Software Engineering

With or Without You (U2)

Ivo Vondrak, 2025

- What is an engineering?

- What is an engineering?
- Engineering is about making ideas alive!

Software Engineering

The most accepted definition

- Software engineering is the branch of computer science that deals with the **design, development, testing, and maintenance of software** applications. Software engineers apply **engineering principles** and knowledge of programming languages to build software solutions for end users.
- Engineering principles mean **systematic and disciplined approach**

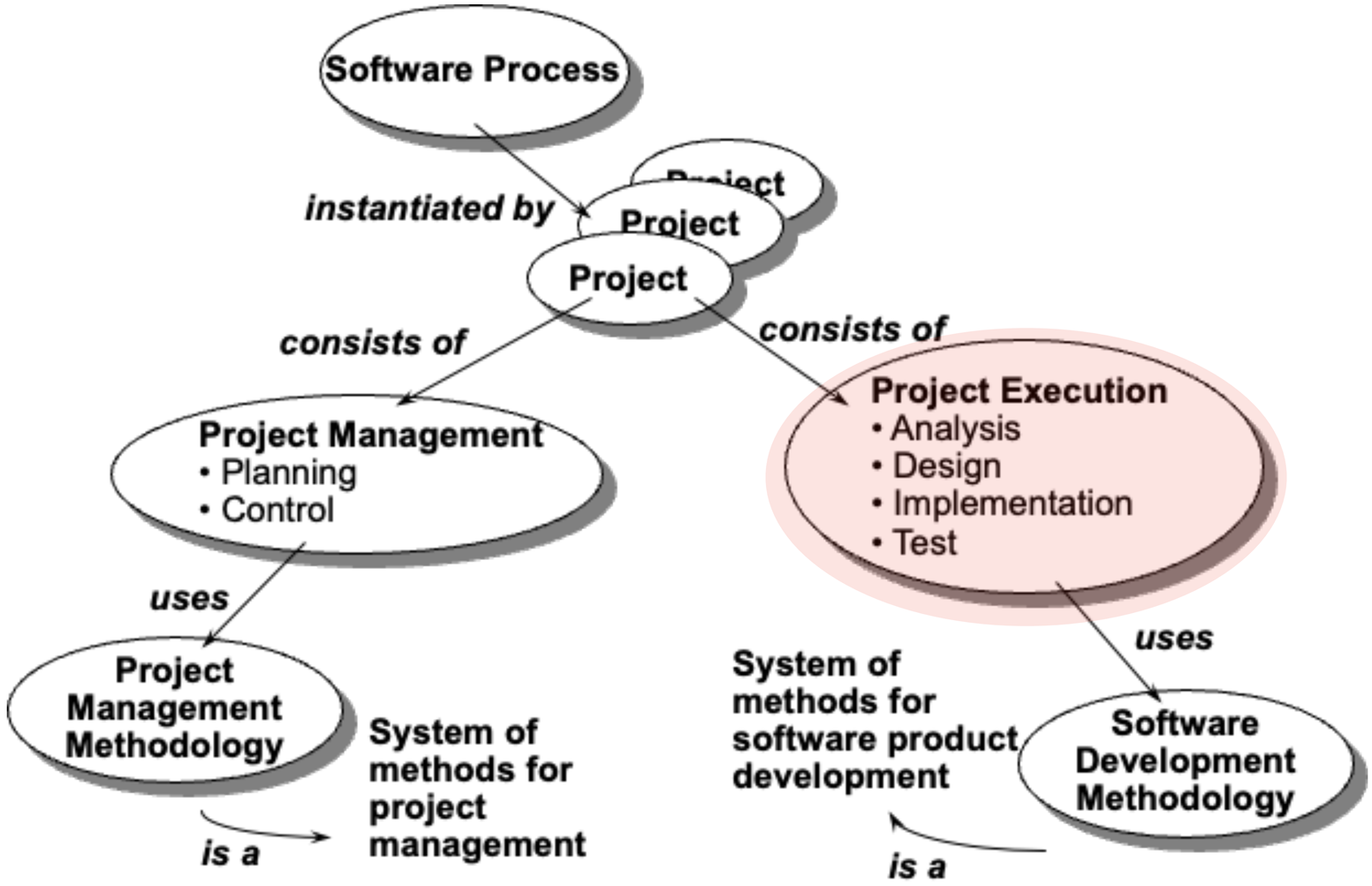
A physician a civil engineer, and a computer scientist were arguing about what was the oldest profession in the world. The physician remarked, "Well, in the Bible, it says that God created Eve from rib taken out of Adam. This clearly required surgery, and so I can rightly claim that mine is the oldest profession in the world. " The civil engineer interrupt, and said, "But even earlier in the book of Genesis, it states that God created the order of heavens and earth from out of chaos. This was the first and certainly the most spectacular application of civil engineering. Therefore, fair doctor, you are wrong: mine is the oldest profession in the world." The computer scientist leaned back in the chair, smiled, and then said“ confidently, **"Ah, but who do you think created the chaos?"**

Unknown but very experienced author 🤪

How to live with …

Software Production Layout

Think about it OR just code it 🤷



The Software Development Life Cycle

SDLC is a framework that defines the steps involved in the development of software — from initial idea to deployment and maintenance

Purpose:

- To ensure quality, predictability, and efficiency.
- Provides a structured approach for managing complex projects.
- Helps to reduce risk and rework.

1. Requirements Analysis
2. System Design
3. Implementation (Coding)
4. Testing
5. Deployment
6. Maintenance

Key Characteristics of SDLC Models

Model defines how the stages of SDLC are executed and connected

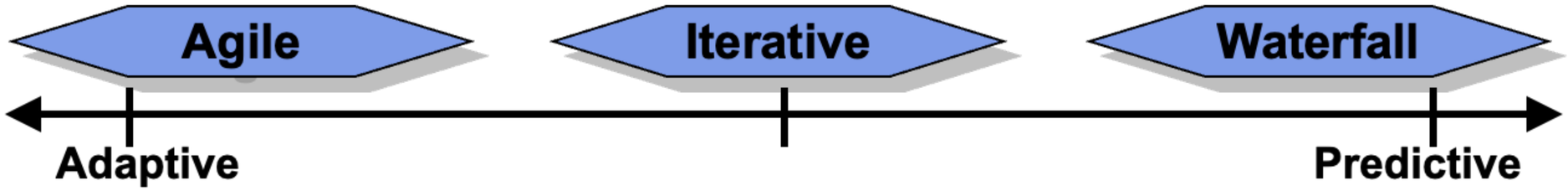
- **Waterfall**: Linear, rigid, sequential
- **Iterative**: Allows revisiting earlier phases
- **Agile**: Adaptive, customer-focused, incremental
- **V-Model**: Emphasizes testing at each stage
- **Spiral**: Risk-driven, combines iterative and waterfall features

Adaptive vs. Predictive Methods

All approaches have their own pros and cons 🤷‍♂️

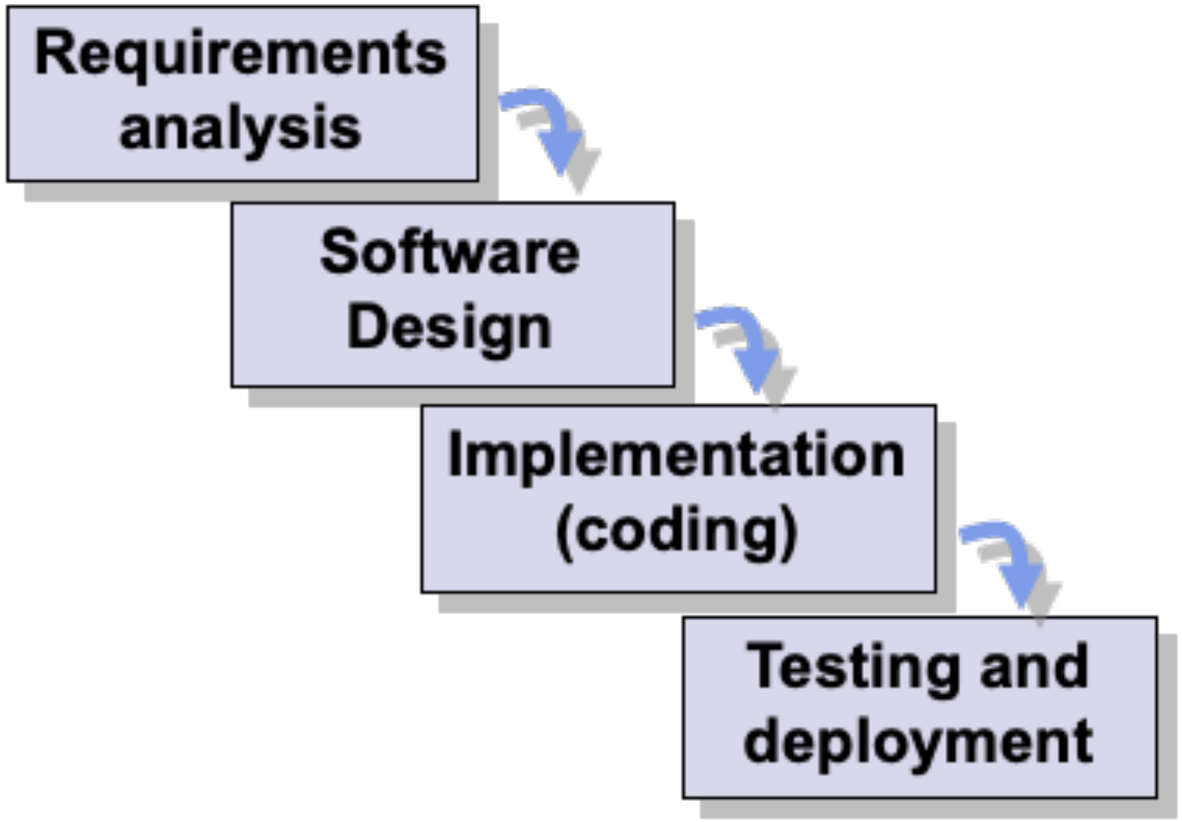
- **Adaptive methods** focus on **adapting quickly to changing realities**. When the needs of a project change, an adaptive team changes as well. An adaptive team will have difficulty describing exactly what will happen in the future.

- **Predictive methods focus on planning the future in detail.** A predictive team can report exactly what features and tasks are planned for the entire length of the development process. Predictive teams have difficulty changing direction. The plan is typically optimized for the original destination and changing direction can cause completed work to be thrown away and done over differently.



The Water Fall Process Model

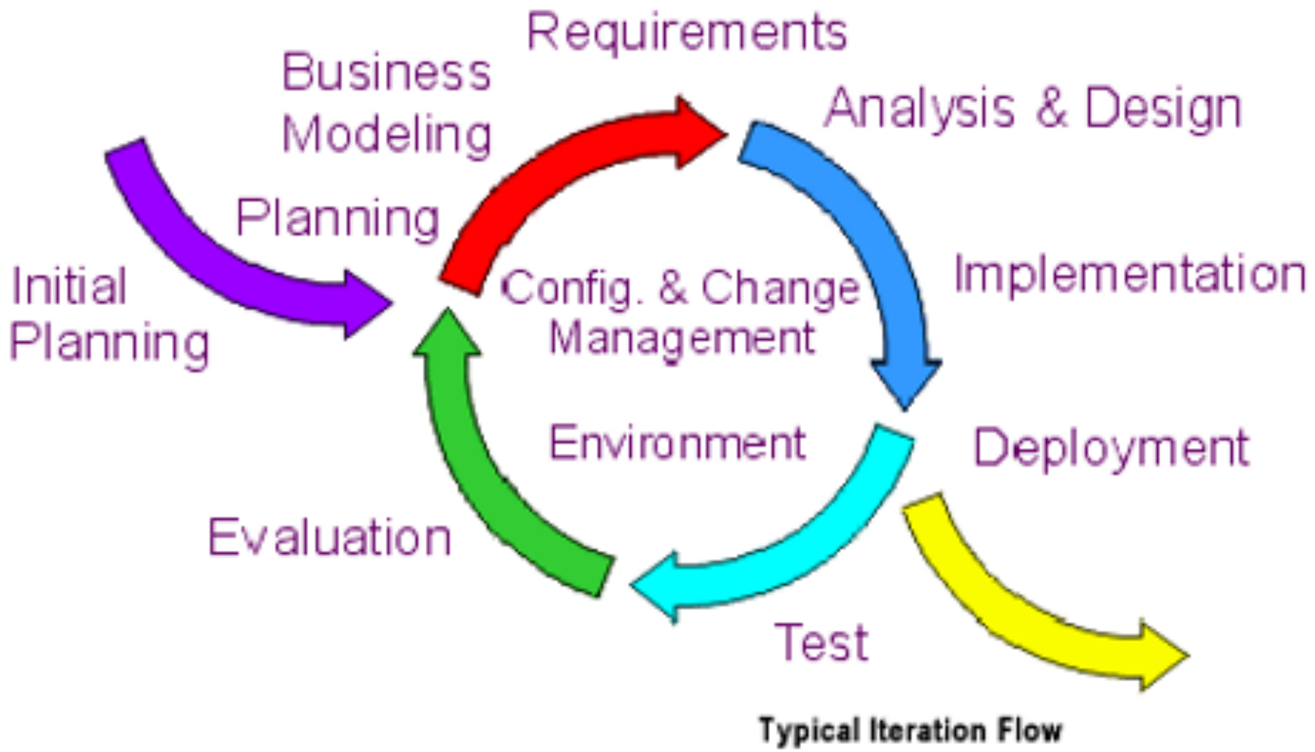
We are not building houses and bridges 🤷‍♂️



- It takes too long to see results.
- It depends on stable, correct requirements.
- It delays the detection of errors until the end.
- It does not promote software reuse and prototyping.

The Iterative Approach

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



Typical Iteration Flow

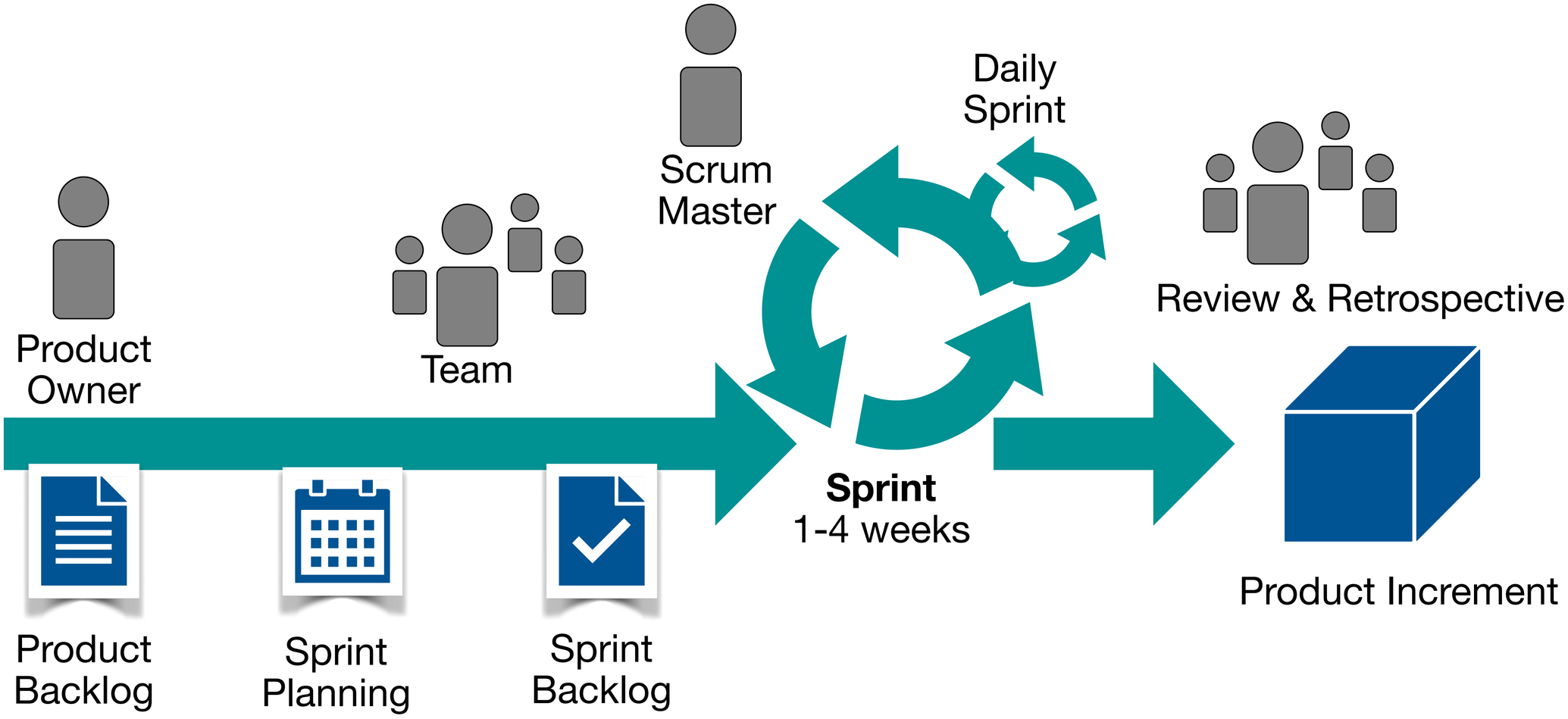
Benefits of an Iterative Approach

We do not know everything at the beginning 🙈

- **Risk Mitigation** – an iterative process lets developers mitigate risks earlier than a sequential process where the final integration is the only time that risks are discovered or addressed.
- **Accommodating Changes** – an iterative process lets developers take into account requirements, tactical and technological changes continuously.
- **Learning as You Go** – an advantage of the iterative process is that developers can learn along the way, and the various competencies and specialties are employed during the entire lifecycle.
- **Increased Opportunity for Reuse** – an iterative process facilitates reuse of project elements because it is easy to identify common parts as they are partially design and implemented instead of identifying all commonality in the beginning.
- **Better Overall Quality** – the system has been tested several times, improving the quality of testing. The requirements have been refined and are related more closely to the user real needs. At the time of delivery, the system has been running longer.

The Agile Method – Scrum

Responding to change over following a plan



Scrum Framework Principles

Working software is the primary measure of progress

- Based on short, fixed-length iterations called **Sprints** (1–4 weeks).
- Each Sprint delivers a **potentially shippable product** increment.
- Roles: **Product Owner, Scrum Master**, and **Development Team**.
- Events: **Sprint Planning** (before each sprint, limit 8h), **Daily Scrum** (during the actual sprint, 15-20 minutes), **Sprint Review** (at the end of the sprint, system previews, limit 4h), **Sprint Retrospective** (feedback from the sprint, answers: **What was done correctly** and **What could be done better**)
- Artifacts: **Product Backlog** (high level document that describes the whole product, what should the system do, requirements etc.) , **Sprint Backlog** (detailed document that describes the information about the current sprint), **Increment** of working software.
- **Focuses on collaboration, transparency, and rapid feedback.**

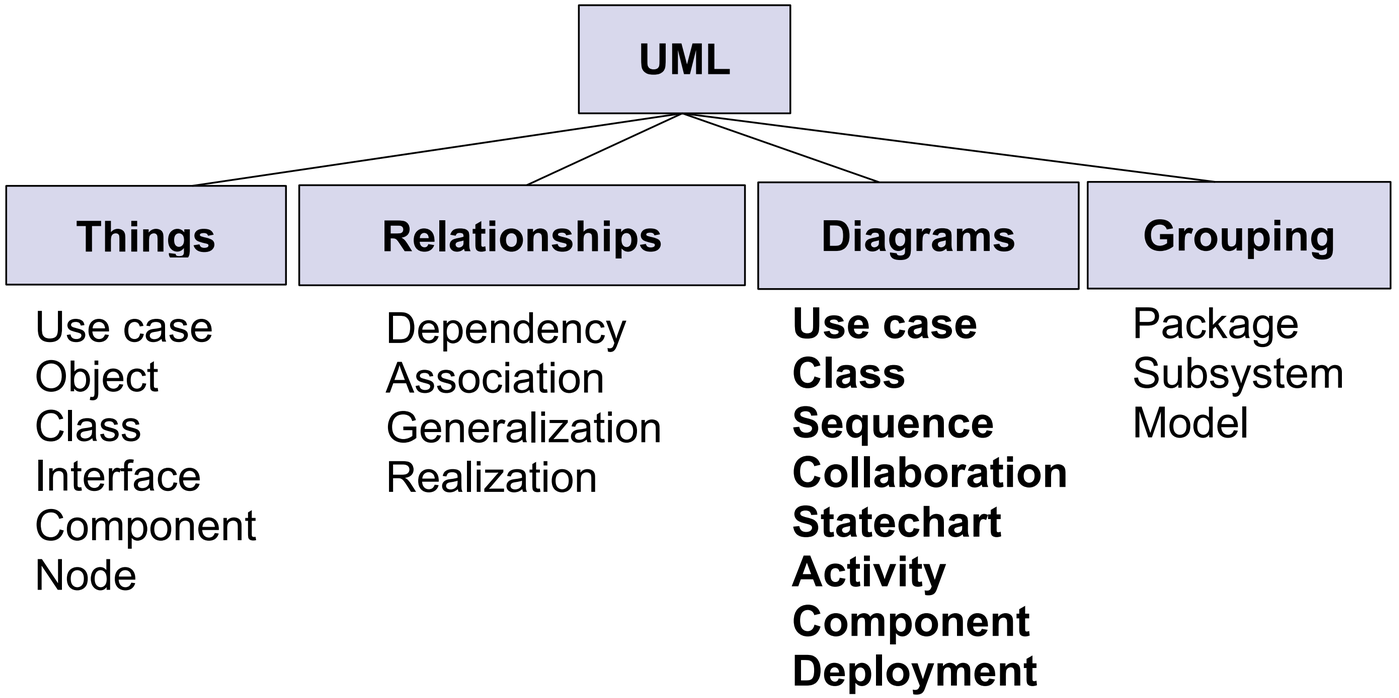
The Unified Modeling Language

A standard way to visualize the design of a software system

- The **Unified Modeling Language** (UML) is a standard language for writing software **blueprints**.
- The UML may be used to **visualize, specify, construct and document** the artifacts of a software-intensive system.
  - Visualizing means **graphical language**
  - Specifying means **building precise, unambiguous, and complete models**
  - Constructing means that **models can be directly connected to a variety of programming languages**
- Created by **Grady Booch, James Rumbaugh, and Ivar Jacobson** (the “Three Amigos”).
- Used in both **object-oriented analysis and design**.

Building Blocks of the UML

https://www.uml.org



Business Modeling

The main goal of the business process modeling is to provide common language for communities of software and business engineers

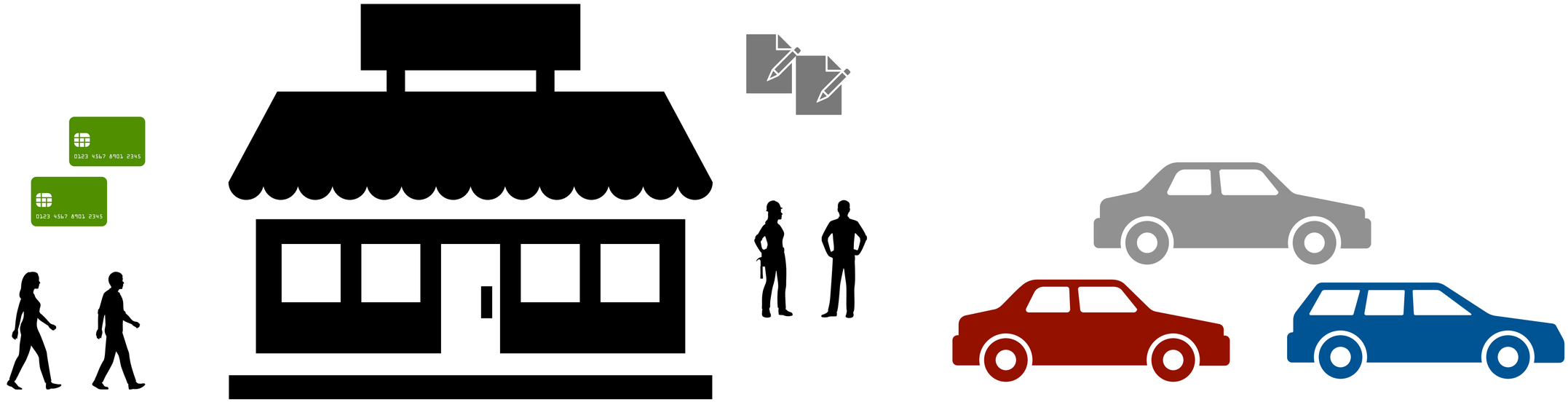
- **Business Process Modeling** (**How & When**). Business process is a set of one or more linked procedures or activities which collectively realize a business objective or policy goal.

- **Domain Modeling** (**Who & What**) captures the most important objects in the context of the system. The domain objects represent the entities that exist in environment in which the system works.

Motivating Example

Requirements statement (sometimes it's only thing you get) 🙈

Develop an information system for a car dealer. The application should collect and provide information about **customers**, their **orders**, **cars**, **payments** etc. The possibility to **communicate with the car manufacturer** to obtain updated offer of available cars or to order a car required by a customer should be the part of the system.



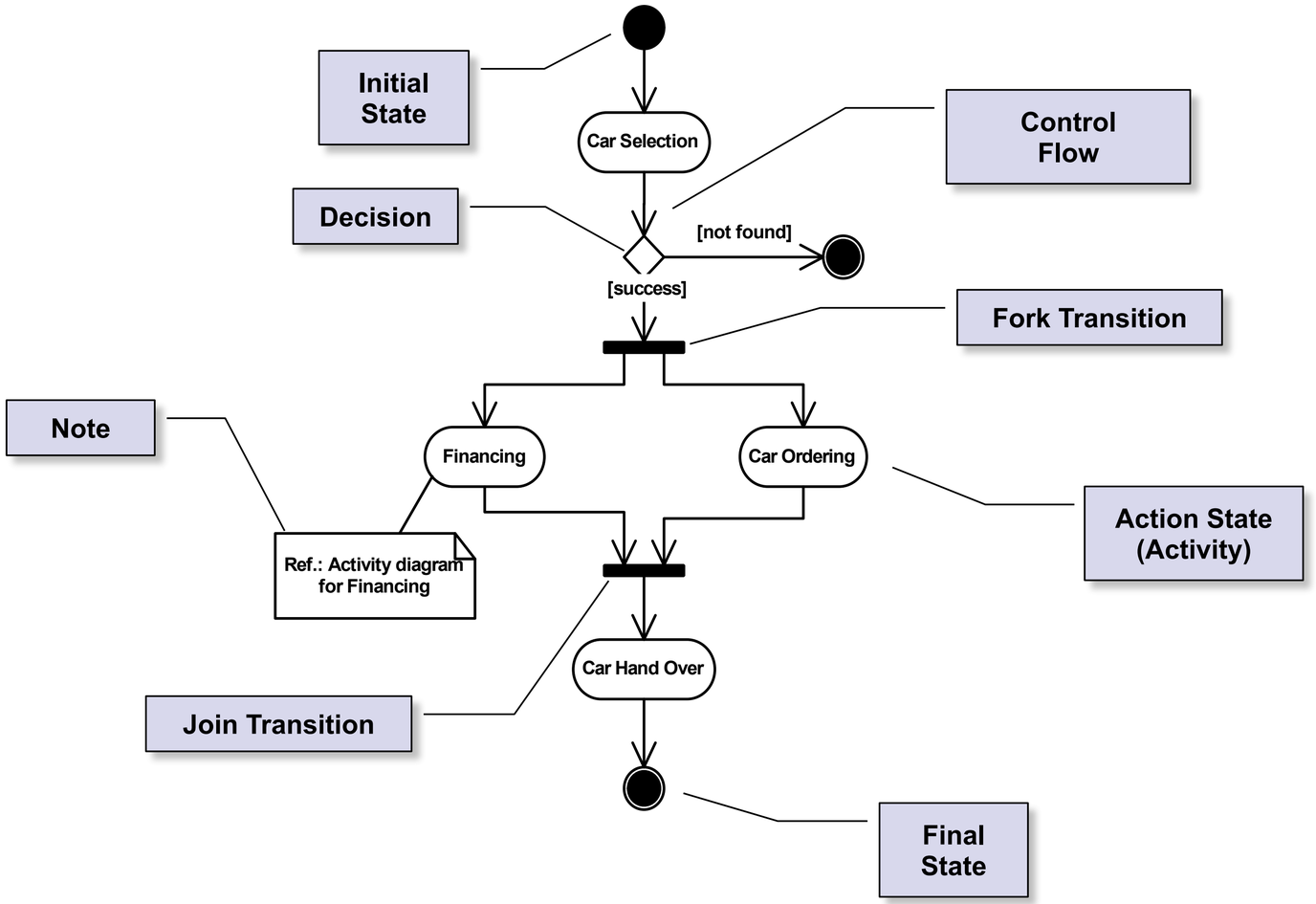
UML Diagrams for Business Modeling

What the hell is going on at the dealership? 🤷

- **Activity Diagram** is a variation of a state machine in which the states represent the performance of **activities** and the transitions are triggered by their completion.
    - The purpose of this diagram is to **focus on flows driven by internal processing**.
- **Class Diagram** is a graph of elements (in the scope of business modeling represented by **workers and entities**) connected by their various static relationships.
    - The purpose of this diagram is **to capture static aspect** of the business domain.

