Eduardo Mosqueira Rey (Coordinator)

Department of Computer Science and Information Technology
Faculty of Computer Science



UNIVERSIDADE DA CORUÑA

Table of Contents

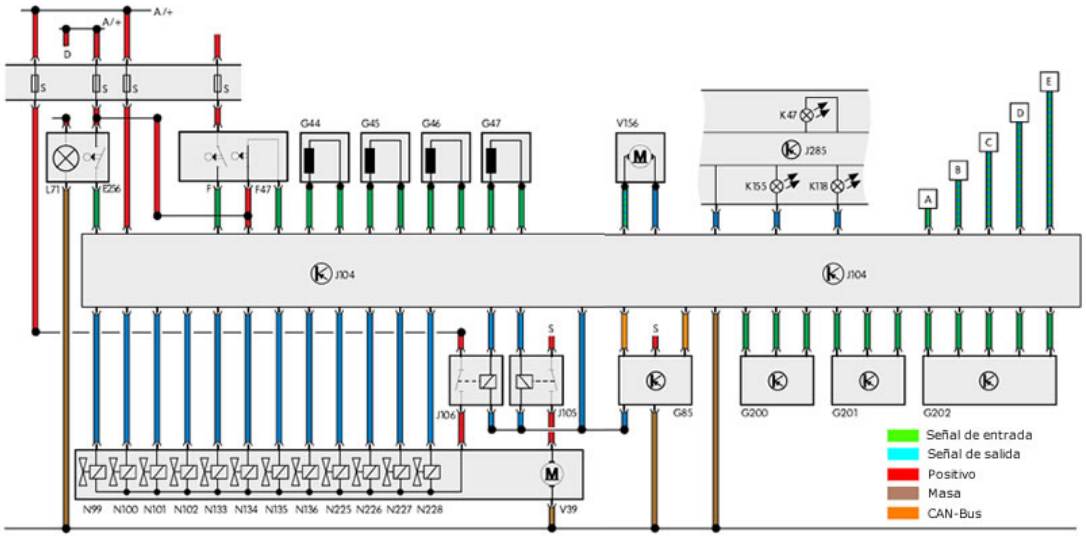
Table of Contents

Modeling

Model

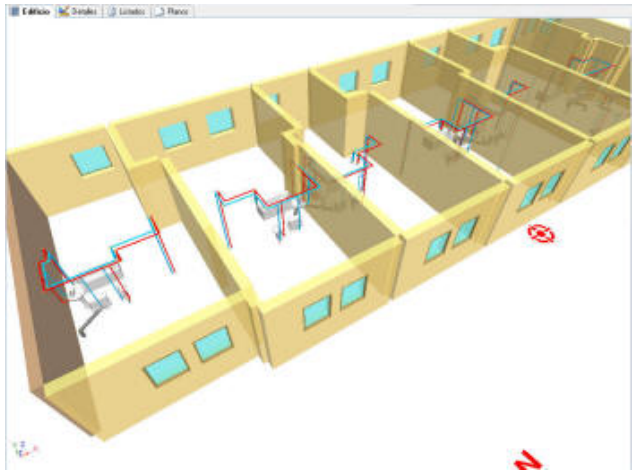
A model is a simplification of reality to better understand the system we are developing.

- Model creation is a common technique in Engineering: Architecture, automobiles, electronic devices, etc.



Modeling

- **Why model?**
  - Systems are too complex to grasp in their entirety.
  - Modeling is a divide-and-conquer technique used to focus on one specific aspect at a time.
  - For example, in architecture, different plans for different purposes (electricity, water, etc.)



Modeling

- **Basic principles of modeling**
  - The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped.
  - Every model may be expressed at different levels of precision.
  - The best models are connected to reality.
  - No single model of view is sufficient. Every nontrivial system is best approached through a small set of nearly independent models with multiple viewpoints.

Analogy with Architecture

- **Building a doghouse**
  - Built by one person.
  - Minimal modeling.
  - Simple process.
  - Simple tools.

Analogy with Architecture

- **Building a house**
  - Built by a team.
  - Elaborate modeling.
  - Well-defined process.
  - Specialized tools.

Analogy with Architecture

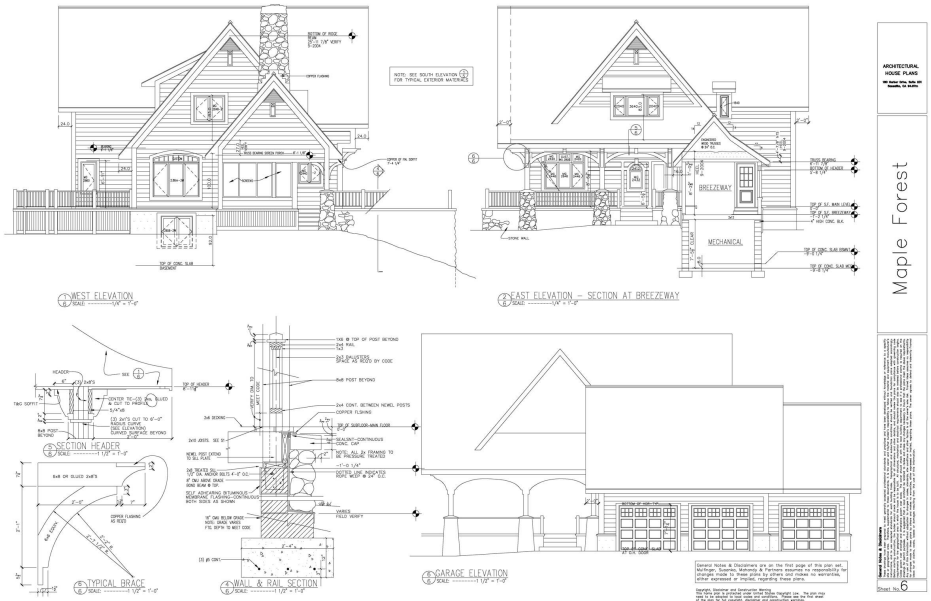
- **Building a skyscraper**
  - Built by multiple teams.
  - Complex modeling.
  - Well-defined process.
  - Sophisticated tools.

Question

If architects adapt their development process to the complexity of their goals,
why are some many skyscraper-sized programs developed as if they were doghouses?

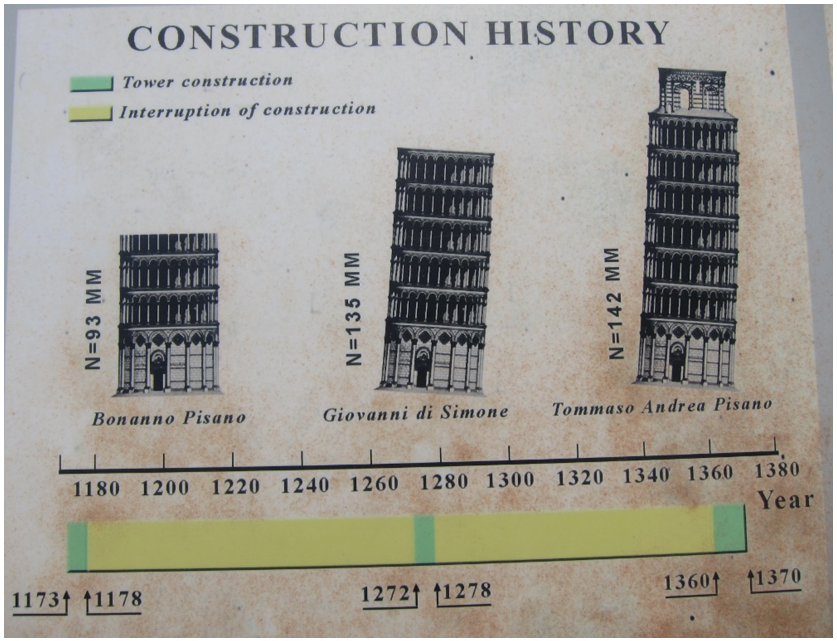
Analogy with Architecture

- **Architecture**
  - Physical limitations prevent future changes.
  - Architects rely on well-known physical laws whose effects can be predicted with accuracy.



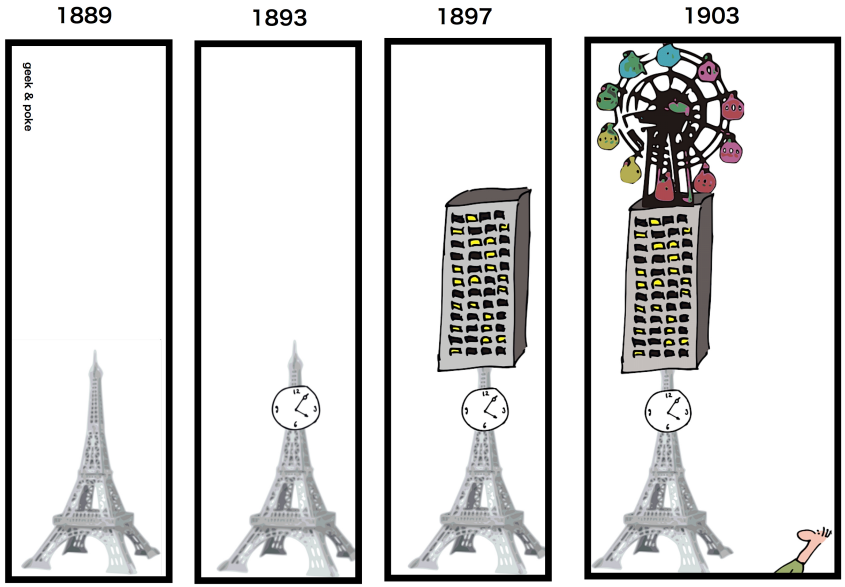
Analogy with Architecture

- But even Architecture has notorious failures: Leaning Tower of Pisa, Tacoma Narrows Bridge, Millennium Bridge, etc.



