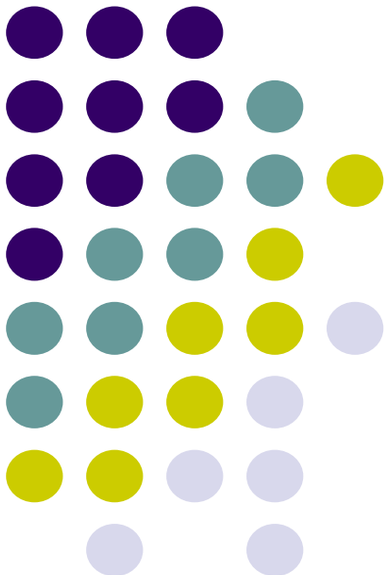
Software Engineering

Prof. Ing. Ivo Vondrak, CSc.
Dept. of Computer Science
Technical University of Ostrava
ivo.vondrak@vsb.cz
http://vondrak.cs.vsb.cz



Motto:

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?"

References

- Humphrey, W. Managing the Software Process, Addison-Wesley/SEI series in Software Engineering, Reading, Ma, 1989
- Gamma, E., Helm,R., Johnson,R., Vlissides,J. Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994
- Icon Computing, Inc. Object-Oriented Design, Idioms and Architectures, Austin, 1996
- Rational Unified Process Whitepaper: Best Practices for Software Development Teams - Rational Software, 1998
- Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process, Addison Wesley Longman, Inc., 1999
- Booch, G., Jacobson, I., Rumbaugh, J.: The Unified Modeling Language User Guide, Addison Wesley Longman, Inc., 1999
- Schmuller, J.: Teaching Yourself UML in 24 Hours, Sams, 1999

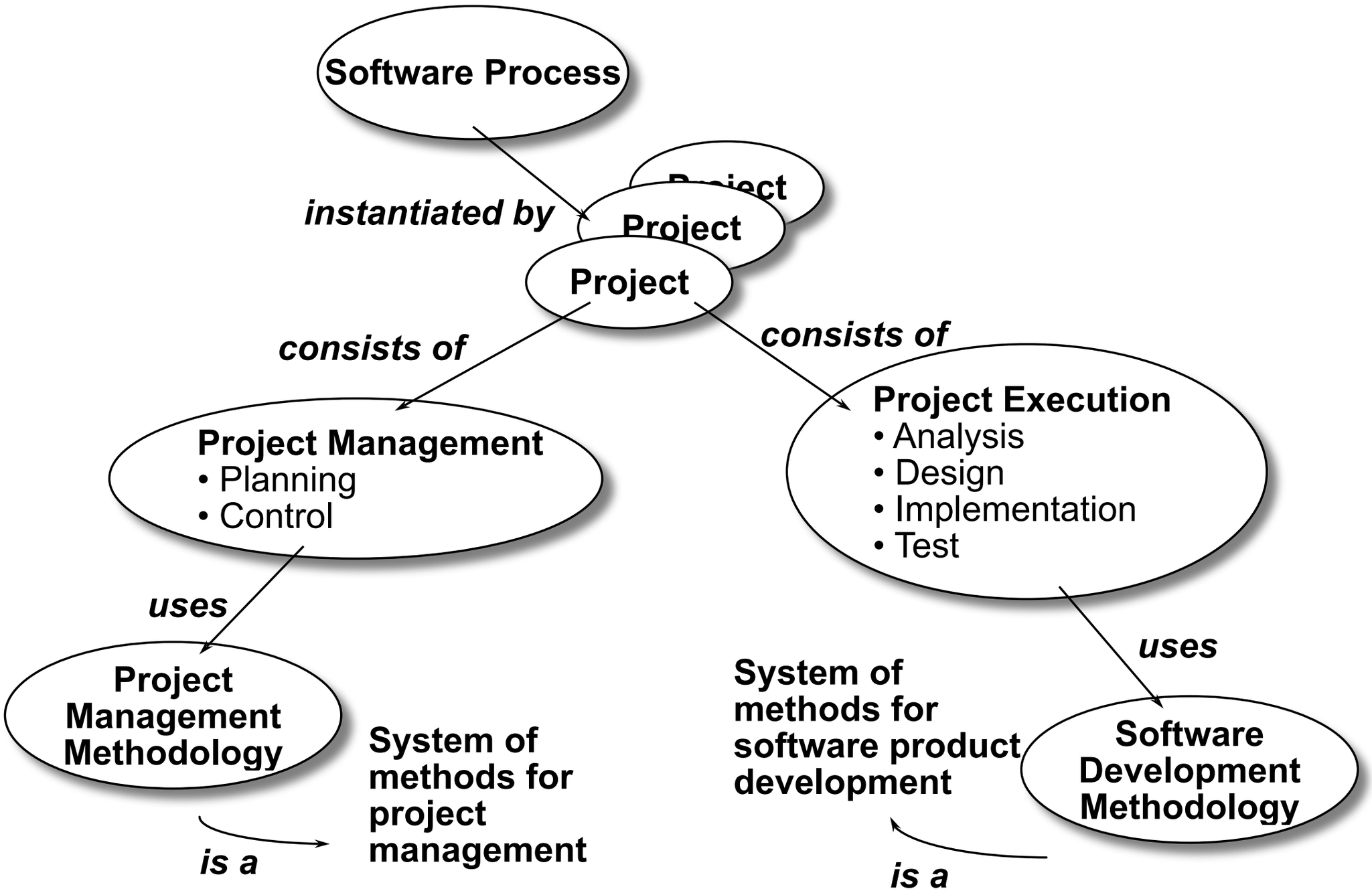
Definition of Software Engineering

Software engineering is an engineering discipline concerned with practical problems of developing large software systems.

Software engineering involves:

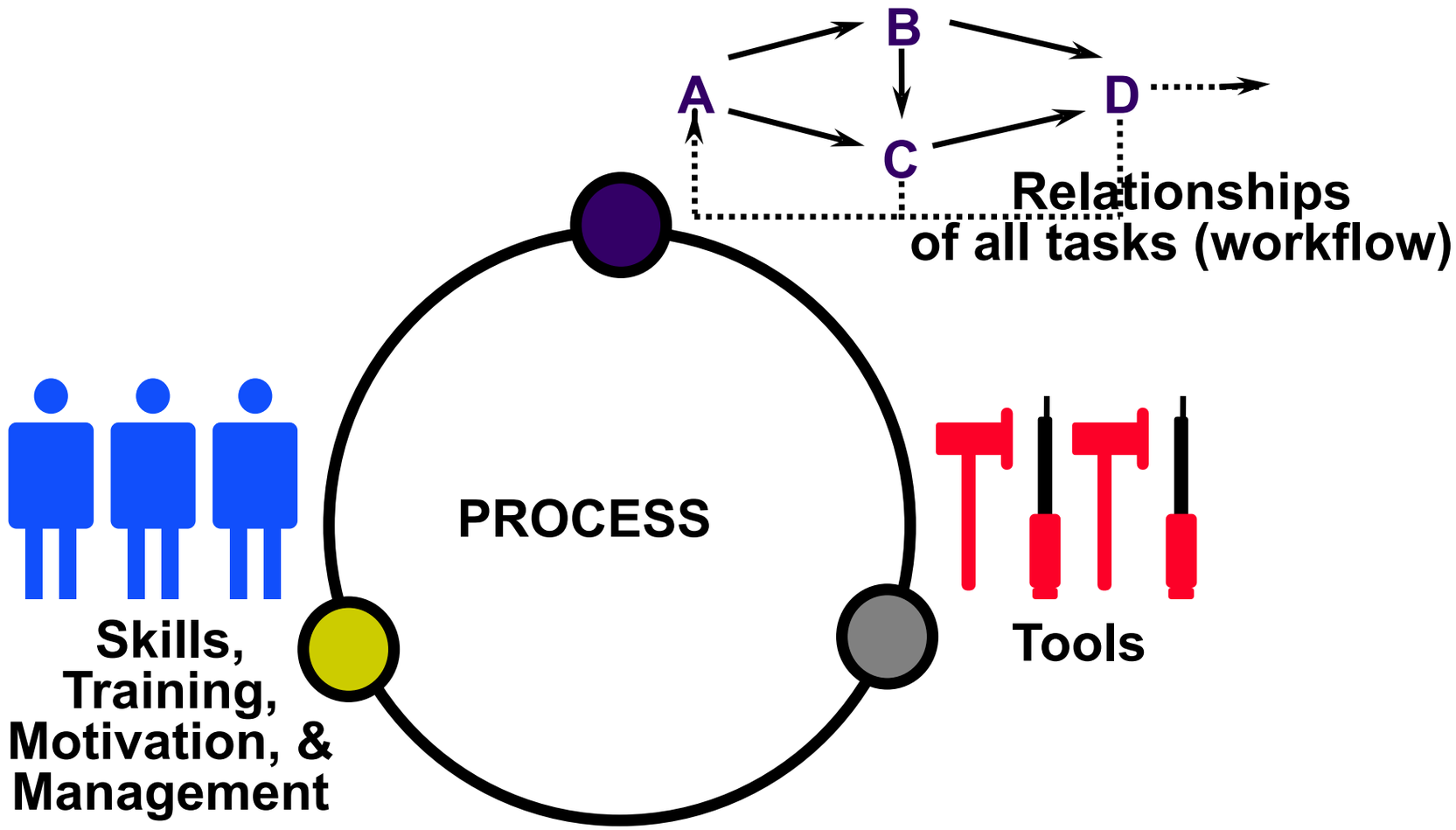
- technical and non-technical issues
- knowledge of specification, design and implementation techniques
- human factors
- software management

Software Production Layout

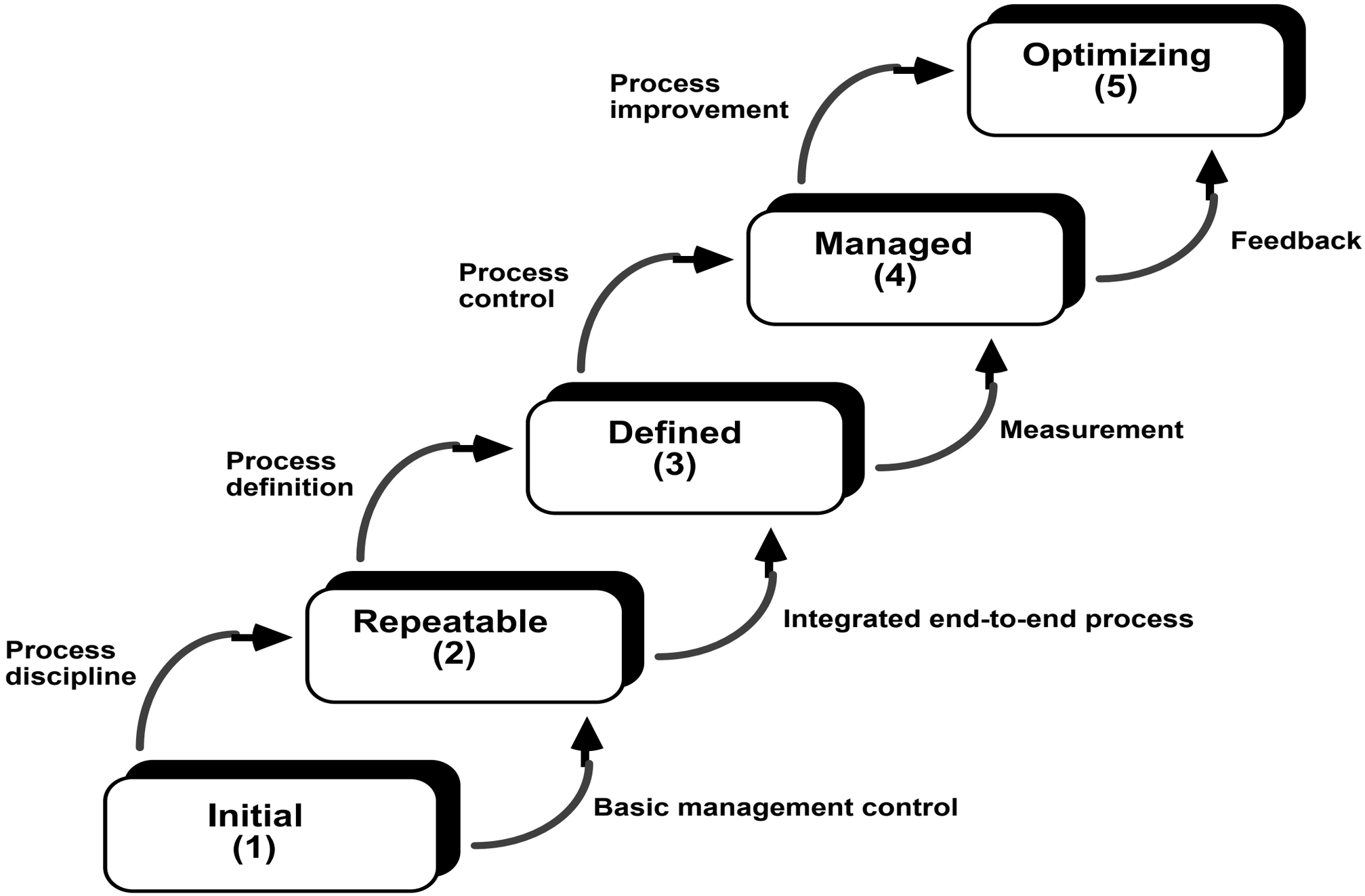


A Definition of Process

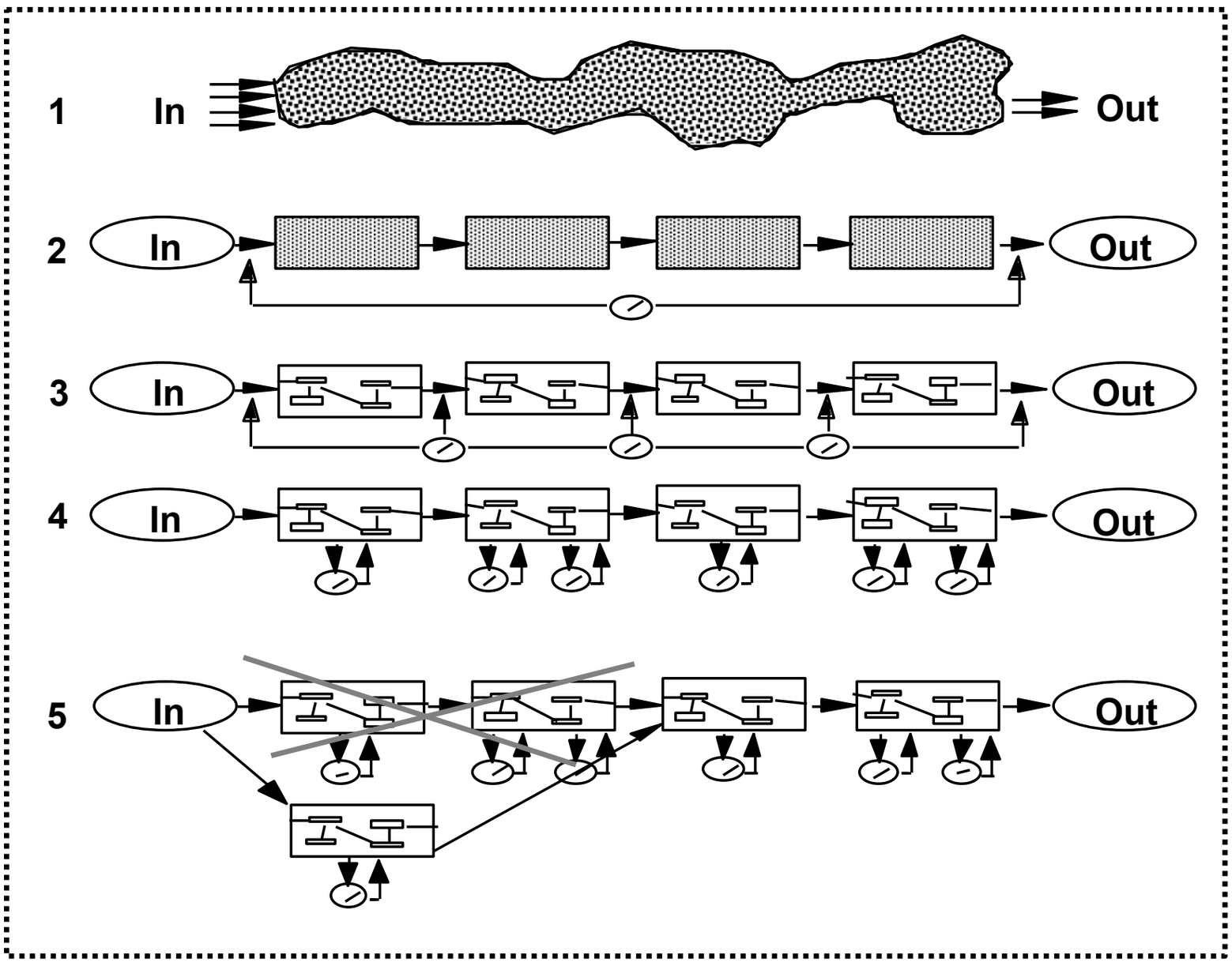
W. Humphrey and P. Feiler: ***"A process is a set of partially ordered steps intended to reach a goal..."(to produce and maintain requested software deliverables).*** **A software process includes sets of related artifacts, human and computerized resources, organizational structures and constraints.**