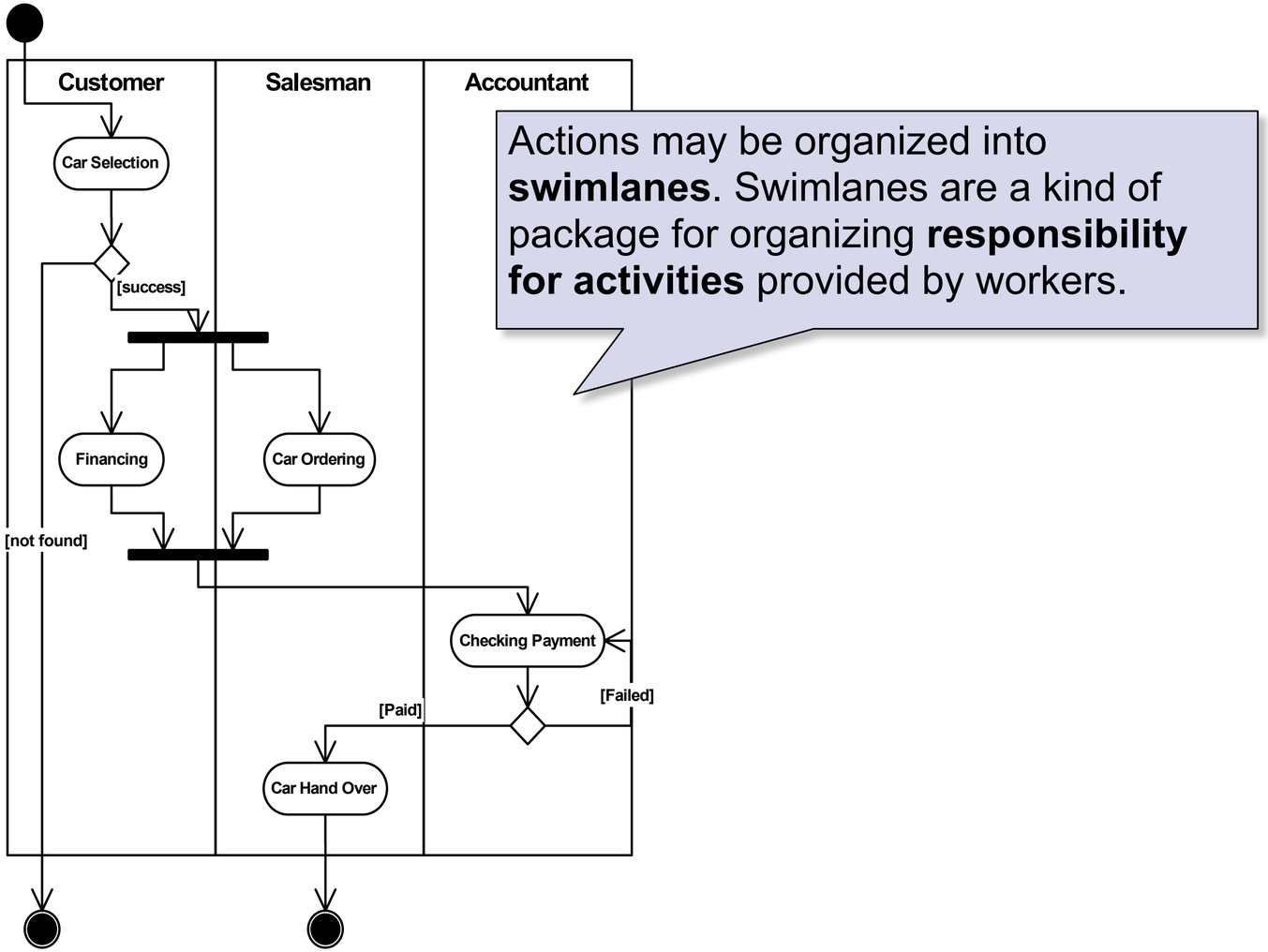
Activity Diagram: Car Sale Process

One of the many business processes but the important one



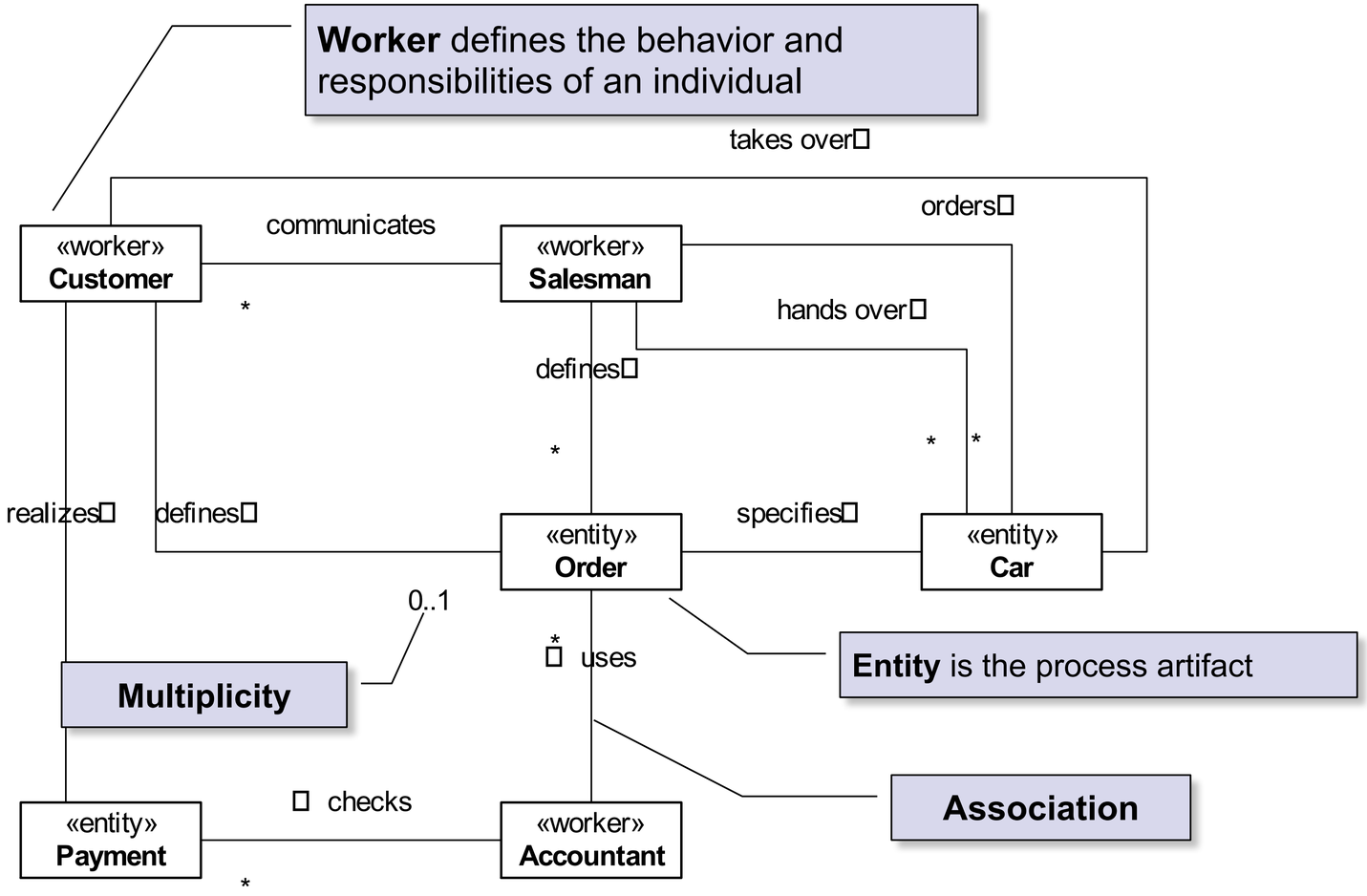
Swimlanes: Packages of Responsibilities

What by whom



Class Diagram: Car Sale Elements

Workers and Entities



Requirements

The goal of the requirements workflow is to describe what the system should do by specifying its functionality. Requirements modeling allows the developers and the customer to agree on that description

- **Use Case Model** examines the system functionality from the perspective of actors and use cases.
  - **Actors**: an actor is someone (user) or some thing (other system) that must interact with the system being developed
  - **Use Cases**: an use case is a pattern of behavior the system exhibits. Each use case is a sequence of related transactions performed by an actor and the system in a dialog.

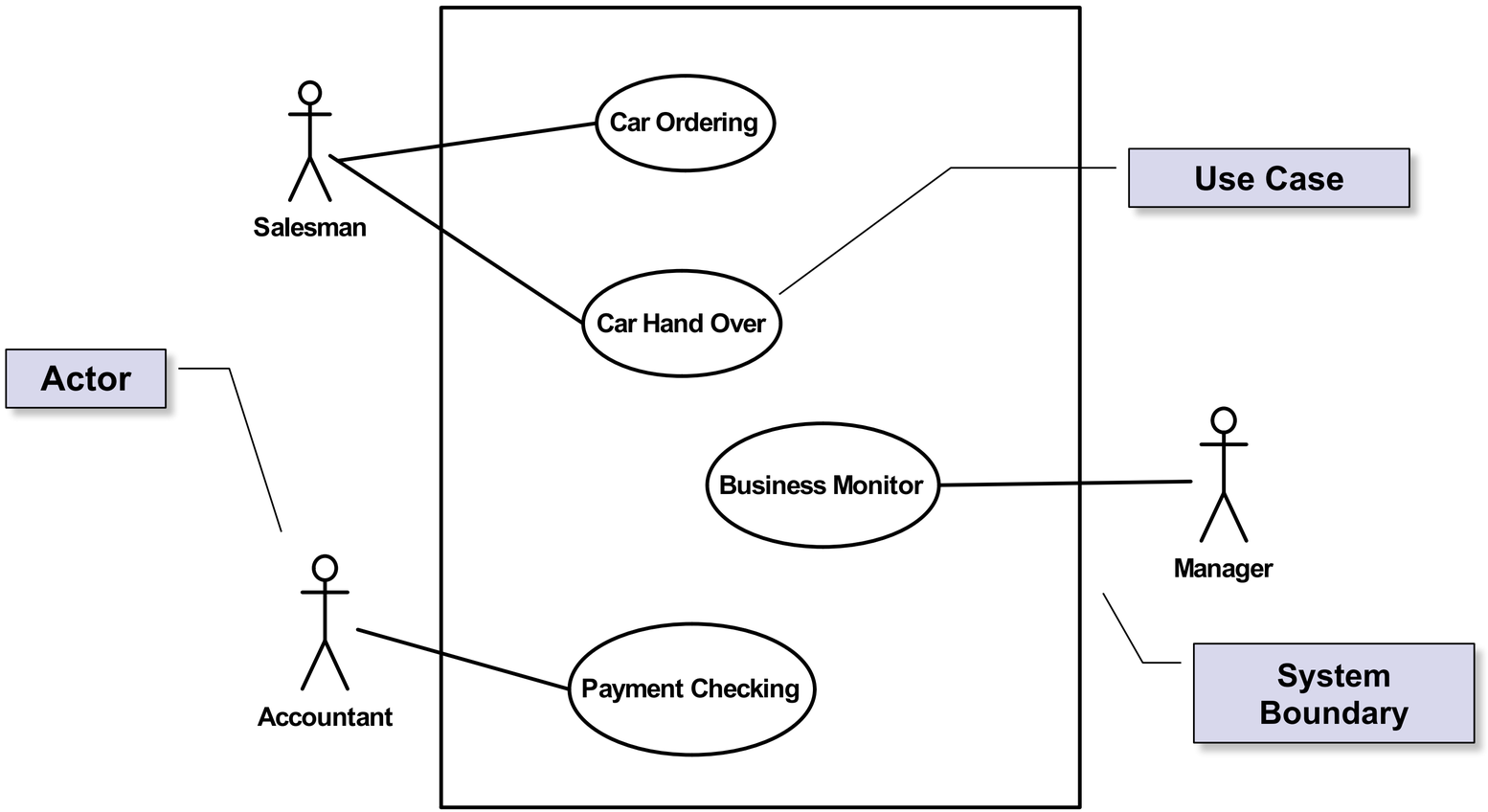
UML Diagrams for Requirements Modeling

Functionality specification

- **Use Case Diagram** shows the relationships among actors and use cases within a system.
  - The purpose of this diagram is to define what exists outside the system (actors) and what should be performed by the system (use cases).
- **Activity Diagram** displays transactions being executed by actor and system in their mutual interaction.
  - The purpose of this diagram is to elaborate functionality of the system specified in a use case diagram.

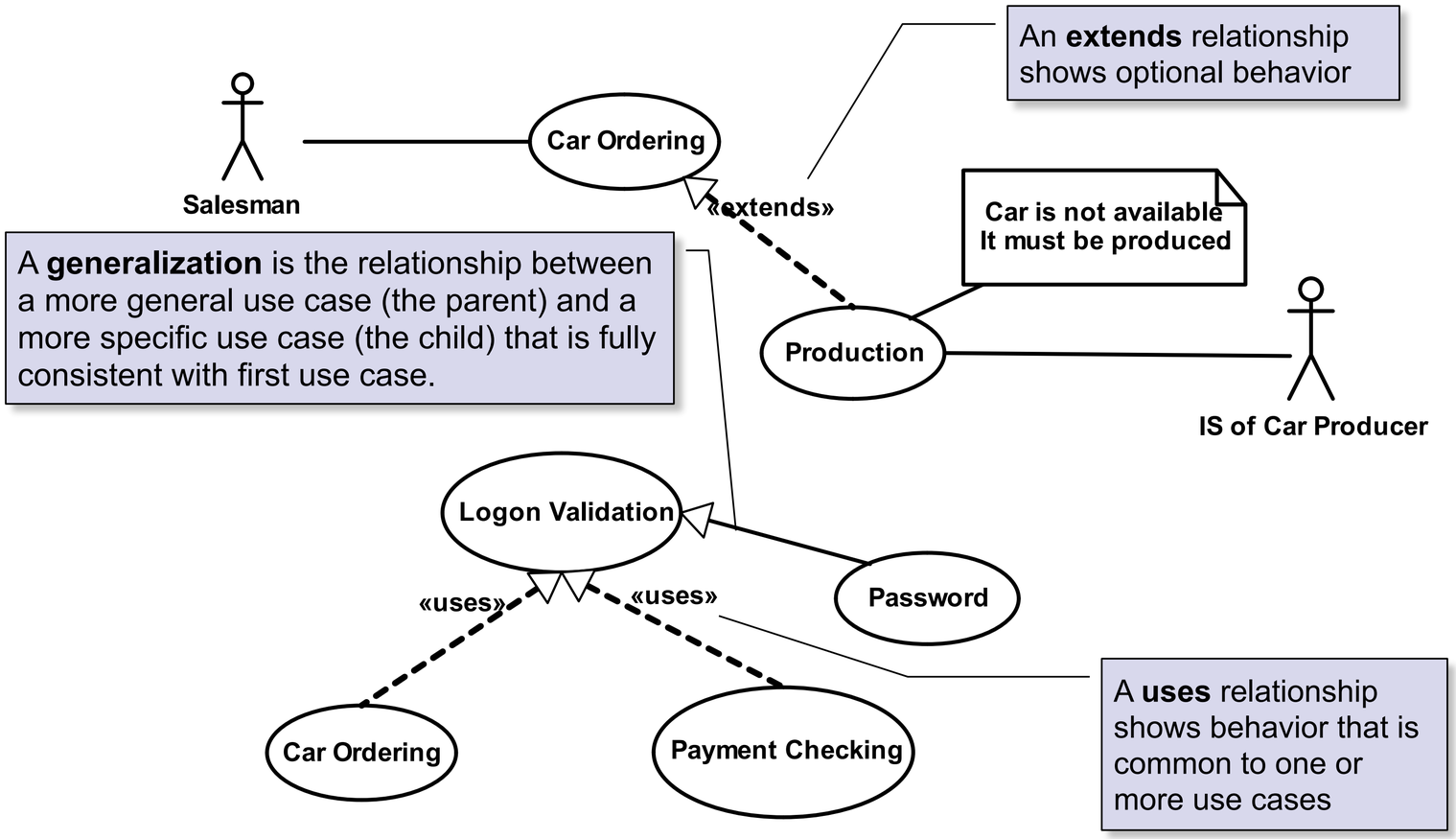
Use Case Diagram: Car Sale

Black box approach



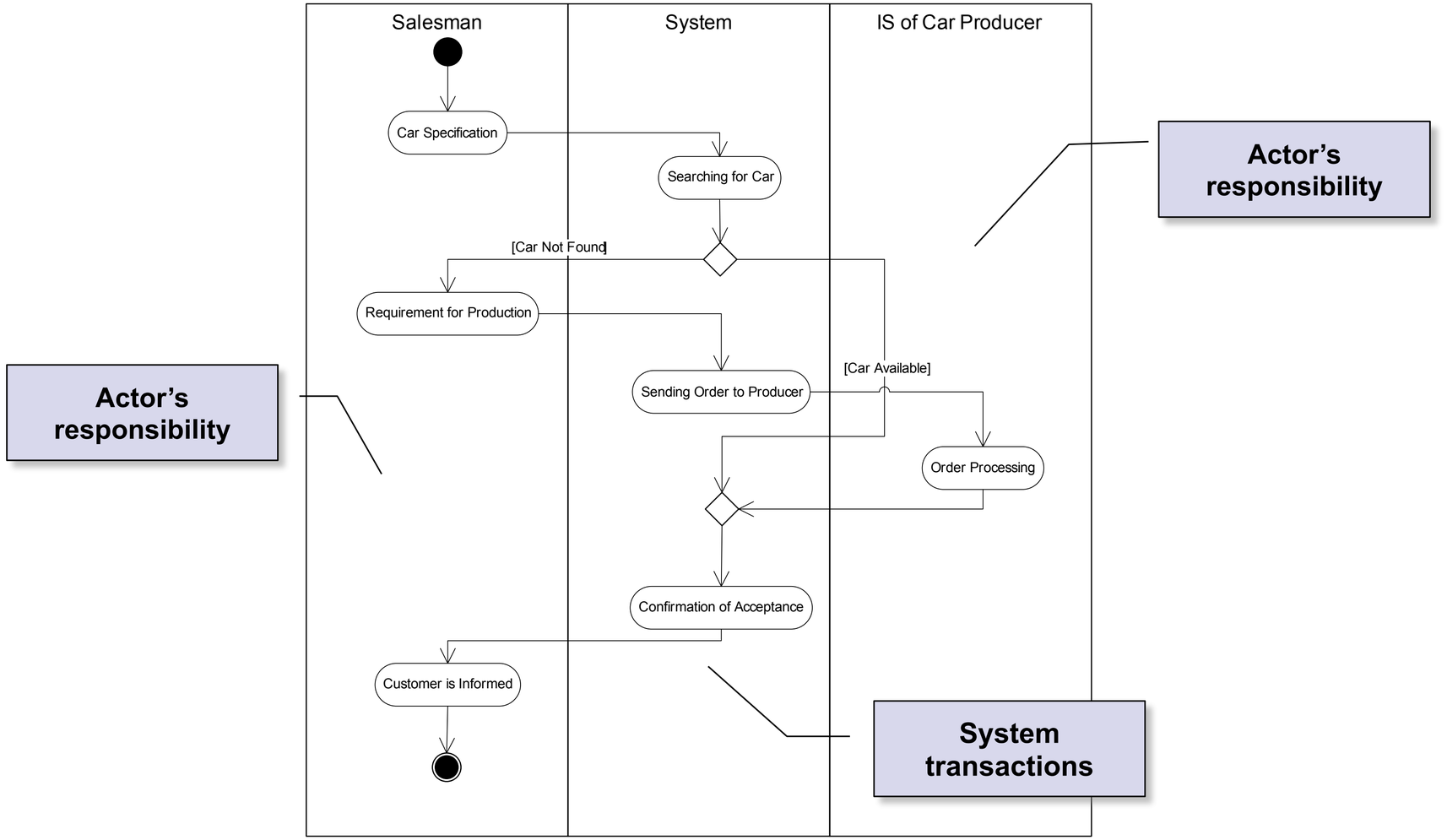
Structuring Use Cases

Relationships among use-cases



Elaborate Functionality of Car Ordering

White box approach



Analysis & Design

The goal of the analysis & design workflow is to show how the system will be realized in the implementation phase.

- **Analysis Model** examines requirements from the perspective of objects found in the vocabulary of the problem domain.

- **Design Model** will further refine the analysis model in light of the actual implementation environment. The design model serves as an abstraction of the source code; that is, the design model acts as a '*blueprint*' of how the source code is structured and written.

- **Deployment Model** establishes the hardware topology on which the system is executed.

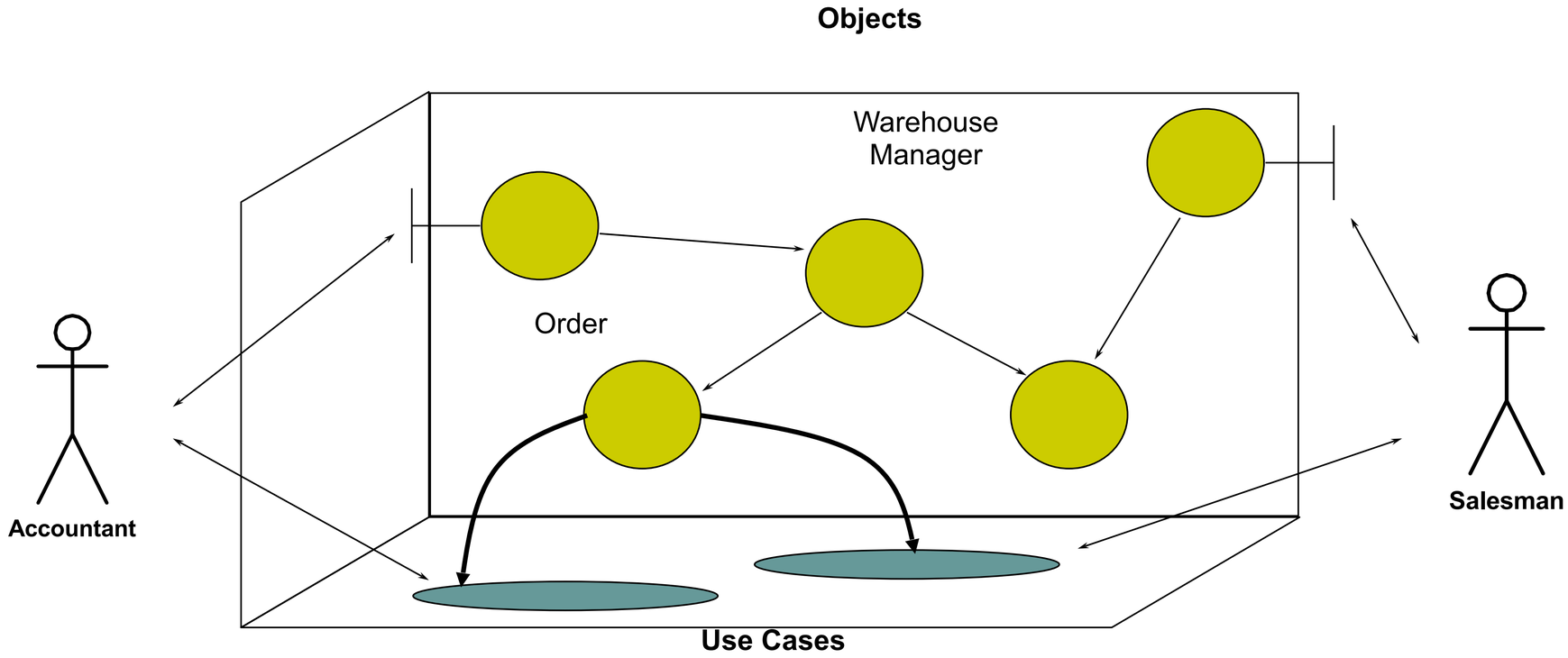
UML Diagrams for Analysis & Design

A lot of work has to be done and this is not always welcome 🤷

- **Class Diagram** shows set of classes, interfaces and their relationships
  - Class diagrams address the **static design view** of the system.
- **Sequence Diagram** is an interaction diagram that emphasizes the time ordering of messages.
- **Collaboration Diagram** displays object interactions organized around objects and their links to one another
  - An interaction diagram that emphasizes the structural **organization of objects that send and receive messages**.
- **Statechart Diagram** shows the life history of a given class and the events that cause a transition from one state to another
- **Deployment Diagram** shows the configuration of run-time processing elements
  - The purpose of this diagram is to **model the topology of hardware** which the system executes.

Use Cases and Objects

Objects are enablers of the sequence of transactions required by the use case. Use cases and objects are different views of the same system. An object can therefore typically participate in several use cases.