Analogy with Architecture

- **Software Design**
  - There are no limitations to future changes ⇒ Designs are dynamic and ever-evolving.
  - There are no physical laws whose effects can be modeled or predicted.
  - Interactions between components can be complex and unpredictable.



Thank god not everything is software

Software Engineering

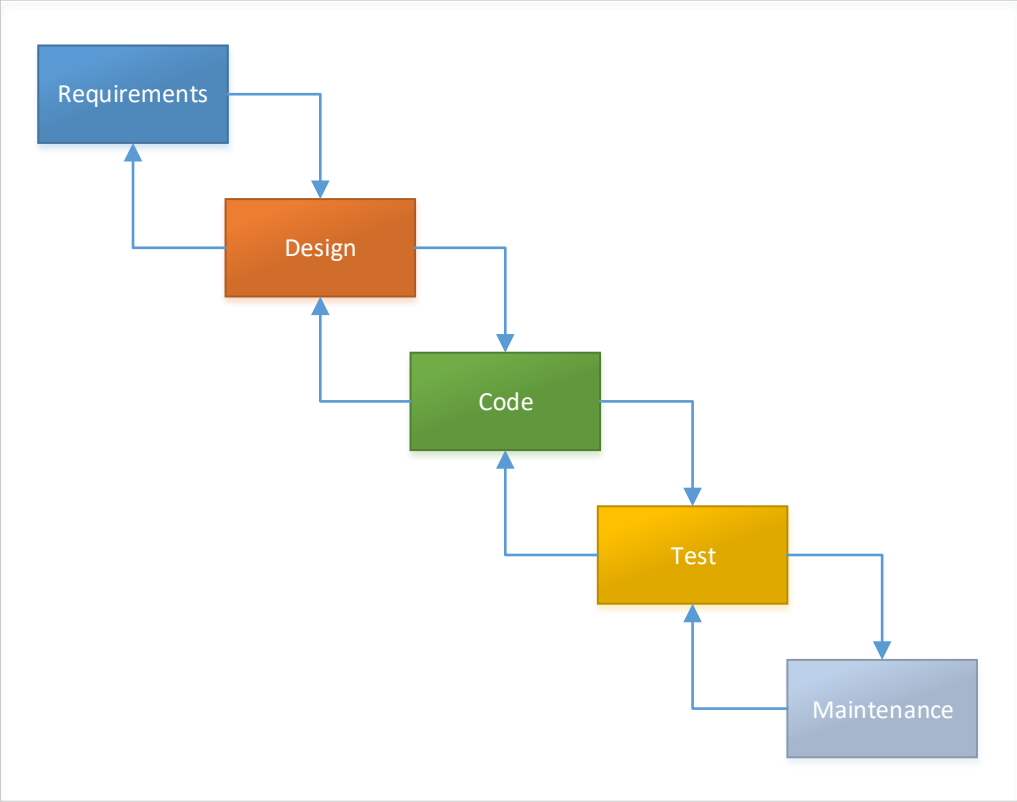
Software Engineering (IEEE)

The systematic design and development of software products and the management of the software process.
That is, the application of Engineering to software.

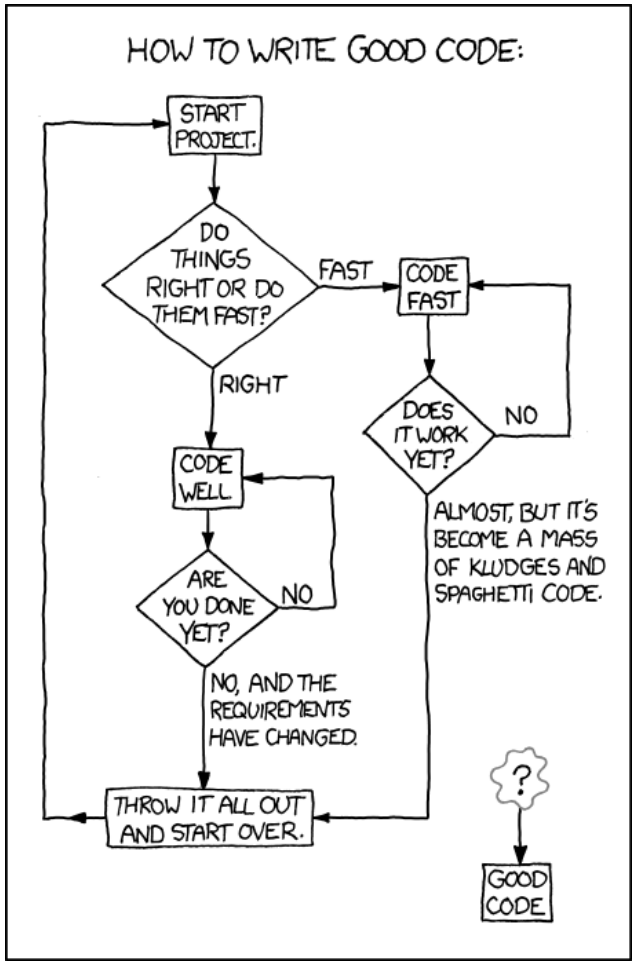
- Software Engineering describes different models of software development.
- These models basically consist in establishing separate phases of the process, with their own goals and tasks: requirements, analysis, design, etc.

Waterfall Model

- Sequential phases.
- A formal phase of requirements analysis before design and coding.



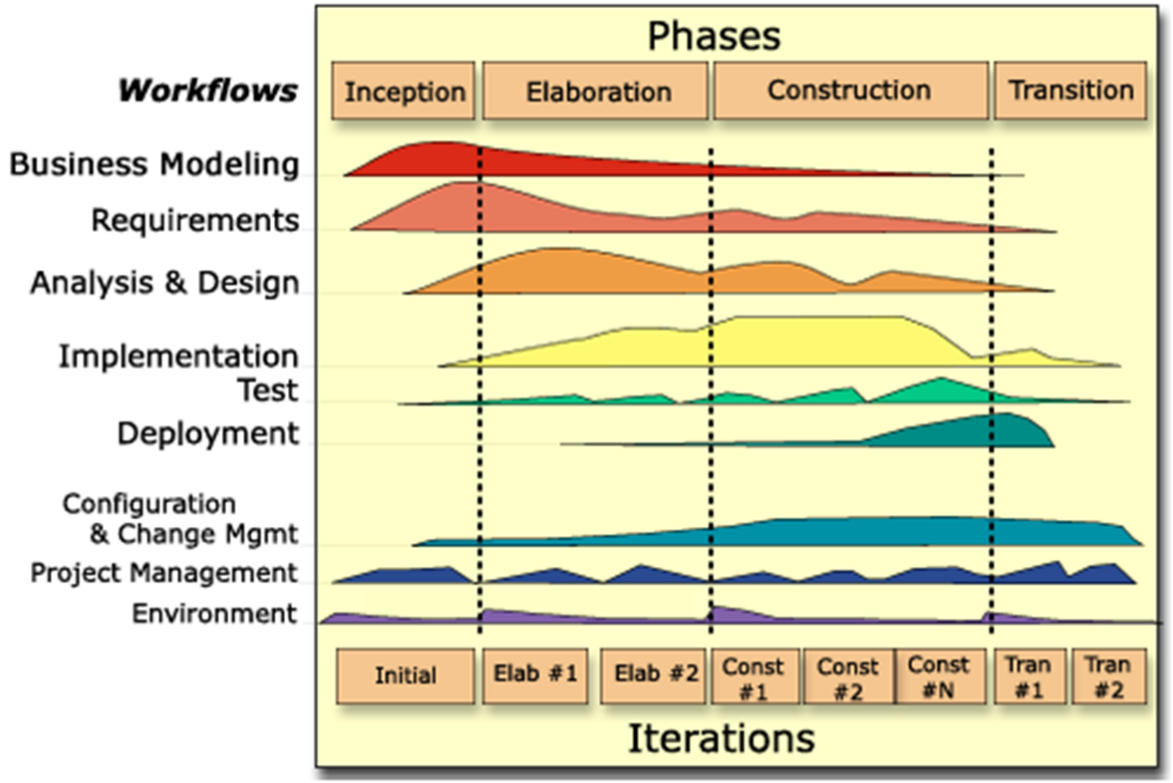
The biggest problem when writing code



fuente: `http://xkcd.com/844/`

Unified Software Development Process (aka Unified Process)

- It follows an iterative and incremental process similar to the spiral model.
- Development is split into phases: Inception, Elaboration, Construction and Transition.
- On each phase one or several iterations are repeated.
- On each iteration, Software Engineering tasks are carried out (e.g., requirements, analysis, design, etc.) with different degrees of effort.



Project Disciplines in Unified Process

- **Business Modeling**
  - The structure and dynamics of the organization are described. Its current problems and possible solutions are also identified.
- **Requirements**
  - Describe what the system must do. Developers and clients must reach a consensus about it.
- **Analysis & Design**
  - Describe the structure and the behavior of the software in light of the specified requirements.
- **Implementation**
  - Classes and objects are built in terms of components (source files, binary files, executables, etc.)
- **Test**
  - Correct functioning is checked according to different aspects: objects as units, object integration, etc.
- **Deployment**
  - The external version of the product is created, distributed and installed in the workplace. User support is also provided.

Design goals

- **Static Model**, which represents the program's structure based on packages and interrelated objects.
- **Dynamic Model**, which represents the program's behavior based on communications between objects.