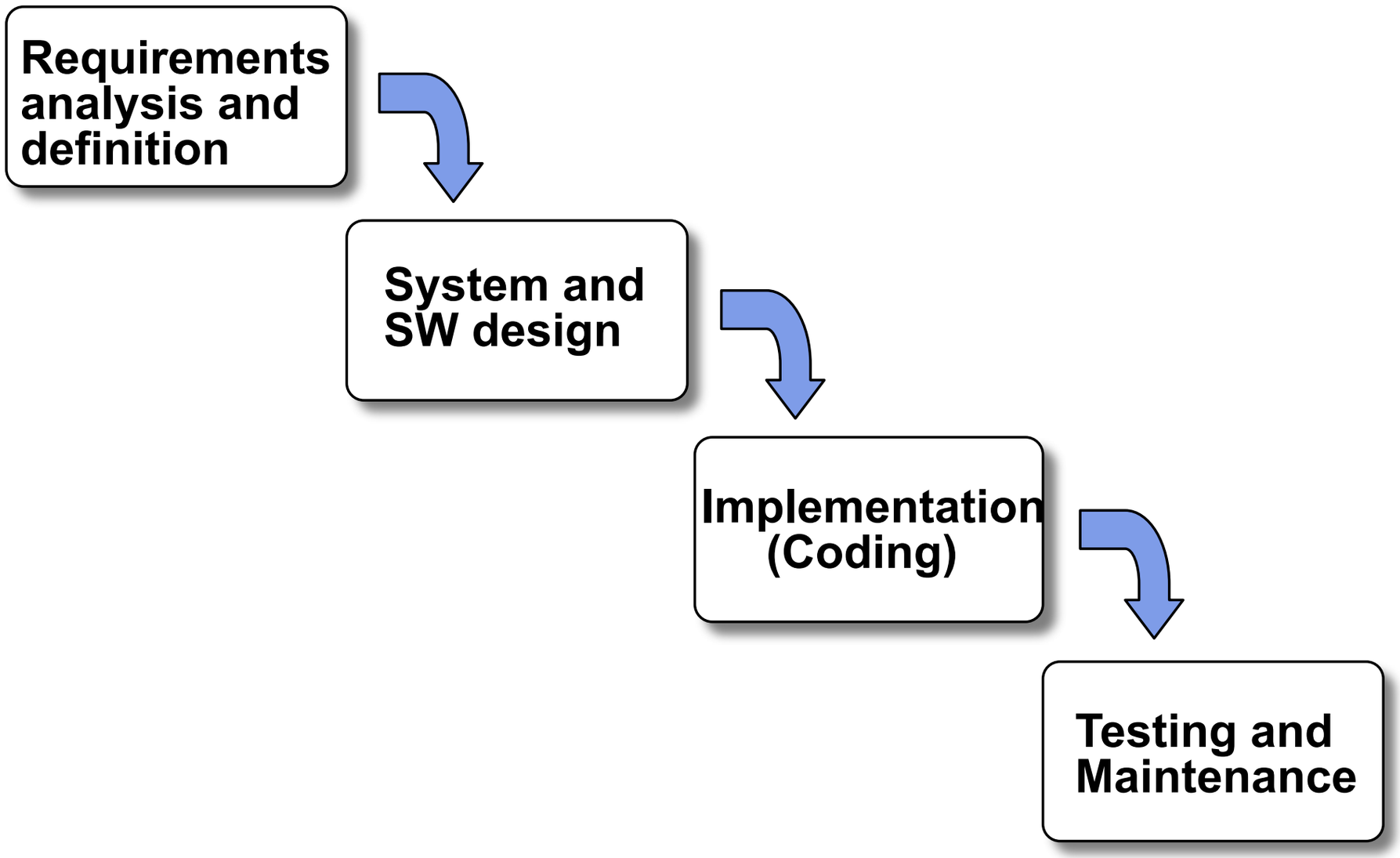
Maturity Levels



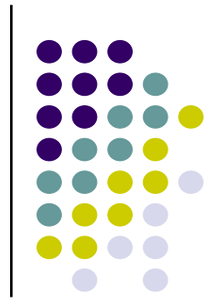
Visibility Into the SW Process



The Waterfall Process Model

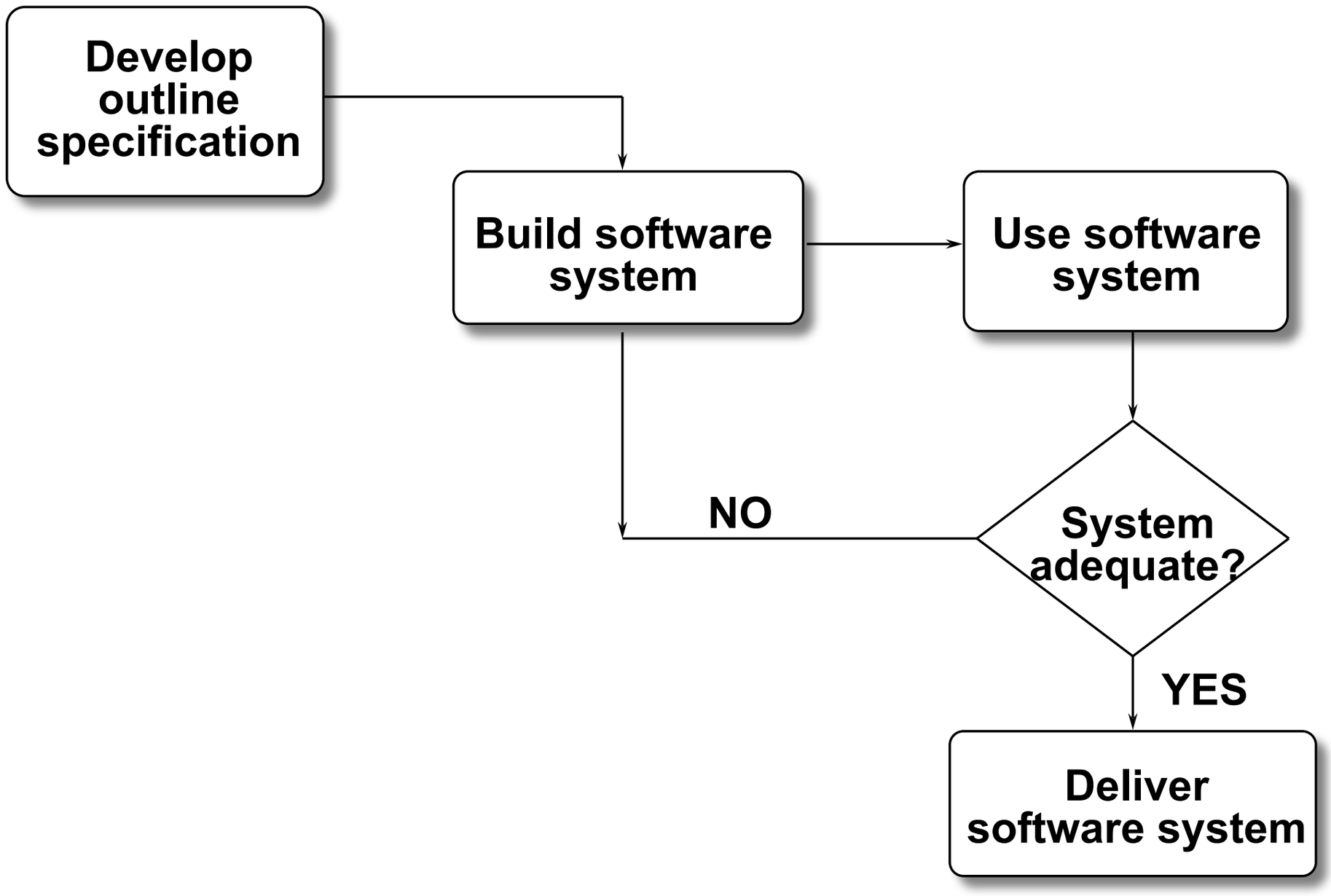


The Waterfall Model: Problems

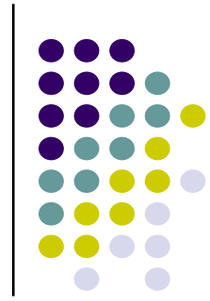


- It takes too long to see results: nothing is executable or demonstrable until code is produced.
- It depends on stable, correct requirements.
- It delays the detection of errors until the end.
- It does not promote software reuse.
- It does not promote prototyping.
- ...

Exploratory Programming



Rational Unified Process®

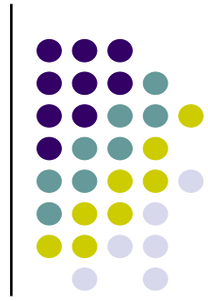


The Rational Unified Process® is a Software Engineering Process. It provides a disciplined approach to assigning tasks and responsibilities within a development organization. Its goal is to ensure the production of high-quality software that meets the needs of its end-users, within a predictable schedule and budget.

Best Practices:

- Develop software iteratively
- Manage requirements
- Use component-based architectures
- Visually model software
- Verify software quality
- Control changes to software

Develop Software Iteratively



Given today's sophisticated software systems, it is not possible to sequentially first define the entire problem, design the entire solution, build the software and then test the product at the end.

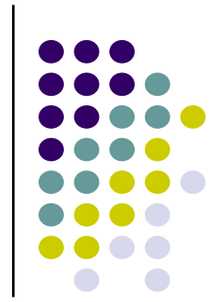
An iterative approach is required that allows an increasing understanding of the problem through successive refinements, and to incrementally grow an effective solution over multiple iterations.

Each iteration ends with an executable release.

Manage Requirements

The Rational Unified Process describes how to **elicit, organize, and document required functionality** and constraints; track and document tradeoffs and decisions; and easily capture and communicate business requirements.

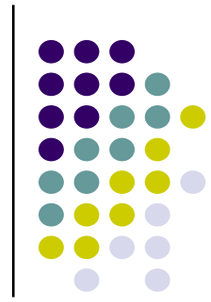
Use Component-based Architectures



The Rational Unified Process provides a systematic approach to **defining an architecture using new and existing components**.

These are assembled in a well-defined architecture, either ad hoc, or in a component infrastructure such as the Internet, CORBA, and COM, for which an industry of reusable components is emerging.

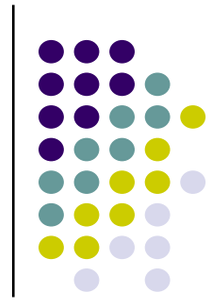
Visually Model Software



The process shows you how to visually model software to capture the structure and behavior of architectures and components. This allows you to hide the details and write code using “**graphical building blocks**”.

The industry-standard **Unified Modeling Language** (UML), created by Rational Software, is the foundation for successful visual modeling.

Verify Software Quality



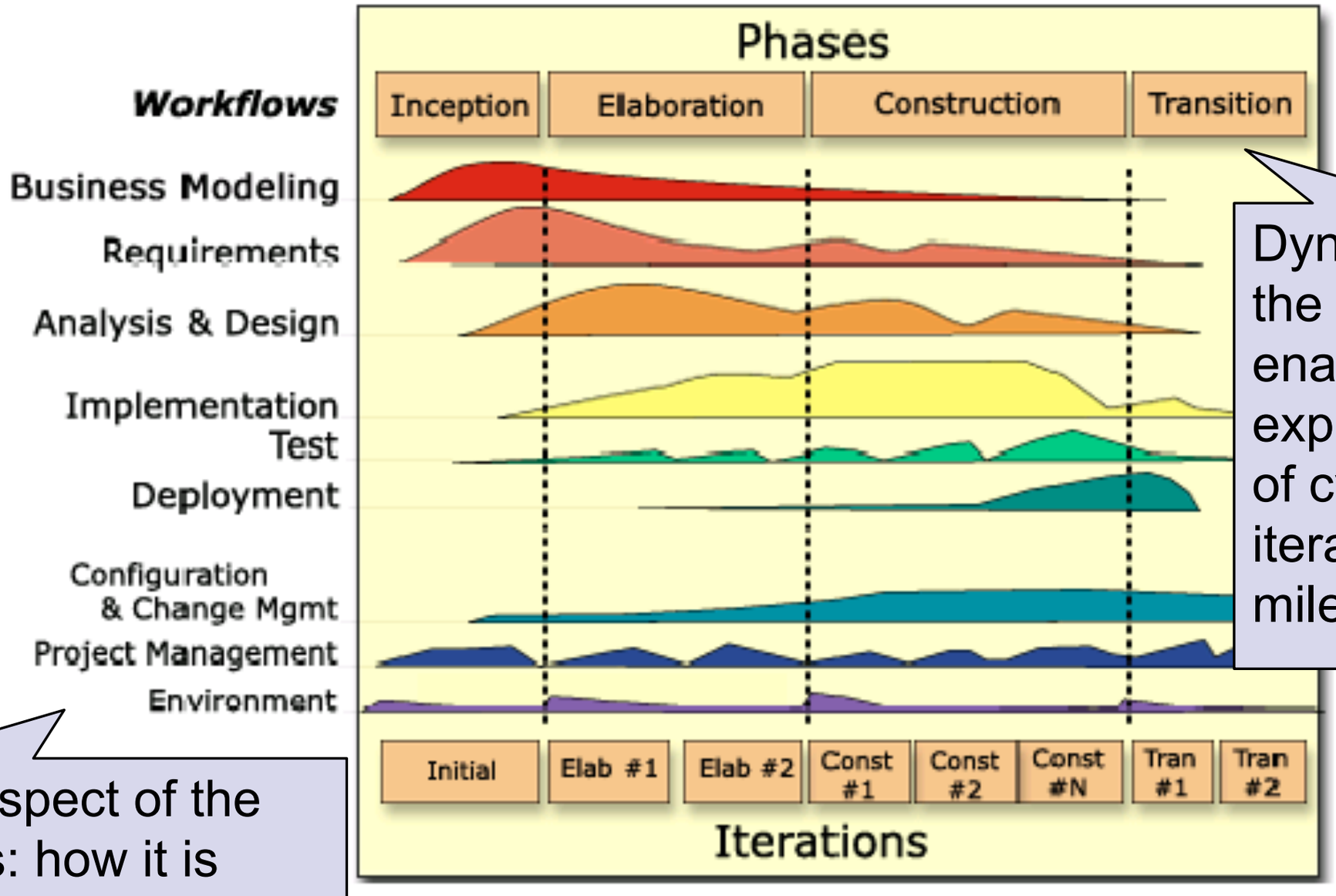
Quality assessment is built into the process, in all activities, involving all participants, using objective measurements and criteria, and not treated as an afterthought or a separate activity performed by a separate group.

Control Changes to Software

The ability to manage change - **making certain that each change is acceptable, and being able to track changes** - is essential in an environment in which change is inevitable.

The process describes how to control, track and monitor changes to enable successful iterative development. It also guides you in how to establish secure workspaces for each developer by providing isolation from changes made in other workspaces and by controlling changes of all software artifacts (e.g., models, code, documents, etc.).

Two Dimensions of the Process



RUP Overview Diagram

Dynamic aspect of the process as it is enacted: it is expressed in terms of cycles, phases, iterations, and milestones.

Static aspect of the process: how it is described in terms of activities, artifacts, workers and workflows.