What is an Object?

object-oriented program = objects + messages

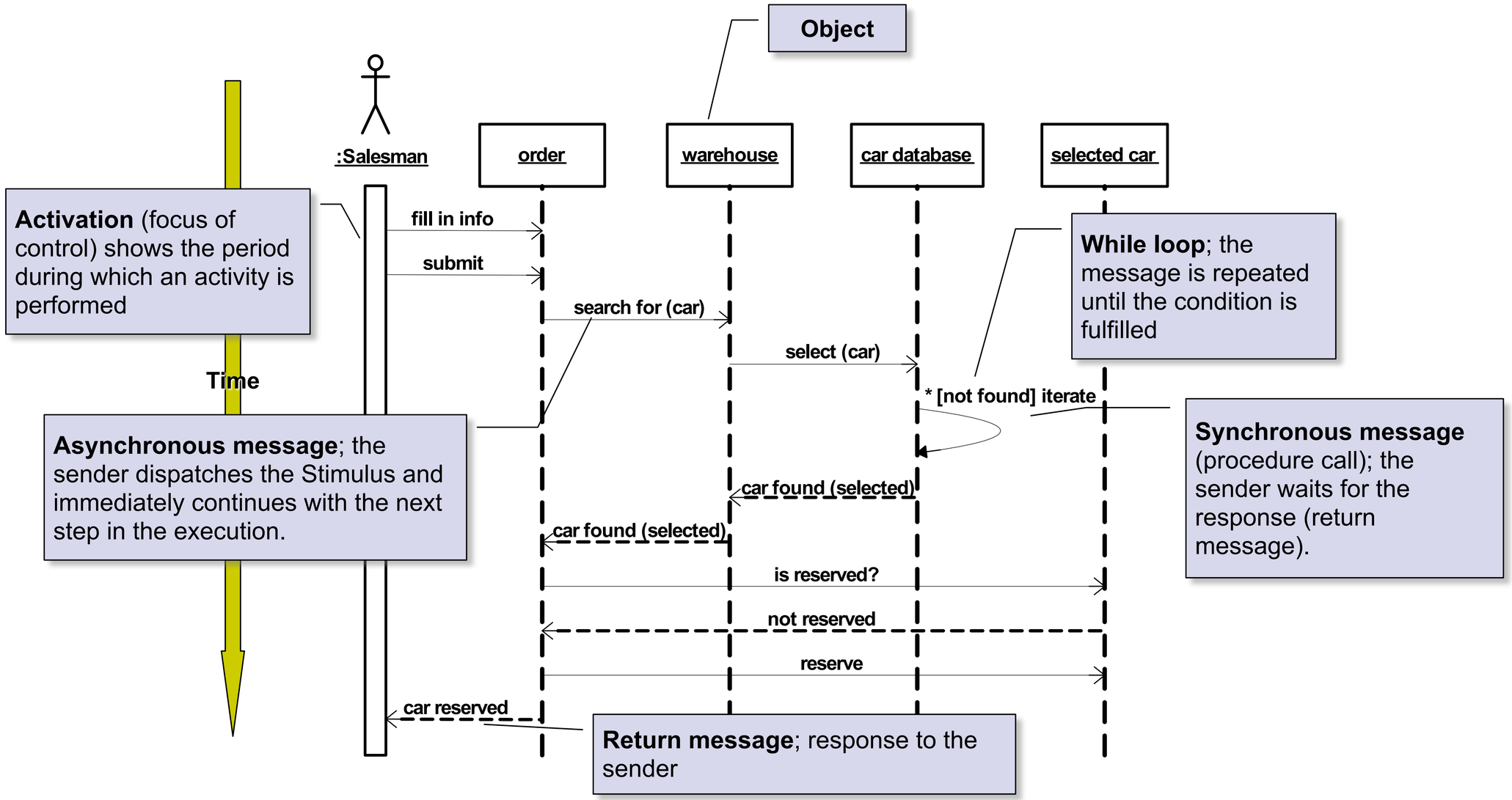
- **An Object is an identifiable individual entity with:**
  - **Identity:** a uniqueness which distinguishes it from all other objects
  - **Behavior:** services it provides in interactions with other objects
- **Secondary properties:**
  - **Attributes:** some of which may change with time
  - **Lifetime:** from creation of the object to its destruction
  - **States:** reflecting different phases in the object's lifecycle

Objects and Their Interactions

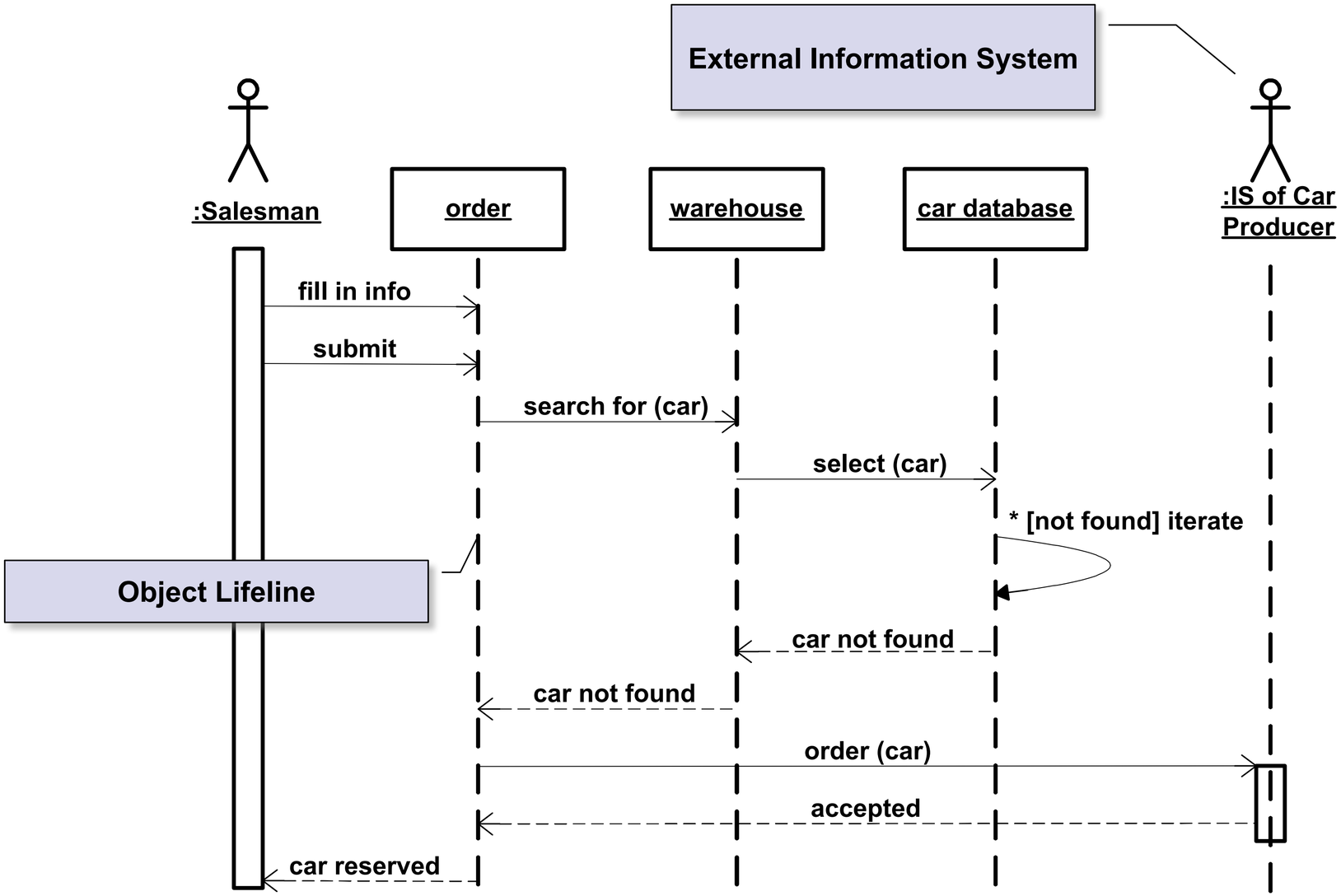
Links and interactions

- **Links:** A link is a physical or conceptual connection between objects (*John Smith* ***works-for*** *Simplex company*). Mathematically, a link is defined as a tuple, that is, an ordered list of objects.
- **A set of interconnected objects constitutes the system**
- **Interactions among objects result in:**
  - Collective behaviors being exercised
  - Changes in the logical configurations and states of the objects and system

Sequence Diagram: Car Ordering



Alternative Scenario

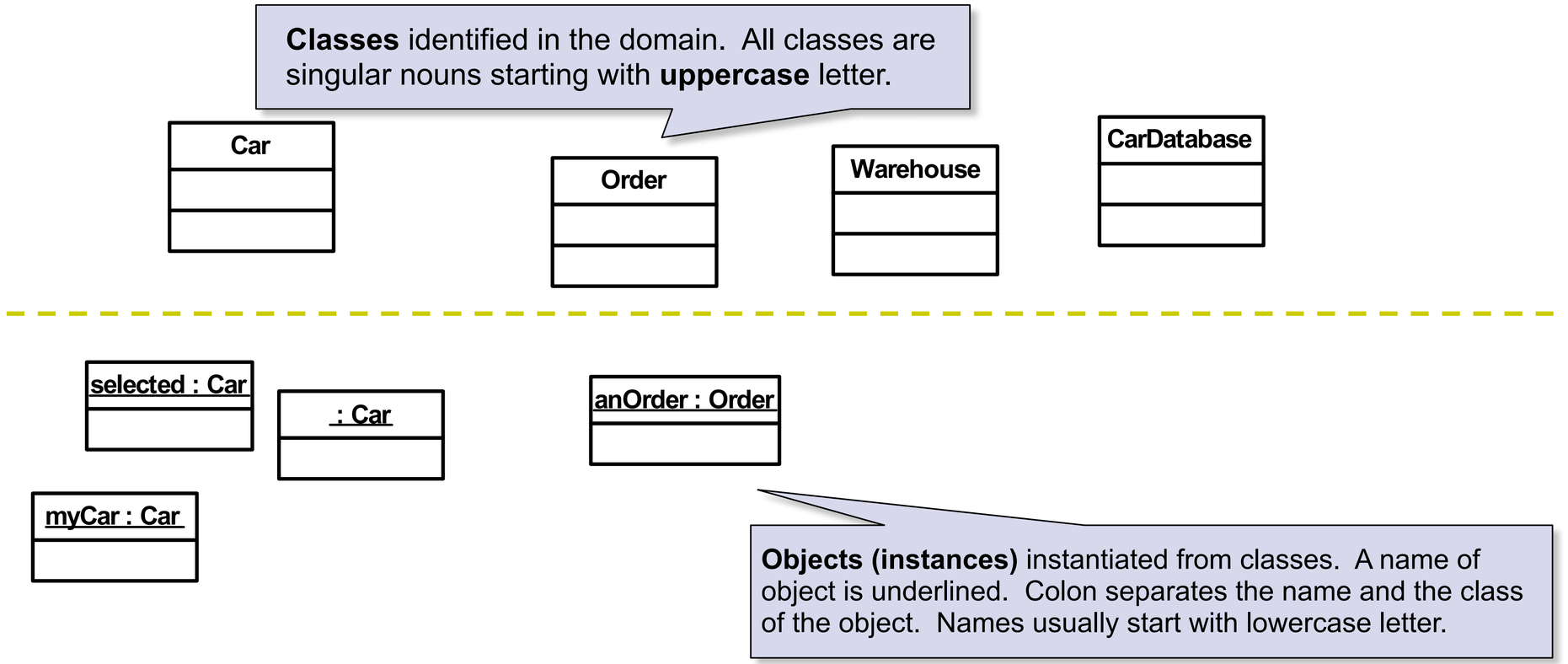


What is a Class?

A class is the descriptor for a set of objects with common structure, behavior, relationships and semantics.

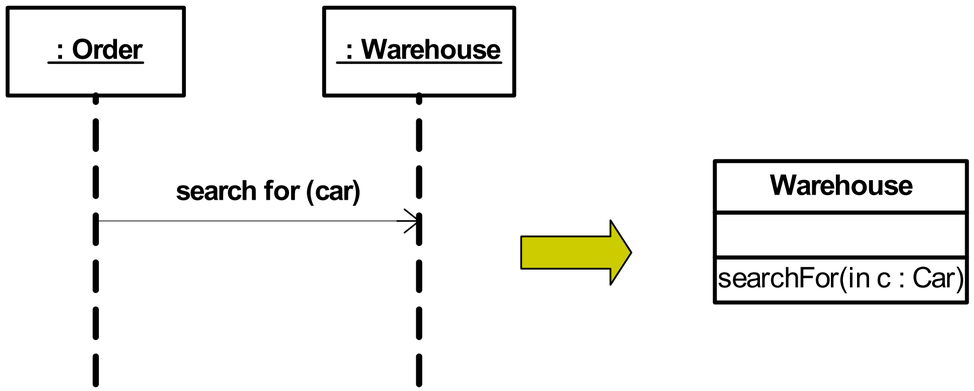
- Classes are found by examining the objects in sequence diagrams.
- Every object is an instance of one class.
- Classes should be named using the vocabulary of the domain.

Objects and Classes



Operations

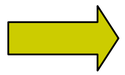
- The **behavior** of a class is represented by its operations.
- **Operation** may be found **by examining interaction diagrams**

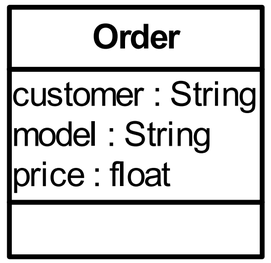


Attributes

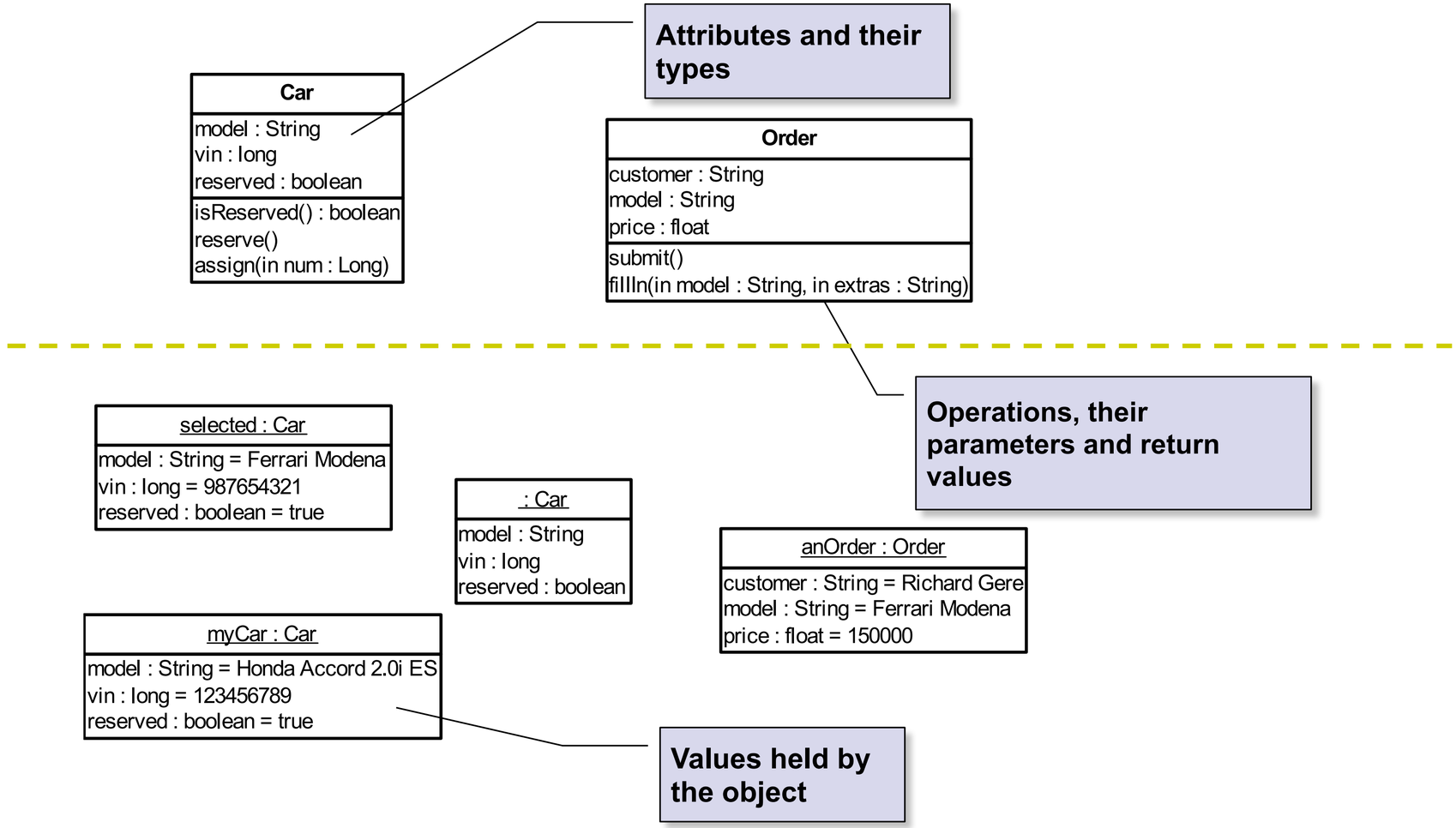
- The **structure** of a class is represented by its **attributes**.
- Attributes may be found by examining class definitions, the problem requirements, and applying domain knowledge.

A **customer** has chosen a **model** of the car and has to pay for it some **price**.





Operations and Attributes

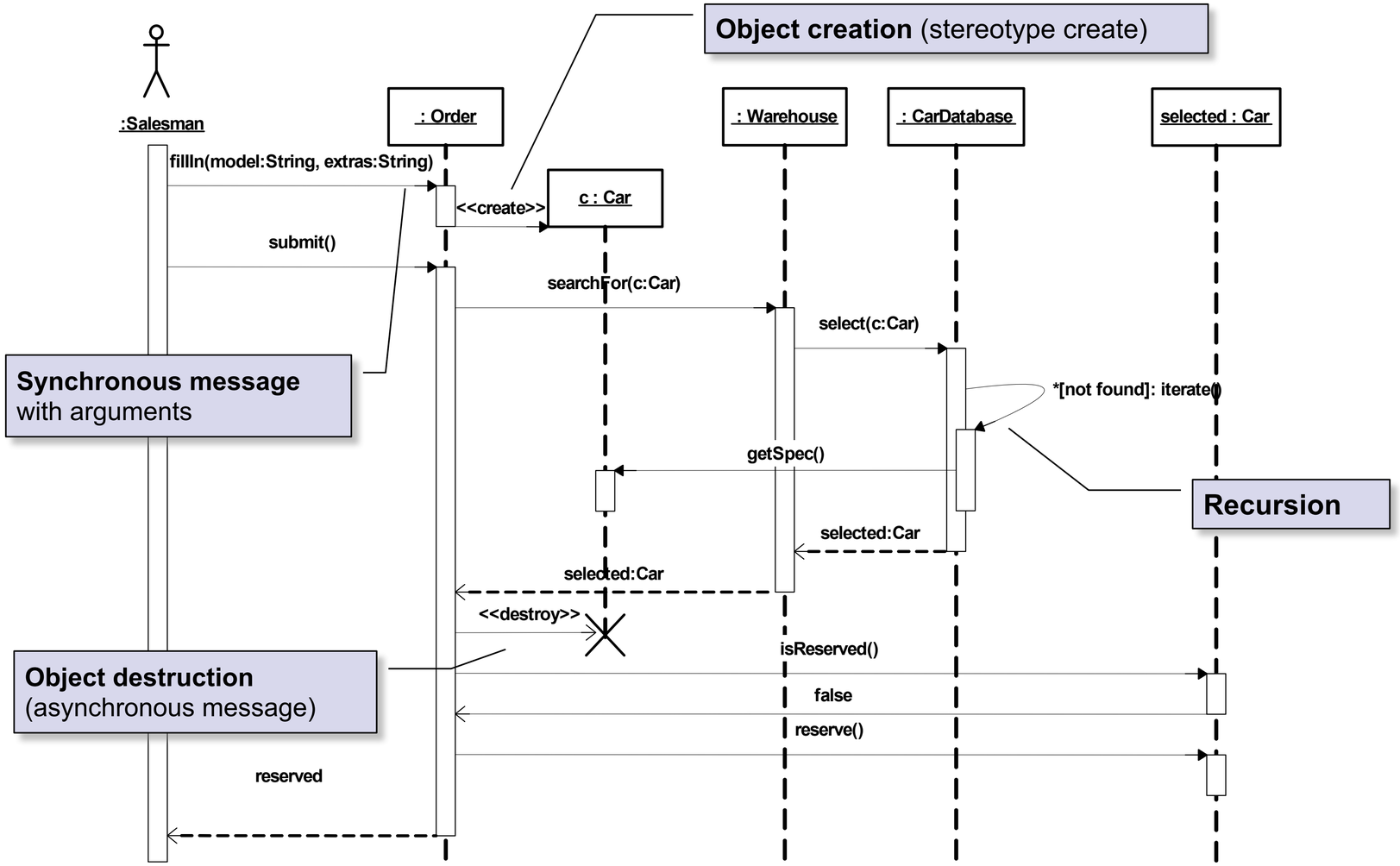


Relationships

Relationships between classes specify a way for communication between objects.

- Sequence and/or collaboration diagrams are examined to determine what links between objects need to exist to accomplish required behavior. **Objects can send messages to each other only if the link between them is established.**
  - An **sequence diagram** emphasizes the time ordering of messages.
  - An **collaboration diagram** emphasizes the structural organization of objects that send and receive messages.

Refined Sequence Diagram



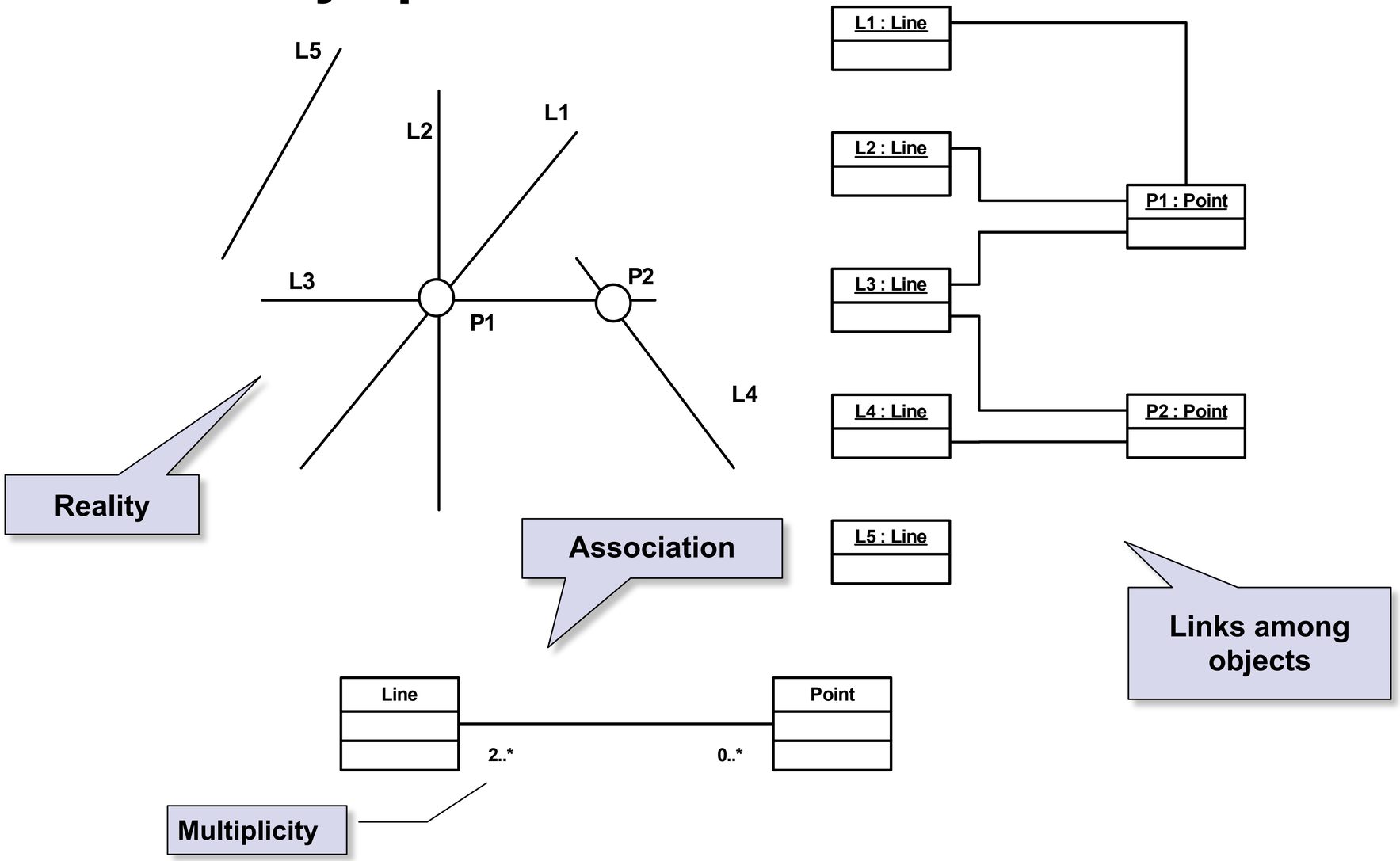
Types of Relationships

Even links have their „classes“

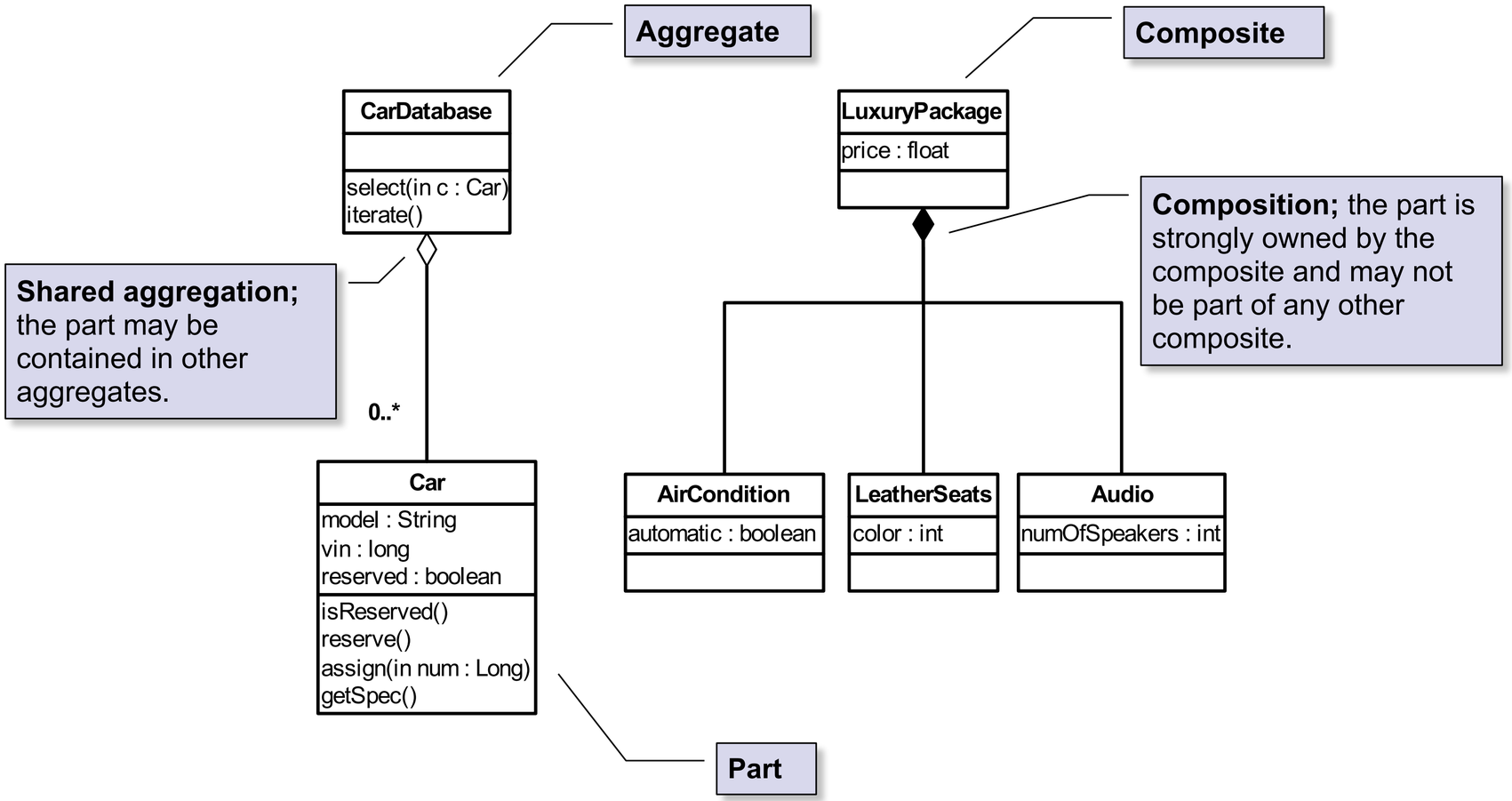
- **Association** describes a group of links with common structure and common semantics (a Person works-for a Company). An association a **bi-directional** connection between classes that describes a set of potential links in the same way that a class describes a set of potential objects.
- **Aggregation** is the “part-whole” or “a-part-of” relationship in which objects representing the **components** of something are associated with an object representing the entire **assembly**.
- **Dependency** is a weaker form of relationship showing a relationship between a **client** and **supplier**.
- **Generalization** is the taxonomic relationship between a more **general element** (the parent) and a **more specific element** (the child) that is **fully consistent** with the first element and that adds additional information.