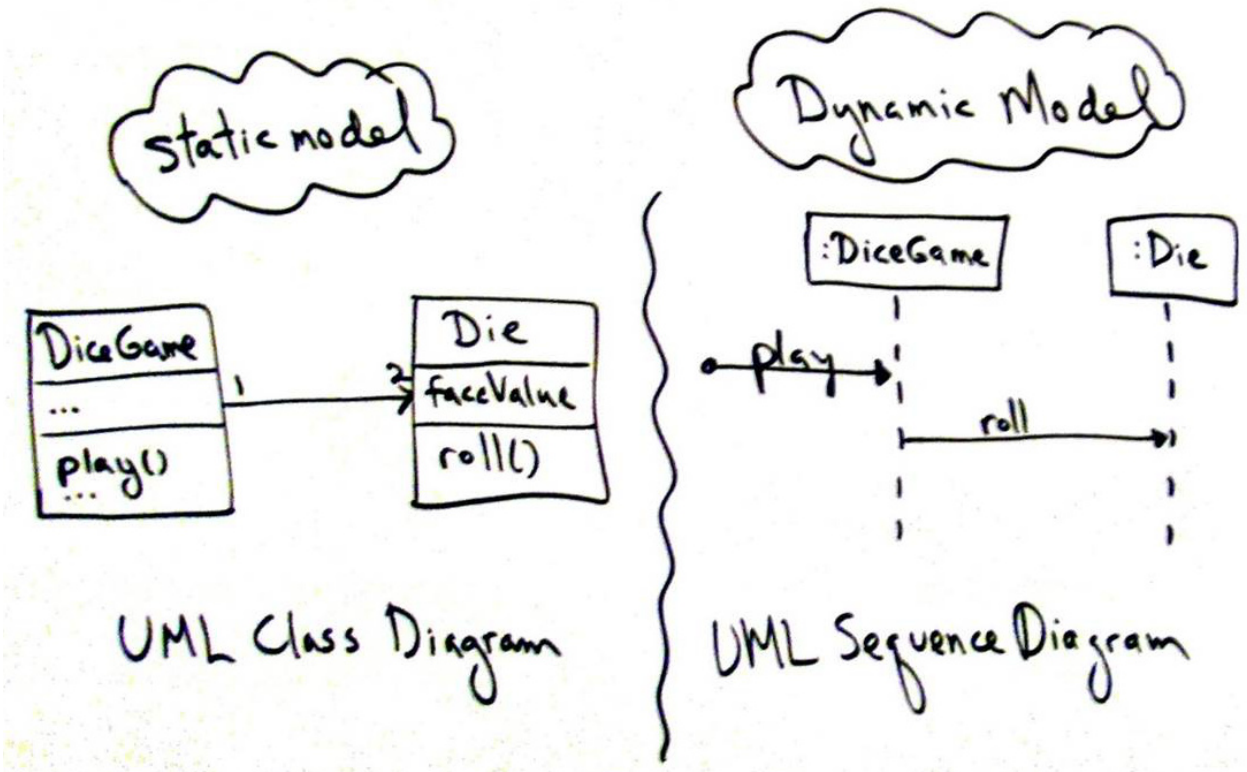
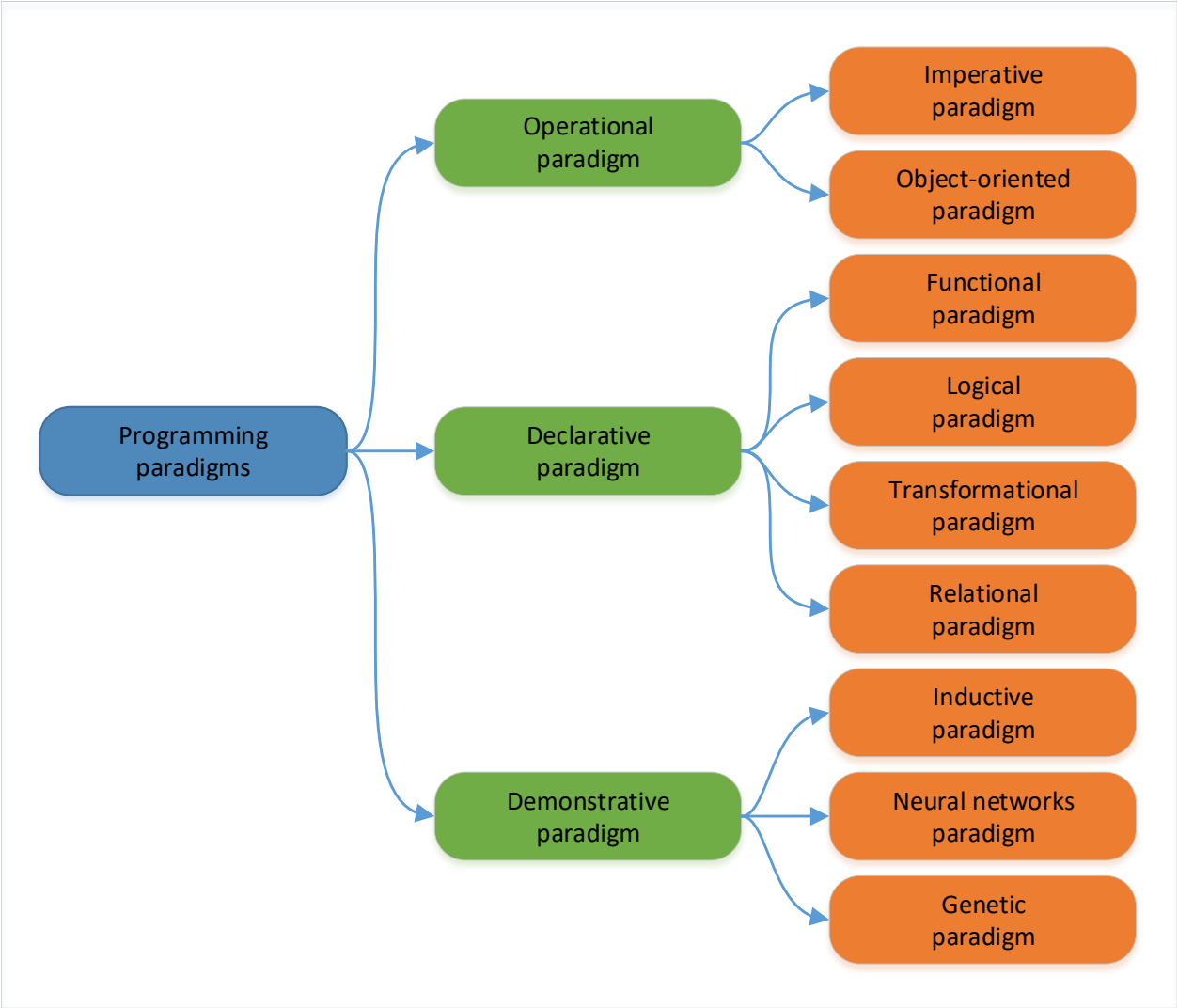
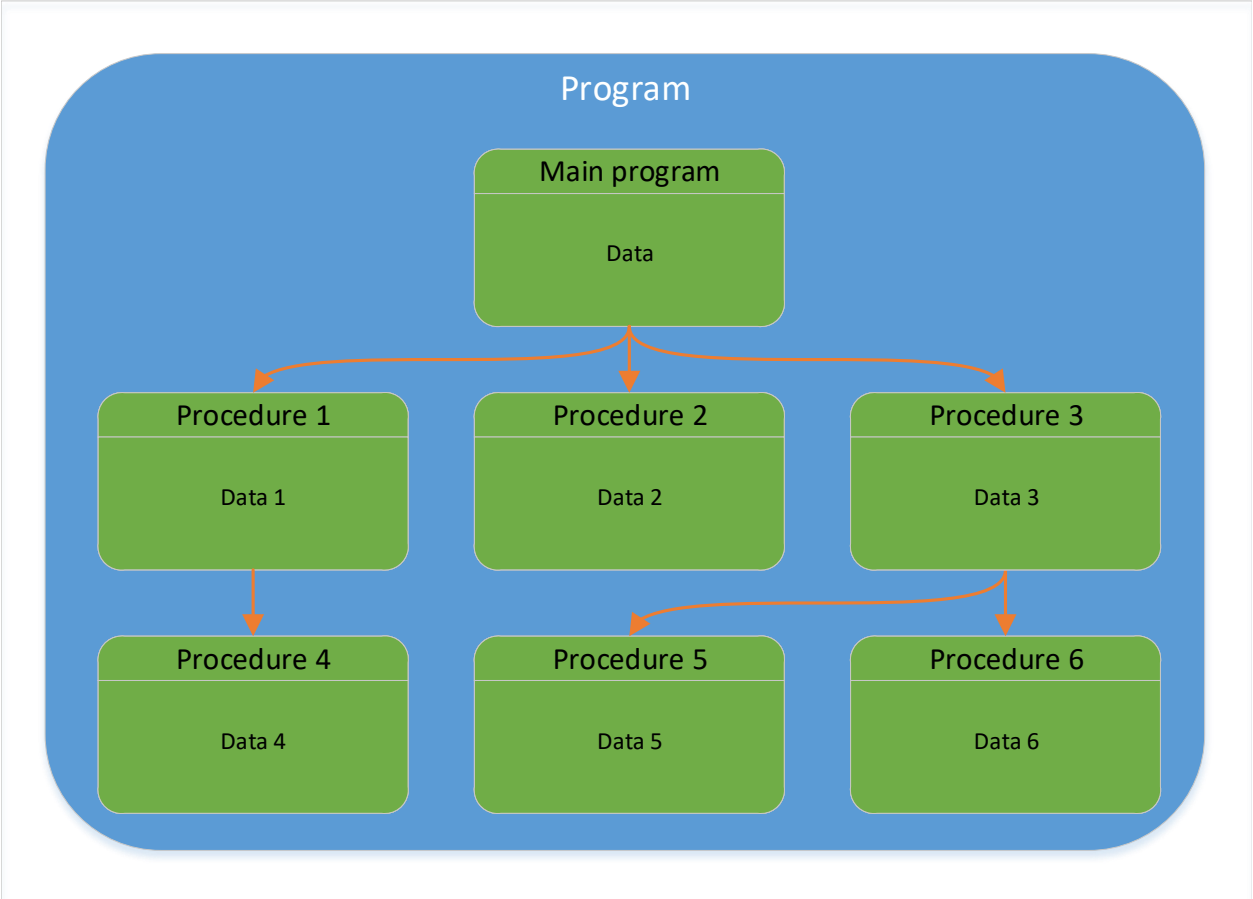