Cycles and Phases

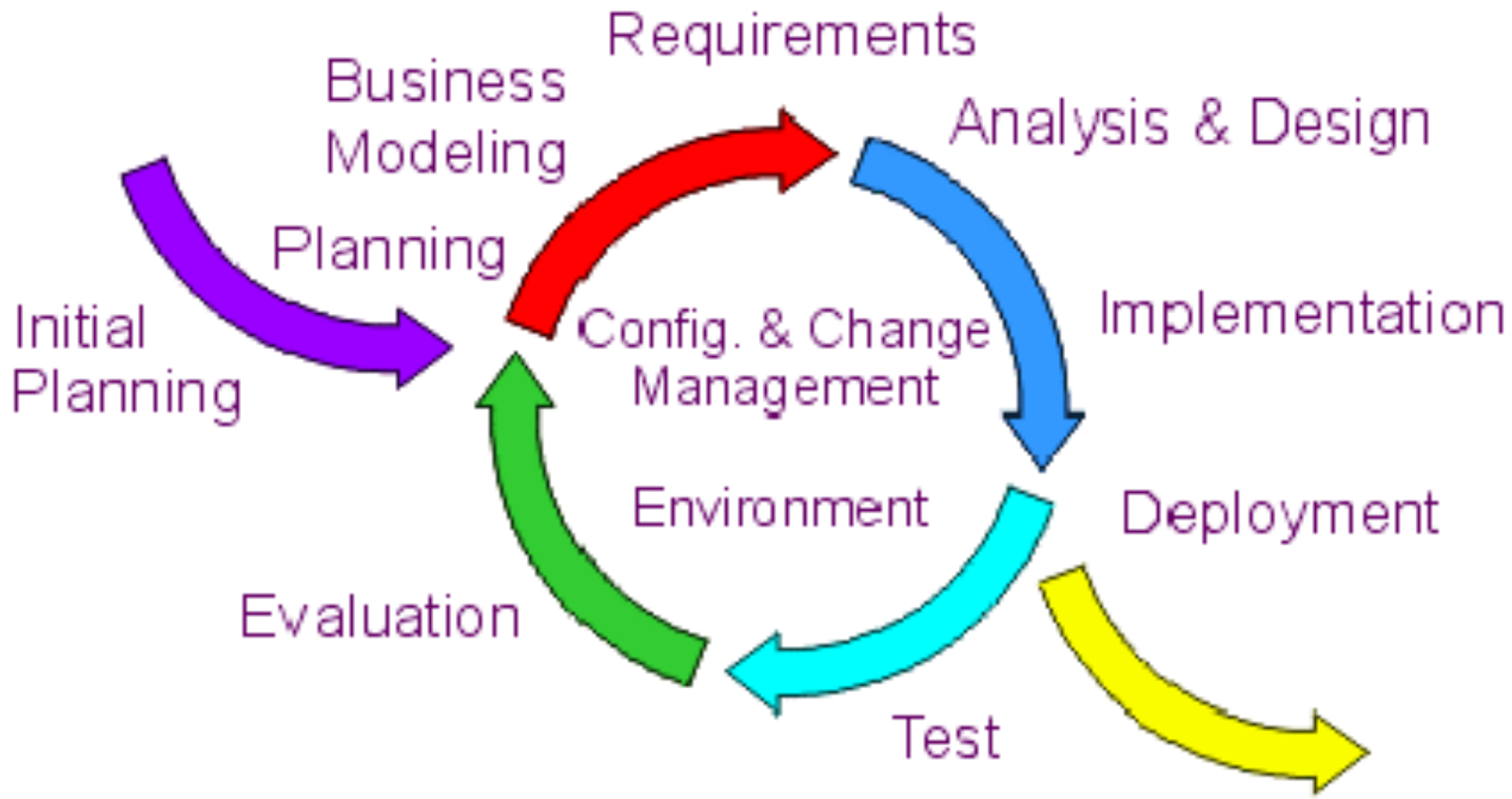
Each cycle results in a new release of the system, and each is a product ready for delivery. This product has to accommodate the specified needs.

The development cycle is divided in **four consecutive phases**

- **Inception:** a good idea is developed into a vision of the end product and the business case for the product is presented.
- **Elaboration:** most of the product requirements are specified and the system architecture is designed.
- **Construction:** the product is built – completed software is added to the skeleton (architecture)
- **Transition:** the product is moved to user community (beta testing, training …)

Iterations

Each phase 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 iteration to become the final system.



Typical Iteration Flow

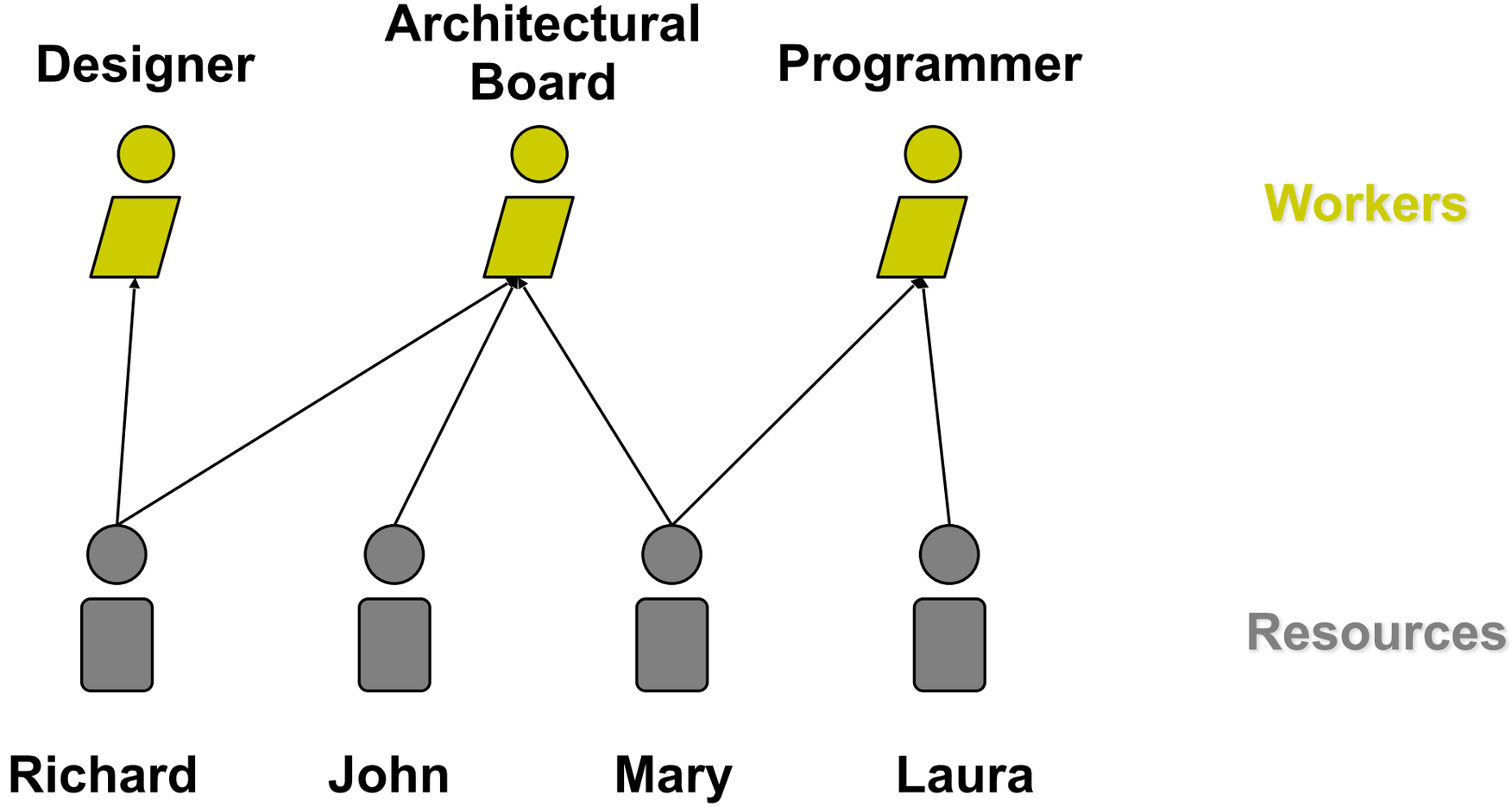
Static Structure of the Process

A process describes **who** is doing **what**, **how**, and **when** using following modeling elements:

- **Workers** (Roles) define the behavior and responsibilities of an individual (designer, analyst, programmer ...), or a group of individuals working together as a team.
- **Artifacts** are things that are produced, modified, or used by a process (model, document, source code …).
- **Activities** are performed by workers to create or update some artifacts (review design, compile code, perform test …).
- **Workflows** are sequences of activities that produce results of observable value (business modeling, implementation …).

Resources and Workers

Allocating resources (individuals) to workers means matching competencies of individuals with the required competencies of the workers.



Management and Technical Artifacts

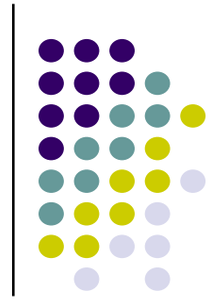
The most important kind of artifact are models.

A model is a simplification of reality, created to better understand the system being created.

Technical artifacts may be divided into:

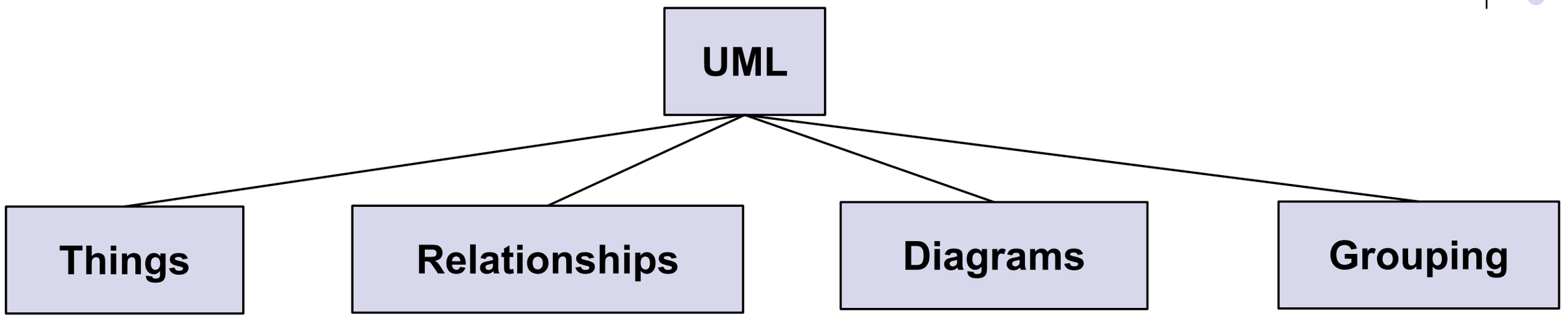
- **Requirements set:** business, domain, use case, and analysis models
- **Design set:** design, and test models
- **Implementation set:** implementation model, source code, configuration, and data files
- **Deployment set:** deployment model, information about the way software is actually packaged

The Unified Modeling Language



- 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

Building Blocks of the UML



Use case
Object
Class
Interface
Component
Node

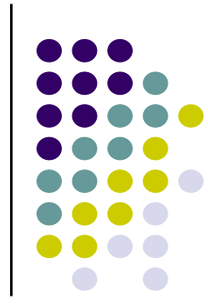
Dependency
Association
Generalization
Realization

Use case
Class
Sequence
Collaboration
Statechart
Activity
Component
Deployment

Package
Subsystem
Model

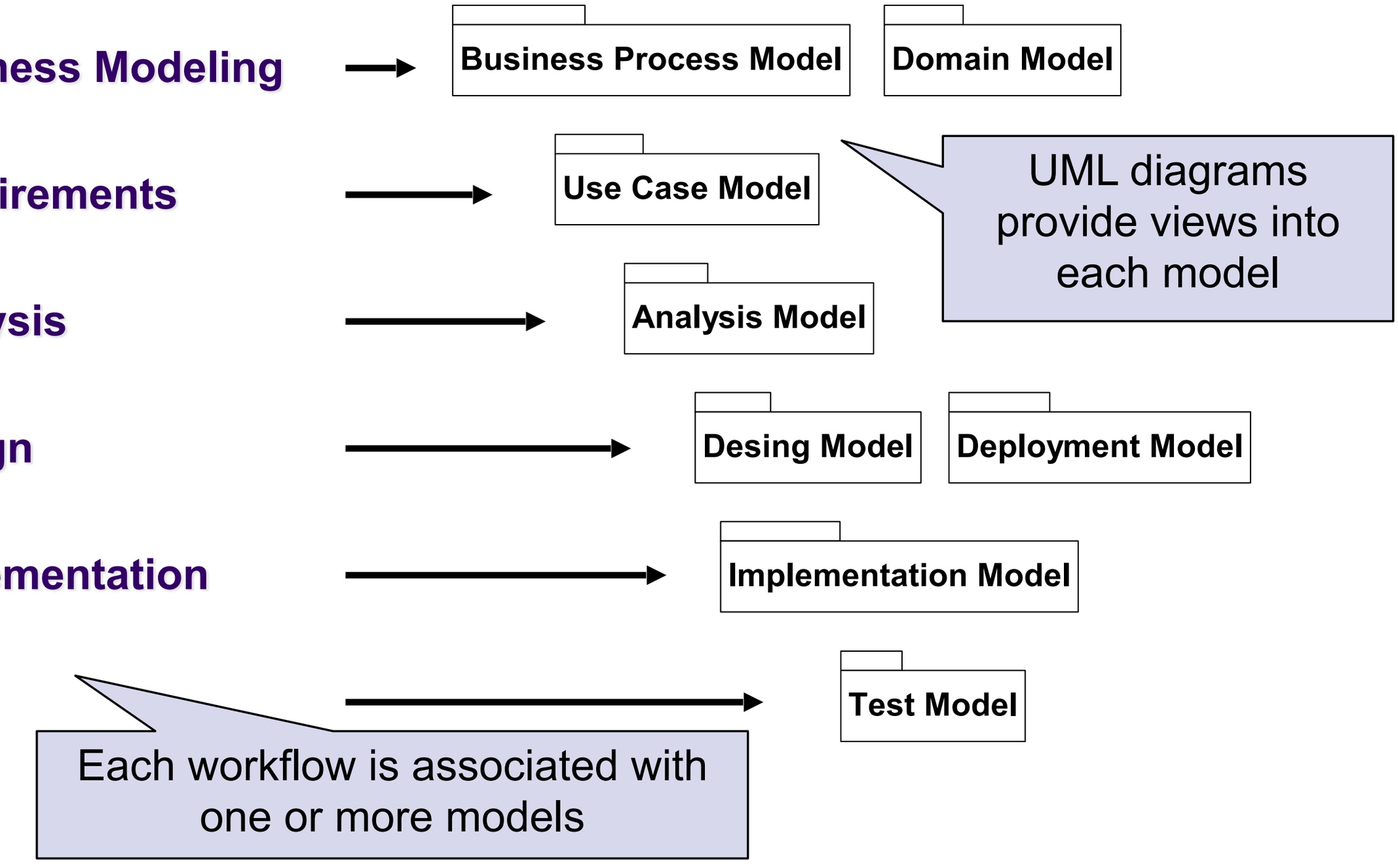
http://www.uml.org

Core Engineering Workflows



- **Business modeling** describes the structure and dynamics of the organization
- **Requirement** describe the use case-based method for eliciting requirements
- **Analysis and design** describe the multiple architectural views
- **Implementation** takes into account sw development, unit test, and integration
- **Test** describes test cases and procedures
- **Deployment** covers the deliverable system configuration

Workflows and Models



Core Supporting Workflows

- **Configuration Management** describes how to control the numerous artifacts produced by the many people who work on a common project (simultaneous update, multiple versions …).
- **Project Management** is the art of balancing competing objectives, managing risk, and overcoming constraints to deliver, successfully, a product which meets the needs of both customers (the payers of bills) and the users.
- **Environment Workflow** provides the software development organization with the software development environment—both processes and tools—that are needed to support the development team.

Exercises

- What is the software engineering about? What is the **definition of the software negineering**?
- What is **the difference between software process and project**? What is the relationship between these both?
- Draw the layout of Rational Unified Process and define what the **cycles, phases and iterations** mean!

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.