Links and Associations

Abstraction of many options …

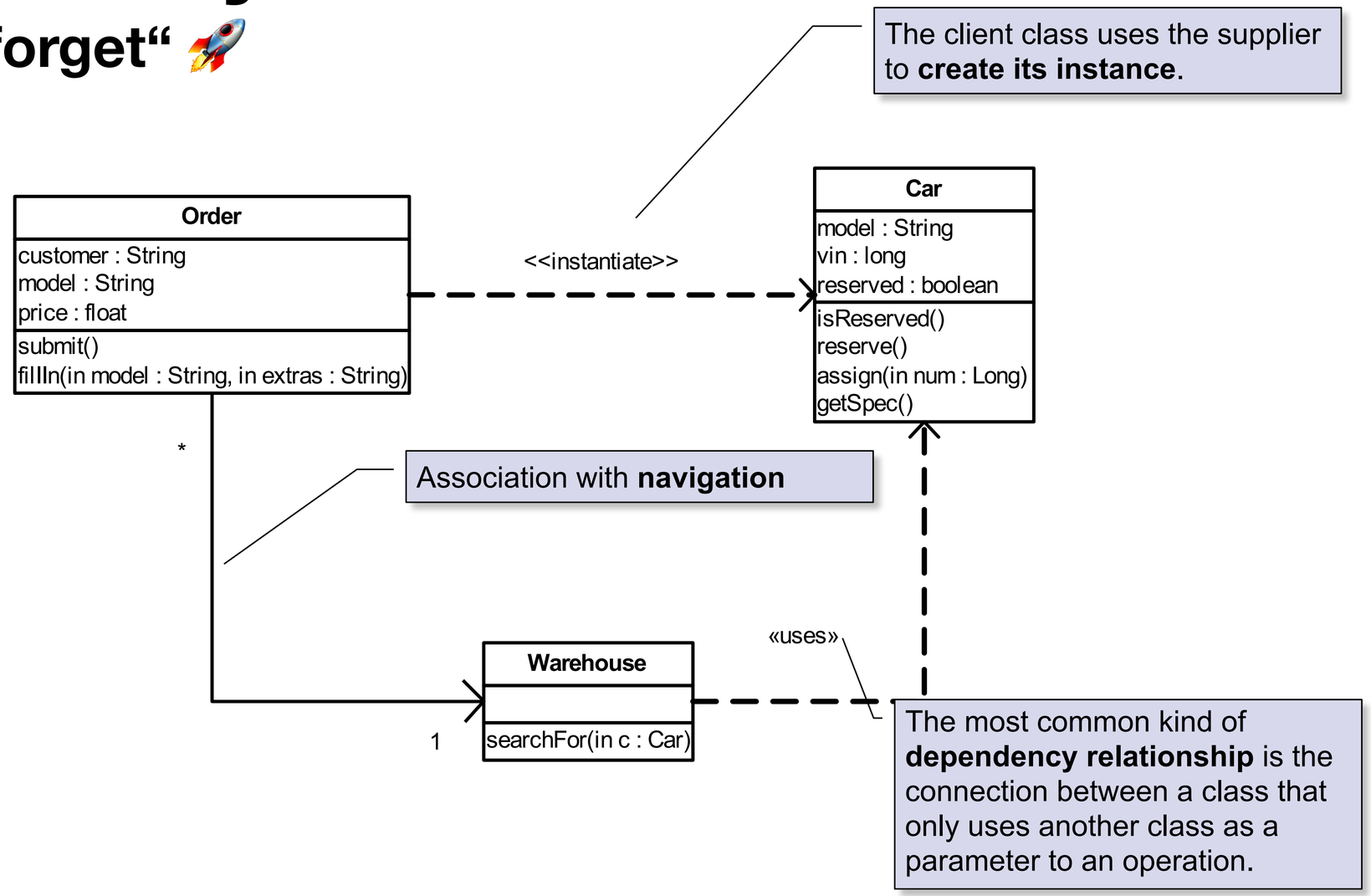


Aggregations



Dependency

„Fire and forget“ 🚀

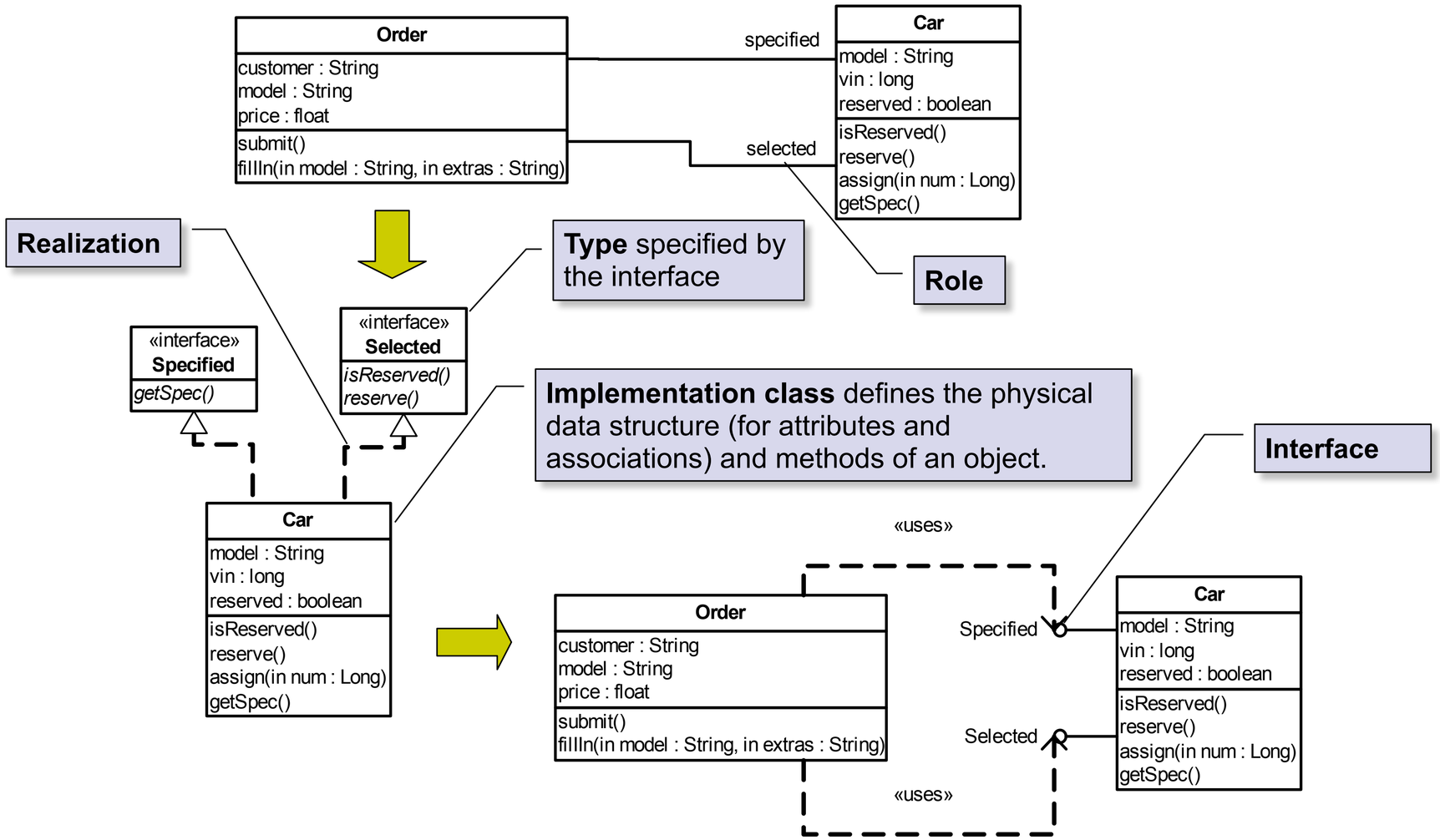


Roles, Types and Interfaces

Let’s make it a little bit complex but much more flexible 🤔

- **Role** is the named specific behavior of an object participating in a particular context (the face it presents to the world at a given moment).
- **Type** specifies a domain of objects together with the operations applicable to the objects (without defining the physical implementation of those objects).
- **Interface** is the named set of externally-visible operations.
- Notions of **role, type and interface are interchangeable**.

Type and Implementation Class

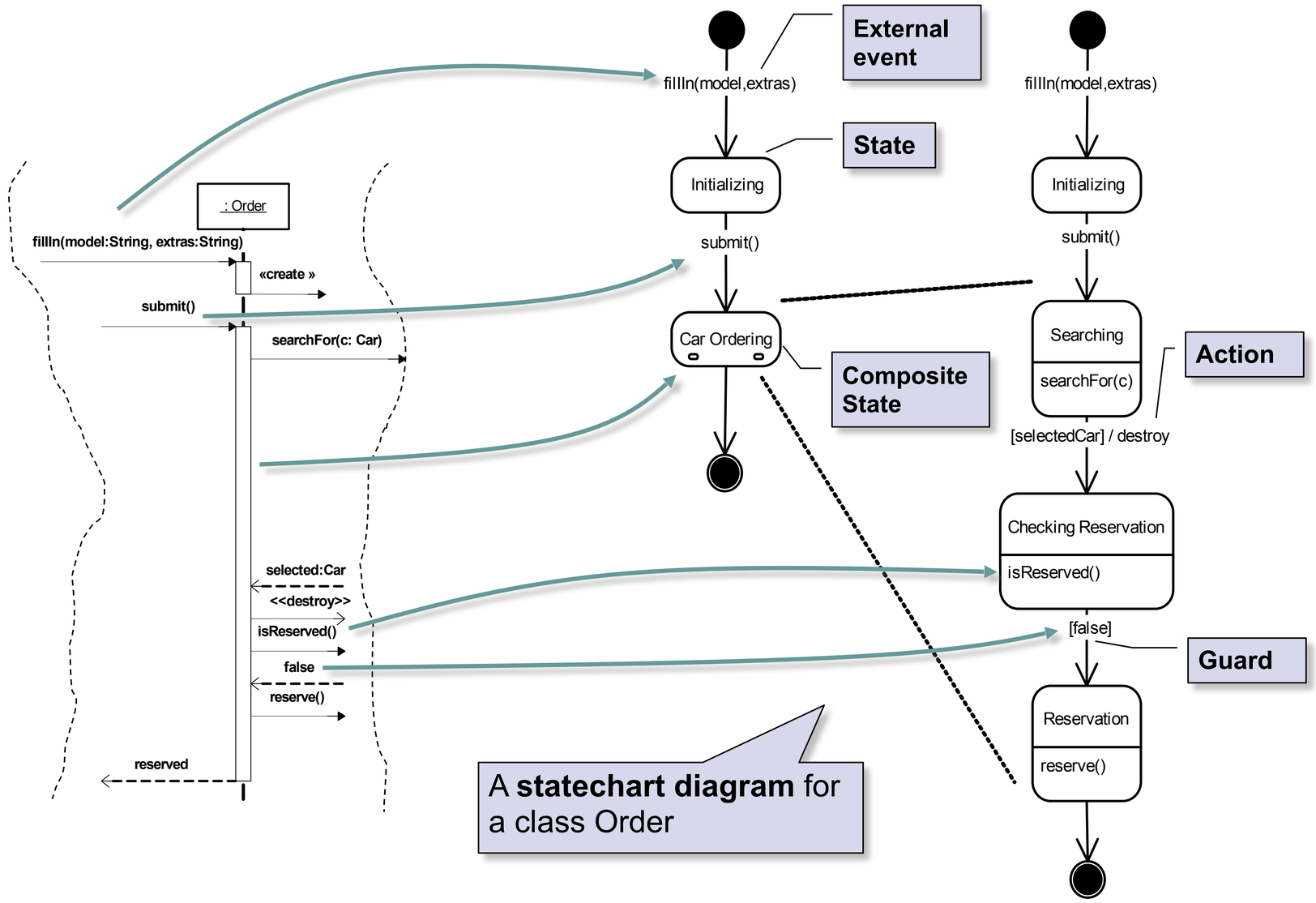


The Dynamic Behavior of an Object

To execute our app objects must behave 😉

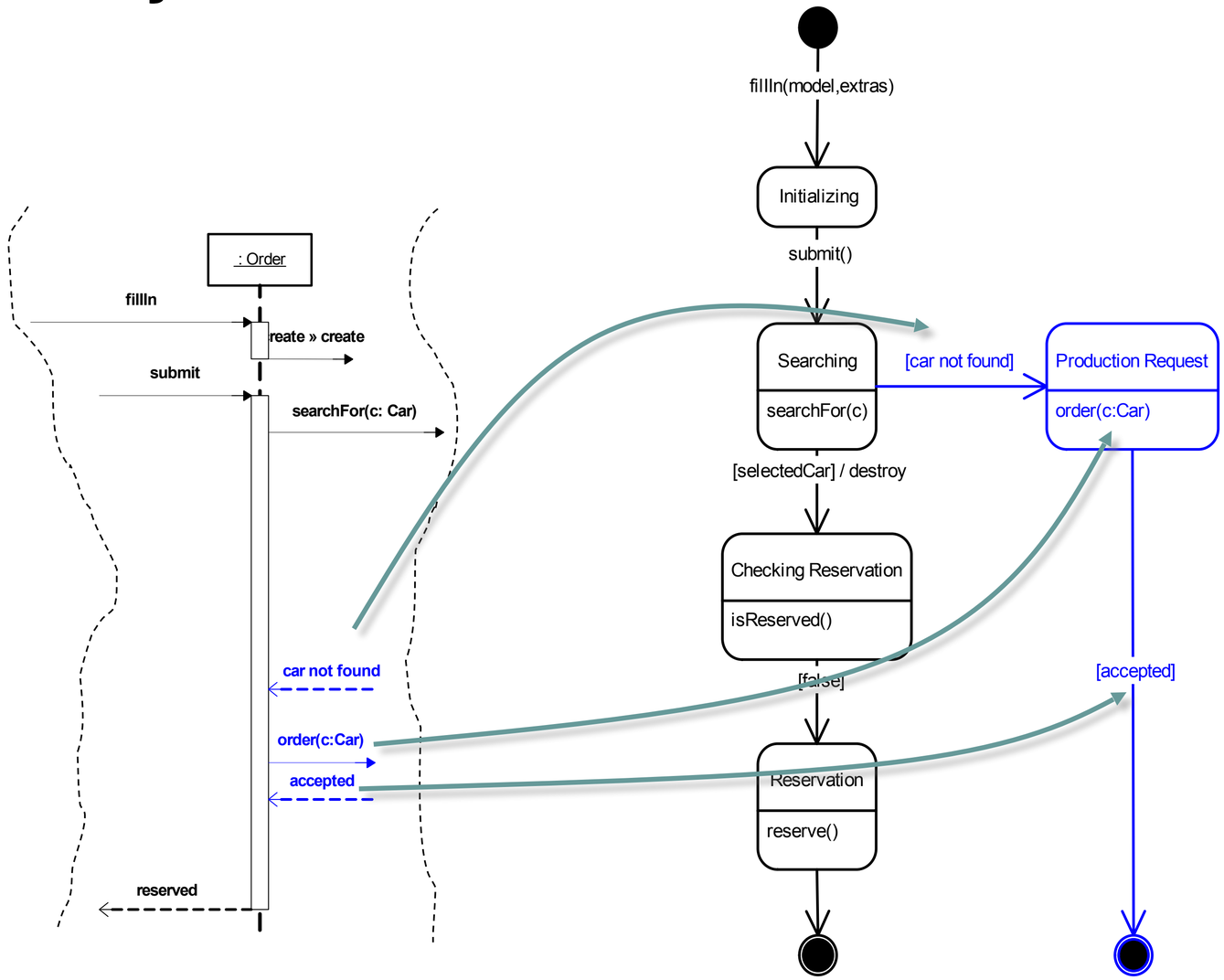
- A **state transition** (statechart) diagram shows
  - The **life cycle** of a given object
  - The **events** causing a transition from one state to another
  - The **actions** that result from a state change
- State transition diagrams are created for objects with **significant dynamic behavior**
- Sequence and/or collaboration diagrams are examined to define statechart diagram of a class

Statechart Diagram



Merging Scenarios

Lifecycle of the object/instance of the class

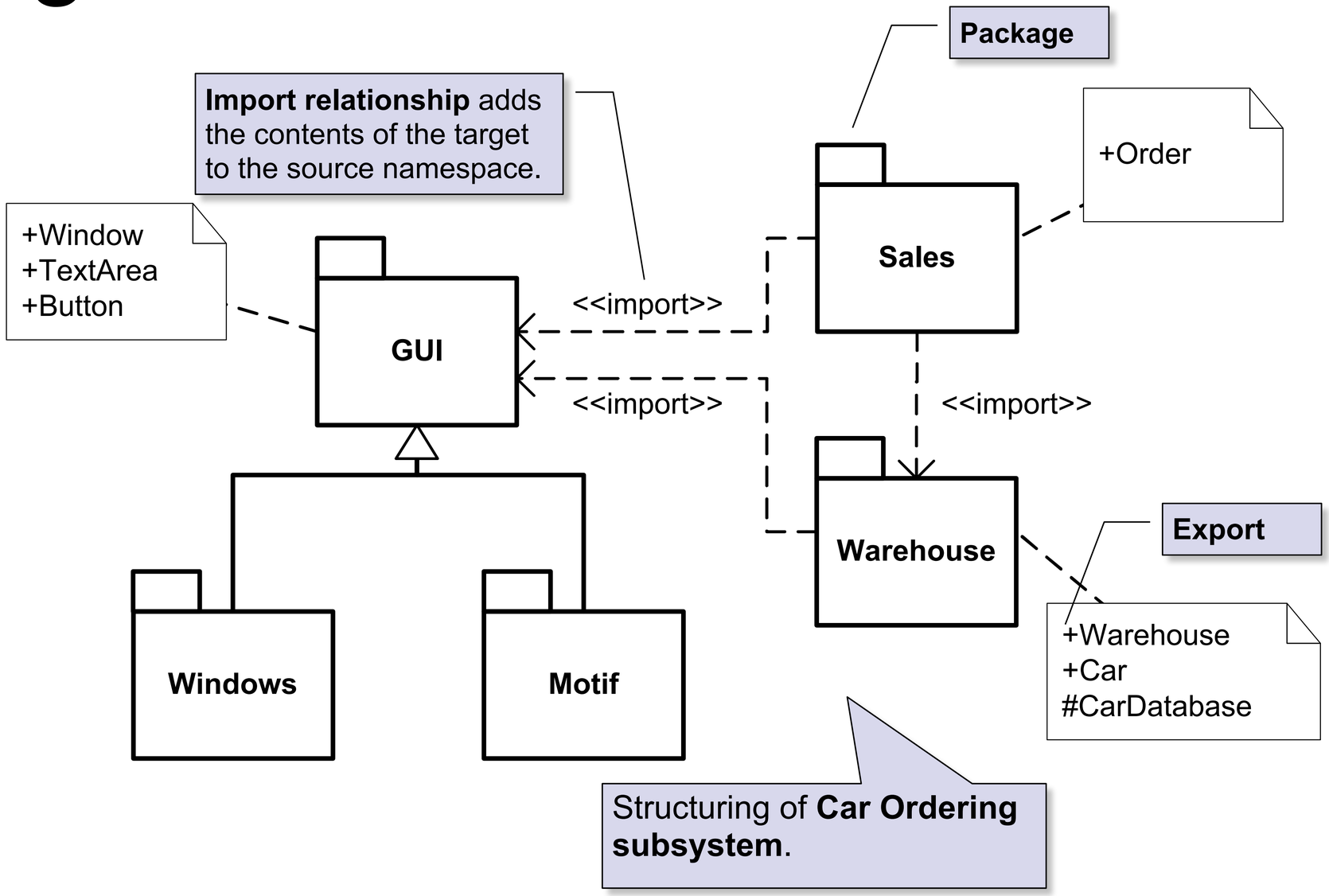


Model Management Overview

How to organize the system

- A **package** is a general-purpose mechanism for organizing elements into groups
  - to manage complexity of modeling structures
- A **subsystem** is a grouping of model elements that represents a behavioral unit in a physical (real) system.
  - to describe the services offered by the subsystem
  - to describe the interface of the subsystem
- A **model** is a simplification of reality, an abstraction of a system, created in order to better understand the system
  - A partitioning of the abstraction that visualize, specify, construct and document that system
- **Subsystems and Models are special cases of Package**

Packages


Package
Import relationship adds the contents of the target to the source namespace.
+Order
+Window
+TextArea
+Button
Sales
GUI
<<import>>
<<import>>
<<import>>
Warehouse
Export
+Warehouse
+Car
#CarDatabase
Windows
Motif
Structuring of Car Ordering subsystem.

Design

The design model will further refine the analysis model in light of the actual implementation environment.

- In analysis **domain objects** are the primary focus.
- In design the **other layers** are added and refined
  - User Interface
  - Distribution
  - Persistence Mechanism

Goal of Design

Mapping of analysis models into a set of software components with precisely defined interactions based on system architecture and already existing components

- Definition of the **system architecture**
- Identifying design **patterns and frameworks**
- Software **components** definition and re-use

Architecture-Centric Development

Abstraction is the key player ☝️🤞

- Models help developers understand and shape both the problem and the solution.
- Model is a simplification of reality that help us master a large, complex system that cannot be comprehended easily in its entirety. The model is not the reality, but the best models are the ones that stick very close to reality.
- Multiple models are needed to address different aspects of the reality. These models must be coordinated to ensure that they are consistent and not too reduntant.

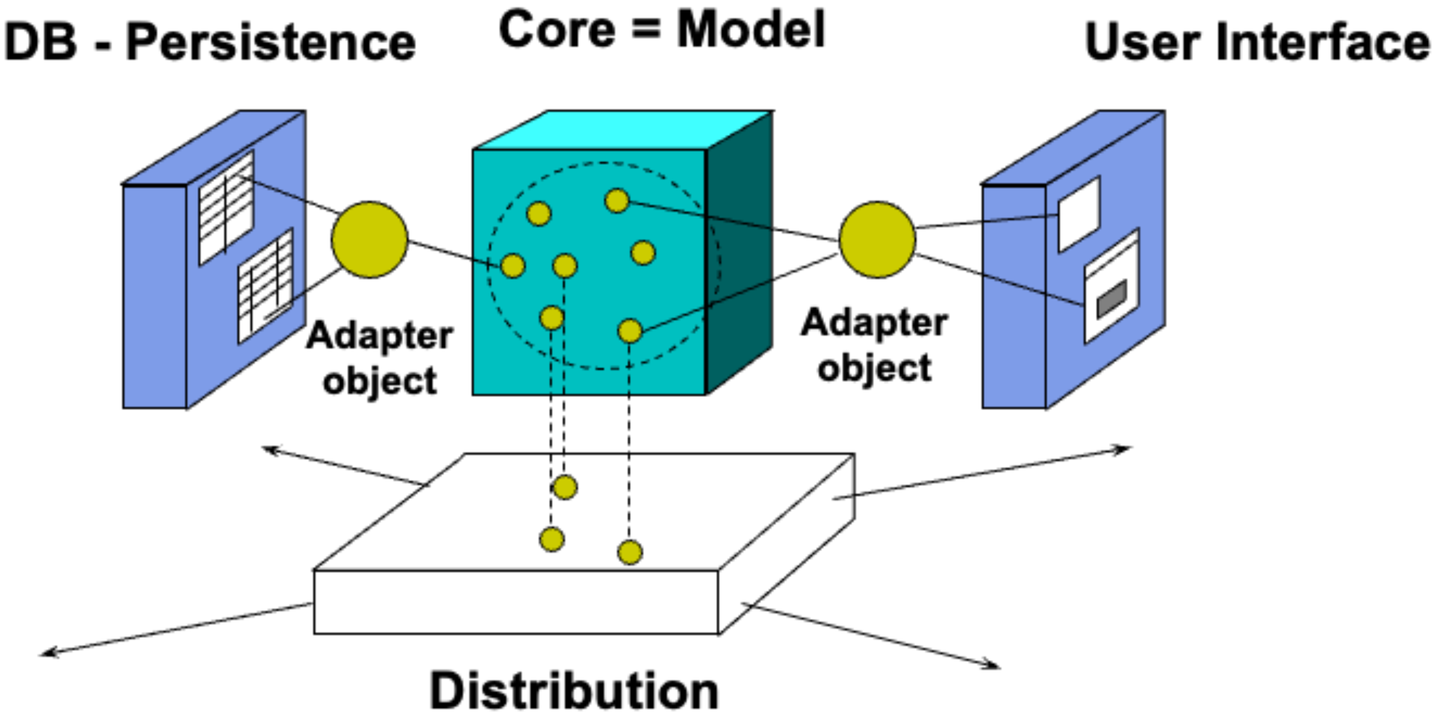
Architecture

Architecture is the skeleton 😐

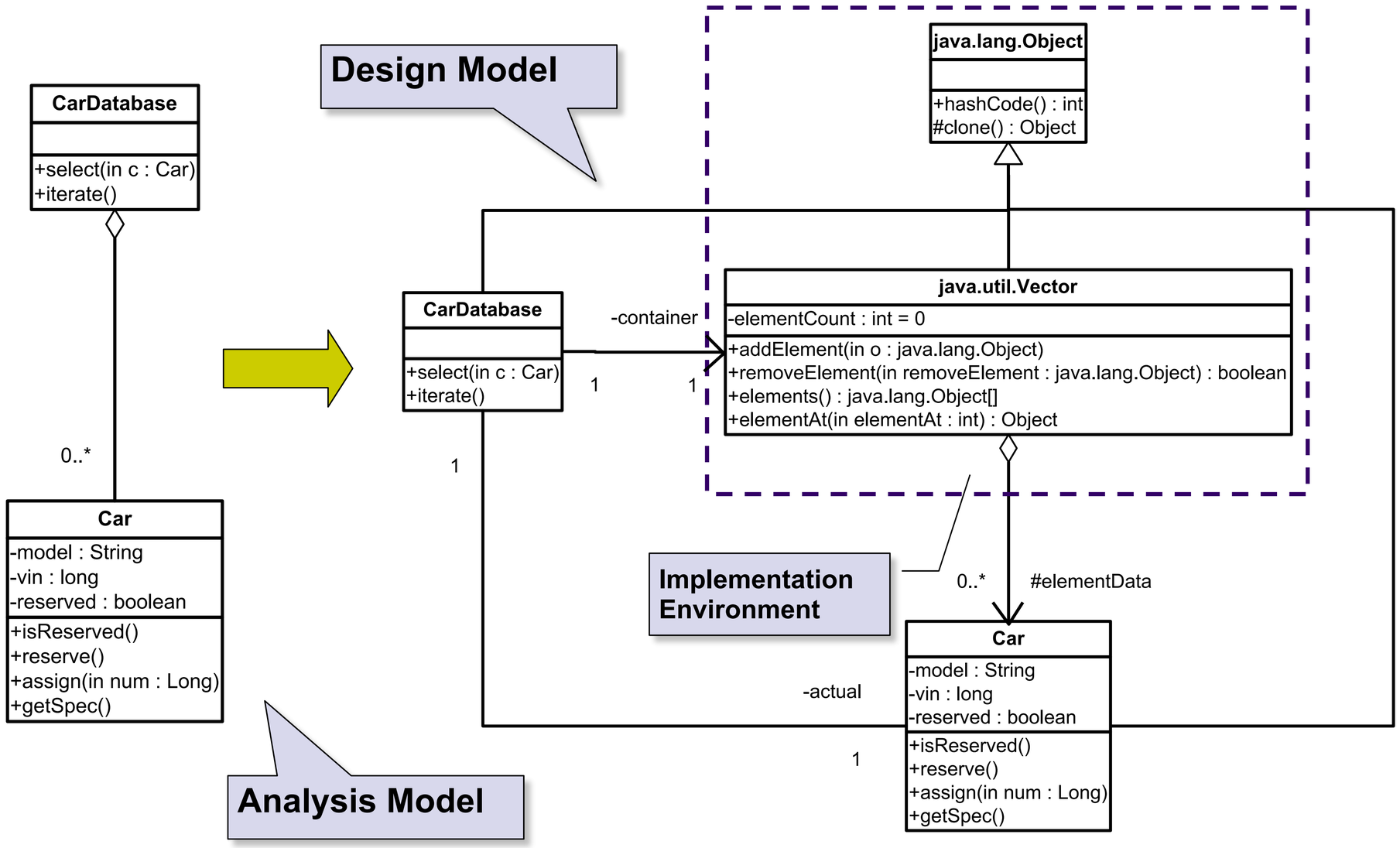
- Architecture is what remains when you cannot take away any more things and still understand the system and explain how it works.
- Architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment

System Architecture

Fundamental organization of any software system



Mapping into Software Components



Design Patterns

Life is all about patterns 🧩

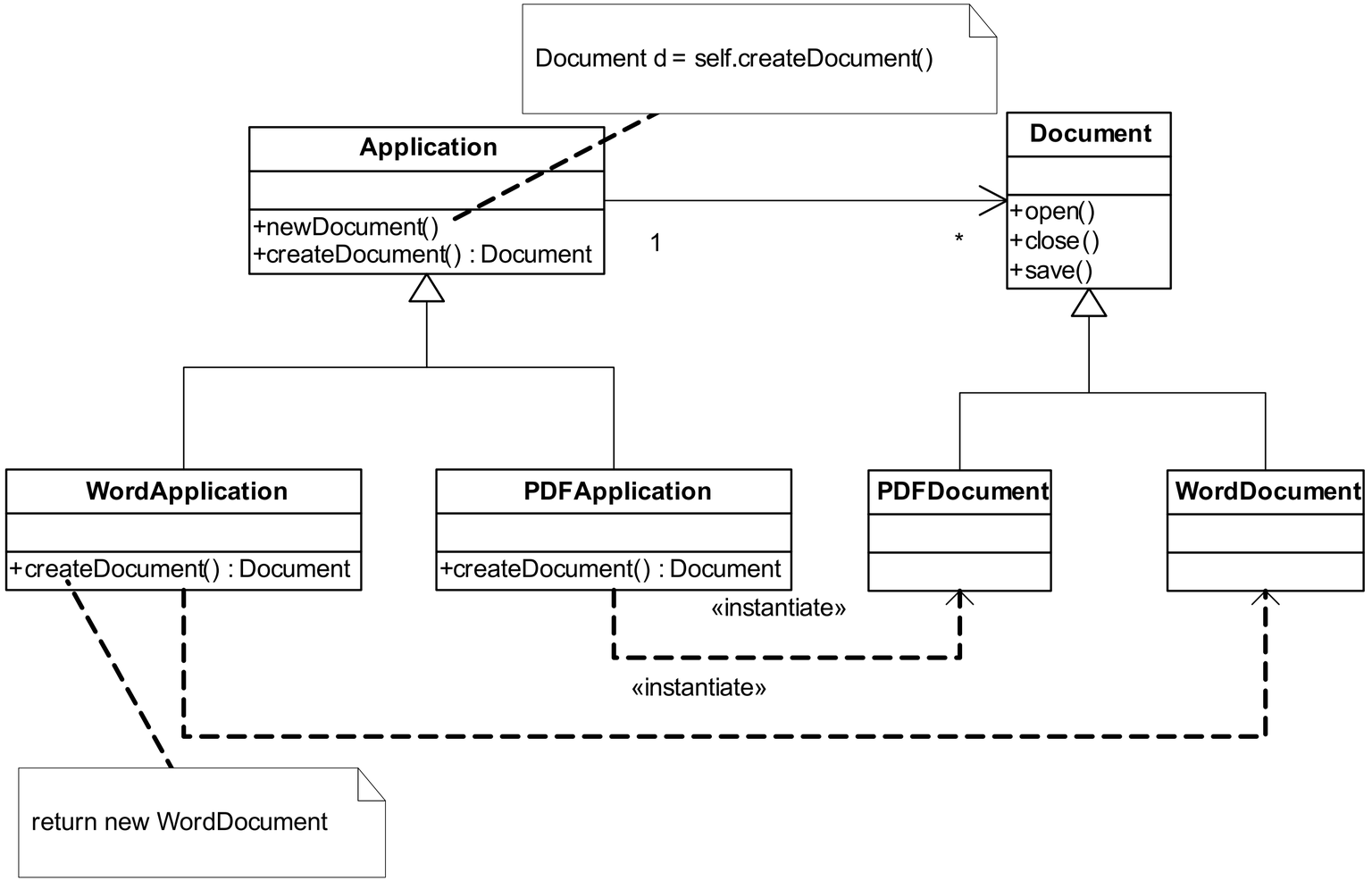
- The design pattern concept can be viewed as an **abstraction of imitating useful parts** of other software products.
- The design pattern is **description of communicating objects and classes** that are customized to solve a general design problem in a particular context.
- Classification of Design Patterns:
  - **Creational patterns** defer some part of object creation to a subclass or another object.
  - **Structural patterns** composes classes or objects.
  - **Behavioral patterns** describe algorithms or cooperation of objects.

Creational Design Patterns

- **Factory Method** define an interface for creating an object, but let subclasses decide which class to instantiate.
- **Factory** provides an interface for creating families of related objects without specifying their concrete classes.
- **Prototype** specifies the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

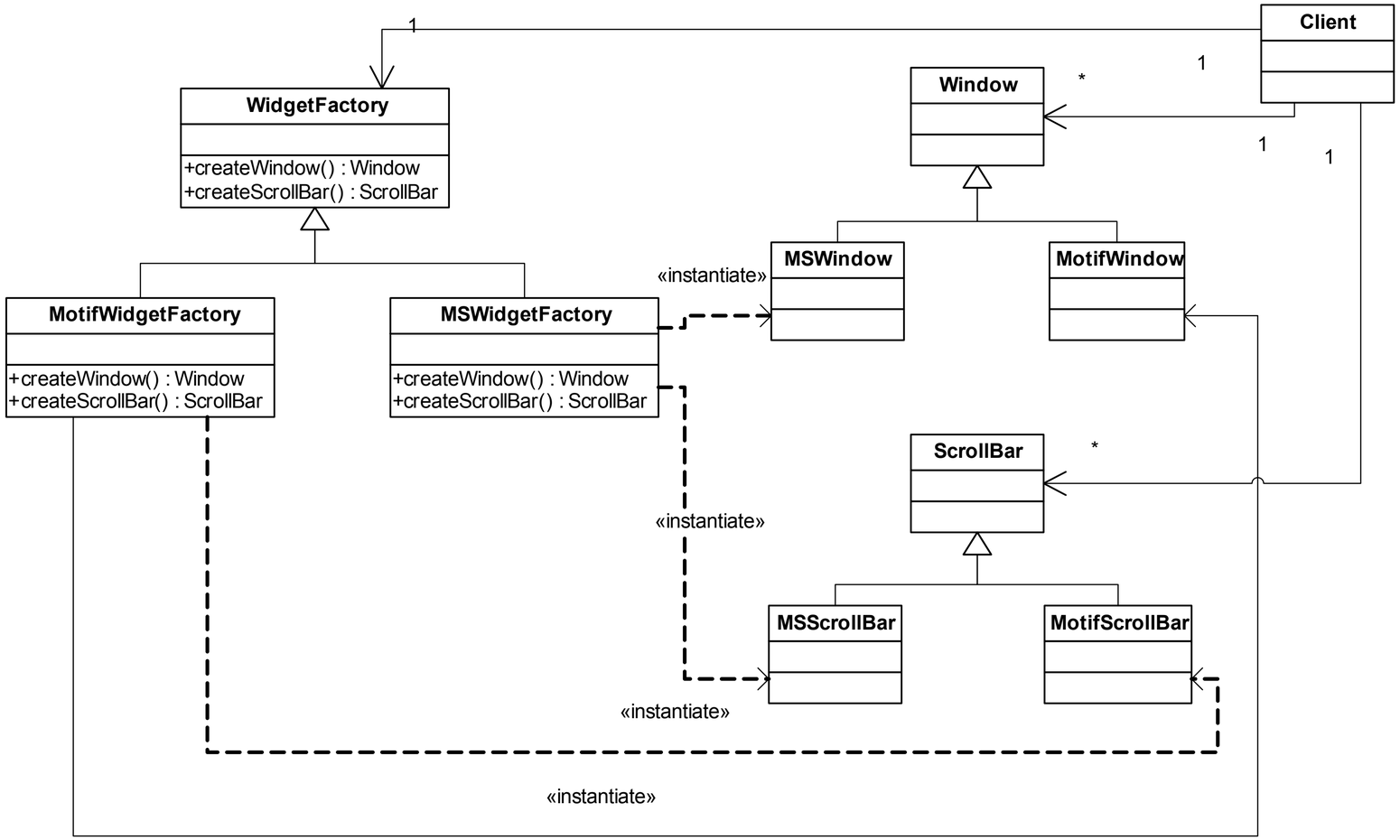
Factory Method

Factory Method define an interface for creating an object, but let subclasses decide which class to instantiate.



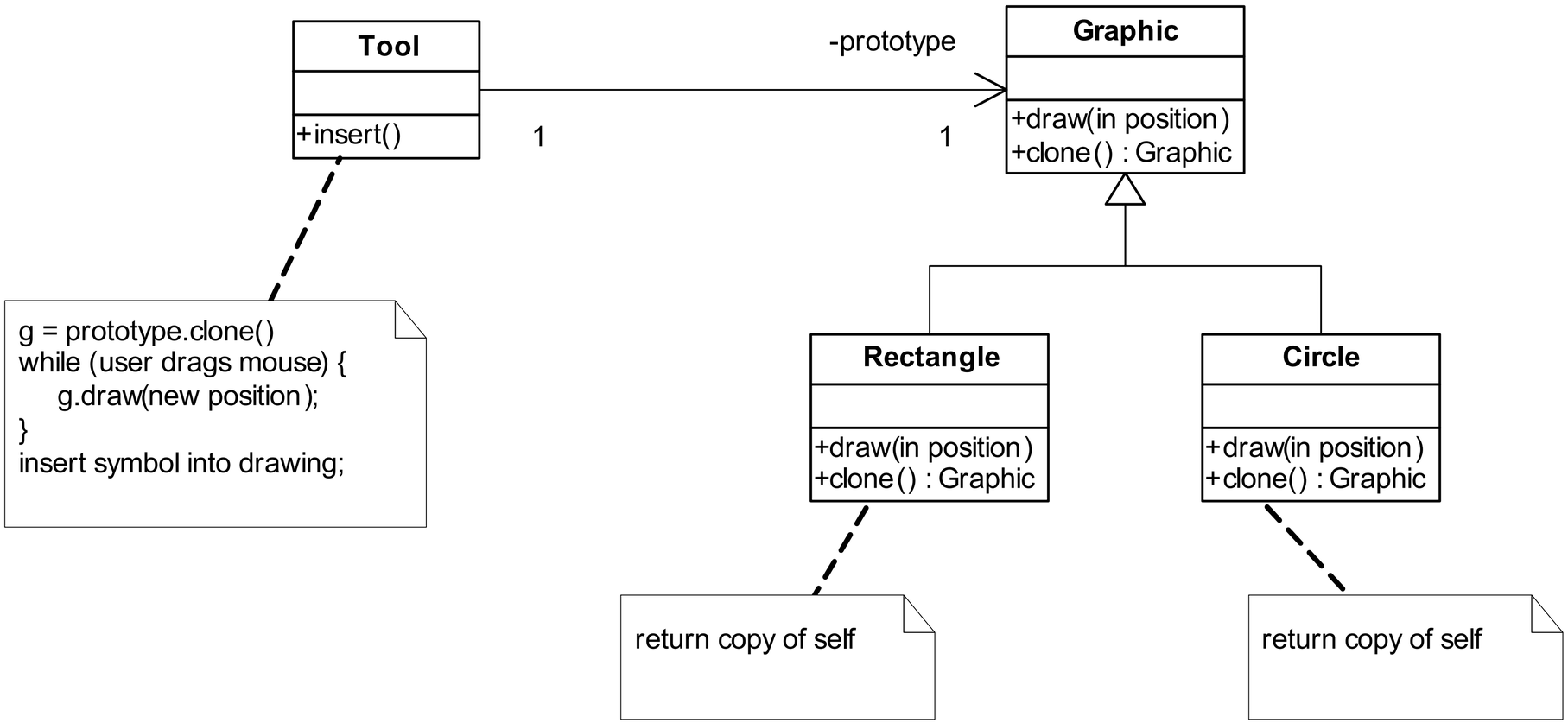
Factory

Factory provides an interface for creating families of related objects without specifying their concrete classes.



Prototype

Prototype specifies the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

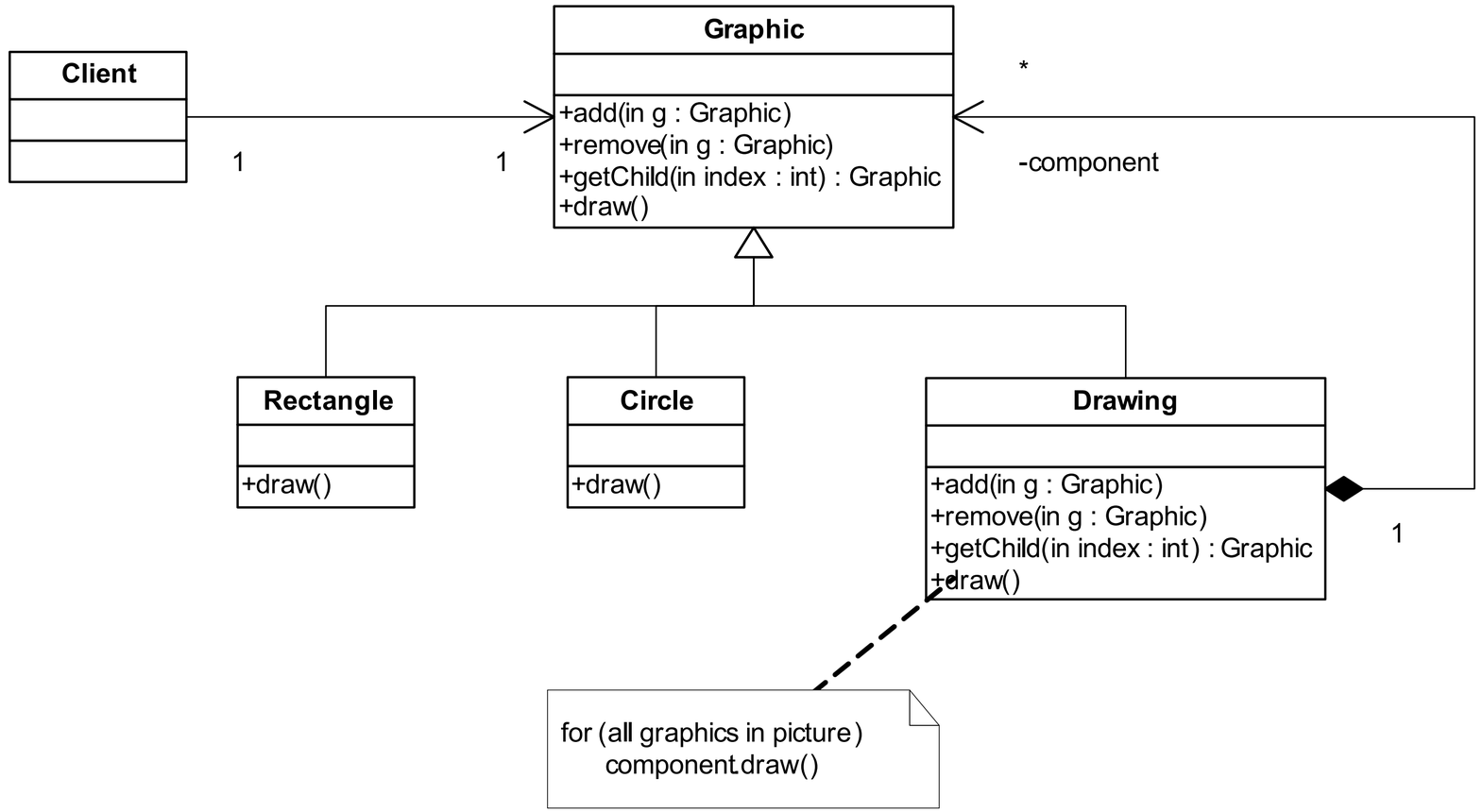


Structural Design Patterns

- **Composite** composes objects into tree structures to represent part-whole hierarchies. Composite lets client treat individual objects and compositions of objects uniformly.
- **Adapter** converts the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.
- **Decorator** attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **Proxy** provides a surrogate or representative for another object to control access to it.

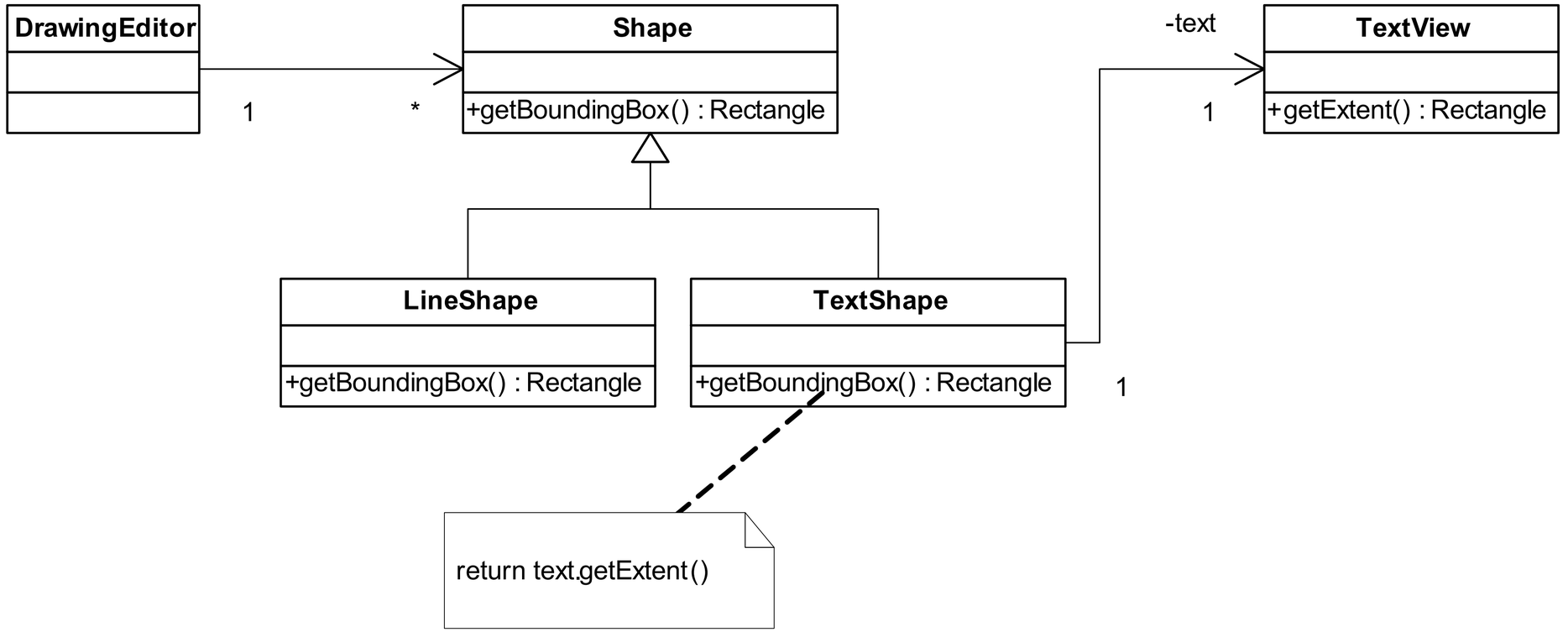
Composite

Composite composes objects into tree structures to represent part-whole hierarchies.
Composite lets client treat individual objects and compositions of objects uniformly.



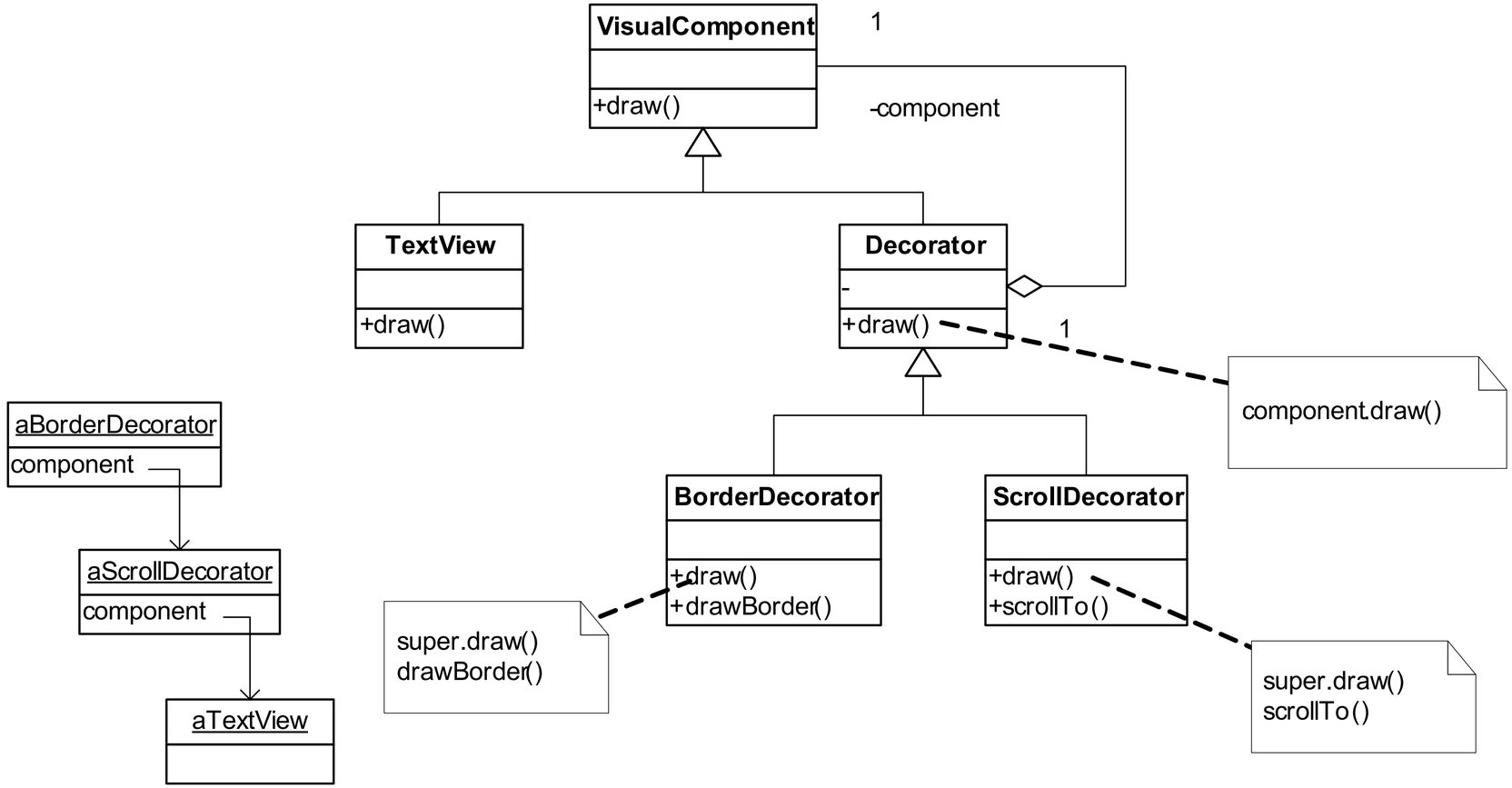
Adapter

Adapter converts the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



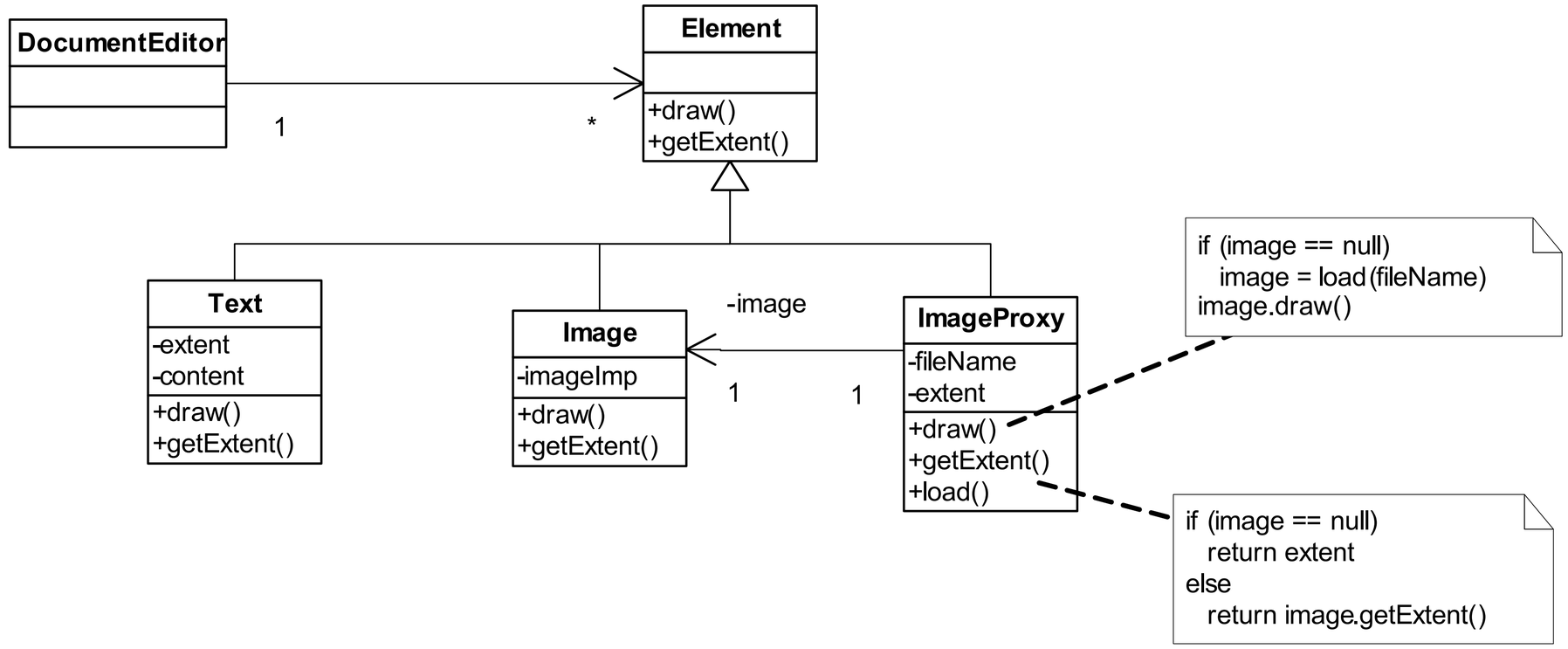
Decorator

Decorator attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.



Proxy

Proxy provides a surrogate or representative for another object to control access to it.