Table of Contents


Programming paradigms
Operational paradigm
Declarative paradigm
Demonstrative paradigm
Imperative paradigm
Object-oriented paradigm
Functional paradigm
Logical paradigm
Transformational paradigm
Relational paradigm
Inductive paradigm
Neural networks paradigm
Genetic paradigm

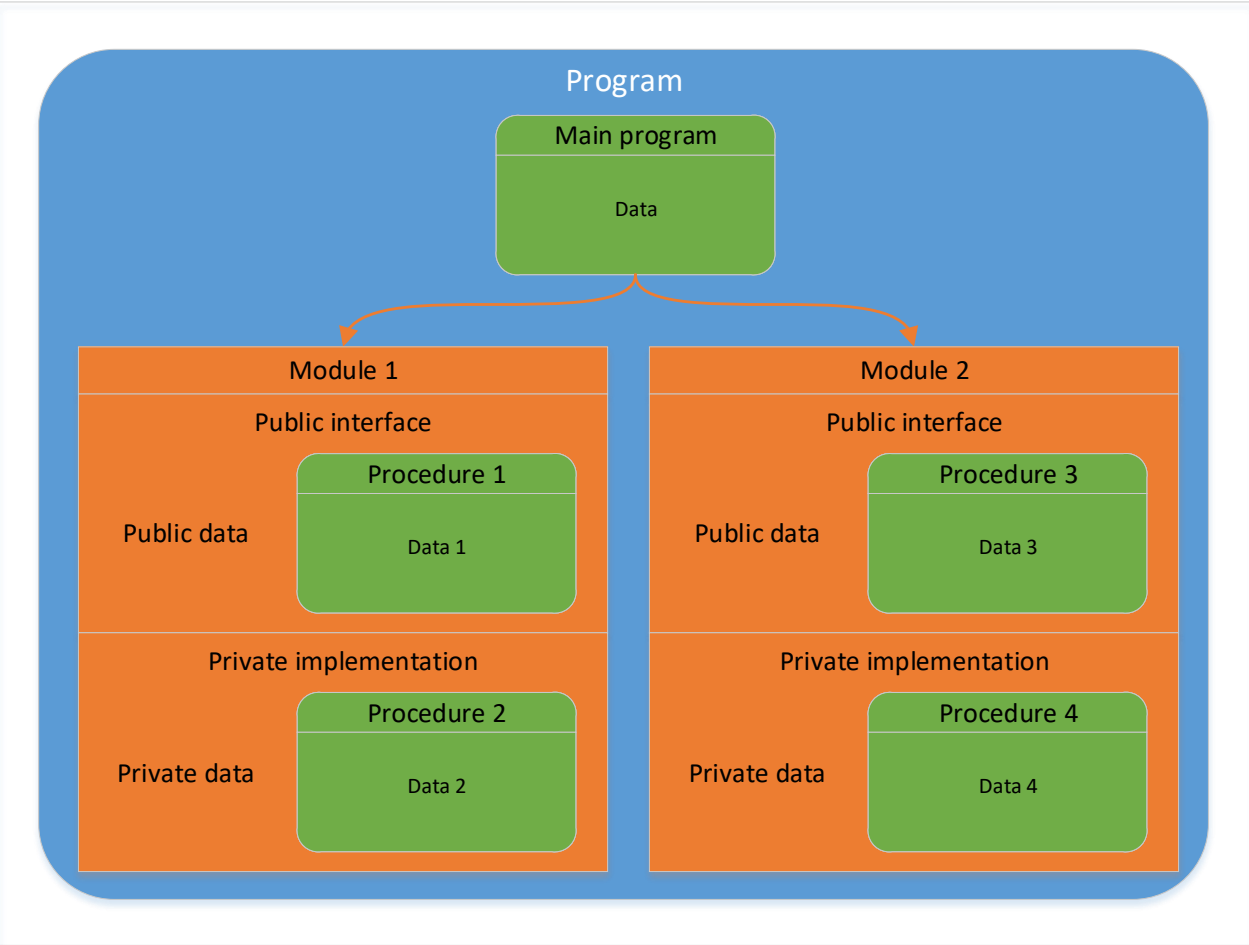
Procedural Programming

- **Procedure**: A set of instructions and named internal data.



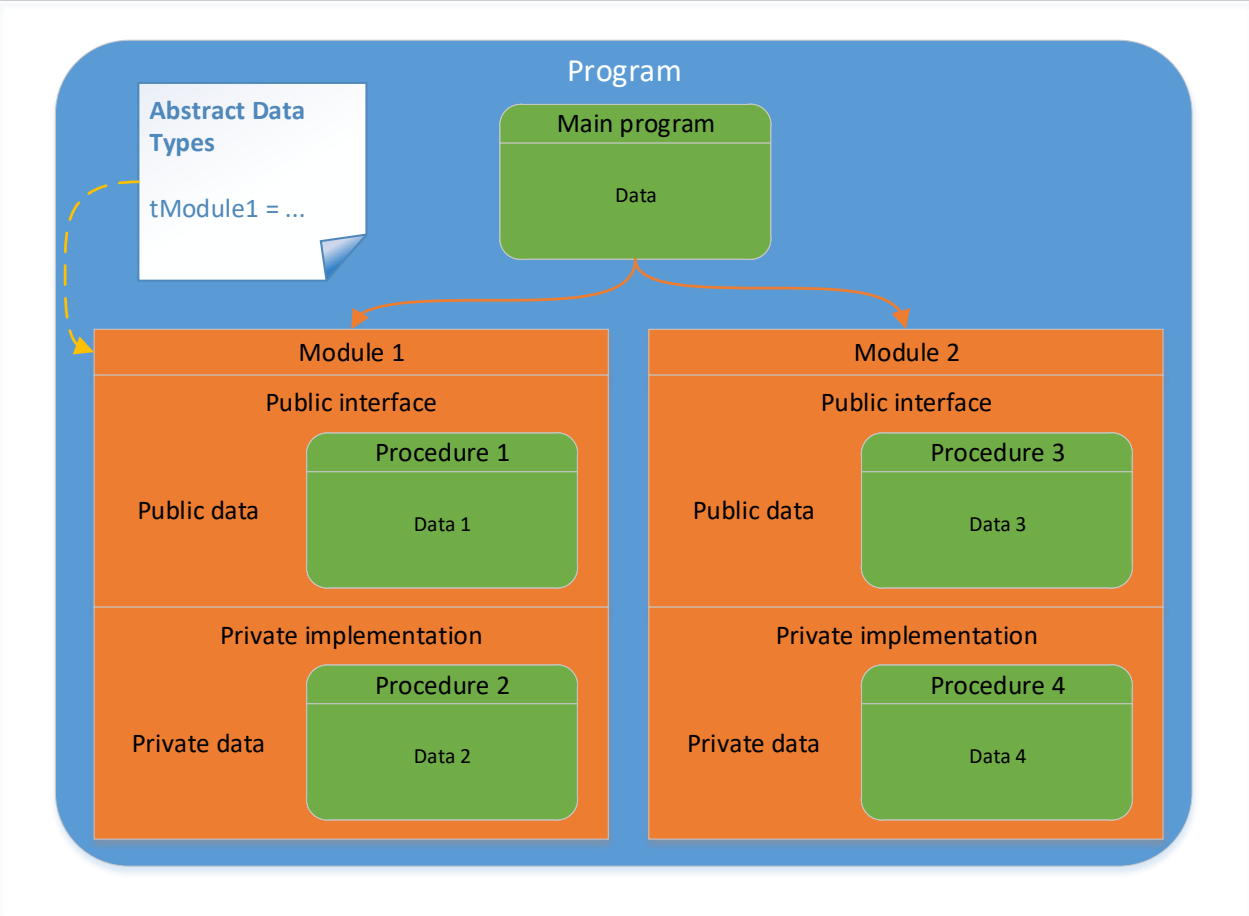
Modular Programming

- **Module**: A grouping of procedures with shared functionalities. Includes procedures and data that are not accessible from outside.