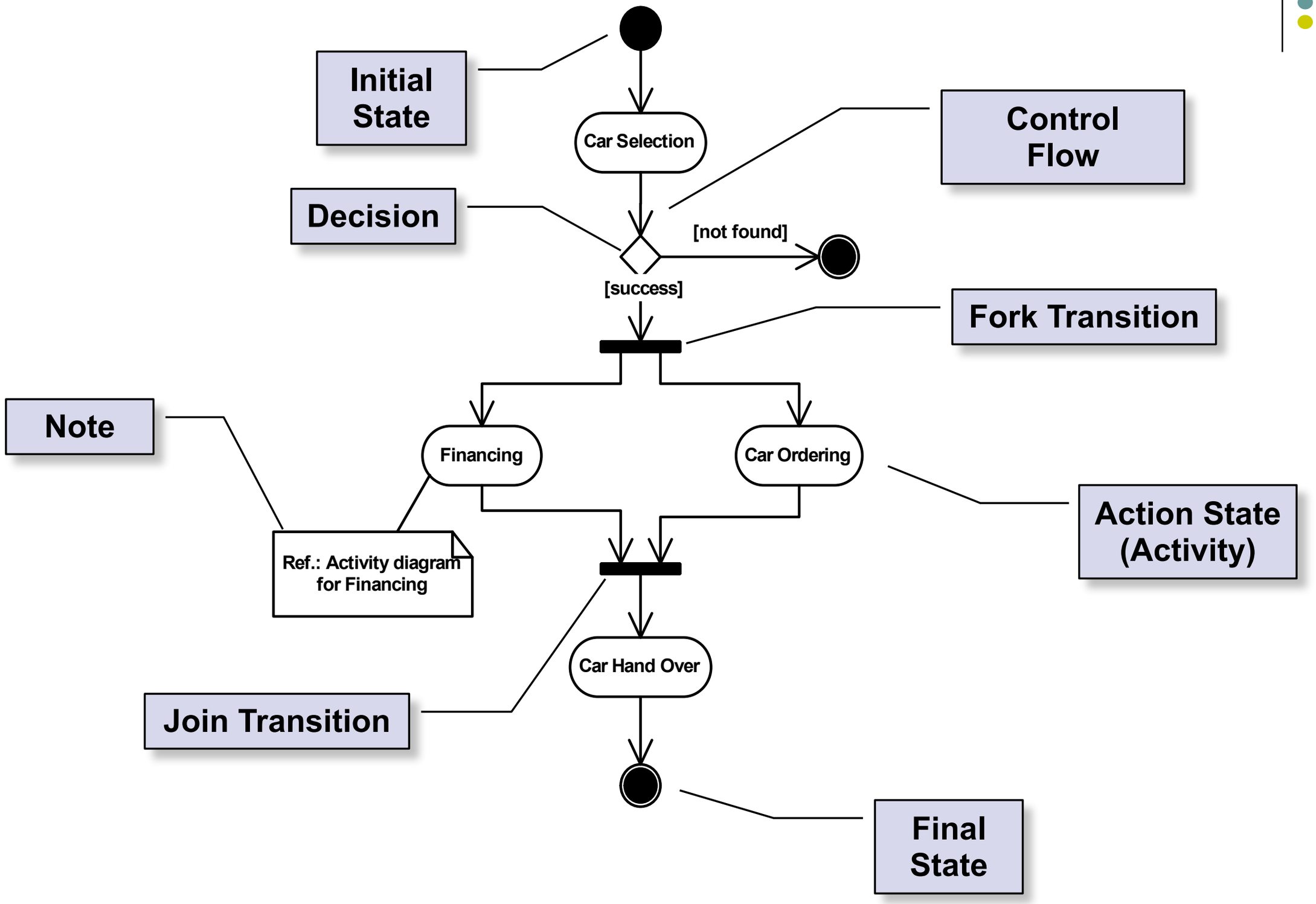
Motivating Example

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. The goal is to make the customer happy!

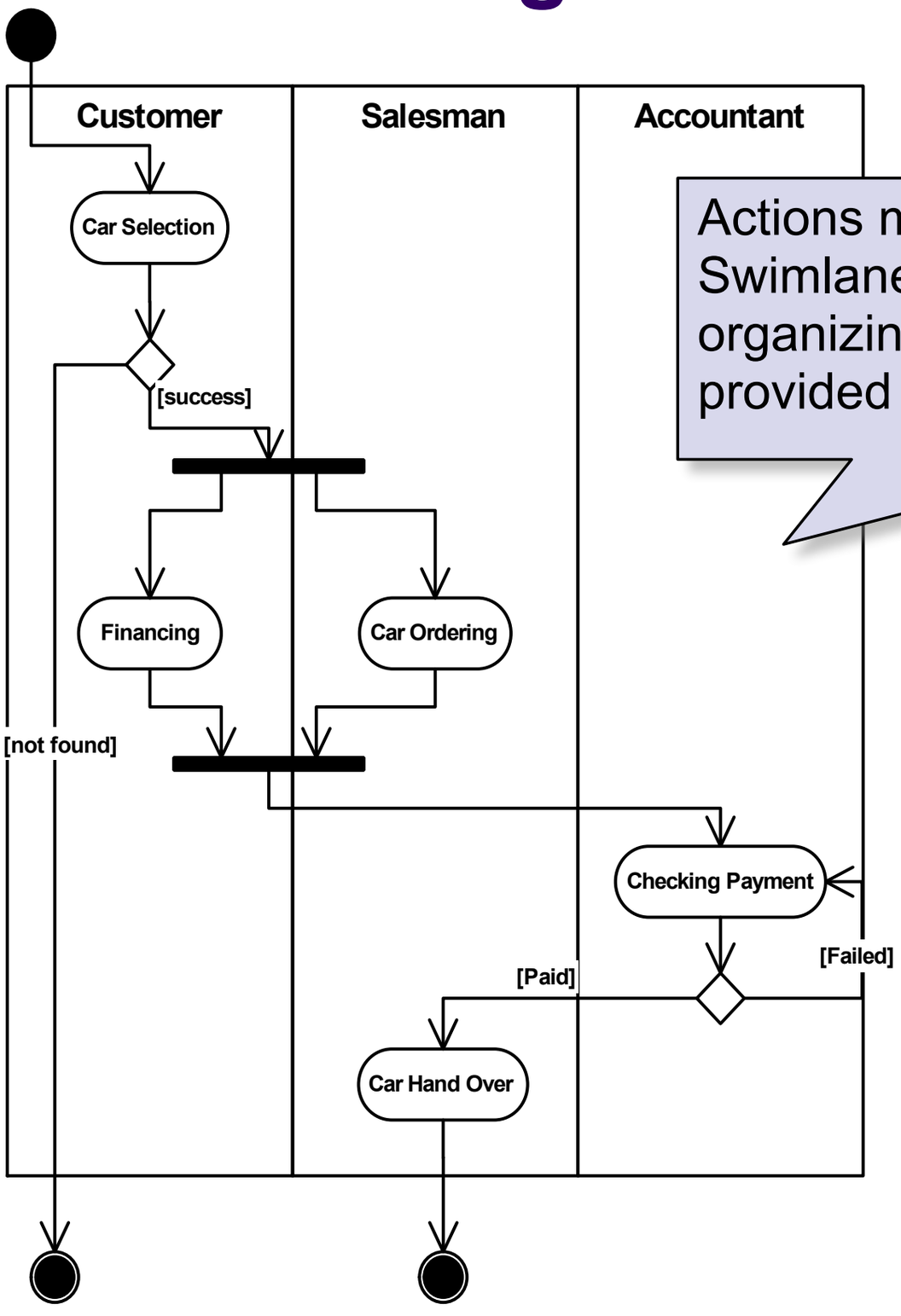
UML Diagrams for Business Modeling

- **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

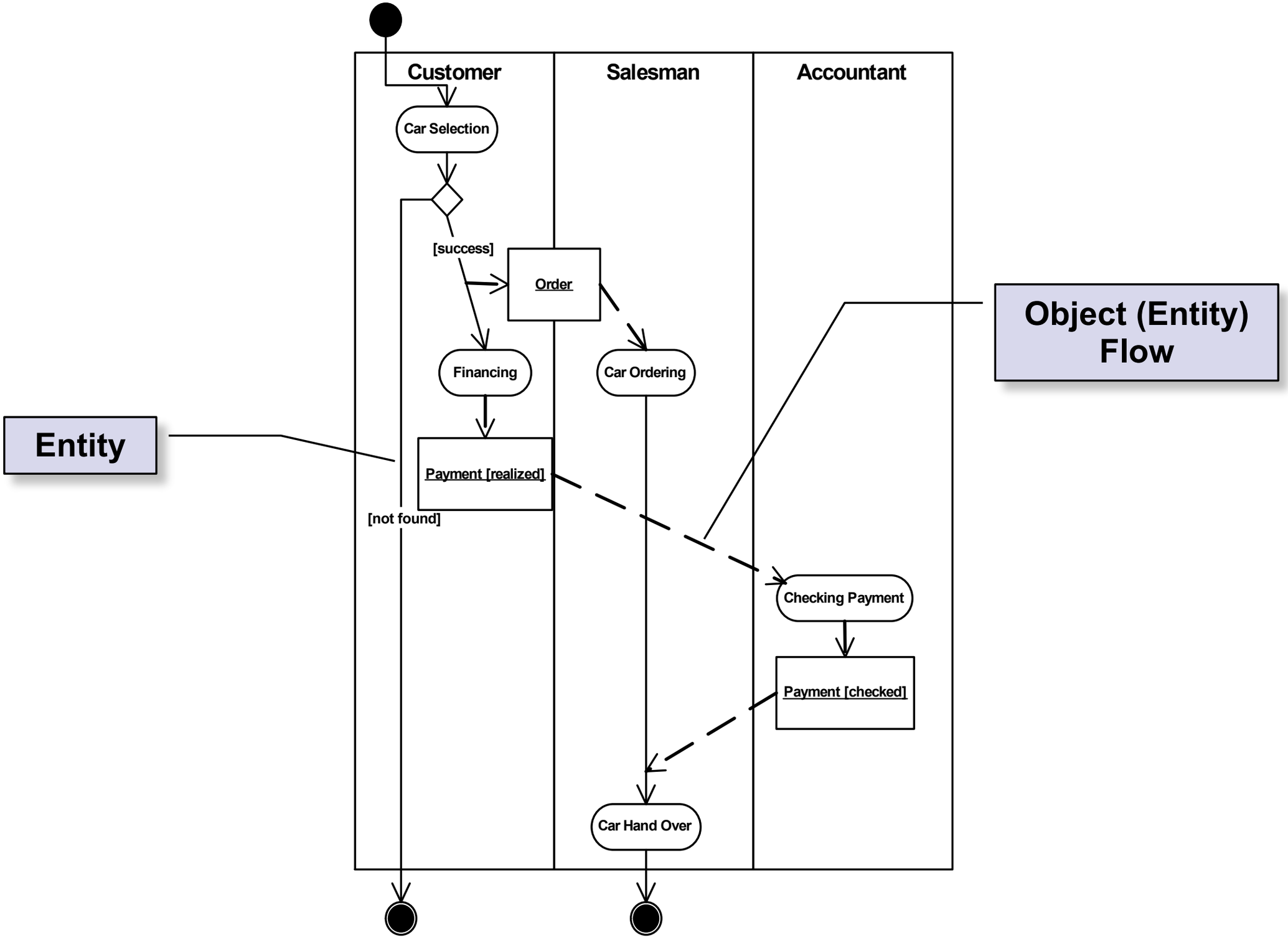


Swimlanes: Packages of Responsibilities

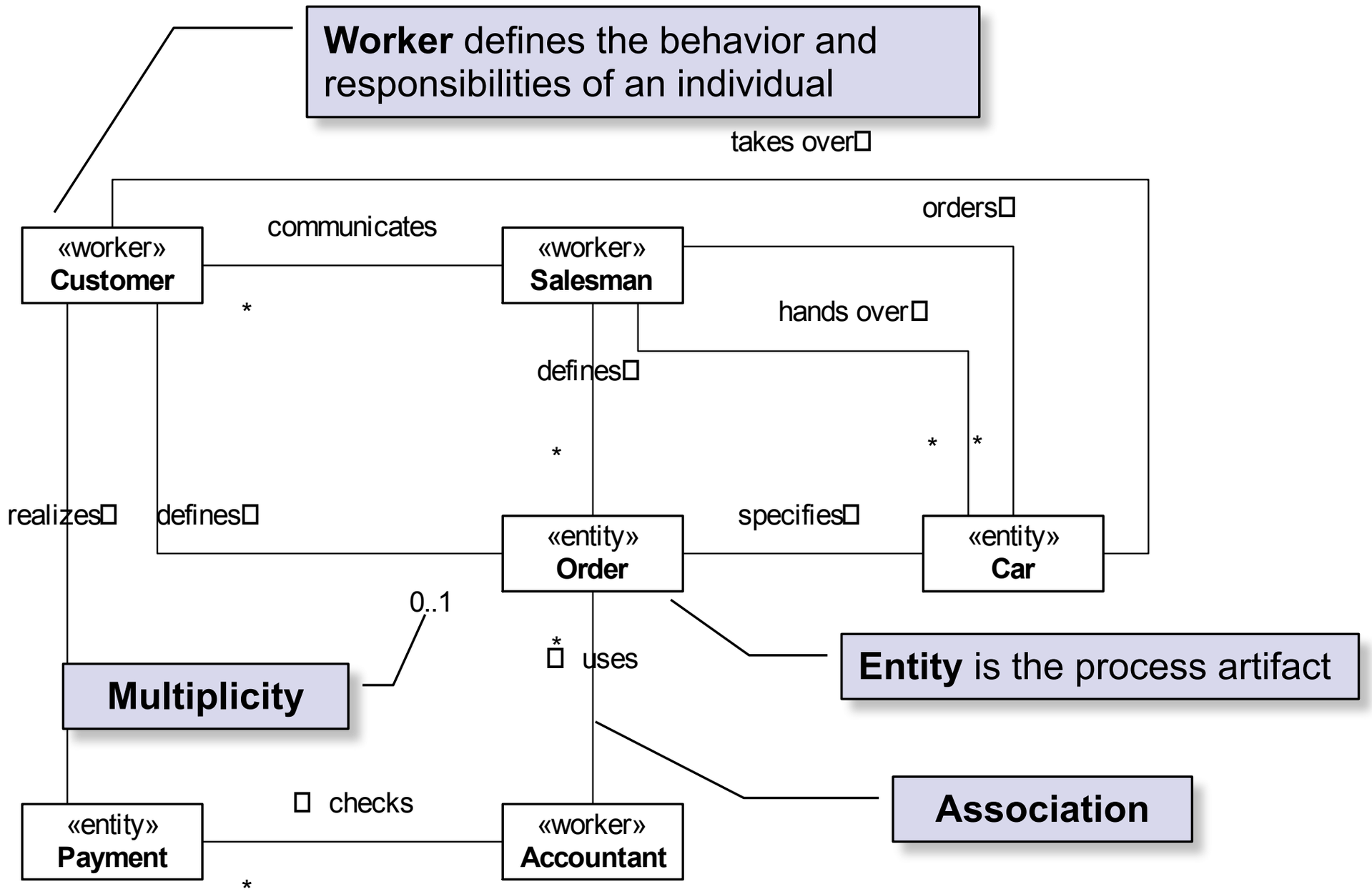


Actions may be organized into **swimlanes**. Swimlanes are a kind of package for organizing **responsibility for activities** provided by workers.

Activities and Entities



Class Diagram: Car Sale Elements



Exercises

- Install the Poseidon CASE from the web site www.gentleware.com.
- Specify **activity diagrams** for the business processes typical for video lending library.