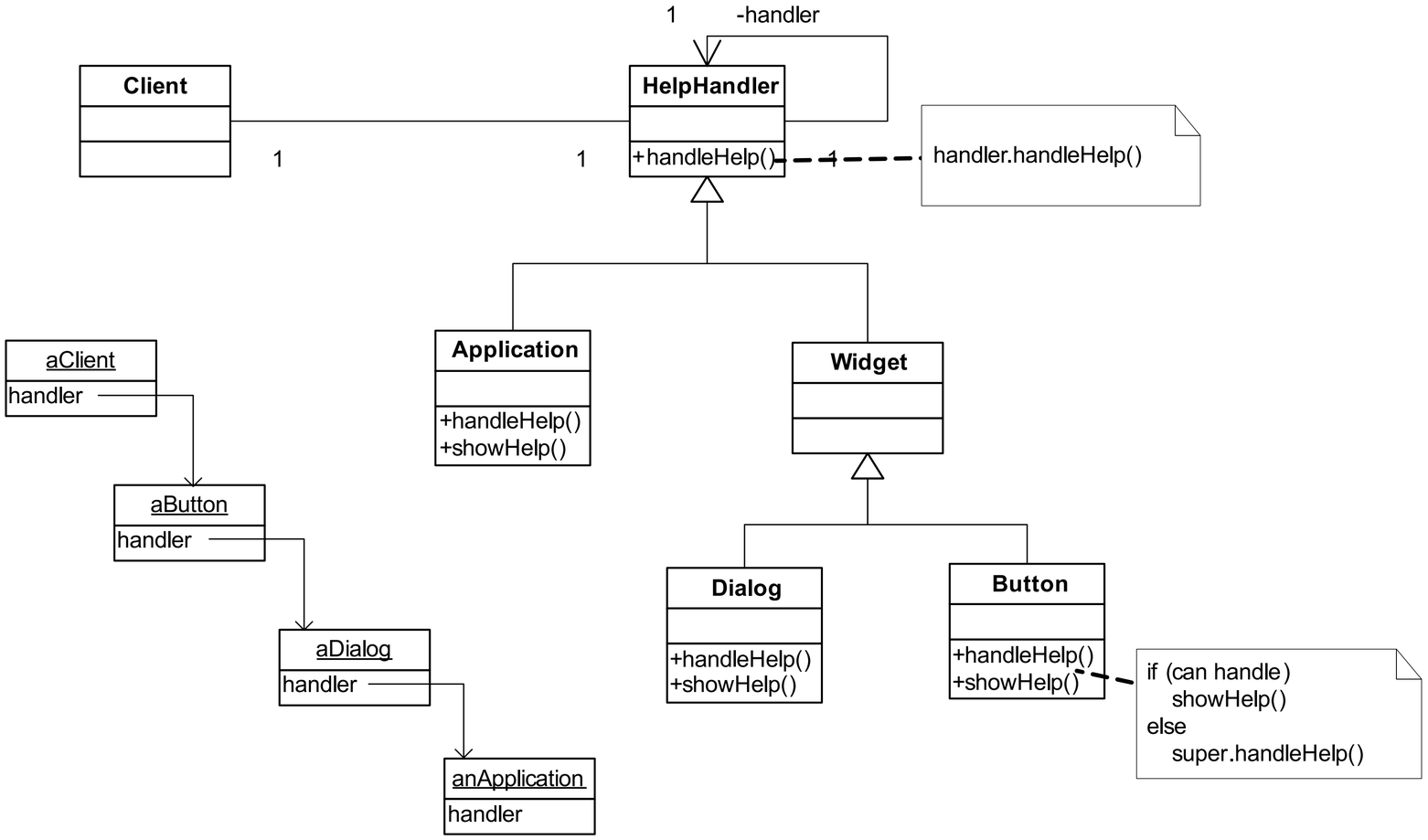


Behavioral Design Patterns

- **Chain of Responsibility** avoids coupling the senders of a request to its receiver by giving more than one object a chance to handle request. Chain the receiving objects and pass the request along the chain until an object handles it.
- **Command** encapsulates a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations.
- **Iterator** provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- **Observer** defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **State** allows an object to alter its behaviour when its internal state changes. The object will appear to change its class.
- **Strategy** defines a family of algorithms, encapsulates each one, and makes them interchangable. Strategy lets algorithm vary independently from clients that use it.

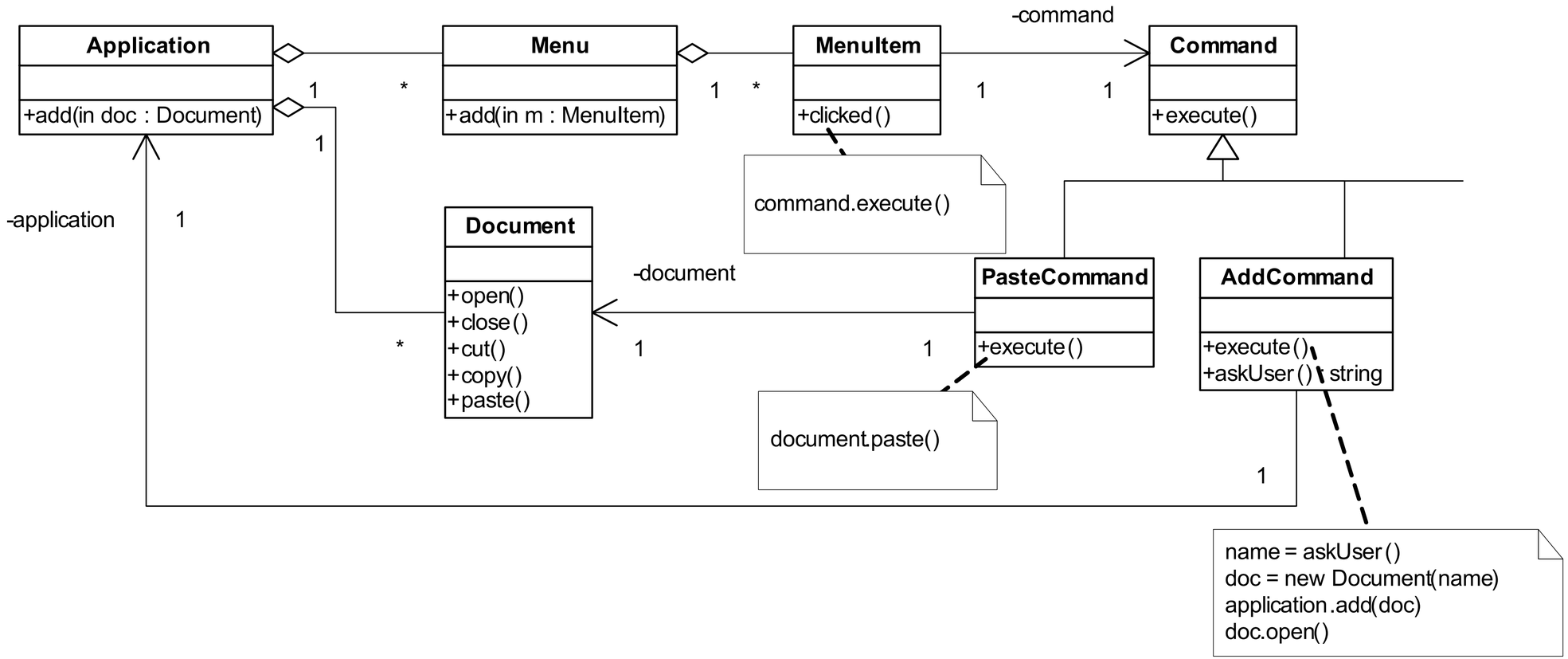
Chain of Responsibility

Chain of Responsibility avoids coupling the senders of a request to its receiver by giving more than one object a chance to handle request. Chain the receiving objects and pass the request along the chain until an object handles it.



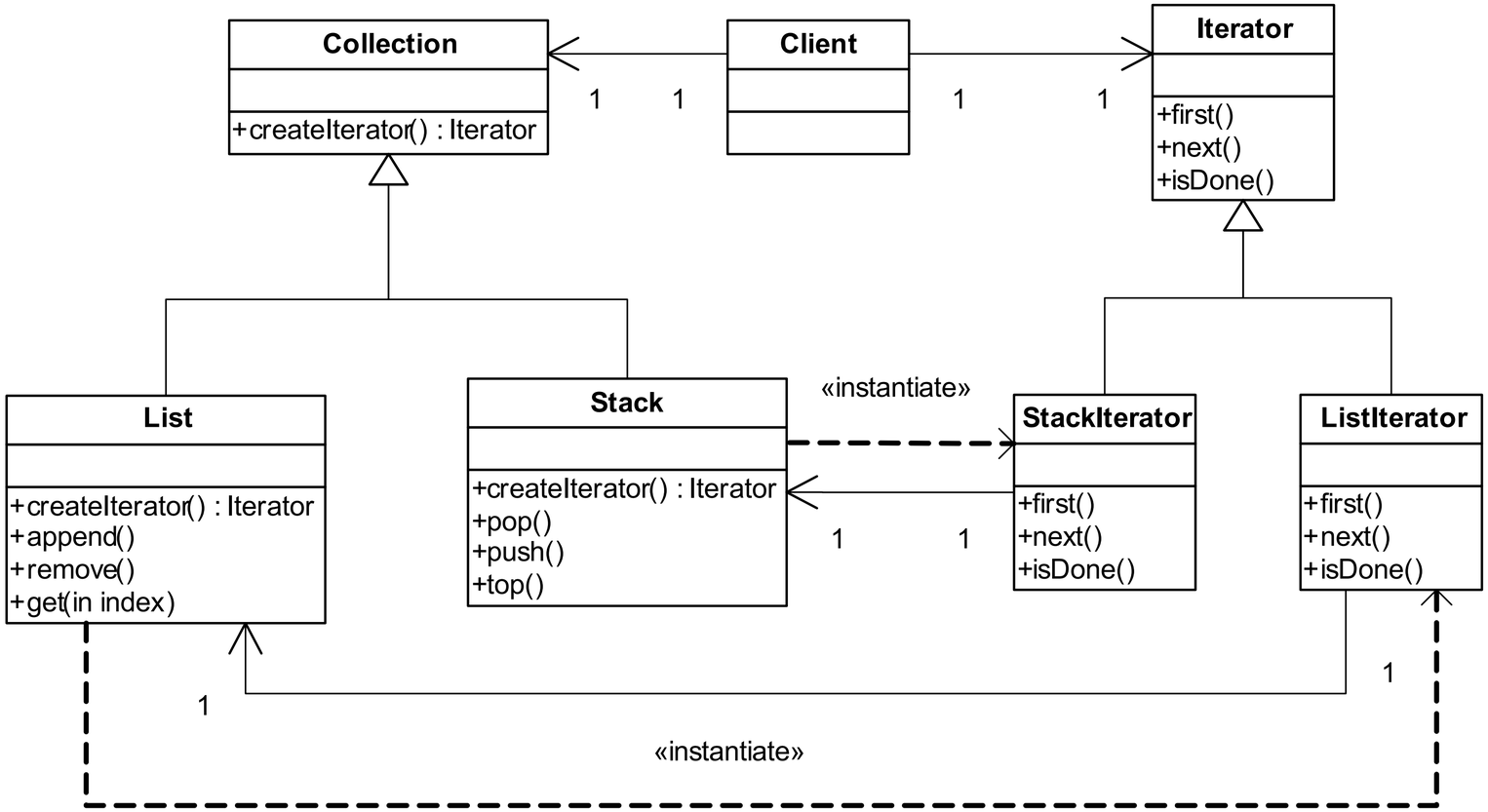
Command

Command encapsulates a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations.



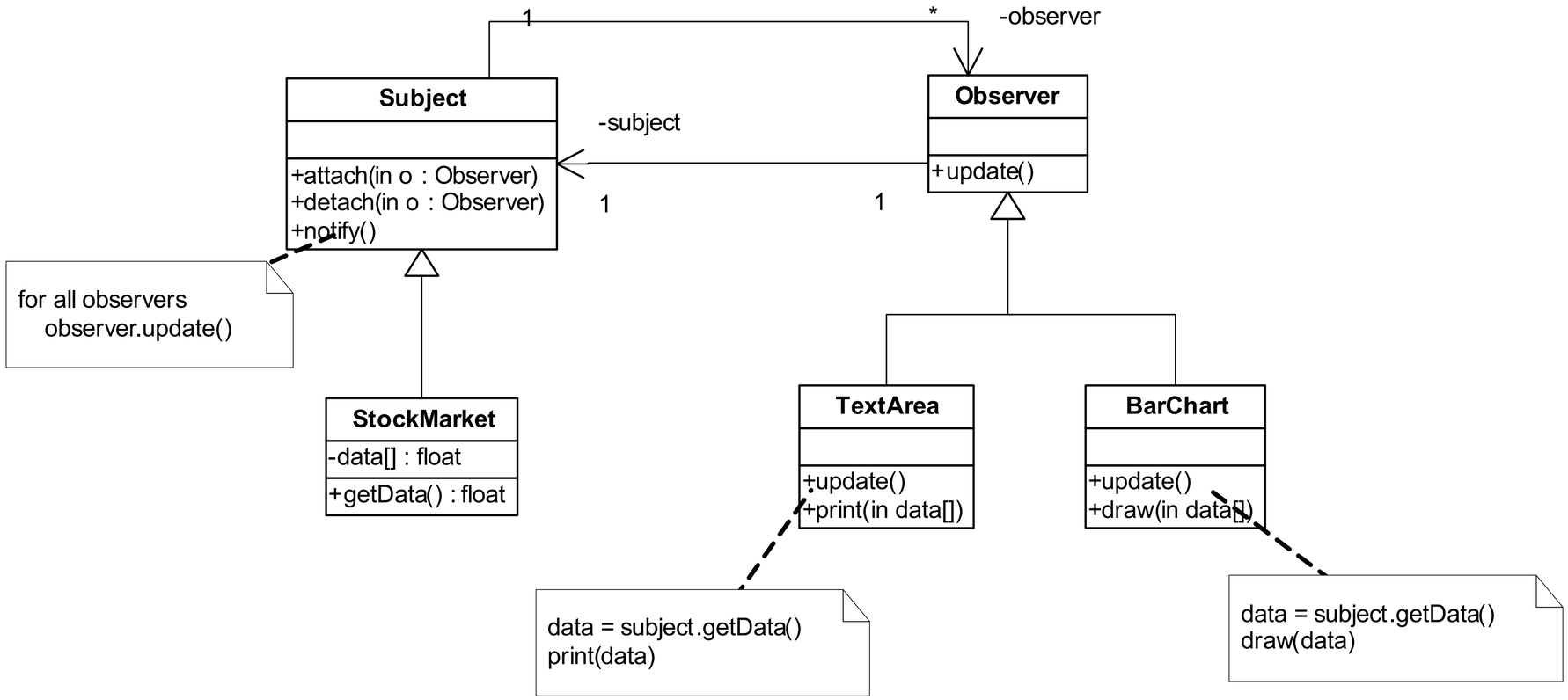
Iterator

Iterator provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



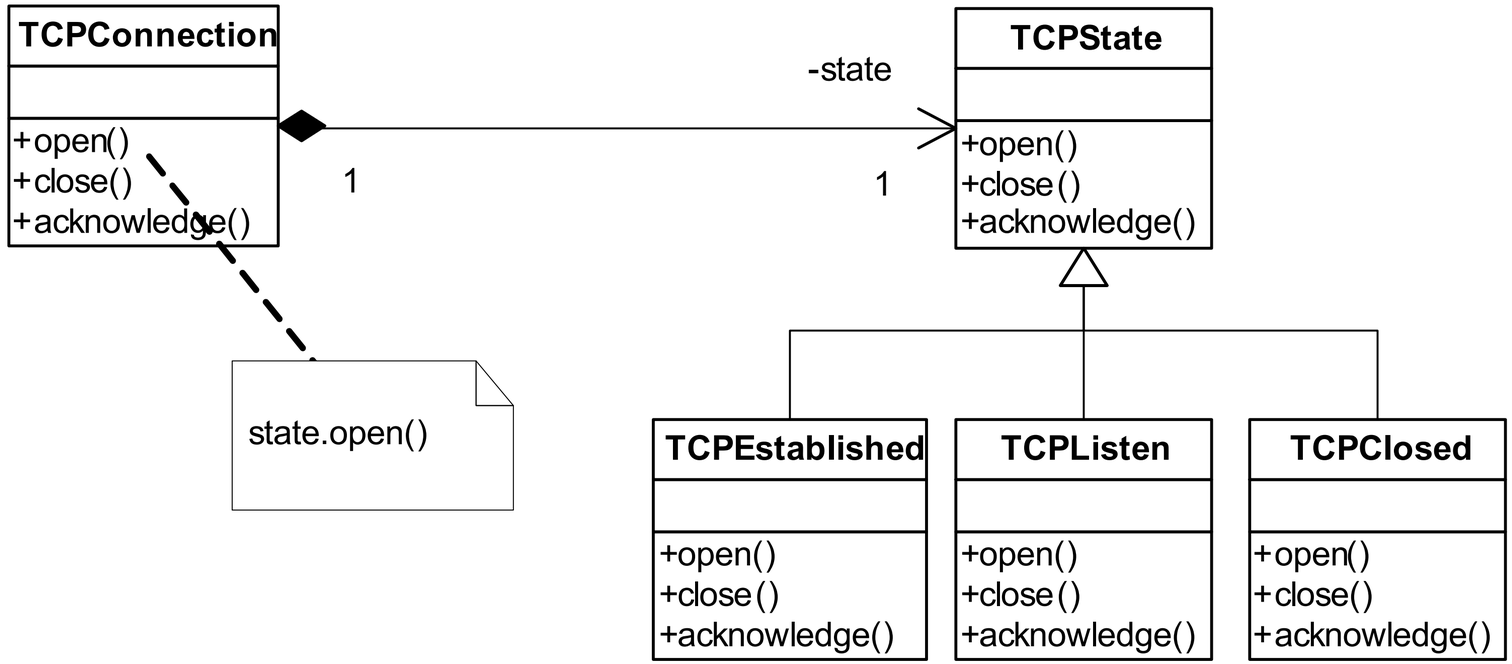
Observer

Observer defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



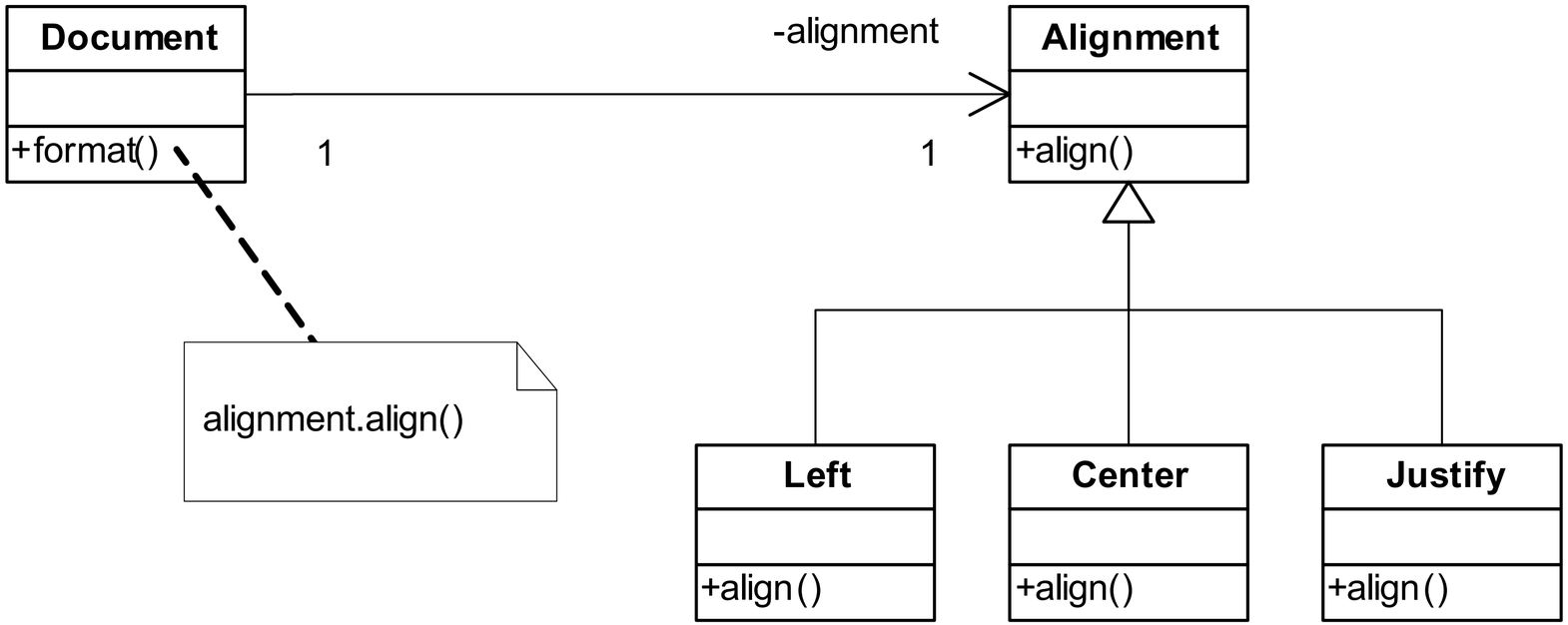
State

State allows an object to alter its behavior when its internal state changes. The object will appear to change its class.



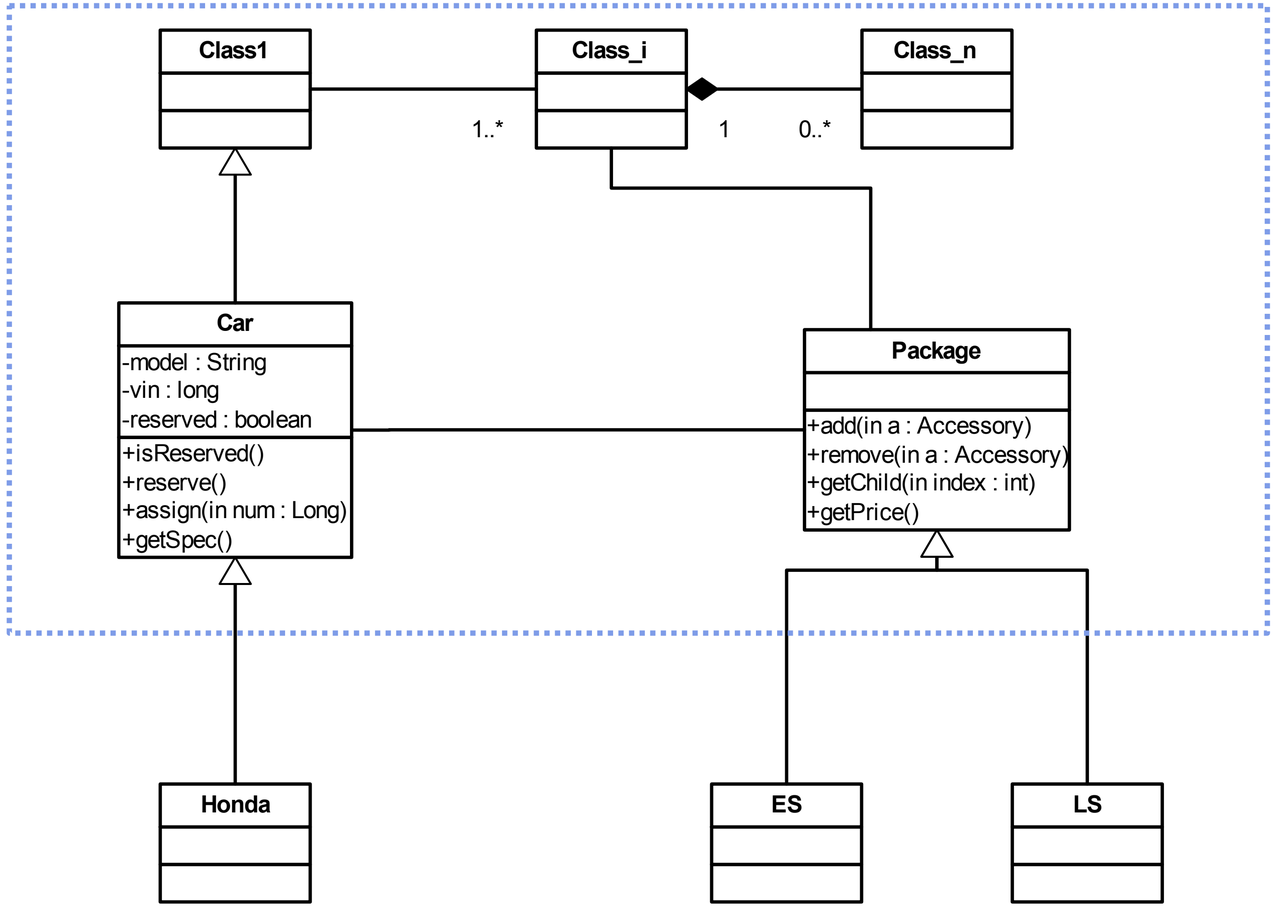
Strategy

Strategy defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets algorithm vary independently from clients that use it.



Frameworks

A Framework is a set of abstract and concrete classes that comprise a generic software system. Framework is calling our classes = full control of flow.



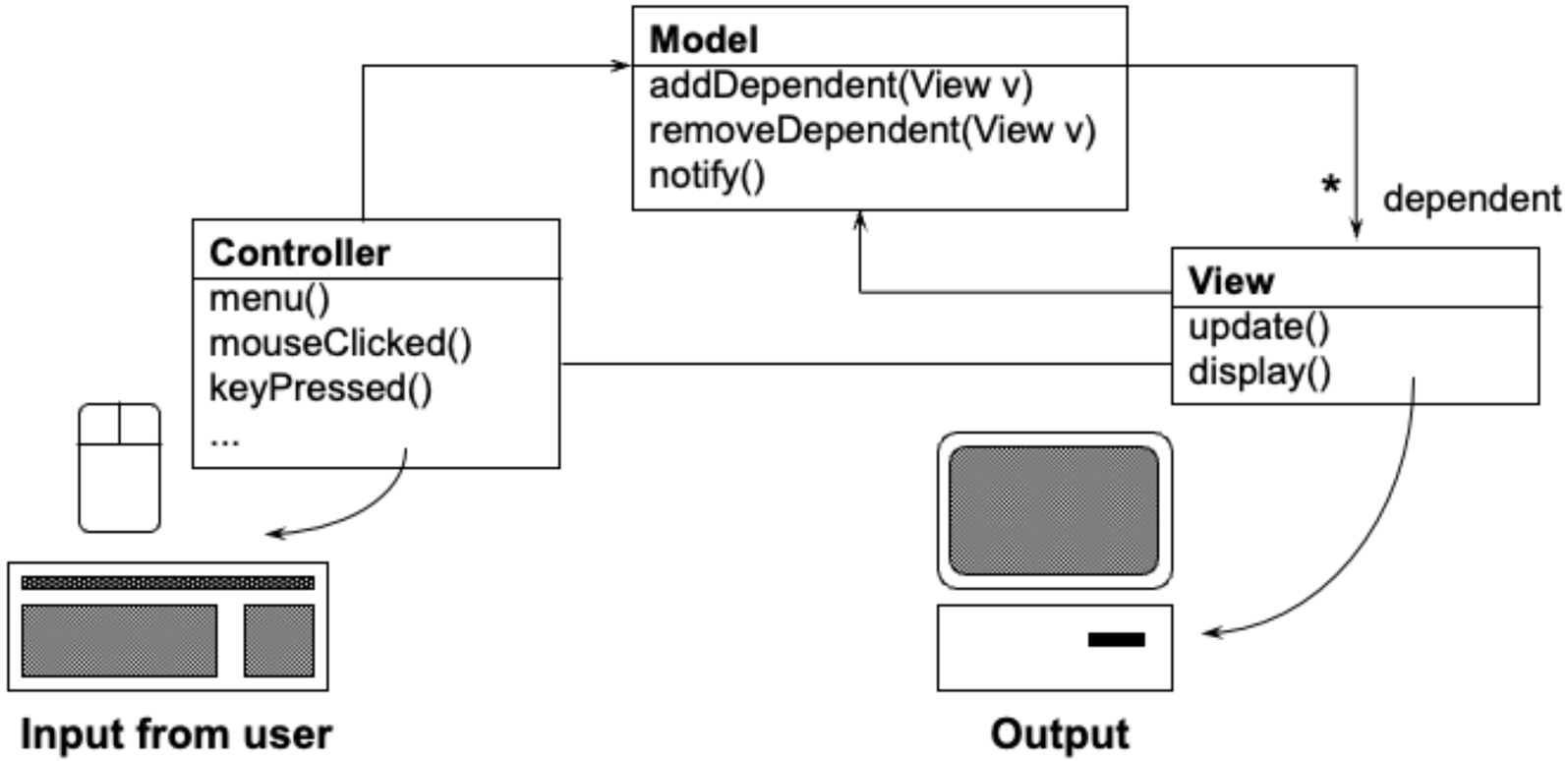
MVC - Application Framework

A Framework is a set of abstract and concrete classes that comprise a generic software system. Framework is calling our classes = full control of flow.