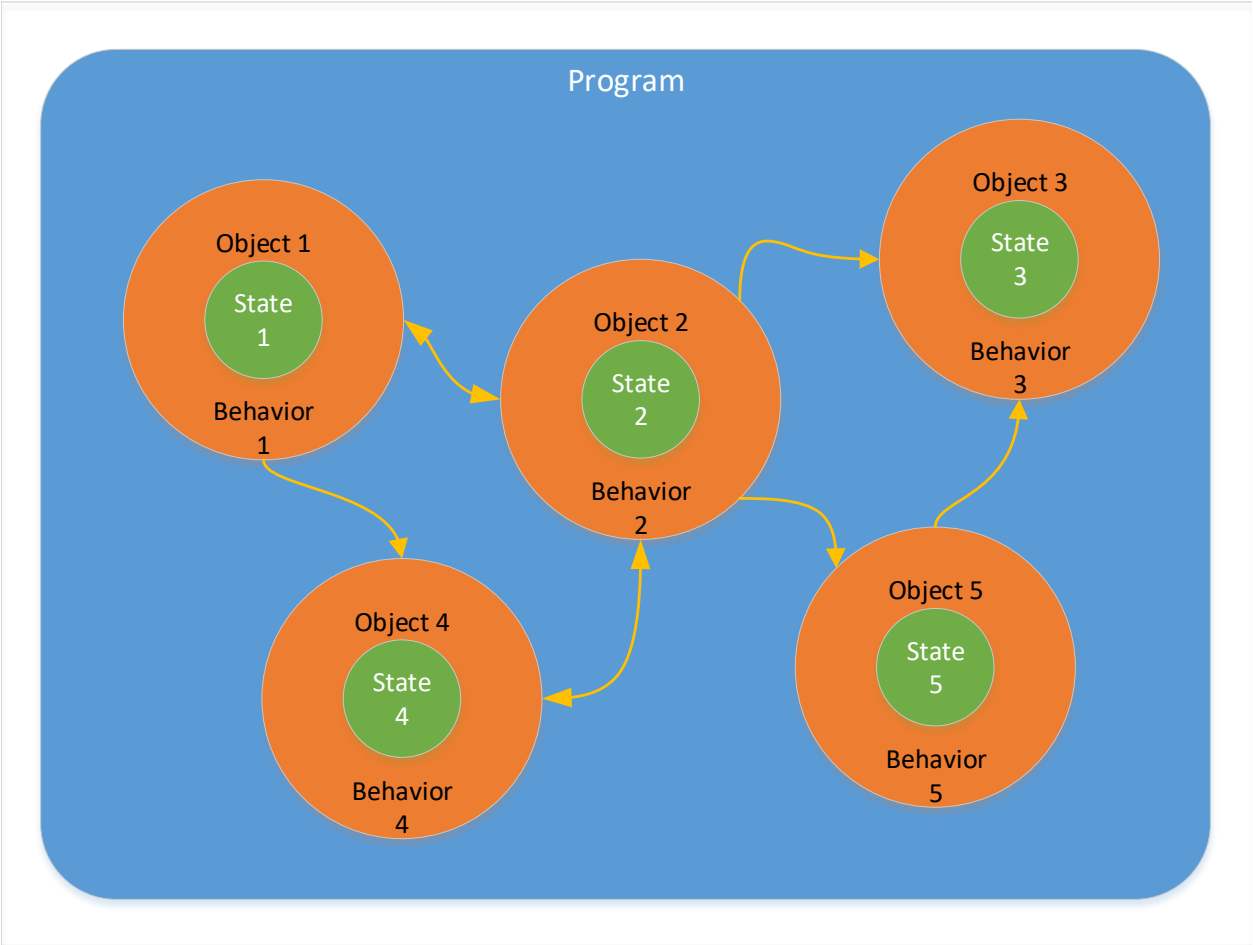
Abstract Data Types

- **Instantiation**: Allows the instantiation of customized data types.



Object-Oriented Programming

- Composed of **objects** (with their own state and behavior) that communicate with each other through messages.



Objects and Abstract Data Types (ADTs)

- **Similarities between Objects and ADTs**
  - Initially, an object would seem to be an ADT by a different name, and using different nomenclature, e.g., "classes" and "objects" instead of "types" and "variables".
  - An object belongs to a *class*, maintains an internal *state* and exports *methods* (i.e., procedures and functions). The internal state is private and can only be manipulated through methods.
  - Objects possess characteristics such as abstraction, encapsulation, modularity, etc., which ADTs already offered.

What do objects offer that is new?

- **A new design philosophy**
  - **Bottom-Up** design instead of **Top-Down** design.
- **Inheritance**
  - Objects can inherit the attributes and the behavior of other objects.
  - Related characteristics, such as **polymorphism** and **dynamic binding**.

Top-Down Design

Top-Down Design

The program is systematically decomposed into smaller, more manageable parts.

- It's a mechanism typical of structured programming.
- Decomposition stops when the parts are so simple that they can be implemented directly.

Top-Down Design

- **Example**: Top-Down design of a card game.

Drawbacks of Top-Down design

- The emphasis is placed on implementing functions that solve the problem, instead of designing the necessary data structures.
- It tends to yield designs that are custom-built for the problem under consideration. This is because the lower-level parts exist only to meet the needs of the higher-level parts.
- The resulting code is difficult to reuse, even for similar projects (e.g., another card game).

Bottom-Up Design

Bottom-Up Design

We begin by identifying all the basic data structures of the program, their behavior and their interactions. The project is built by combining these basic building blocks.

- It's the common design mechanism in object-oriented programming.
- The basic building blocks are "autonomous" objects that encapsulate their own internal state and react to messages.
- The resulting objects can be easily reused for similar problems.

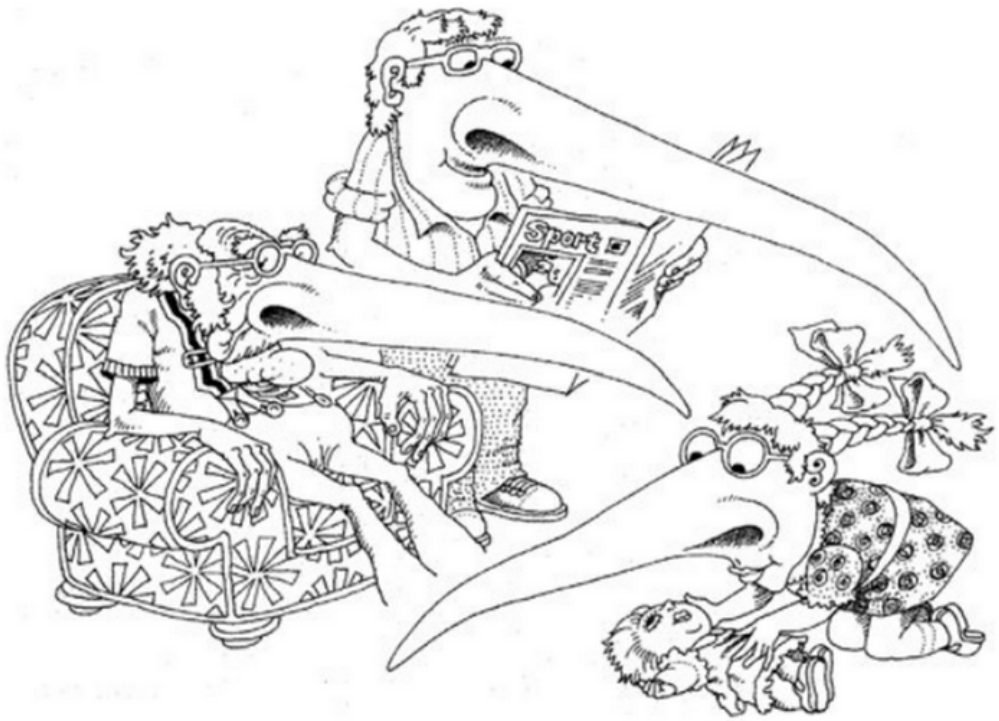
Bottom-Up Design

Exercise: Card game

- What are the objects in a card game?
- Which kind of internal state (i.e., data) do they store?
- How do they communicate with each other?
- What is their behavior (i.e., their methods)?

Inheritance

It's an "IS_A" relationship between classes. A class reuses the structure and the behavior that were previously defined for other class(es).



A subclass may inherit the structure and behavior of its superclass.

Inheritance

- It allows to define new classes that inherit the characteristics of their "parent" classes while adding their own specialized characteristics.
- It can also be used to generalize the common characteristics of several "subclasses" into a "superclass".

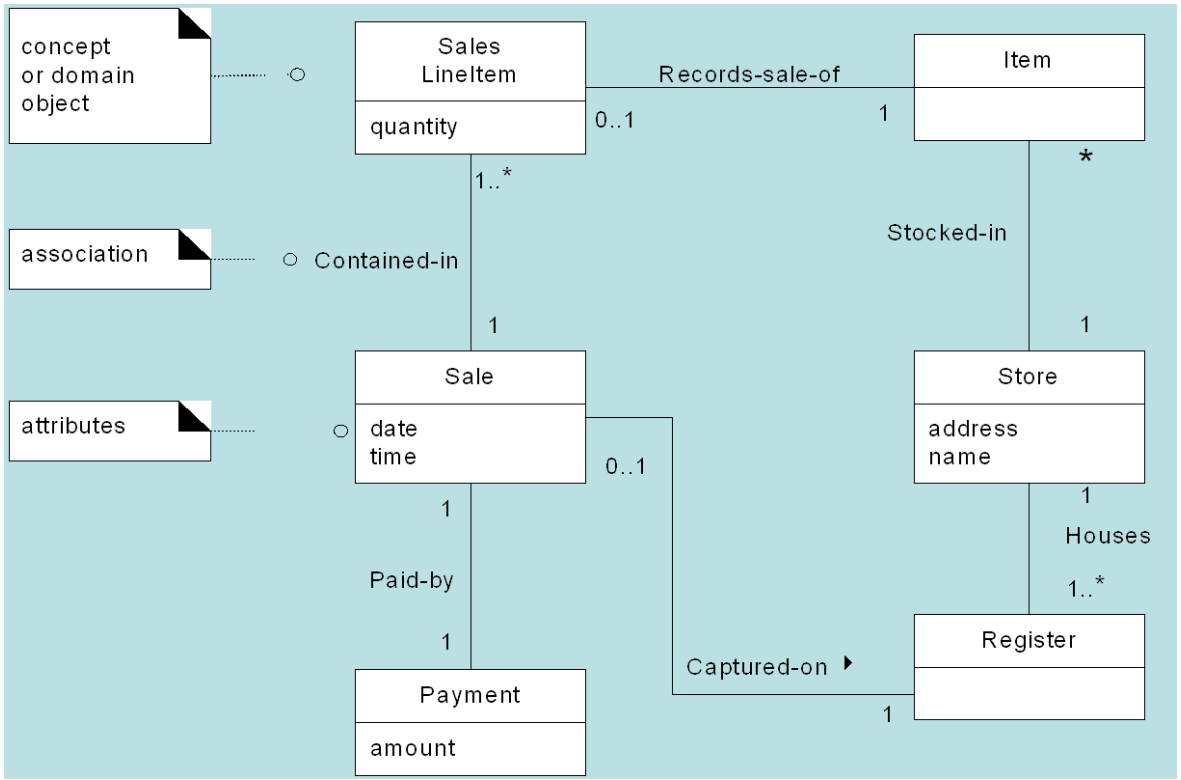
Exercise: Card game

Can you think of an example of inheritance for card games?