Alternative Approaches

- Integrated Definition – IDEF (U.S. Air Force)
  - http://www.idef.com
- Architecture of Integrated Information Systems – ARIS (prof. Scheer)
  - http://www.ids-scheer.com
- Business Process Modeling – BPM (prof. Vondrak)
  - http://www.bpr.cz
- …

Requirements

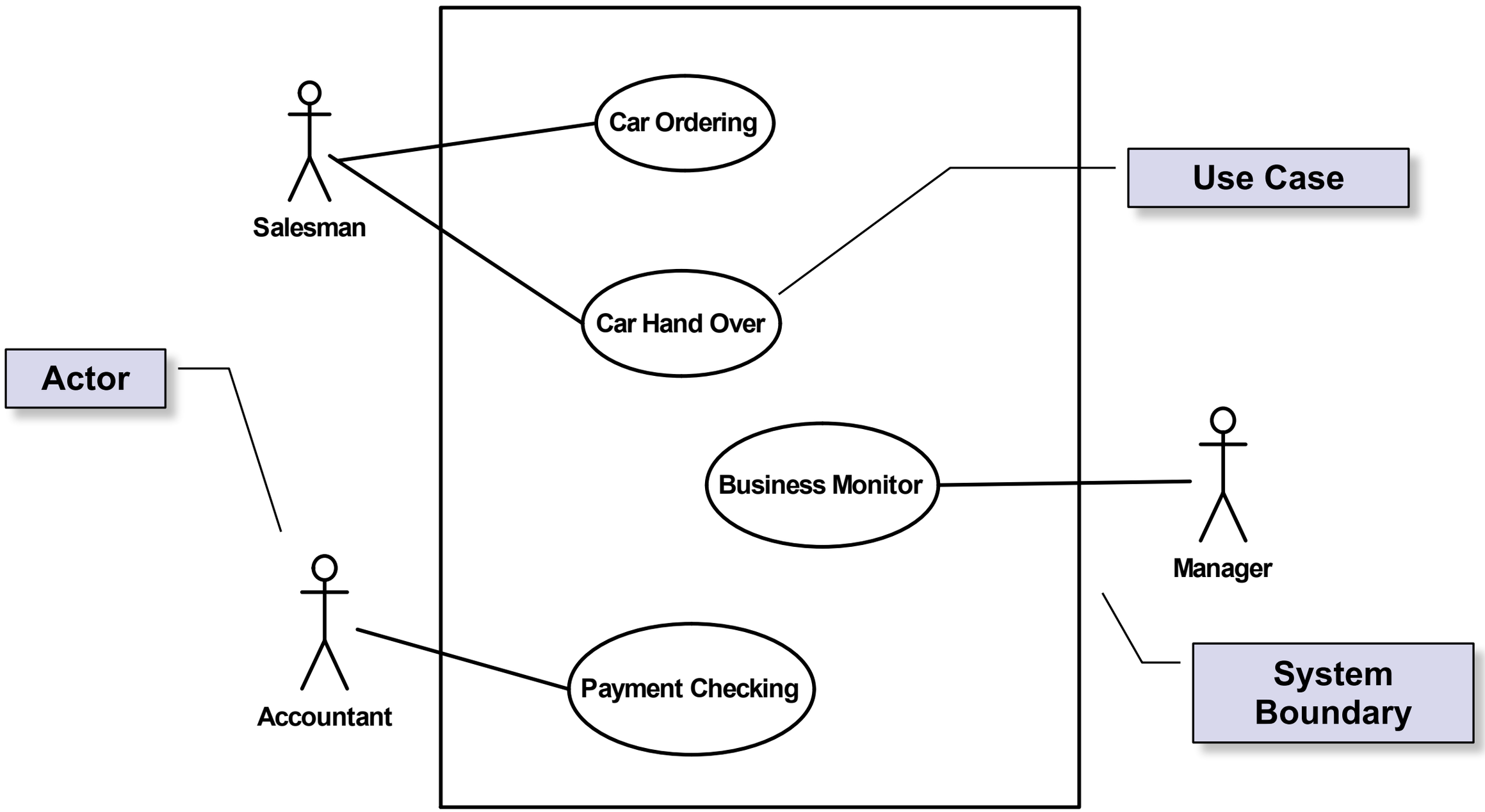
The goal of the requirements workflow is to describe **what** the system should do by specifying its functionality. Requirements modeling allows the developers and the customer to agree on that description.

- **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.

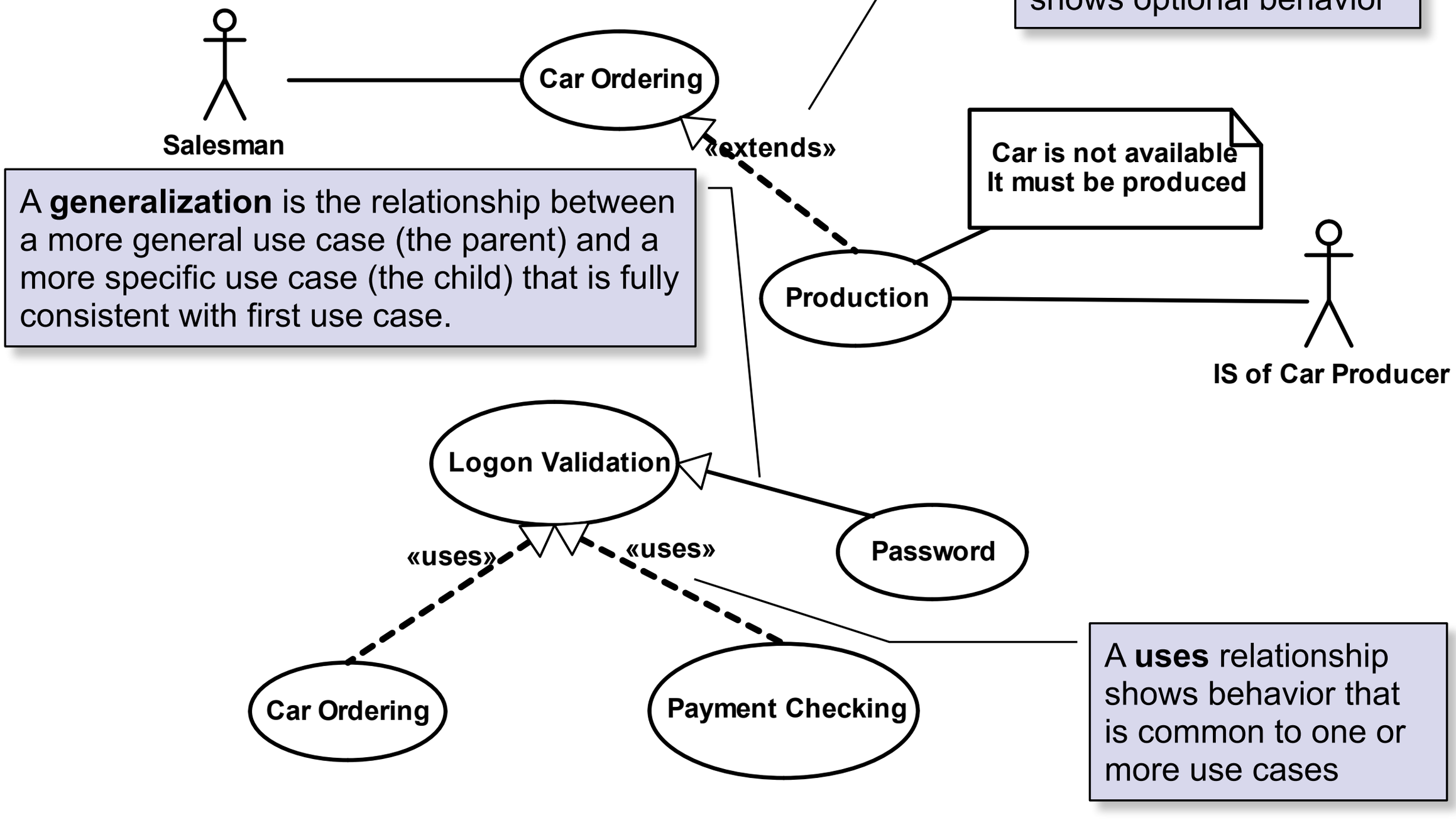
UML Diagrams for Requirements Modeling

- **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

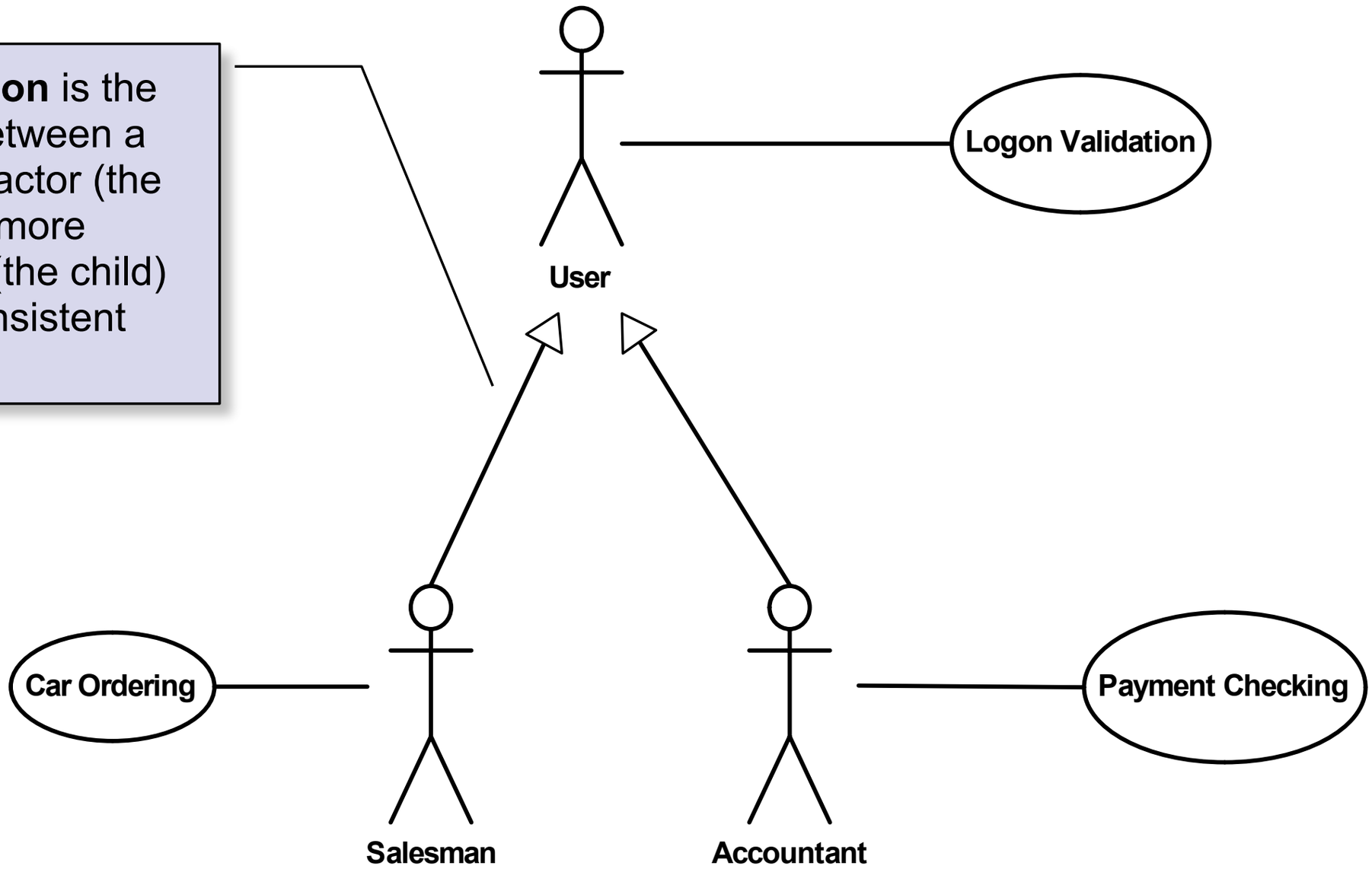


Structuring Use Cases

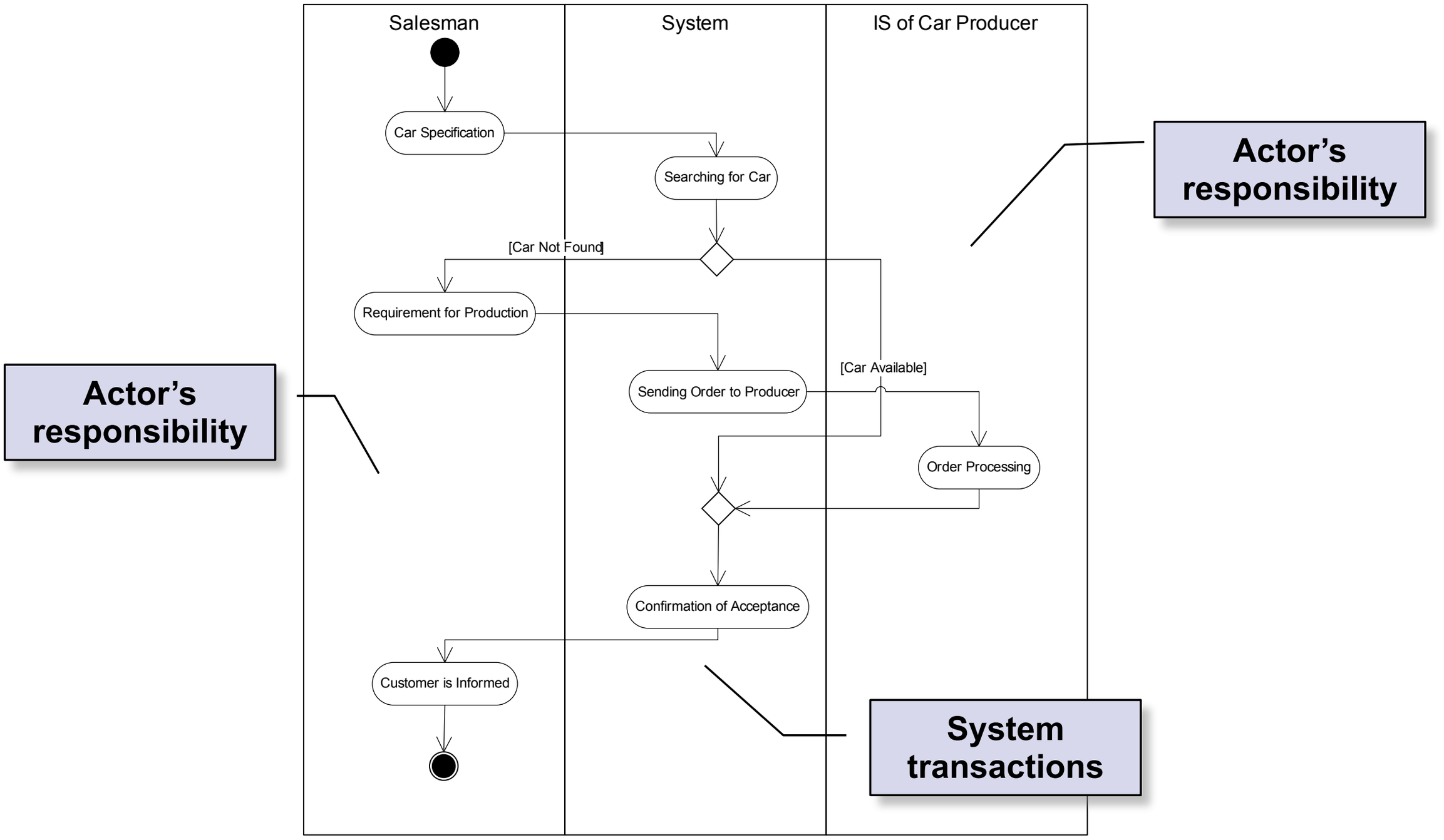


Structuring Actors

A **generalization** is the relationship between a more general actor (the parent) and a more specific actor (the child) that is fully consistent with first actor.



Elaborate Functionality of Car Ordering



Exercises

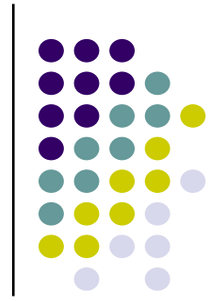
- For the business processes specified during business modeling workflow identify **actors and their use cases**.
- Specify **activity diagram** for the lending process.

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.