- Any application can be divided into **two parts**: (1) **The domain model**, which handles data storage and processing. (2) **The user interface**, which handles input and output.
- **Model-View-Controller** Architecture:
- **Model**: domain state and behavior (dynamic object).
- **View**: display current state of dynamic object.
- **Controller**: effect user-evoked behavior from the dynamic object.

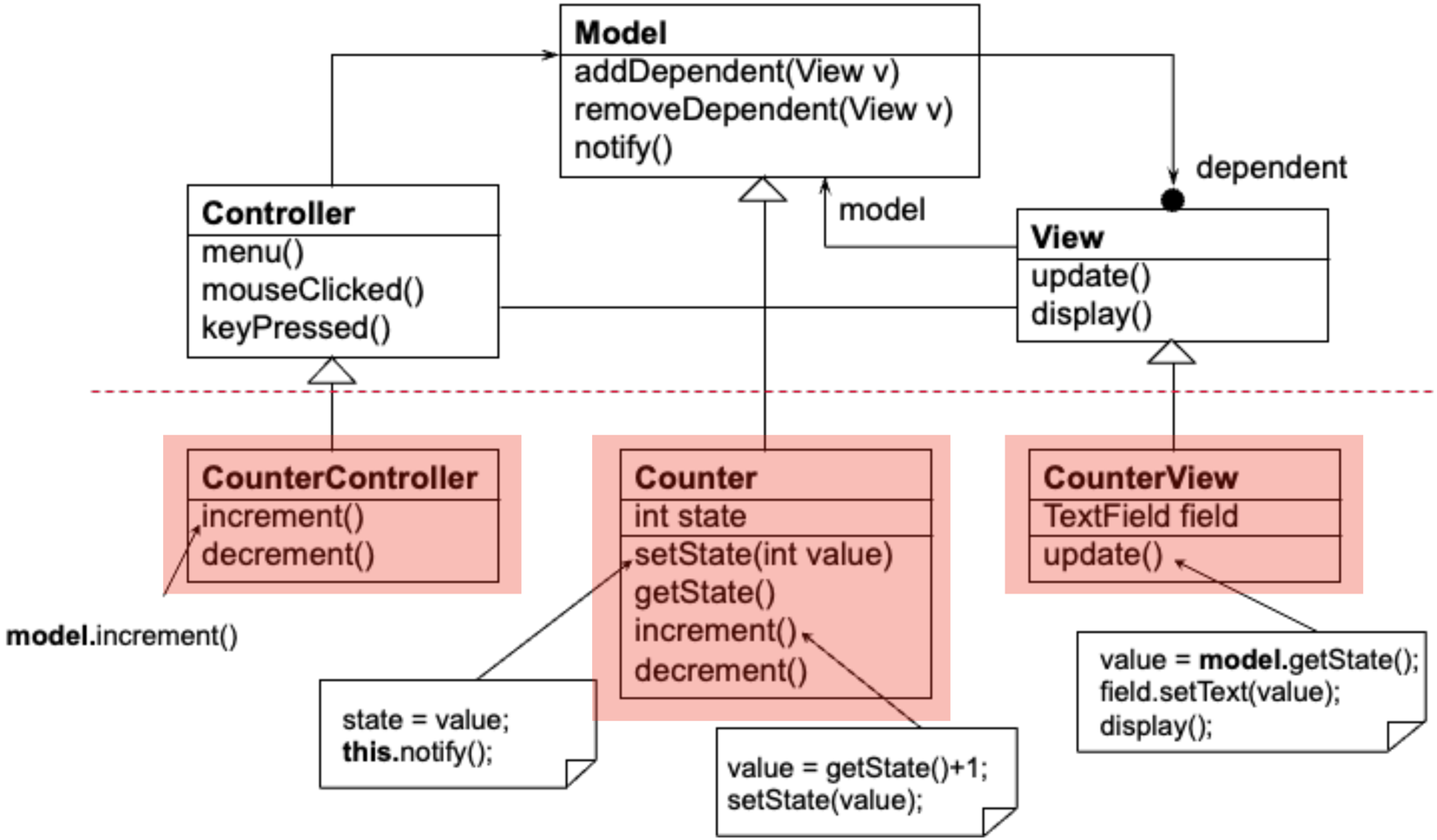
MVC Classes

Trygve Reenskaug created MVC while working on Smalltalk-79 as a visiting scientist at the Xerox Palo Alto Research Center (PARC) in the late 1970s. 💡



MVC Example

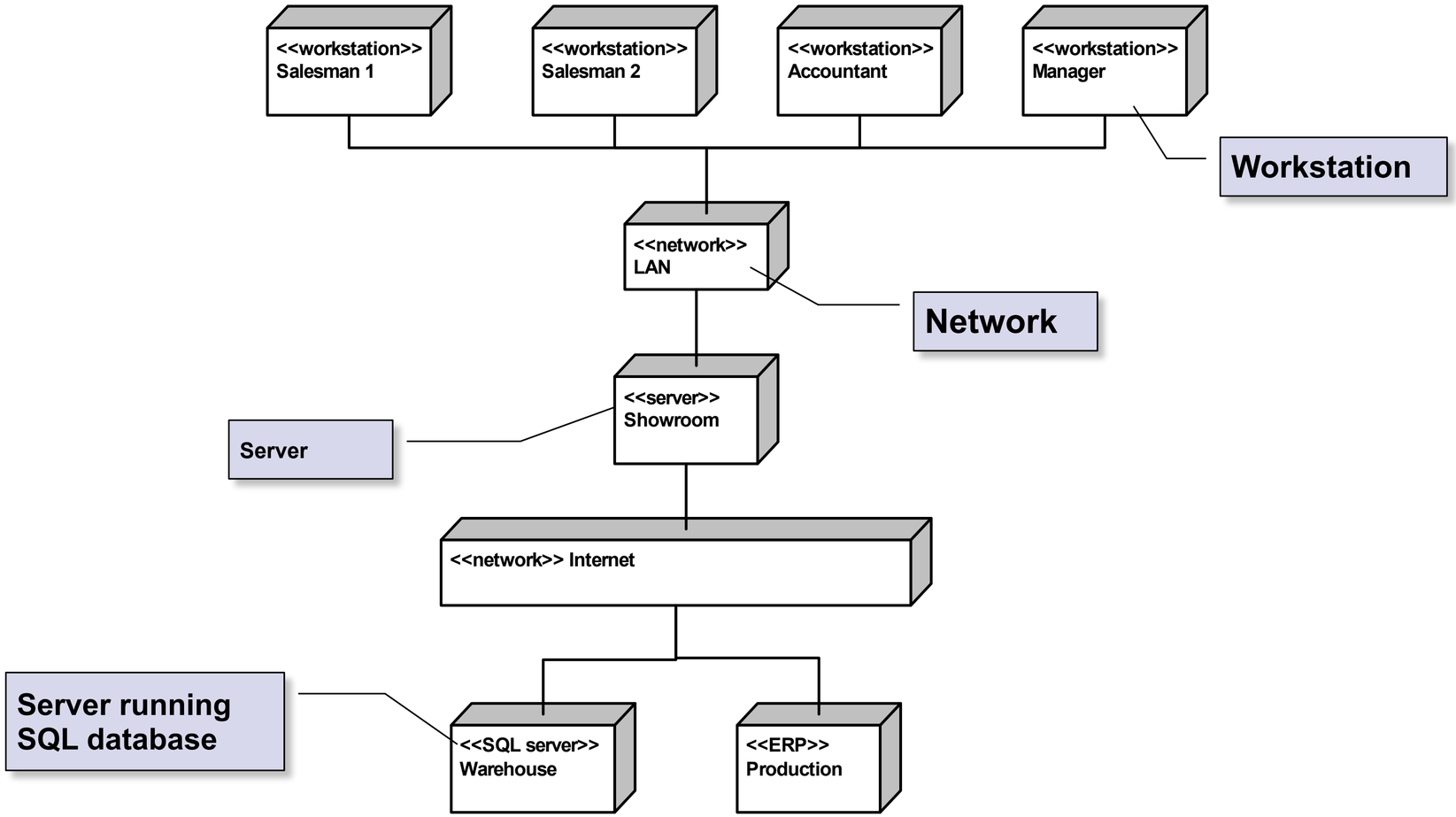
How to increment a decrement a counter.



Deployment Model

- Deployment diagram shows the **configuration of run-time processing elements.**
- The purpose of this diagram is **to model the topology of hardware** which the system executes.

Deployment Diagram



Implementation

The goal of the implementation workflow is to flesh out the designed architecture and the system as a whole.

- **Implementation Model** describes how elements in the design model, such as design classes, are implemented in terms of components such as a source code files, executables, and so on. The implementation model also describes **how the components are organized** according to the structuring and modularization mechanism available in the implementation environment and the programming language (e.g. **package in Java**).

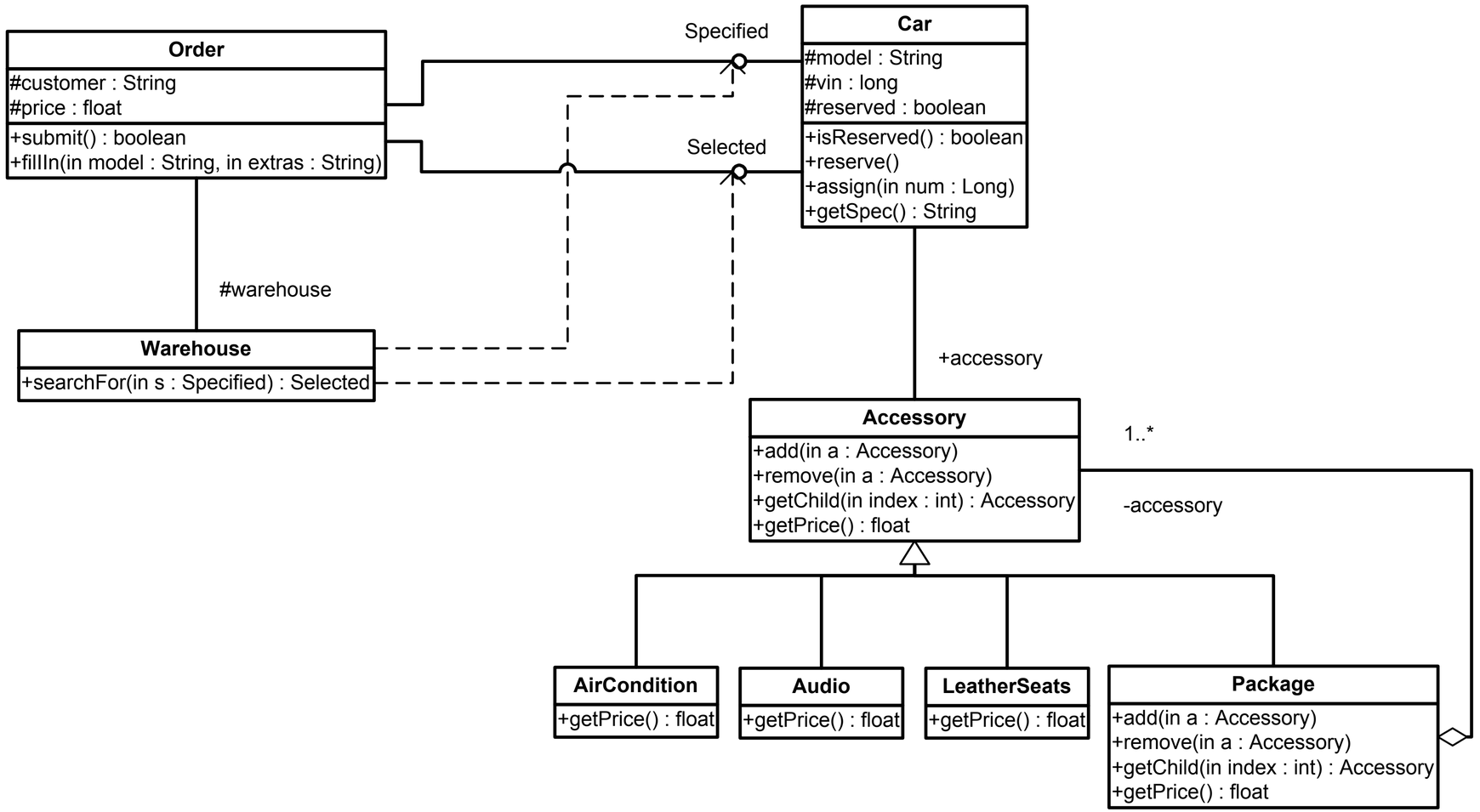
Mapping

How to represent blueprint to a real code

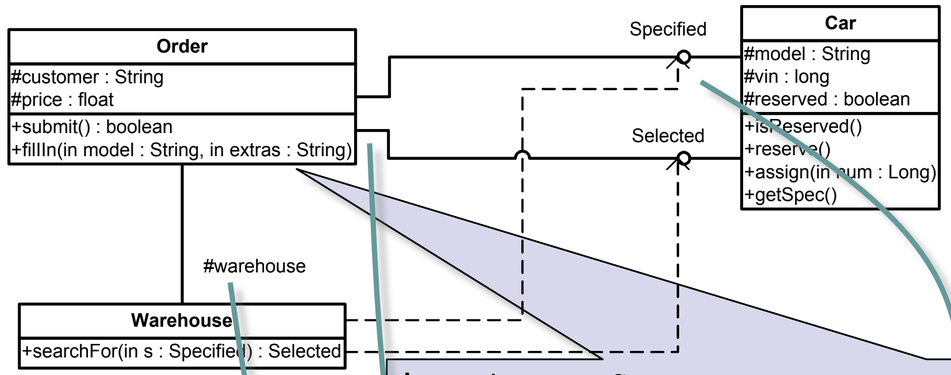
Analysis & Design	Source code (Java)
Class	Class
Role, Type, Interface	Interface
Operation	Method
Attribute (Class)	Static variable
Attribute (instance)	Instance variable
Association	Instance variables
Dependency	Local variable or argument in method or return value
Interaction between objects	Call to a method
Use Case	Sequence of calls
Package/Subsystem	Package

Design Results

Blueprint



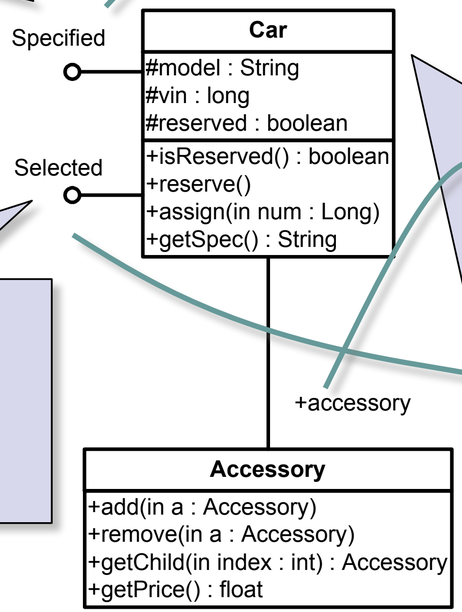
Source Code: Order.java




```
import cars.Car;
import warehouse.Warehouse;
public class Order {
   protected String customer;
   protected float price;
   protected Specified specified;
   protected Selected selected;
   protected Warehouse warehouse;

   public void fillIn(String model, String extras) {
        Specified specified = new Car(model, extras);
   }
   public boolean submit() {
      // warehouse is assigned through network
      selected = warehouse.searchFor(specified);
      if (selected.isReserved())
         return false;
      selected.reserve();
      return true;
   }
}
```

Specified.java, Selected.java, Car.java

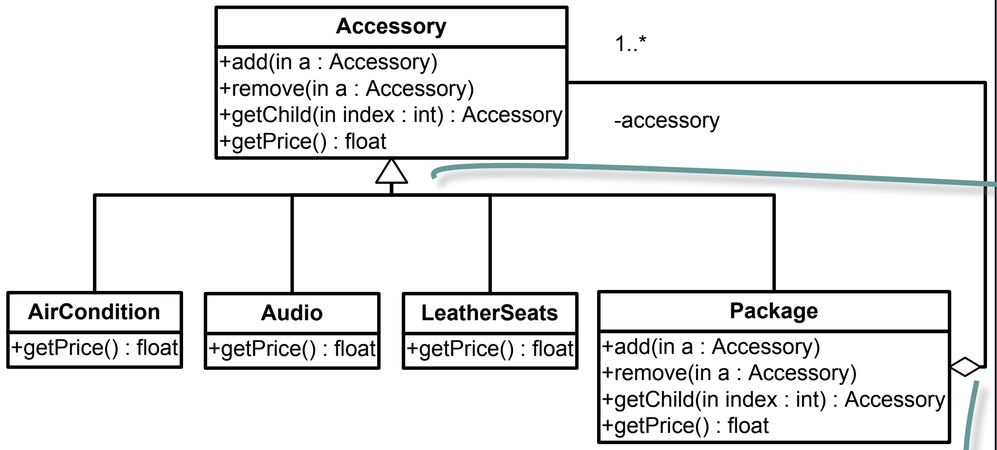
```
package cars;
public interface Specified {
   String getSpec();
}
```




```
package cars;
public interface Selected {
   boolean isReserved();
   void reserve();
}
```

```
package cars;
public class Car implements Specified, Selected {
   protected String model;
   protected long vin;
   protected boolean reserved;
   public Accessory accessory;

   public boolean isReserved() {
      return reserved;
   }
   public void reserve() {
      reserved = true;
   }
   public void assign(long num) {
      vin = num;
   }
   public String getSpec() {
      return model;
   }
}
```

Accessory.java



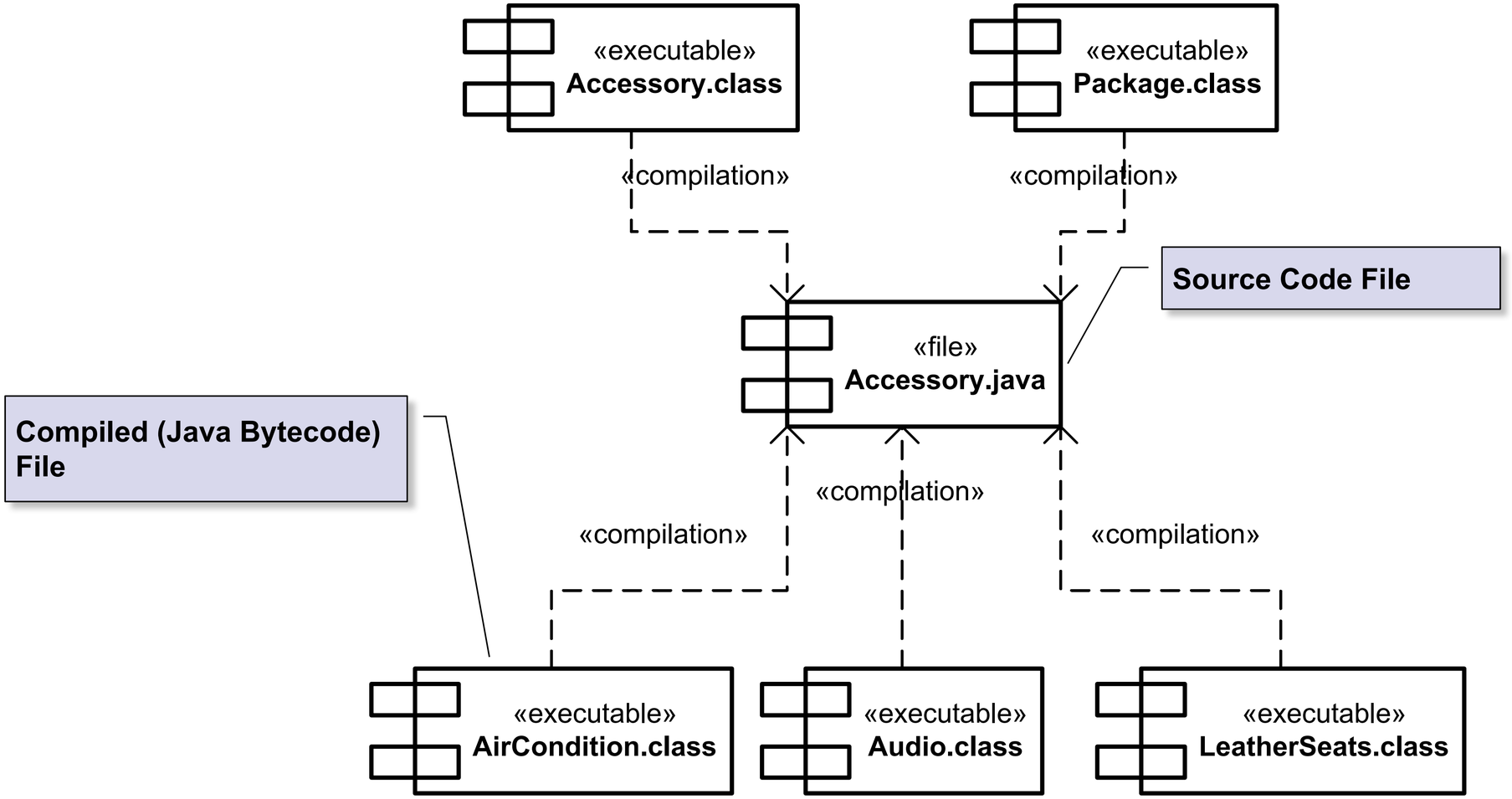

```
package cars;
public abstract class Accessory {
   public void add(Accessory a) {}
   public void remove(Accessory a) {}
   public Accessory getChild(int index) {
      return null;
   }
   public abstract float getPrice();
}

class AirCondition extends Accessory {
   public float getPrice() {
      return 2000.0f;
   }
}
// other "friend" classes
class Package extends Accessory {
   private Accessory[] accessory;

   public void add(Accessory a) {/*code*/}
   public void remove(Accessory a) {/*code*/}
   public Accessory getChild(int index) {
      return accessory[index];
   }
   public float getPrice() {
      float sum=0;
      for (int i=0; i < accessory.length; i++) {
         sum = sum + accessory[i].getPrice();
      }
      return sum;
   }
}
```

UML Diagrams for Implementation

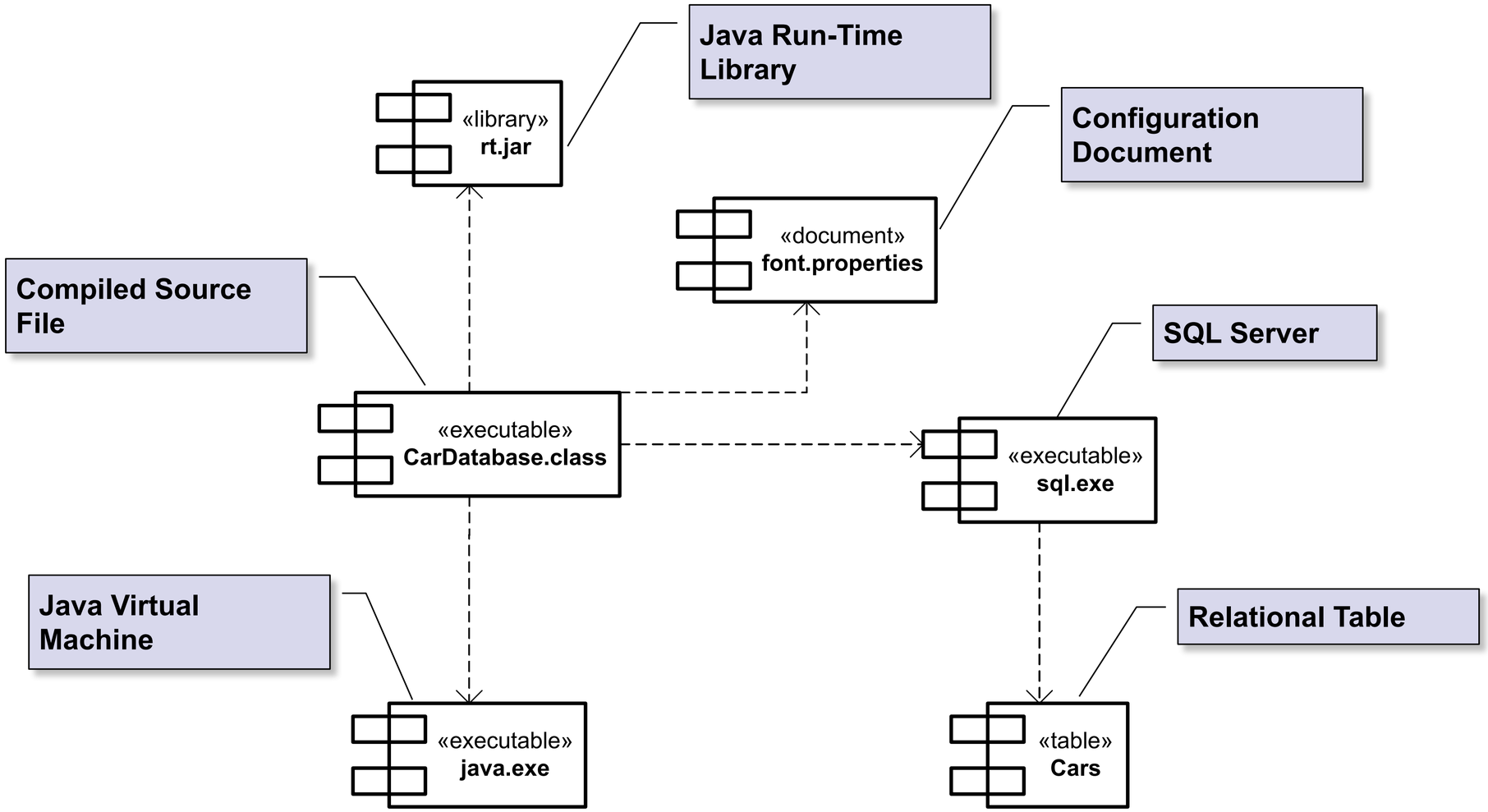
How to manage all software components

- **Component Diagram** illustrates the organization and dependencies among **software components – physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces**. A component may be
  - A source code component
  - A binary or executable component
  - Others (database tables, documents) …
- **Deployment Diagram** is refined to show the configuration of run-time processing elements and the software components, processes, and objects that execute on them.