Domain Model

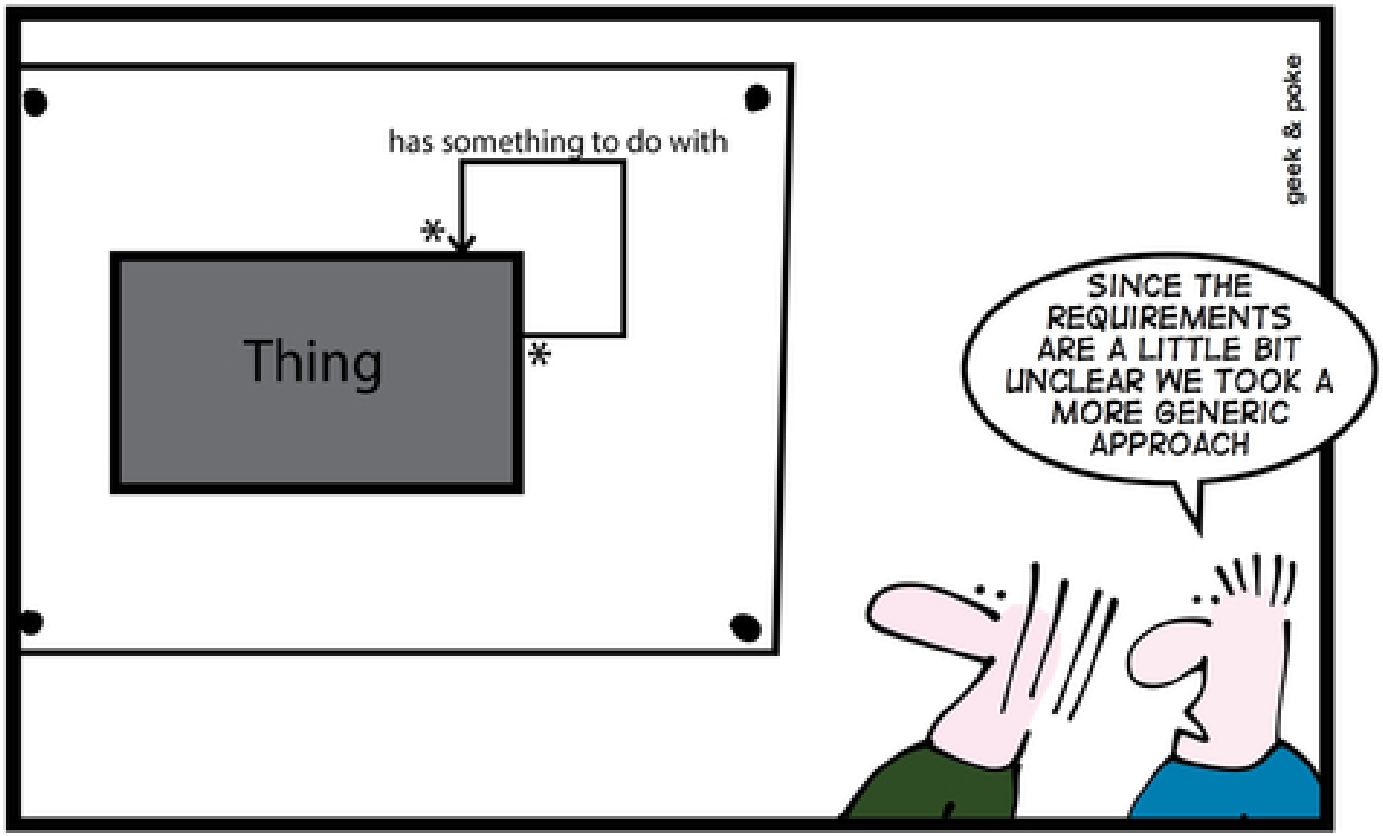
Domain Model (or Conceptual Model)

It's an object-based model that shows a "general overview" including the main **classes**, their **attributes** and their **relations**.



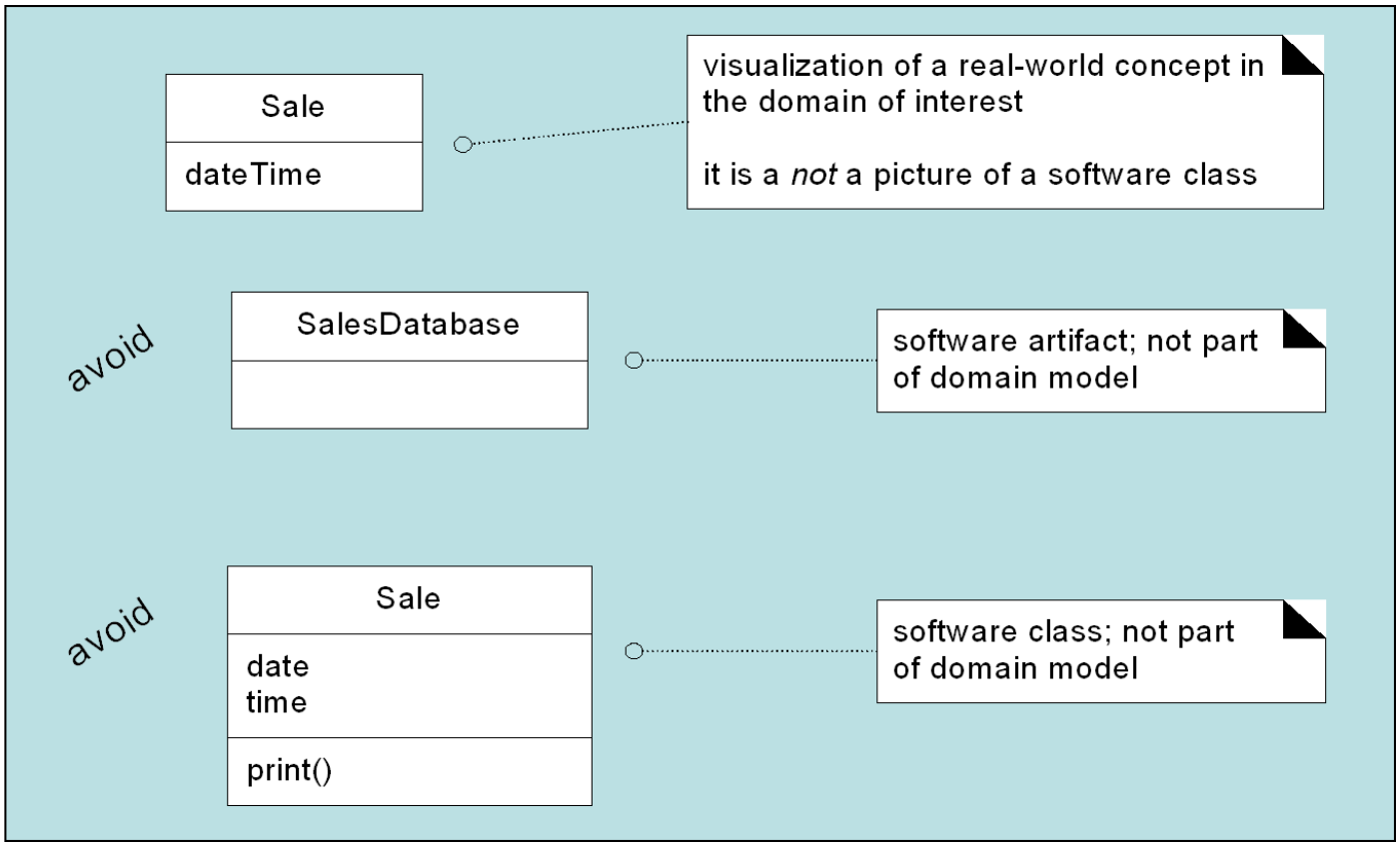
Domain Model

- It must be possible to obtain the domain model from the requirements.



Domain Model

- It must describe real-situation objects, NOT software artifacts.



How To Find Conceptual Classes

- **Identifying nouns and noun phrases**: Nouns in textual domain descriptions (e.g., use cases) are candidates for conceptual classes and attributes.
- **Example Point of Sale (POS)**:

Main Success Scenario (or Basic Flow):

1. **Customer** arrives at a **POS checkout** with **goods** and/or **services** to purchase.
2. **Cashier** starts a new **sale**.
3. **Cashier** enters **item identifier**.
4. System records **sale line item** and presents **item description**, **price**, and running **total**. Price calculated from a set of price rules.

Cashier repeats steps 2-3 until indicates done.

5. System presents total with **taxes** calculated.
6. Cashier tells Customer the total, and asks for **payment**.
7. Customer pays and System handles payment.
8. System logs the completed **sale** and sends sale and payment information to the external **Accounting** (for accounting and **commissions**) and **Inventory** systems (to update inventory).
9. System presents **receipt**.
10. Customer leaves with receipt and goods (if any).