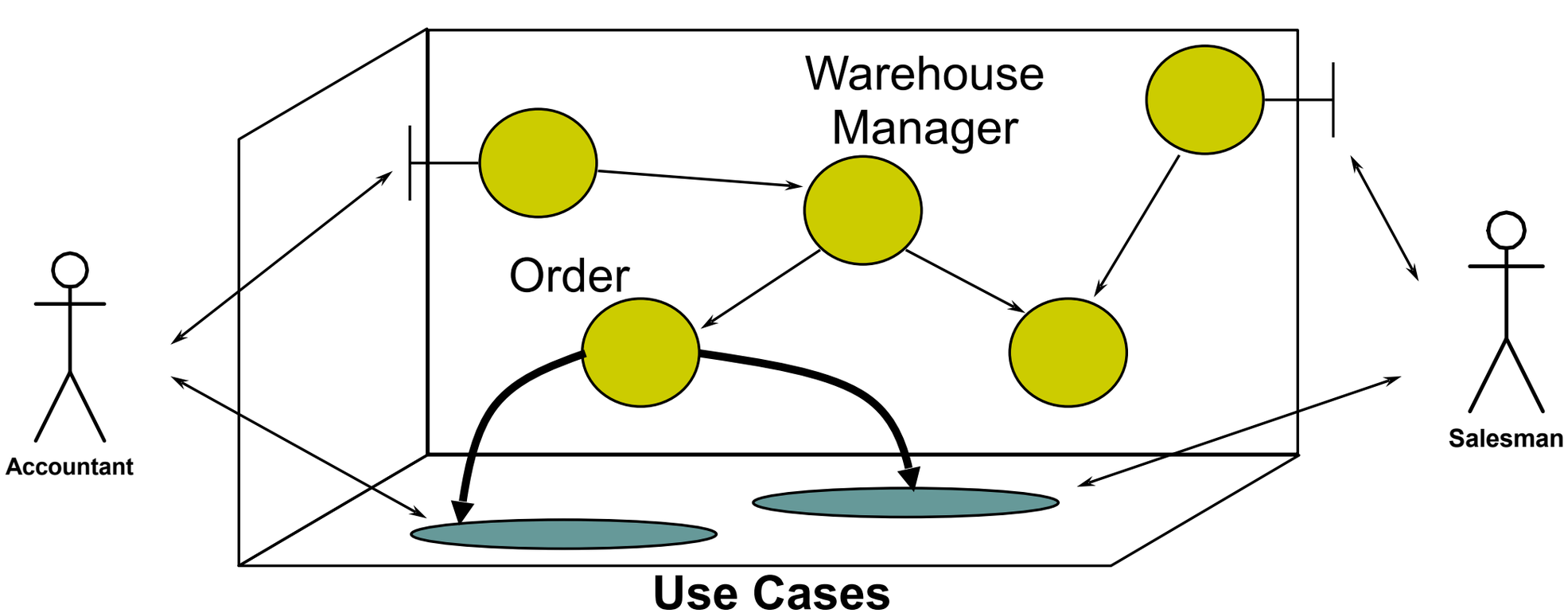
UML Diagrams for Analysis & Design



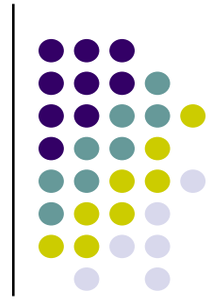
- **Class Diagram** shows set of classes, interfaces and their relationships
  - Class diagrams address the static design view of the system.
- **Sequence Diagram** is an interaction diagram that emphasizes the time ordering of messages.
- **Collaboration Diagram** displays object interactions organized 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?



- **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

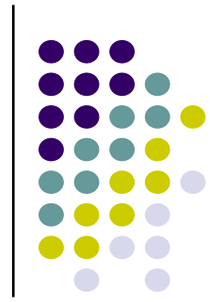
Views of Object

- **External**
  - responsibilities of that object
  - services provided
  - visible properties and associations
  - externally visible protocols and interactions
- **Internal**
  - data representations
  - implementations of behaviors

Relationships Among Objects

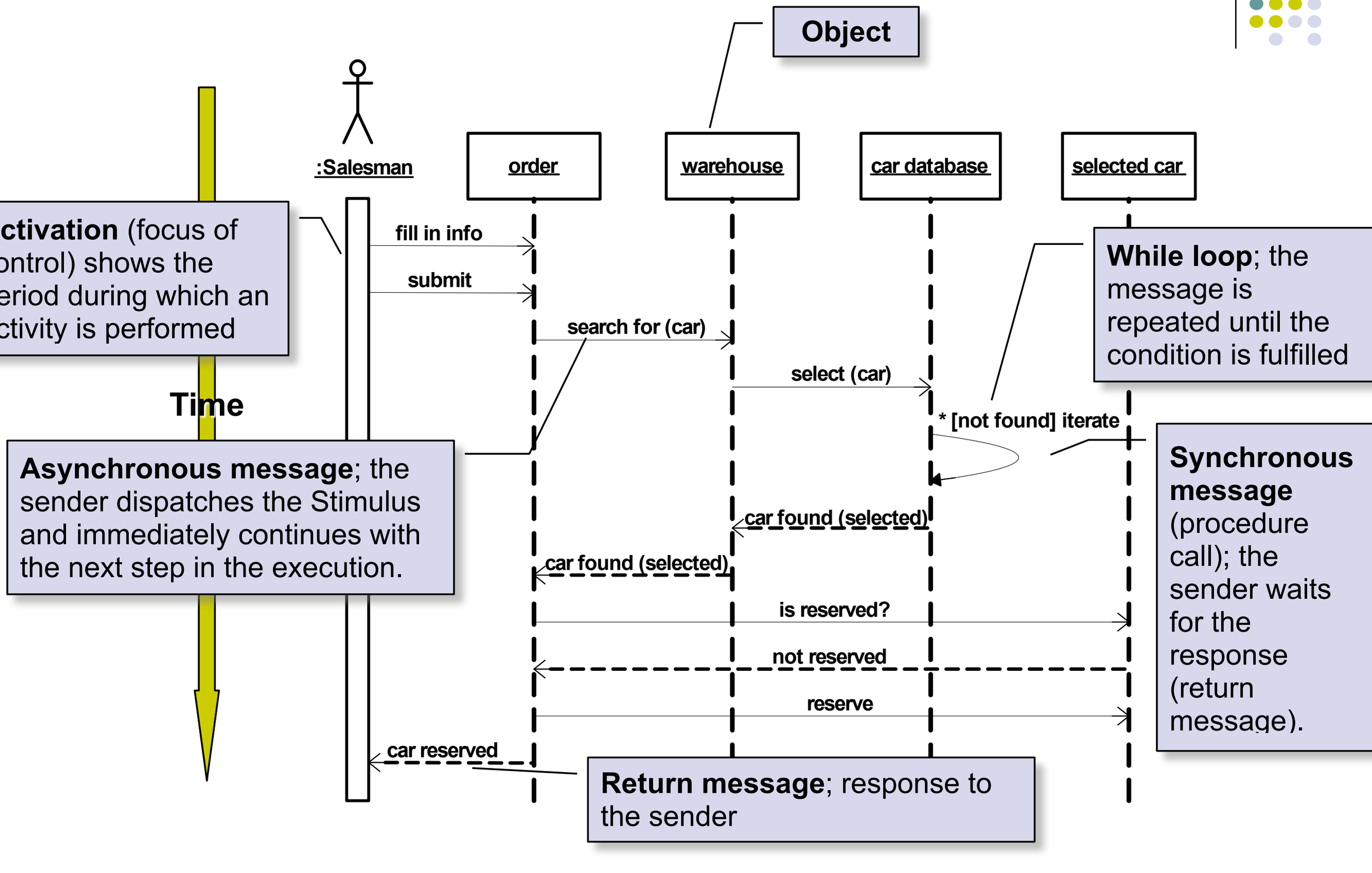
- **Links**
  A link is a physical or conceptual connection between objects (*John Smith* ***works-for*** *Simplex company*). Mathematically, a link is defined as a tuple, that is, an ordered list of objects.

Objects and Their Interactions

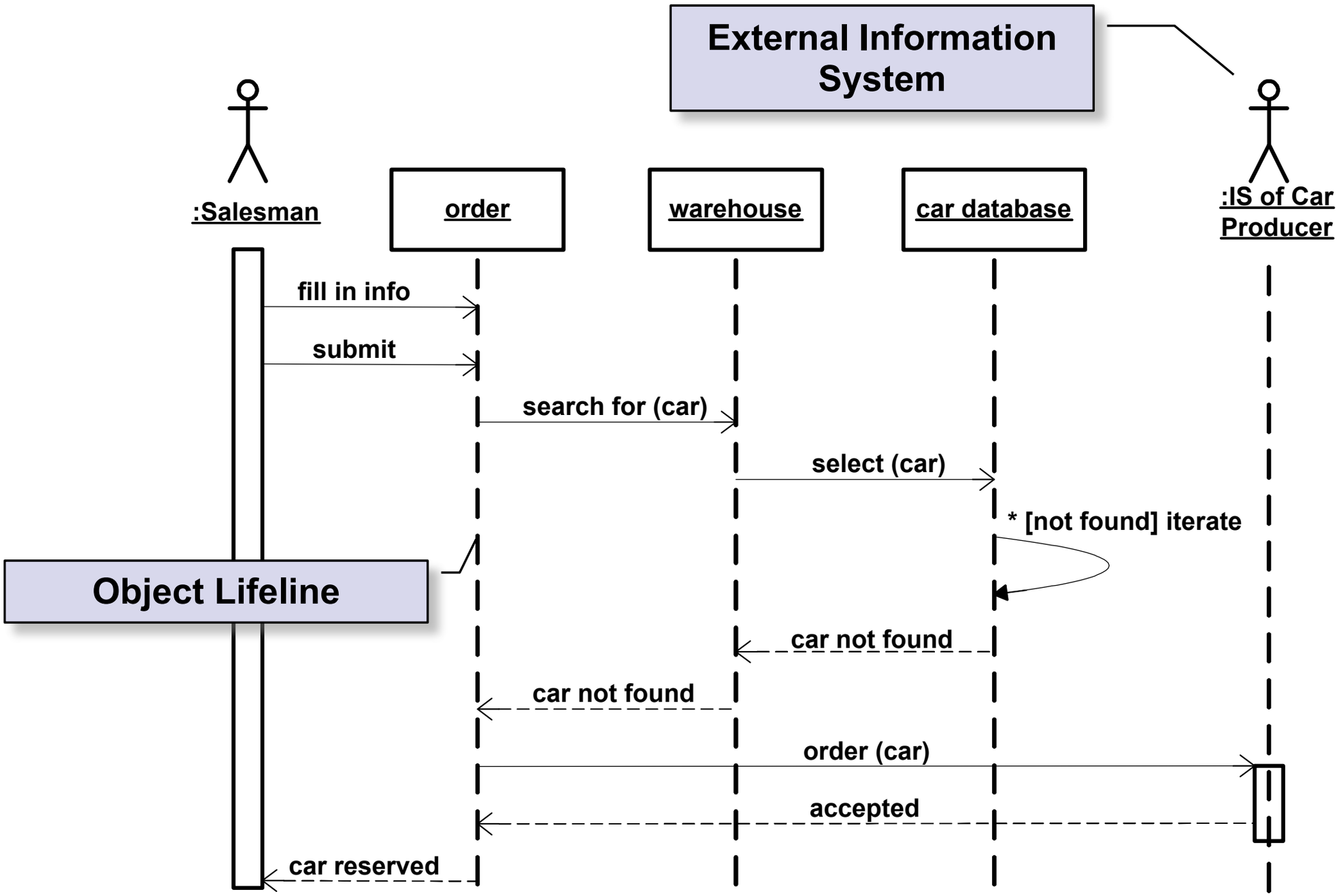


- **A set of interconnected objects constitutes the system**
- **Interactions between objects result in:**
  - Collective behaviors being exercised
  - Changes in the logical configurations and states of the objects and system

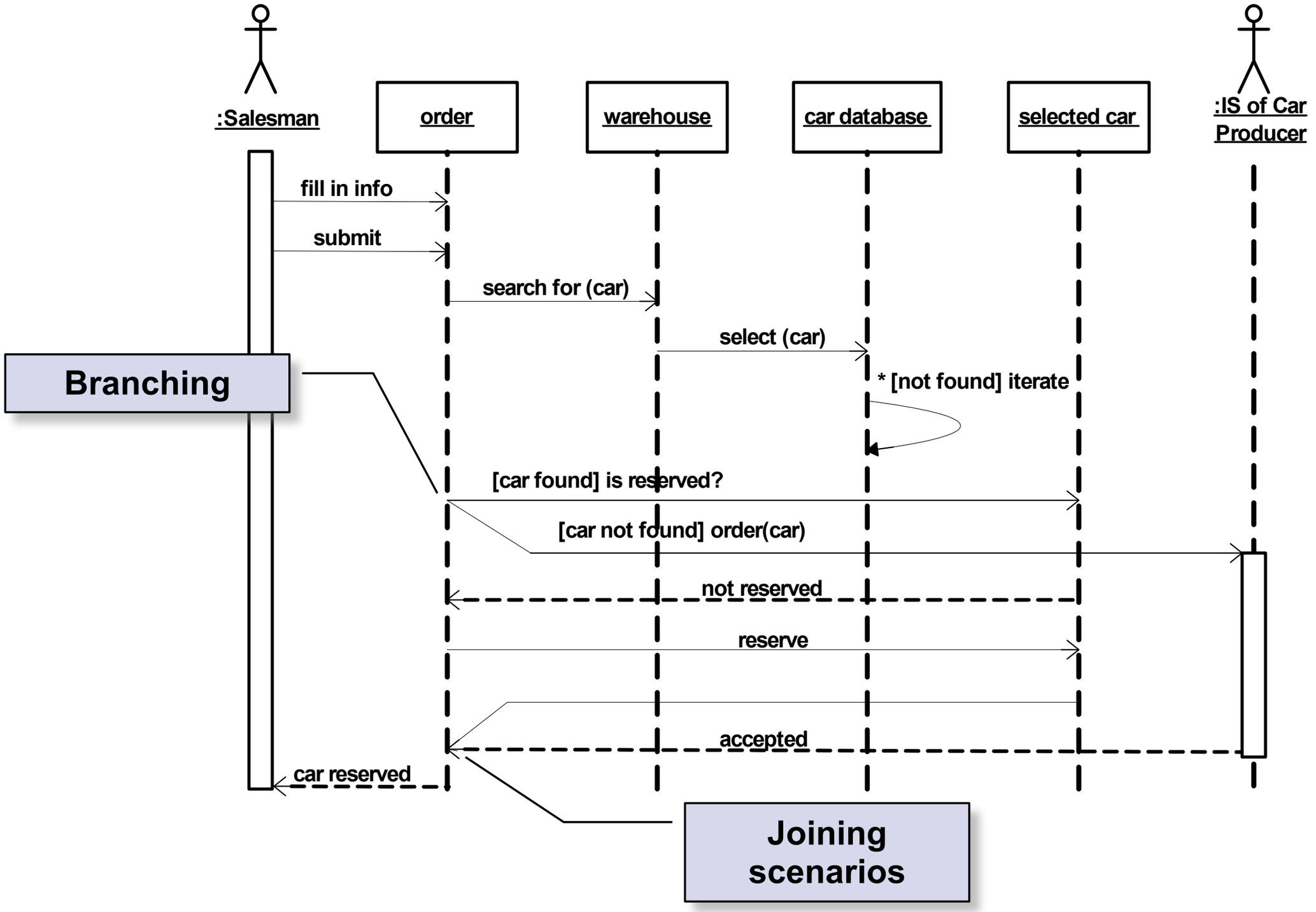
Sequence Diagram: Car Ordering



Alternative Scenario



Merging Scenarios



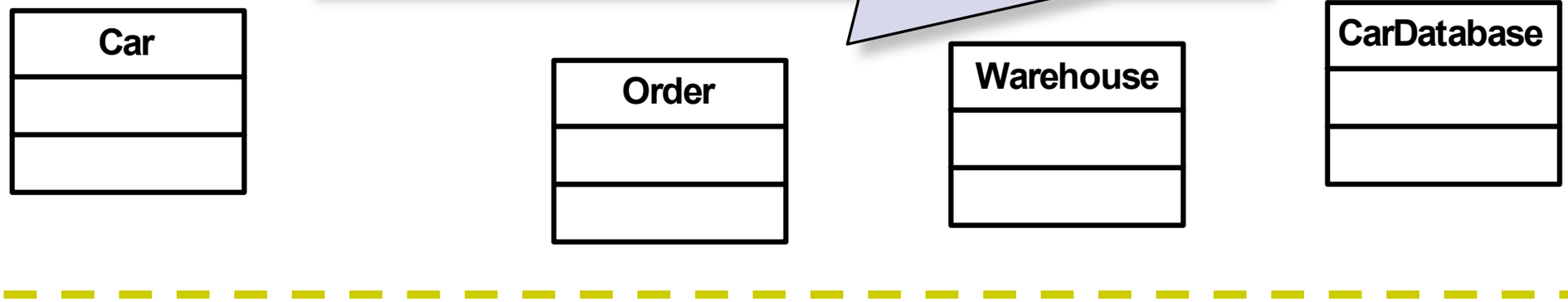
What is a Class?