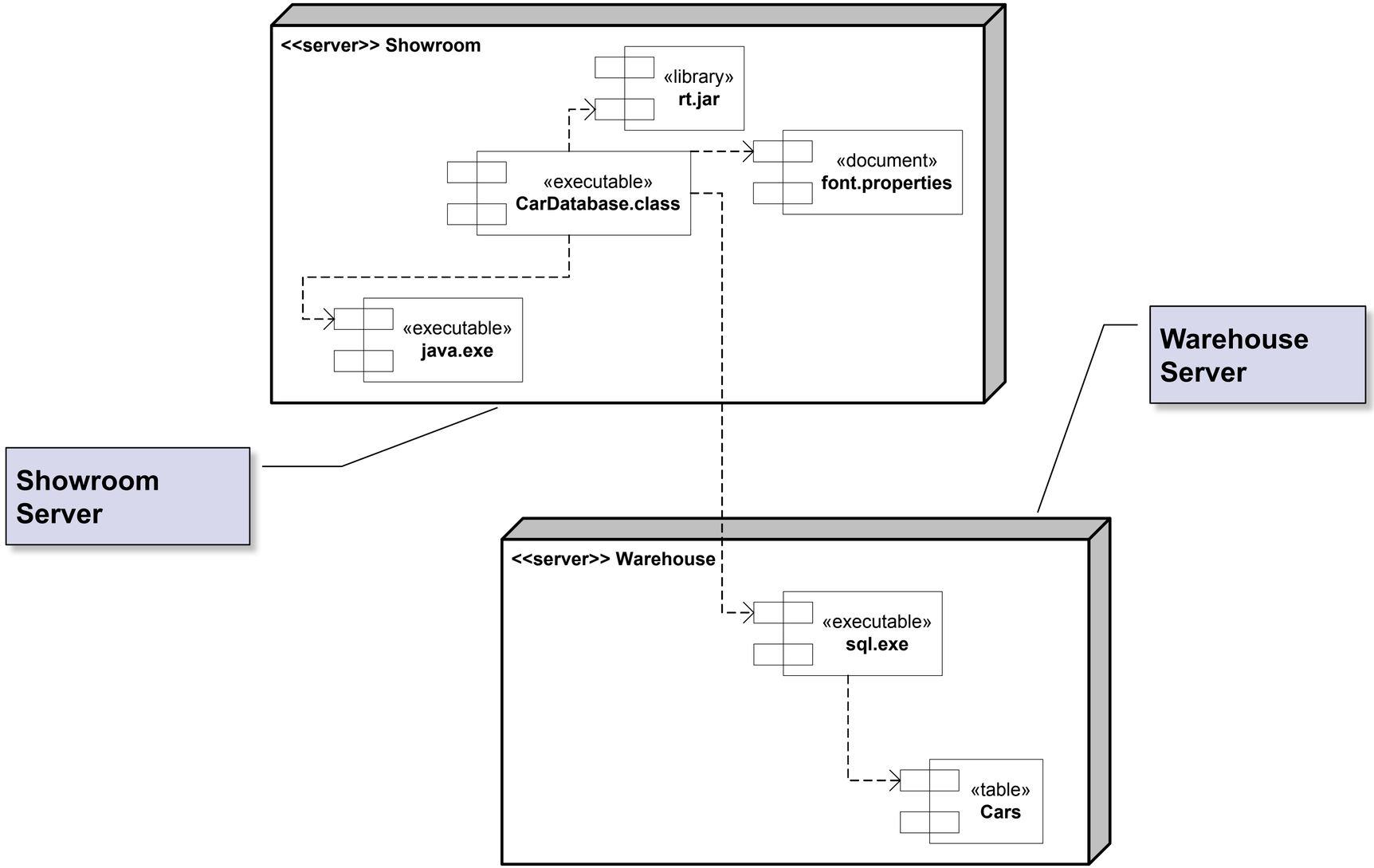
Component Diagram: Binaries



Component Diagram: Run-Time



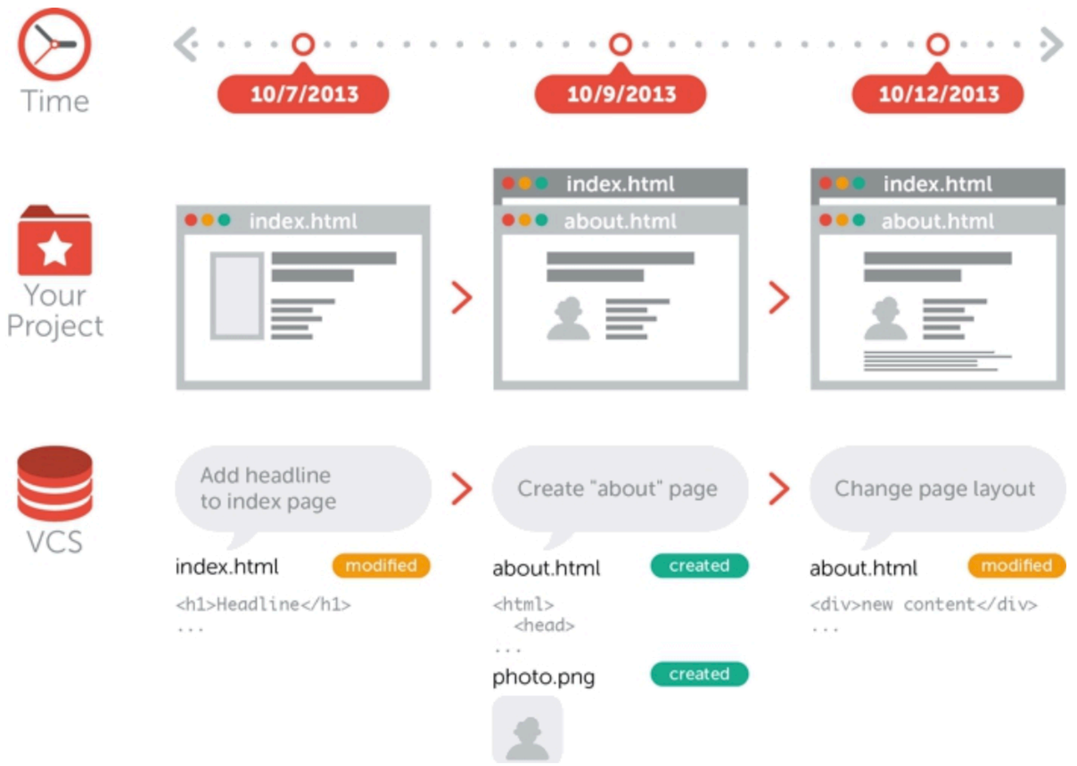
Refined Deployment Diagram



Version Control

A version control system (VCS) records the changes you make to your project‘s files.

- **Keeps track** of changes in code and documents over time.
- **Allows multiple developers** to work on the same project.
- Helps to **revert mistakes** and compare versions.
- Enables **branching, merging, and collaboration**.
- **Without version control, teamwork is chaos.**



Source: Learn Version Control with Git by Tobias Günther

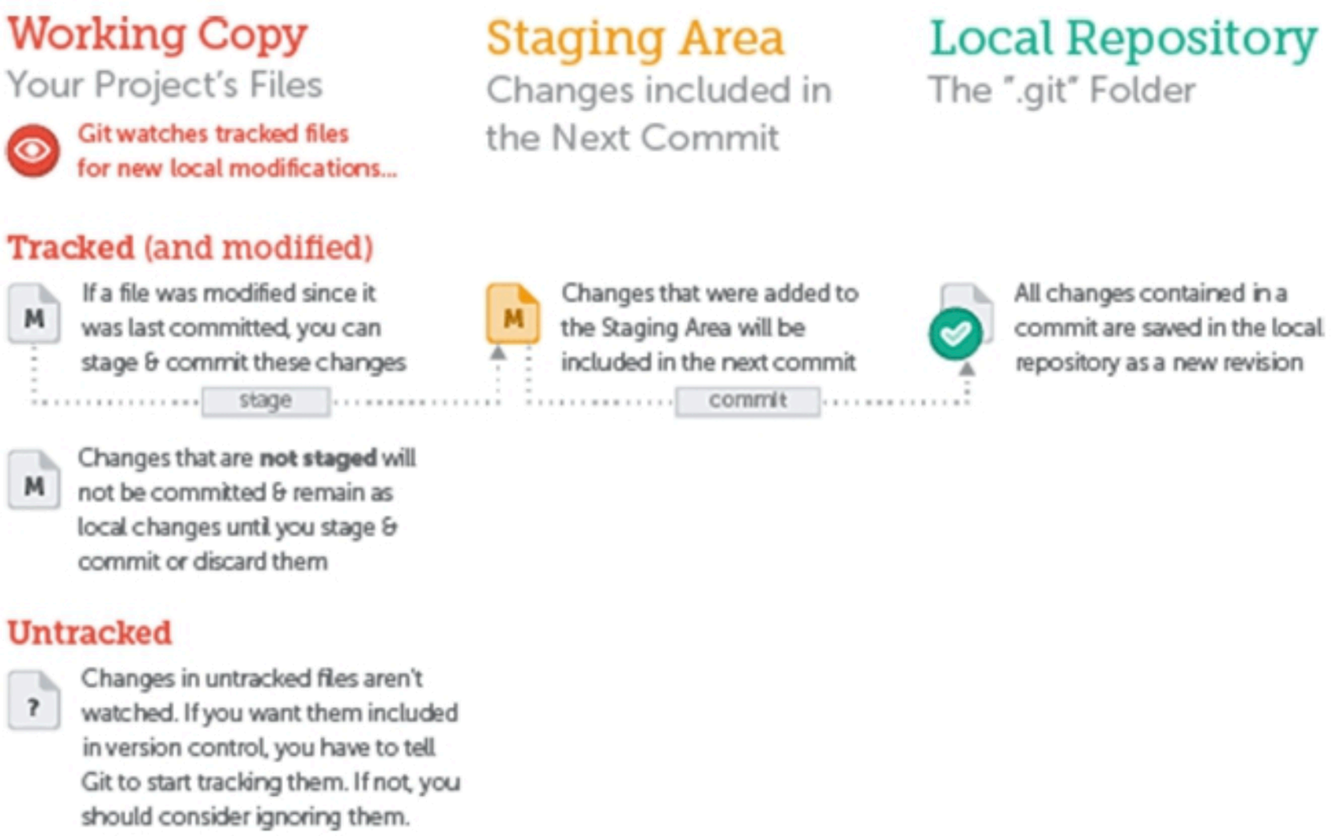
What is Git?

Git is a distributed version control system created by Linus Torvalds (2005) used for tracking changes, collaboration, and integration across teams.

- **Repository (repo)**: A directory where all your project files and history are stored.
- **Commit**: A snapshot of your work with a unique ID and message.
- **Branch**: A separate line of development — great for new features or bug fixes.
- **Merge**: Combines changes from different branches.
- **HEAD**: The current position in the commit history.
- Git ≠ GitHub → Git = tool, GitHub = hosting platform for Git repositories.

Working on the Project

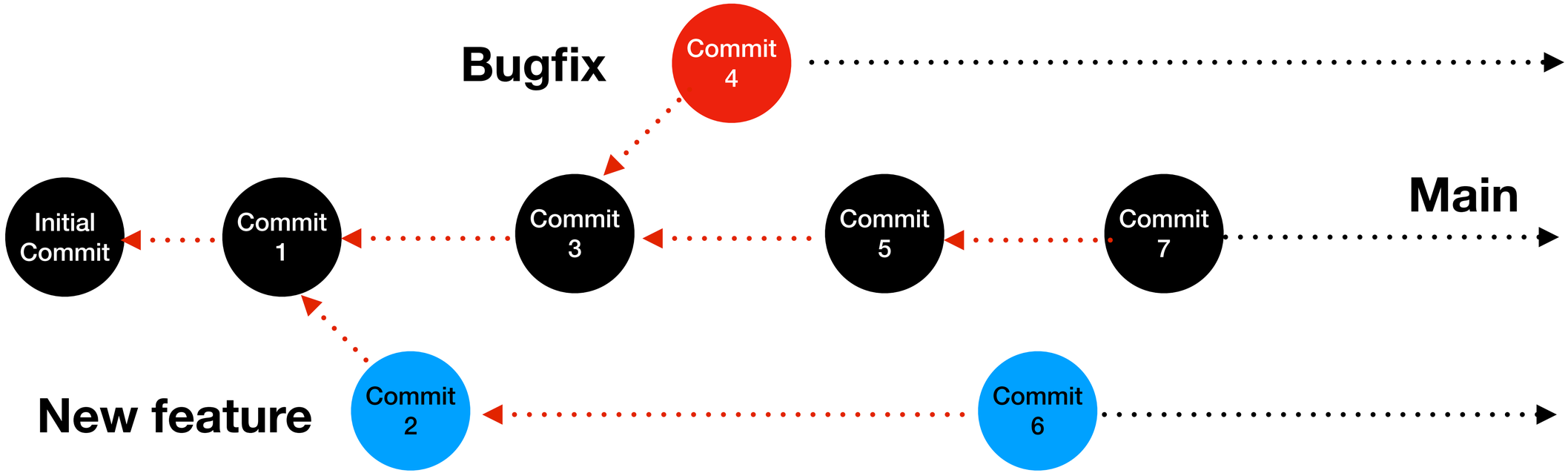
At some point after working on your files for a while, you‘ll want to save a new version of your project. Or in other words: you‘ll want to commit some of the changes you made to your tracked files.



Source: Learn Version Control with Git by Tobias Günther

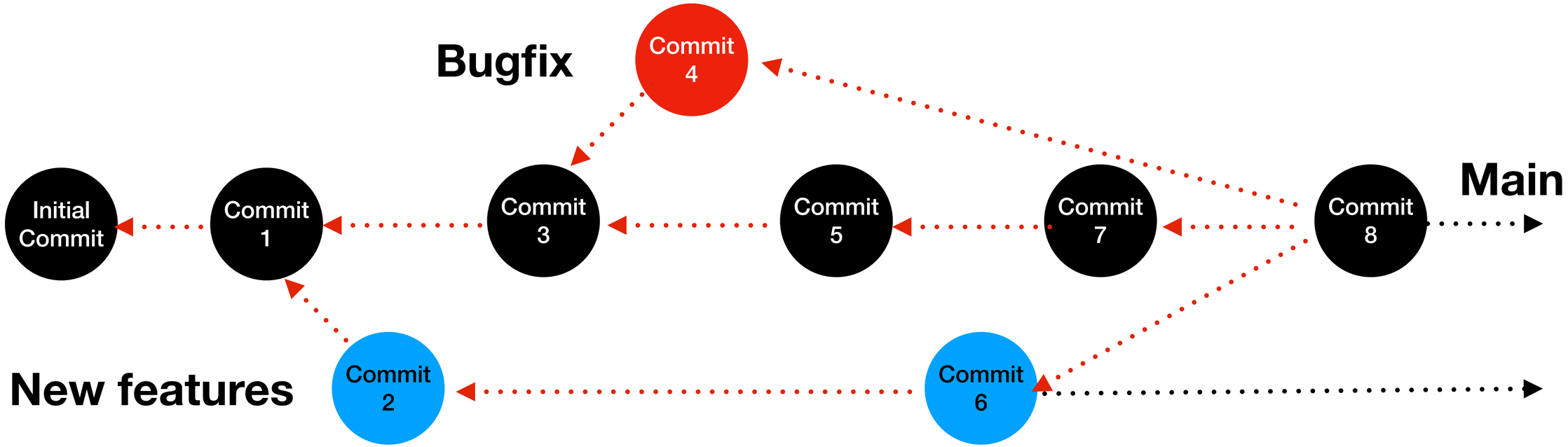
Branching and …

Every project has multiple different contexts where work happens: main production, new features, bugfixes, experiments and this work always happens in multiple of these contexts in parallel.



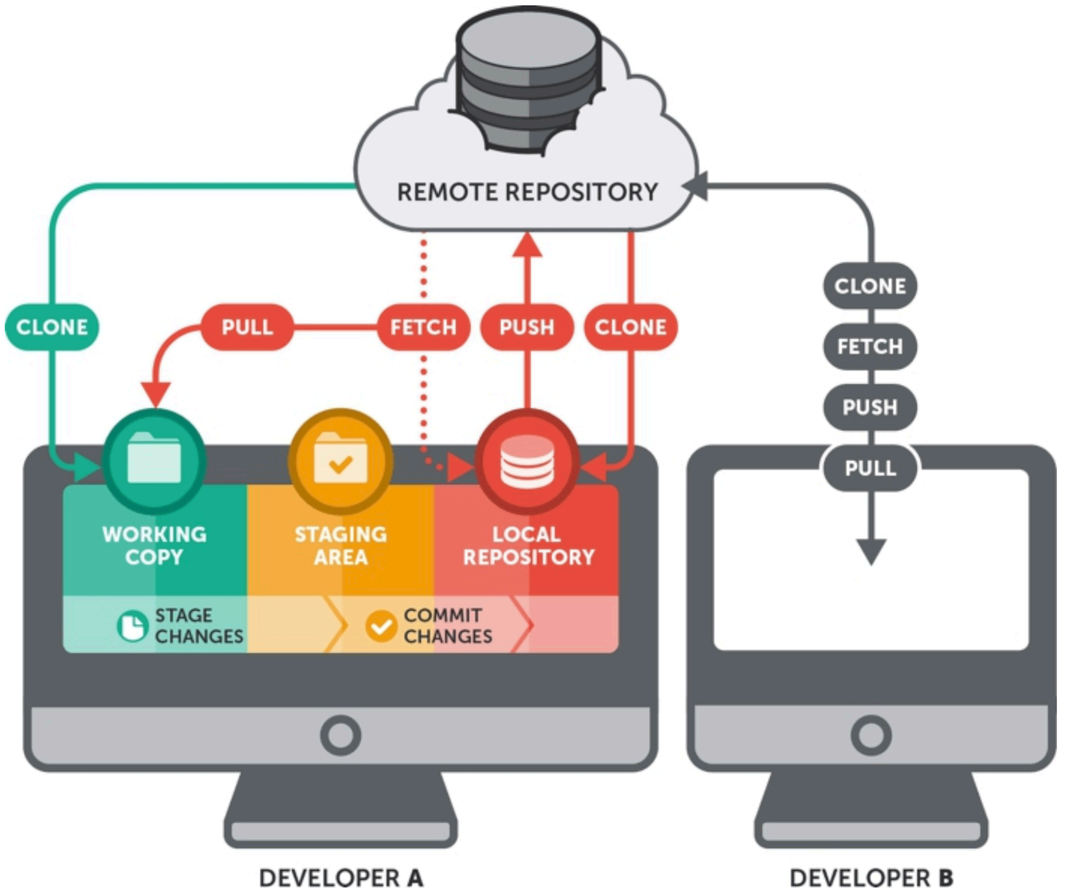
… Merging

Merging changes means to integrate changes from one branch into another.



Local and Remote Repositories

Push sends local commits to the remote repository. Pull fetches and merges updates from the remote repository to local copy.



Source: Learn Version Control with Git by Tobias Günther

When you forget to pull before push.

MERGE CONFLICT HELL

Testing

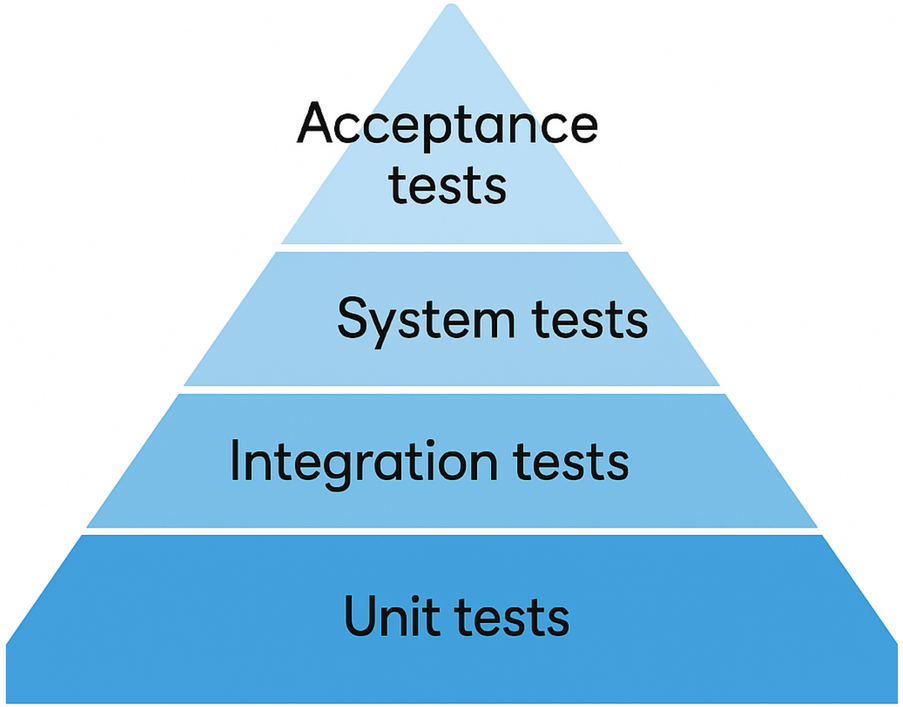
Software testing is the process of verifying that a system meets its requirements and behaves as expected.

Goals:

- Detect defects early
- Improve software quality
- Increase confidence before release

Levels of testing:

- **Unit tests** – test individual components (fast, many)
- **Integration tests** – test interactions between modules
- **System tests** – test the entire system as a whole
- **Acceptance tests** – verify the system meets business needs



Unit Testing

Validate individual functions, classes, or modules

- Fast to run
- Easy to automate
- Should not depend on external systems
- **AAA pattern (Arrange – Act – Assert)**

```
@Test
void shouldReturnCorrectSum() {
    // Arrange
    Calculator calc = new Calculator();

    // Act
    int result = calc.add(2, 3);

    // Assert
    assertEquals(5, result);
}
```

Integration Testing

Integration testing verifies that different components or modules of a software system work correctly together.

- **Big Bang Integration**: All modules are combined and tested at once. Quick setup but hard to debug errors - you don’t know which module caused the failure. **Not suitable for large projects.**
- **Top-Down Integration**: Start testing from the top (main logic, APIs) - gradually add lower modules. **Missing parts are replaced by stubs** (temporary fake components).
- **Bottom-Up Integration**: Start with low-level components - build up to higher-level modules. **Use drivers to simulate higher-level parts** that call tested code.
- **Sandwich / Hybrid**: Combines both top-down and bottom-up approaches.
- **AAA pattern (Arrange – Act – Assert) is empoyed**

System Testing

System testing validates the entire integrated software system as a whole to ensure it meets its specified requirements.

Functional testing

- Does the system perform the tasks it was designed for (**use cases**)?
- Are workflows complete (e.g., login → purchase → confirmation)?

Non-functional testing

- **Performance**: How fast does the system respond under load?
- **Security**: Can unauthorized users access restricted areas?
- **Usability**: Is the system intuitive for end users?
- **Reliability**: Does it work consistently over time?

Acceptance Testing

Acceptance testing is performed by end users or clients to confirm that the system meets their business goals and expectations.

User Acceptance Testing (UAT): Conducted by actual end users. Tests real workflows (e.g., submitting orders, managing profiles).

Business Acceptance Testing (BAT): Ensures business processes and rules align with company needs (e.g. tax calculation …).

Operational Acceptance Testing (OAT): Focuses on system readiness for deployment (backups, monitoring, recovery).

Alpha / Beta Testing

- **Alpha**: Performed internally by the development team or limited users.
- **Beta**: Conducted by external users before full release.

Test-Driven Development (TDD)

A development process where tests are written before the code itself.

Red - Green - Refactor

- Write a **failing test** (🔴): Before writing any code, you define what the system should do. Create a unit test that expresses this desired behavior. Since the feature doesn’t exist yet, the test will fail — this is expected!

- Write the Minimum Code to **Make It Pass** (🟢): Implement just enough code to make the test pass — nothing more. Don’t worry about optimization or design yet. The goal is to move quickly from failing to passing.

- **Refactor** for Clarity and Maintainability: Once all tests are green, improve your code without changing behavior. Your safety net: the tests will confirm that nothing breaks during refactoring.

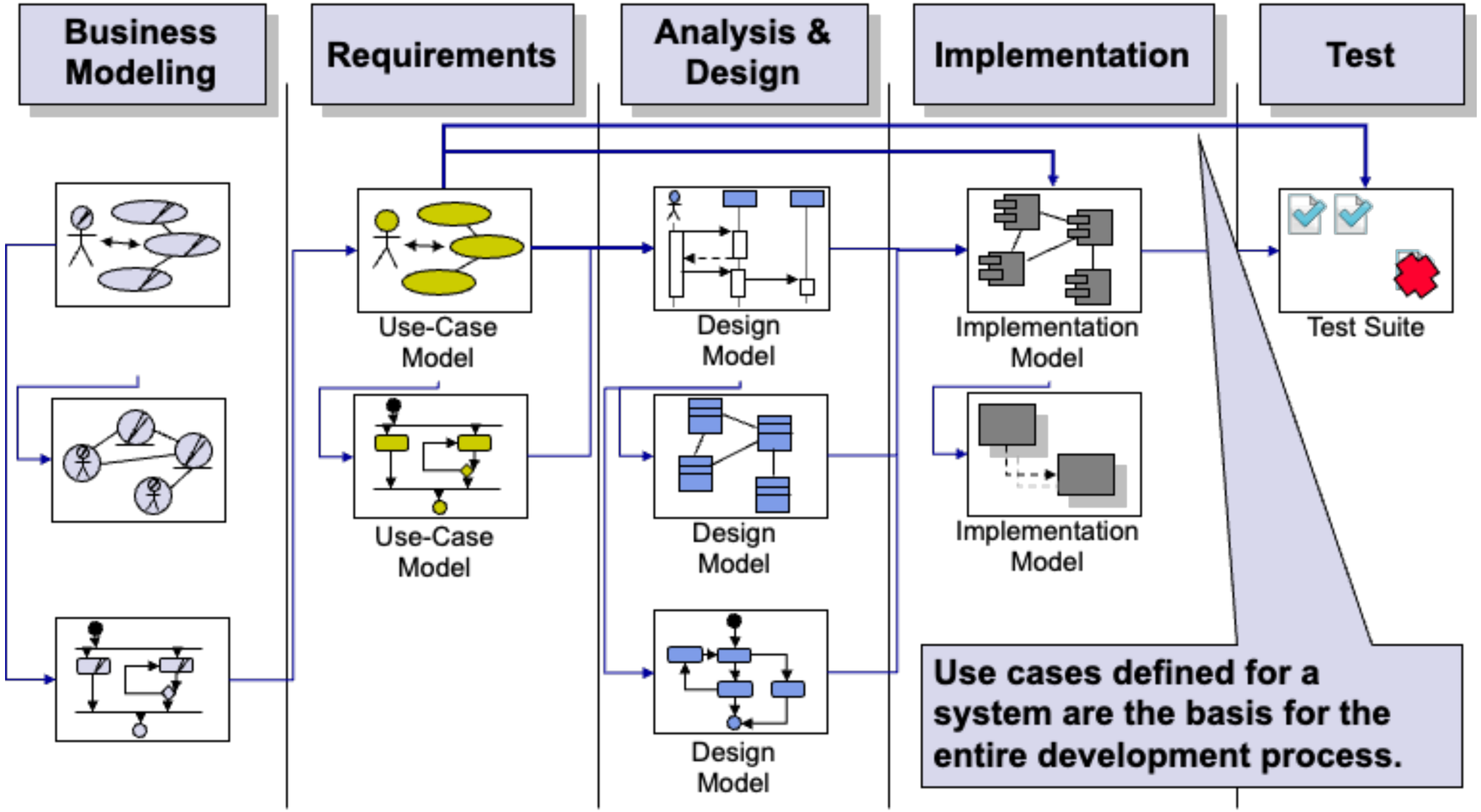
TDD Example

Red - Green - Refactor

```
// Step 1: Write the test
@Test
void shouldReturnTrueForPalindrome() {
    assertTrue(StringUtils.isPalindrome("level"));
}
// Step 2: Implement code
boolean isPalindrome(String s) {
    return new StringBuilder(s).reverse().toString().equals(s);
}
// Refactor
boolean isPalindrome(String s) {
    if (s == null) return false;
    String cleaned = s.toLowerCase().replaceAll("\\s+", "");
    return new StringBuilder(cleaned).reverse().toString().equals(cleaned);
}
```

Use Case Based Development

Architecture and use cases: two sides of the one coin - software engineering



… or without software engineering

I don't believe in coding design tools. I've been programming for more than 30 years now (40 years if you go back to my first class in programming). I think in code. Code is my design language and procrastination is my friend.

Peter Vogel 12/27/2016

Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

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

That is, while there is value in the items on
the right, we value the items on the left more.

Kent Beck
Mike Beedle
Arie van Bennekum
Alistair Cockburn
Ward Cunningham
Martin Fowler
James Grenning
Jim Highsmith
Andrew Hunt
Ron Jeffries
Jon Kern
Brian Marick
Robert C. Martin
Steve Mellor
Ken Schwaber
Jeff Sutherland
Dave Thomas

Summary