Example: Point of Sale (POS)

Register	Item	Store	Sale
Sales LineItem	Cashier	Customer	Ledger
Cash Payment	Product Catalog	Product Description	

Example: Monopoly Game

Conceptual Classes

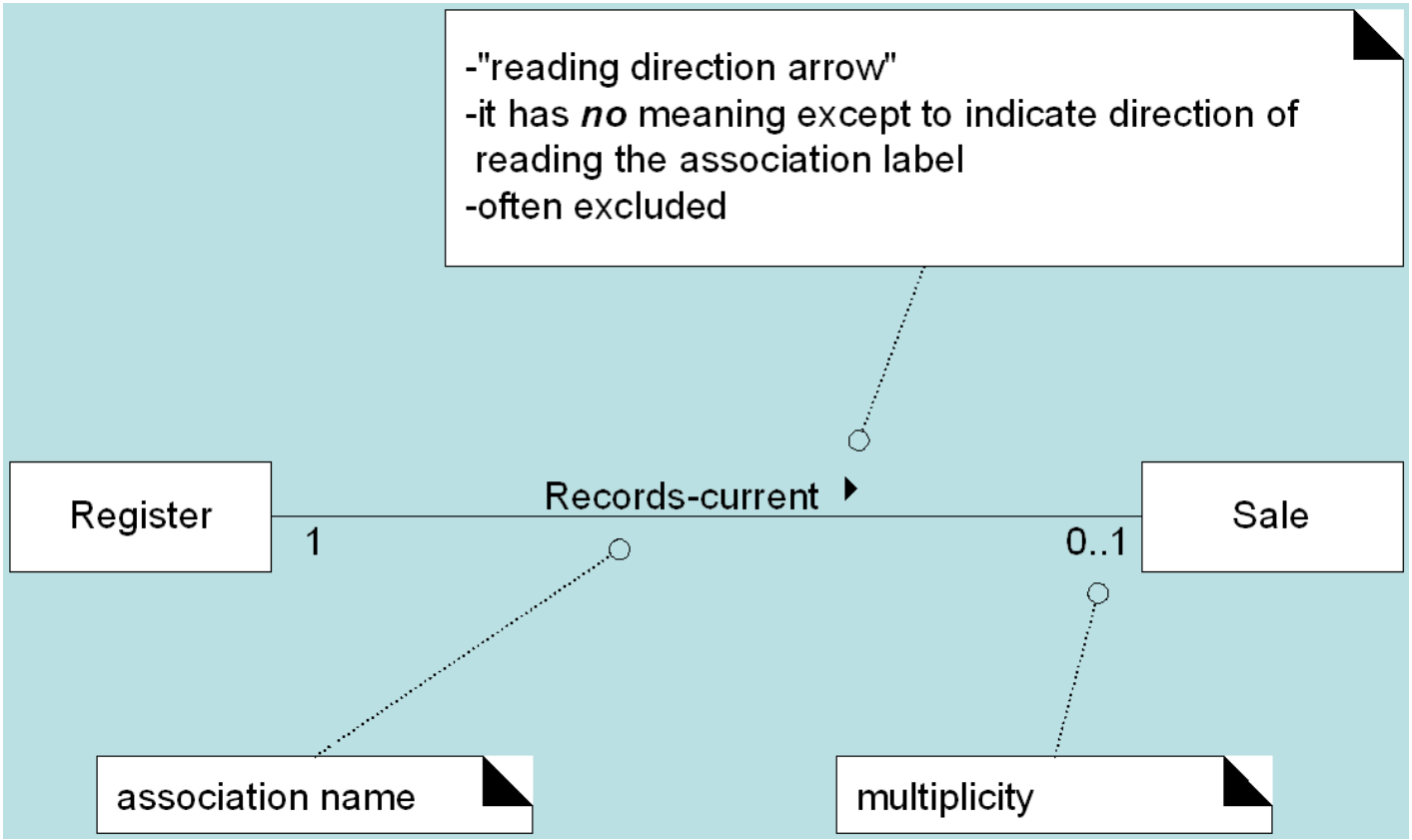
Conceptual Association

Conceptual Association

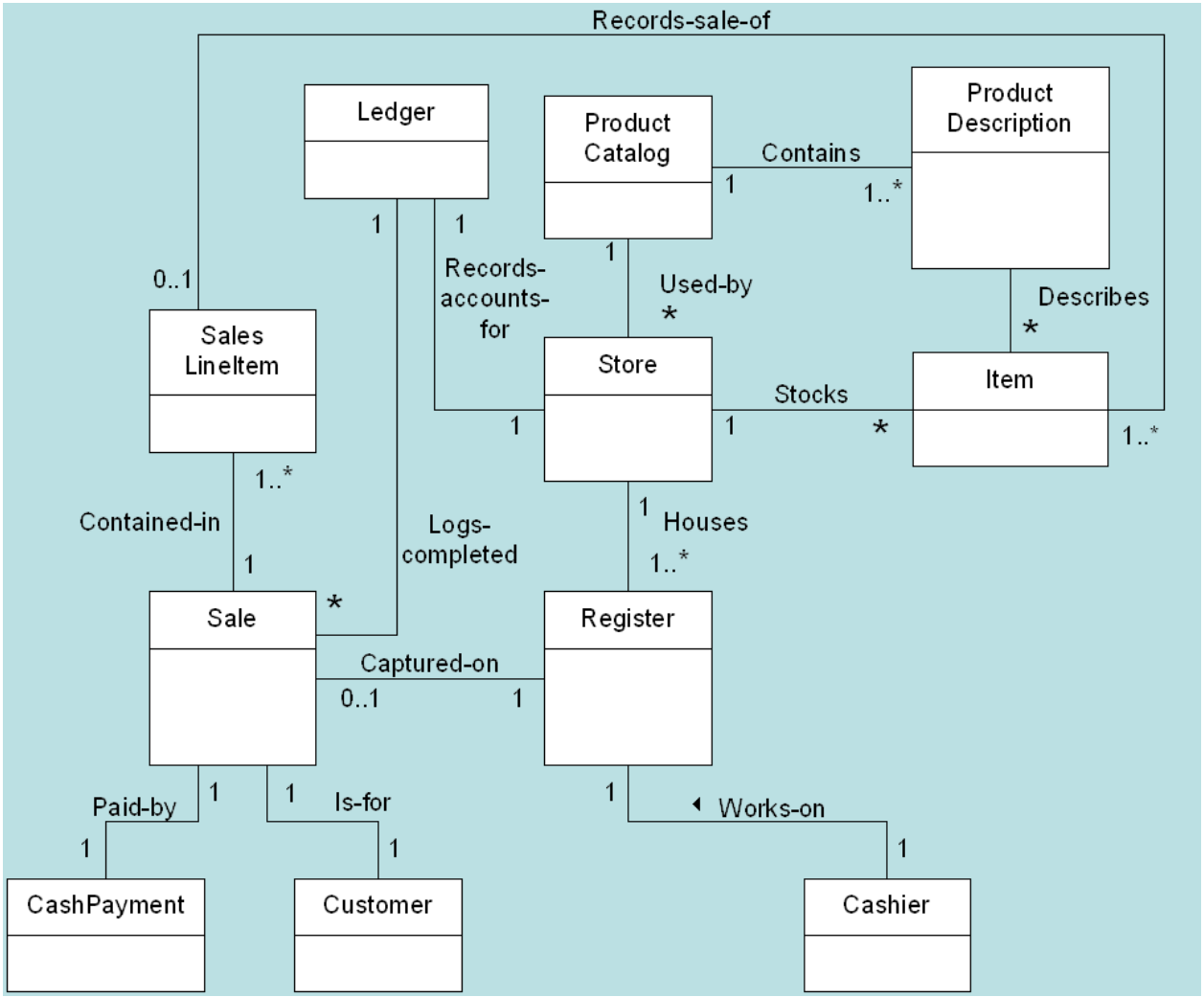
The semantic relationship between two or more classifiers that involve connections among their instances.

- **An association is NOT**: A statement about data flows, database foreign key relationships, instance variables, or object connections in software.
- **An association IS**: A statement that a relationship is meaningful from a purely conceptual perspective in the real world.
- That said, many conceptual associations are naturally represented as associations in a Design Model.