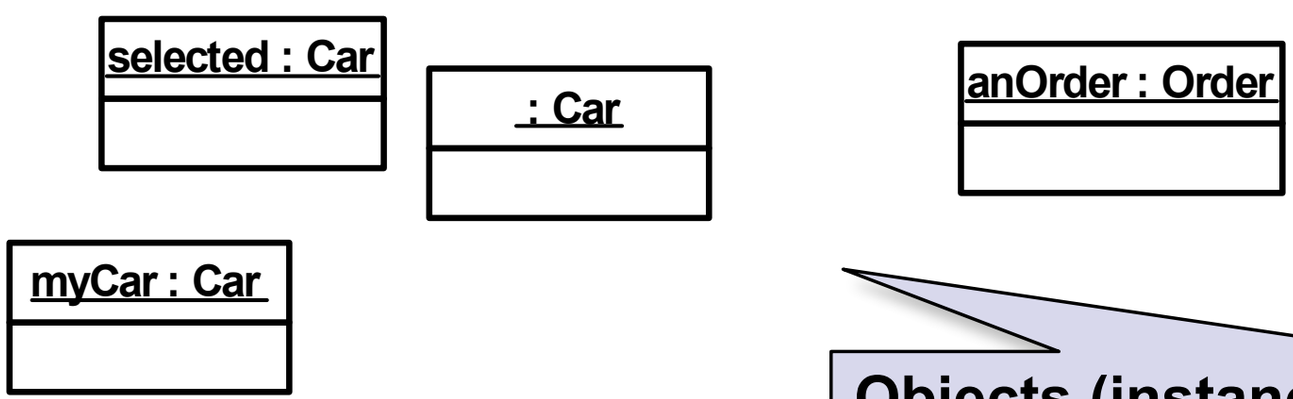
A **class** is the descriptor for a set of objects with common **structure**, **behavior**, **relationships** and **semantics**.

- 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

Classes identified in the domain. All classes are singular nouns starting with **uppercase** letter.

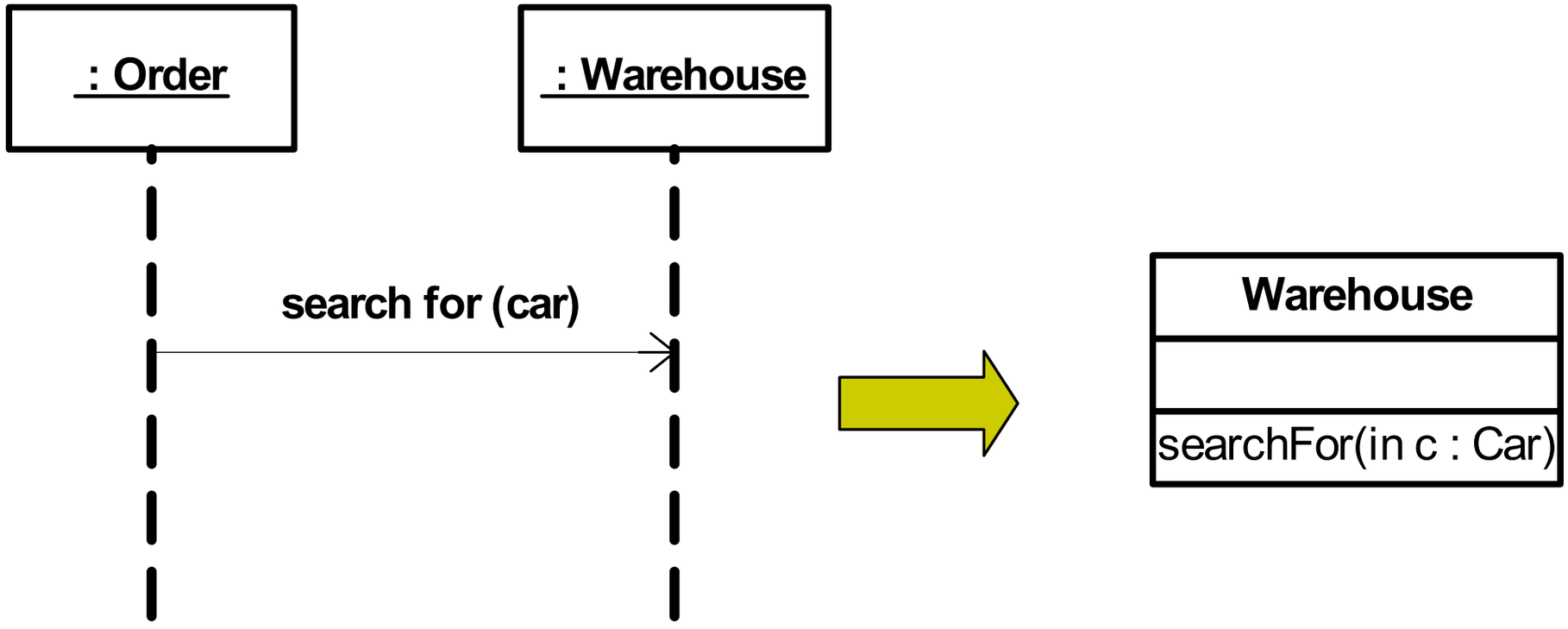




Objects (instances) instantiated from classes. A name of object is underlined. Colon separates the name and the class of the object. Names usually start with lowercase letter.

Operations

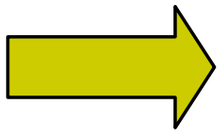
- The **behavior** of a class is represented by its **operations**.
- Operation may be found by examining interaction diagrams

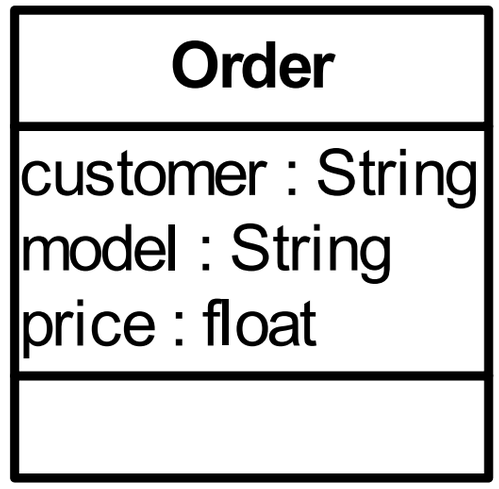


Attributes

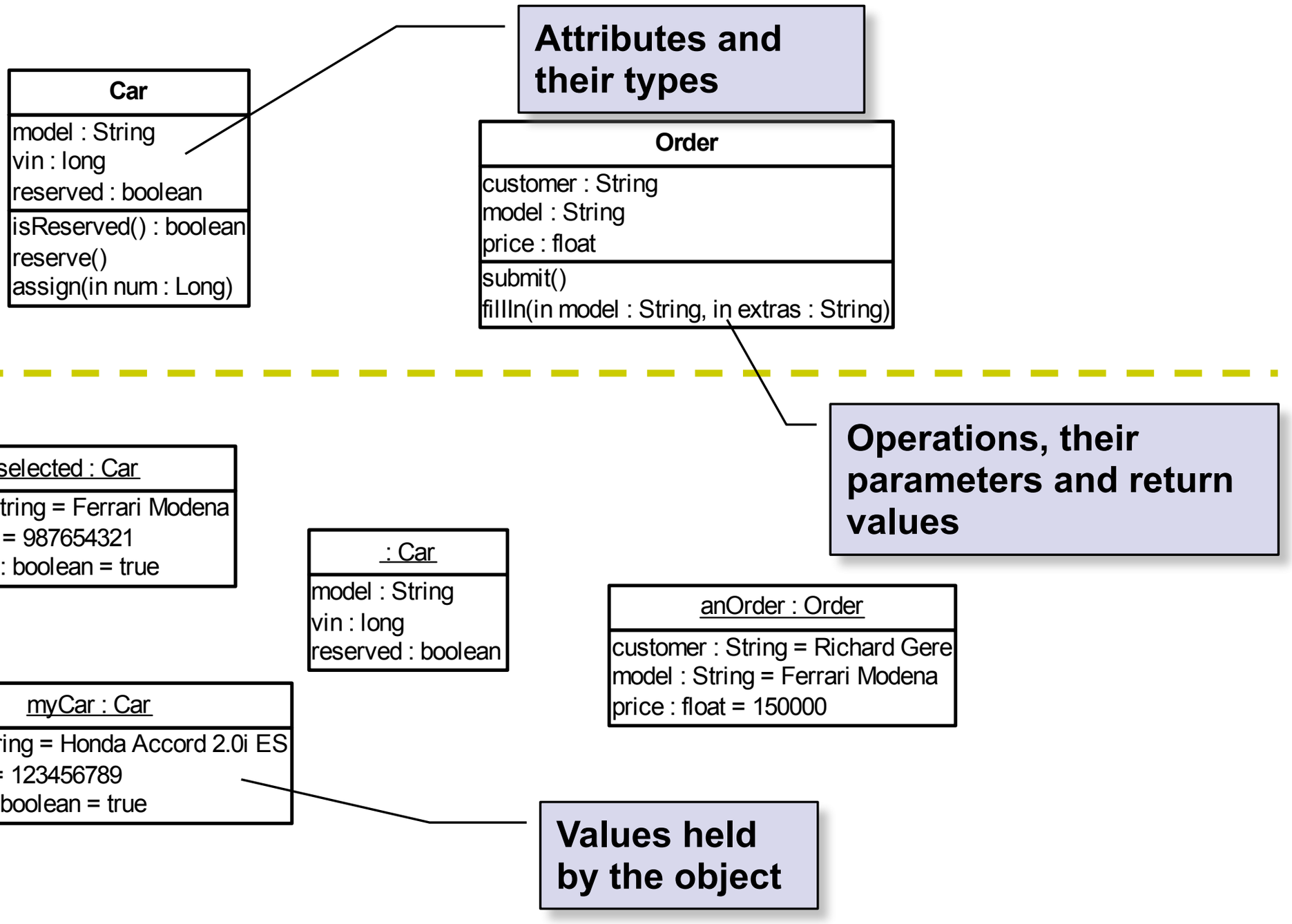
- The **structure** of a class is represented by its **attributes**.
- Attributes may be found by examining class definitions, the problem requirements, and applying domain knowledge.

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

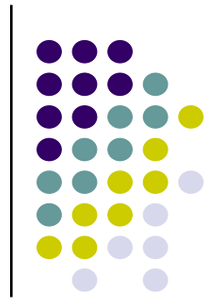




Operations and Attributes

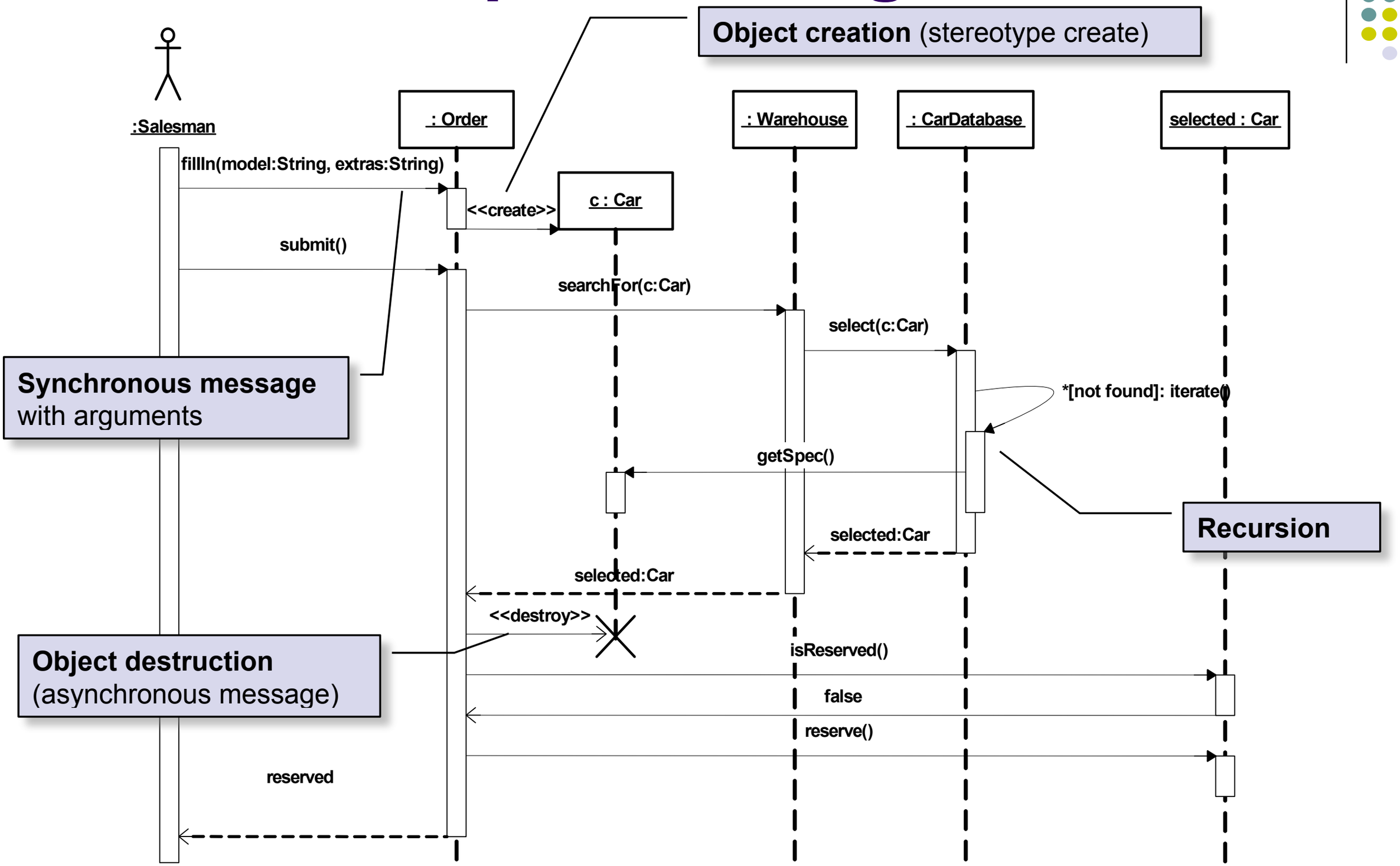


Relationships

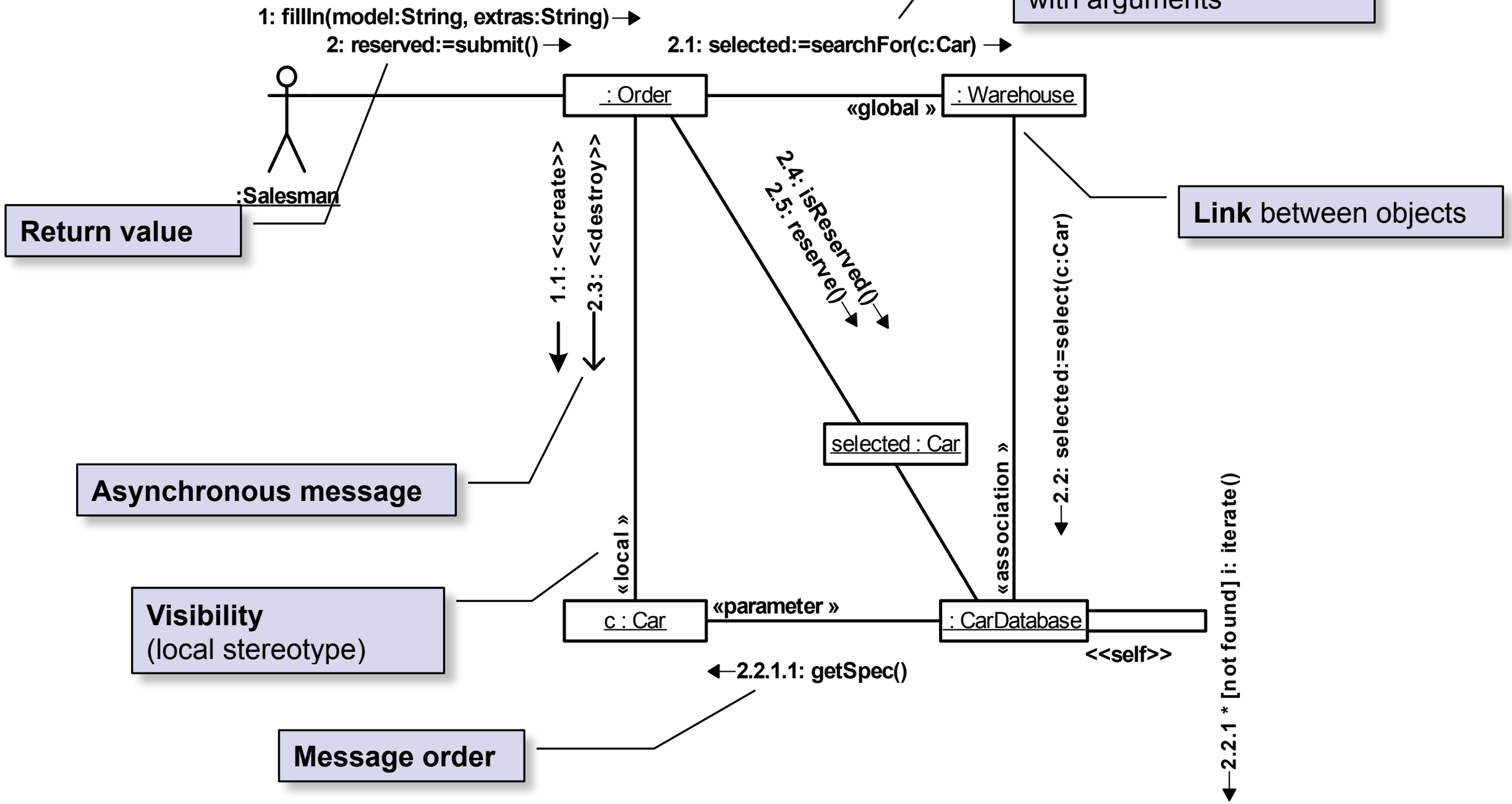


- Relationships between classes specify a **way for communication between objects**.
- Sequence and/or collaboration diagrams are examined to determine what links between objects need to exist to accomplish required behavior. **Objects can send messages to each other only if the link between them is 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