Notation for conceptual associations



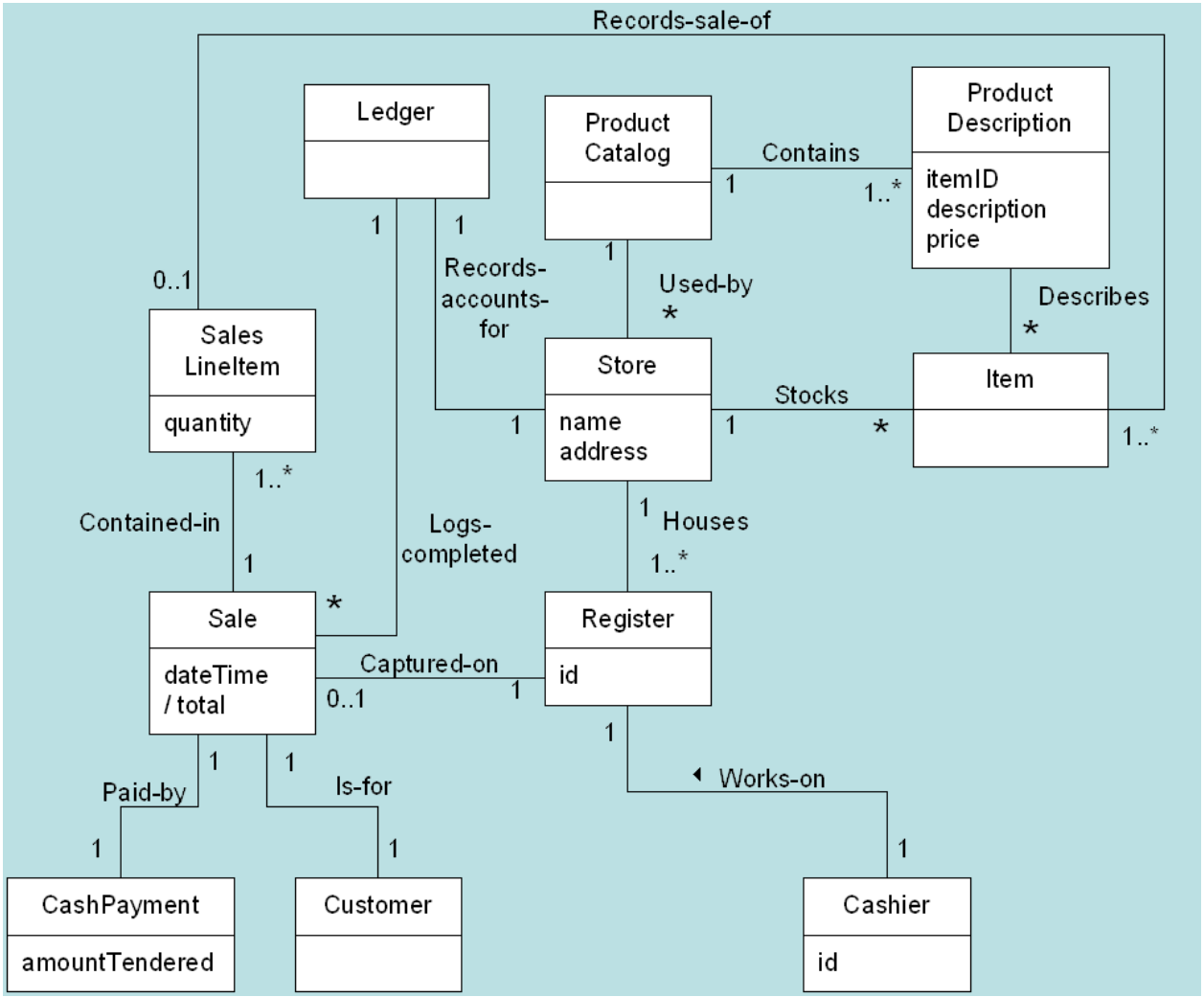
Example: Point Of Sale



Attributes in Domain Models

- Attributes are preferably primitive data types (e.g., boolean, number, character) or non-trivial but common data types (e.g., date, time, address, phone, etc.).
- Specific types are often omitted, as they are not relevant at the most conceptual level.
- Attributes should never refer to another class in the domain. Relationships between classes must be specified via associations.

Example: NextGen Point of Sale System

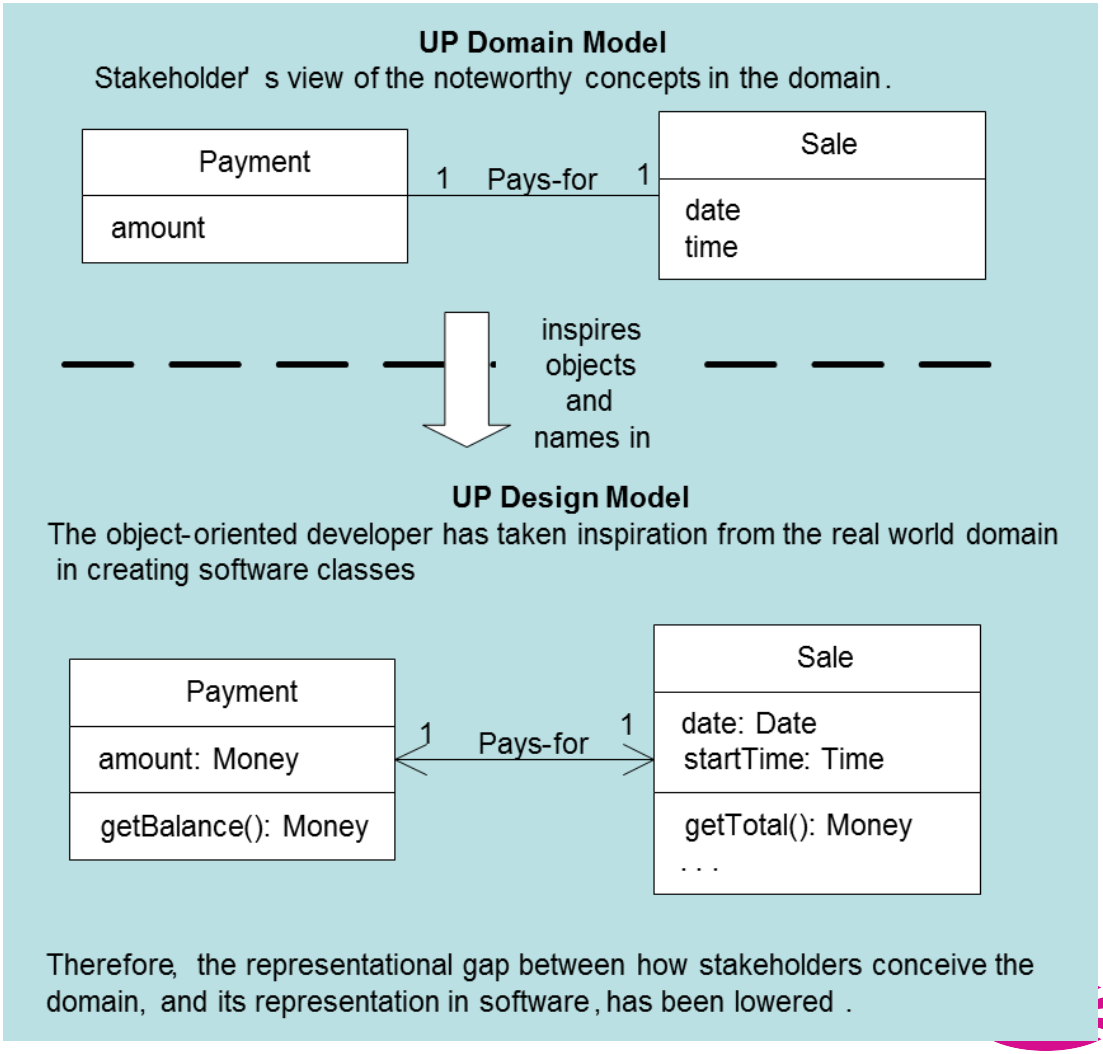


Example: Monopoly Game

Conceptual Diagram

Domain Model vs. Design Model

- Conceptual classes are the main inspiration for designing software objects.
- This bridges the gap between the mental model and the software model.



Assigning Responsibilities

Assigning Responsibilities

Consists in identifying responsibilities (i.e., contracts or obligations) and assigning them to objects.

- It's a metaphor that equates software objects with entities that have responsibilities towards each other in order to achieve a goal.
- A responsibility is not a method. A method is a means to fulfilling a responsibility.
- Responsibilities enhance encapsulation, as a client does not need to know how a given service implements its responsibilities.

Types of Responsibilities

- **Doing**
  - Creating an object
  - Doing a calculation
  - Initiating actions in other objects
  - Coordinating activities in other objects
  - Etc.
- **Knowing**
  - Knowing about private data
  - Knowing about related objects
  - Knowing about things it can calculate

How to Assign Responsibilities to Objects

Responsibility of creating

- A class B has the responsibility of creating an instance of class A if:
  - B "contains" o "aggregates" objects of type A
  - B records instances of A
  - B closely uses A
  - B has the initializing data for A

- The more conditions it matches, the better.
- If multiple classes match the conditions, we prioritize the one that matches the first condition.

Example: Monopoly

Who should create the Squares in the Monopoly game?

Ejemplo: Monopoly

Solution:

Ejemplo: Monopoly

Which design relationship appears between those two classes?

How to Assign Responsibilities to Objects

General criterion

- We must assign a responsibility to a class that possesses the information to carry it out.

- A responsibility needs information about other objects, about the object itself, about what can be obtained, etc.

Example: Monopoly

Which class has the responsibility of obtaining a square from a name?

- `Square s = xxx.getSquare(name)`

Example: Monopoly

Solution:

Benefits of Object Orientation

- **Reusability**
  - Well-designed objects can be combined to create bigger systems.
- **Modularity**
  - Objects are self-contained and have clearly-defined interfaces with other objects.
- **Comprehension**
  - As each object encapsulates its own data and methods, the object can be separately developed and tested.

Benefits of Object Orientation

- **Naturalness**
  - Identifying the necessary objects feels more "natural" than the functional decomposition and progressive refinement of Top-Down design.
- **Extensibility**
  - Inheritance allows defining and using functionally incomplete modules, which can be later expanded without affecting its client modules.
- **Scalability**
  - Unlike conventional systems, the effort spent on object-oriented designs does not grow exponentially with the complexity of the project.

Drawbacks of Object Orientation

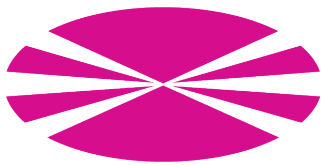
- **Learning curve**
  - Learning an object-oriented programming language and mastering its techniques is more difficult than learning an imperative PL.
- **Change of approach**
  - Typically, developers start with imperative languages and top-down philosophies. OO requires a change of mindset.
- **Inefficient reuses**
  - Objects are not inherently reusable. They must be expressly designed to be reusable, which requires extra effort.
- **Forced object orientation**
  - "*I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail.*"
  - Imposing object orientation where it's not appropriate can yield inadequate designs (e.g., the `Math` class in Java avoids this pitfall).

Unit 1: Introduction

Software Design (614G01015)

David Alonso Ríos
Eduardo Mosqueira Rey (Coordinator)

Department of Computer Science and Information Technology
Faculty of Computer Science



UNIVERSIDADE DA CORUÑA