Collaboration Diagram



Synchronous message with arguments

Return value

Link between objects

Asynchronous message

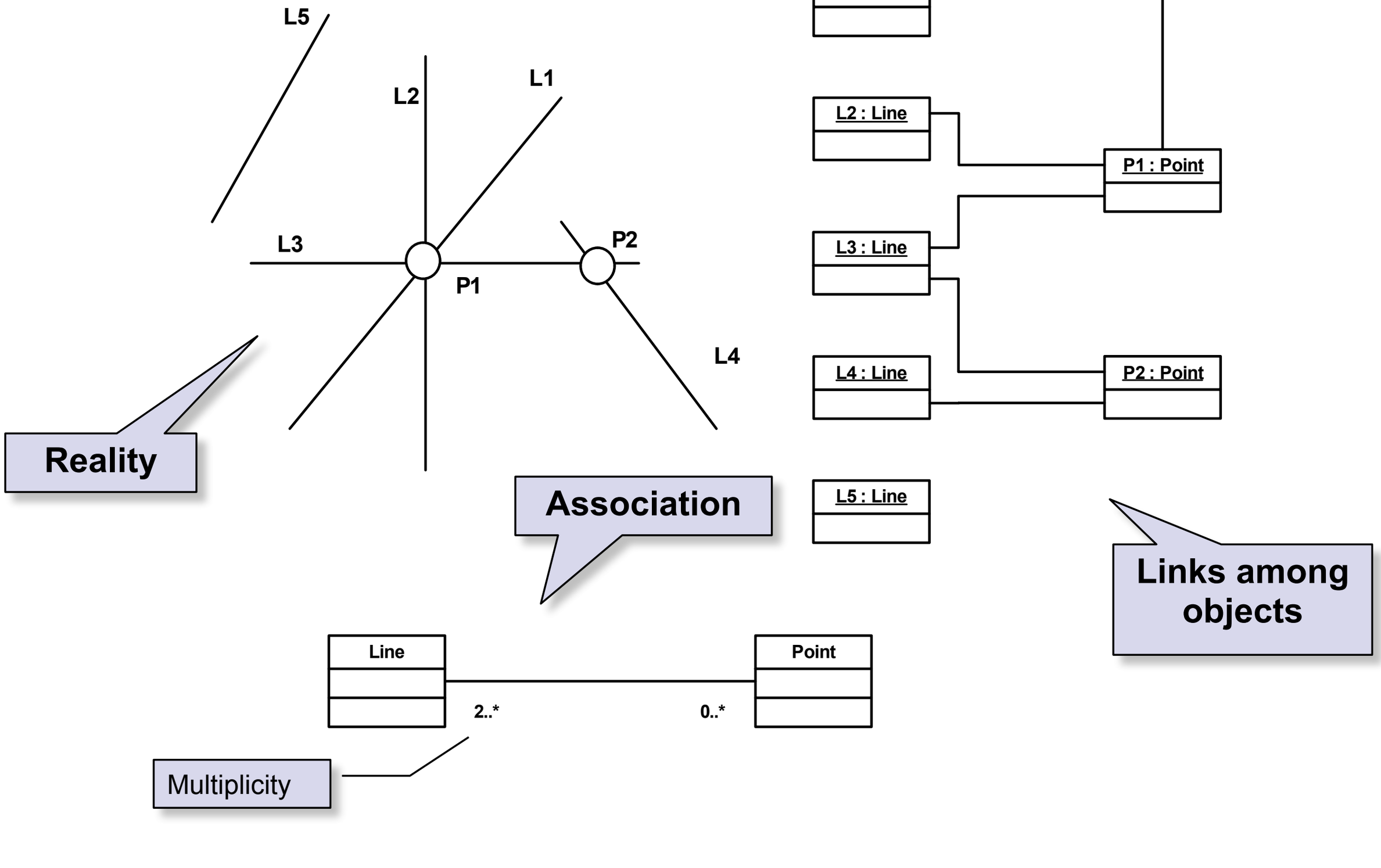
Visibility (local stereotype)

Message order

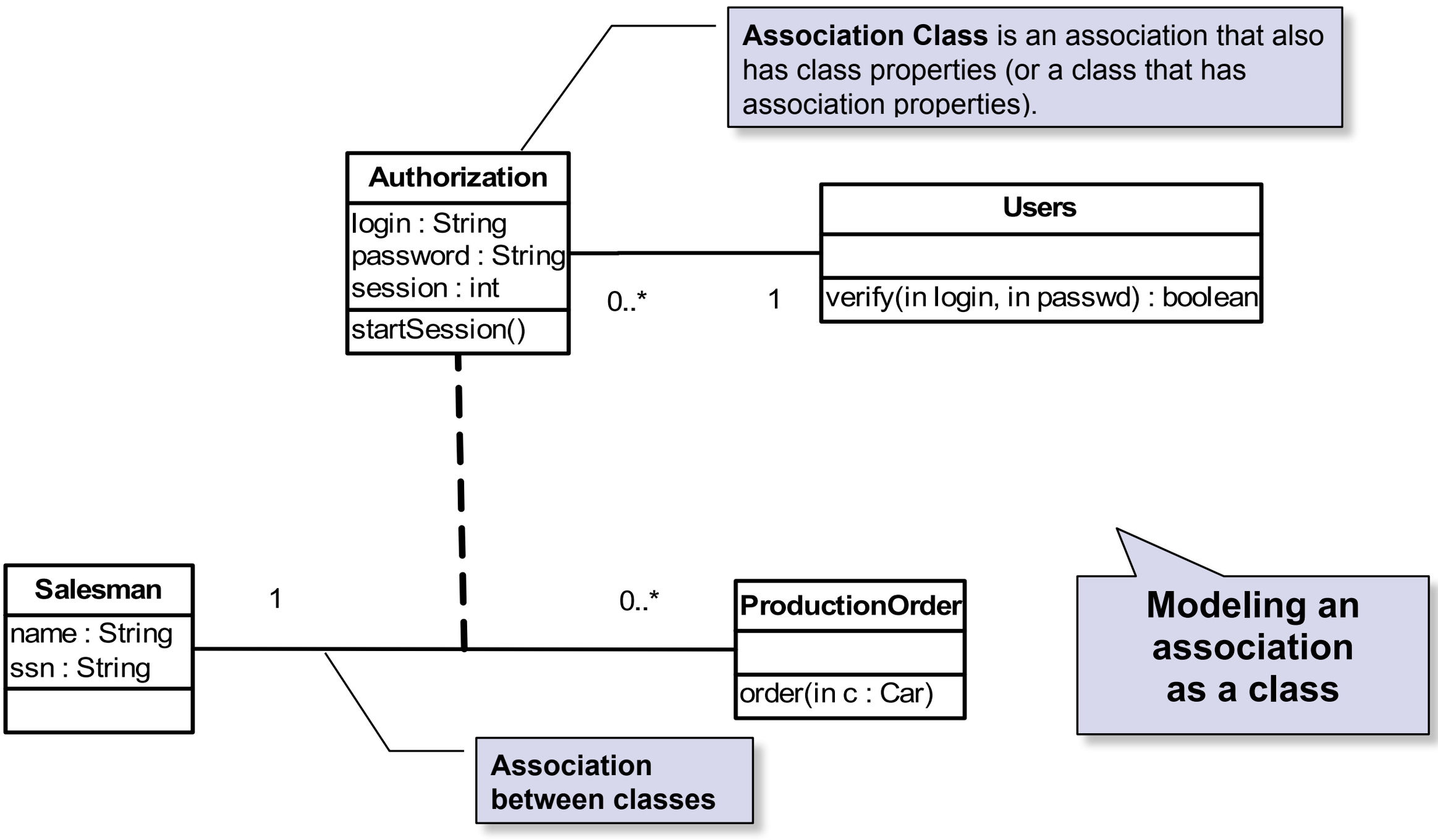
Types of Relationships

- **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.

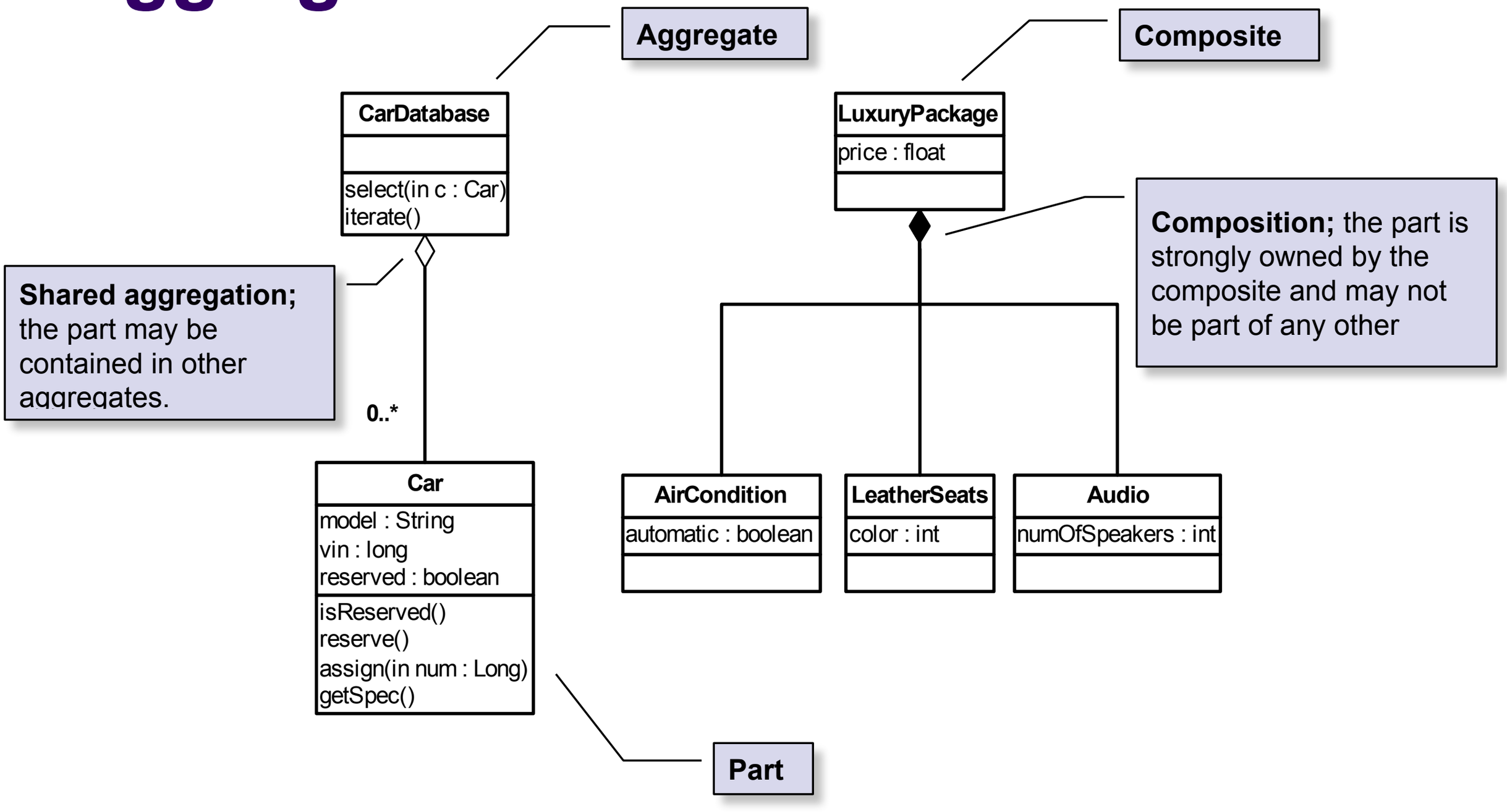
Links and Associations



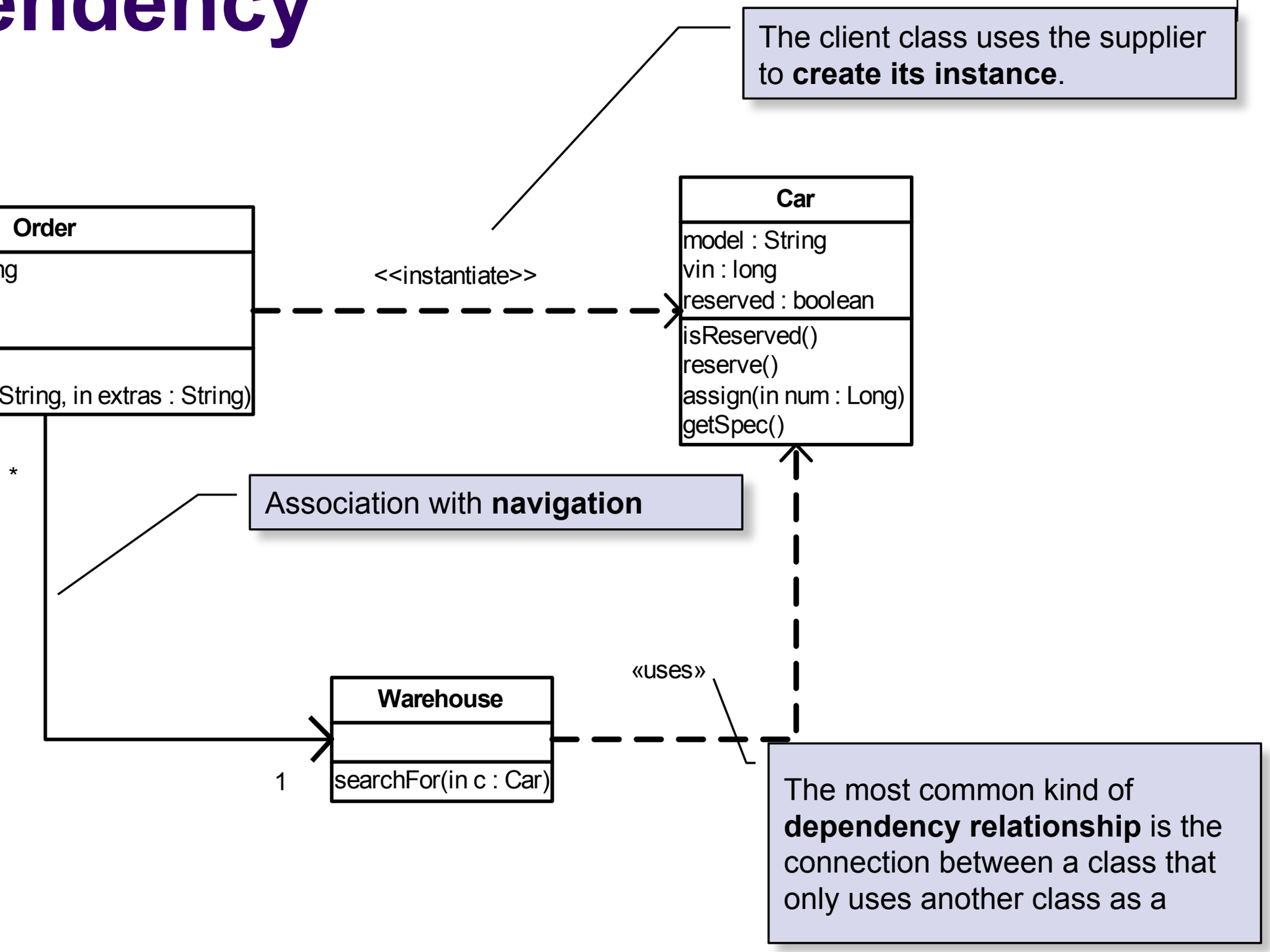
Association Classes



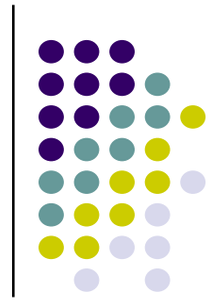
Aggregations



Dependency

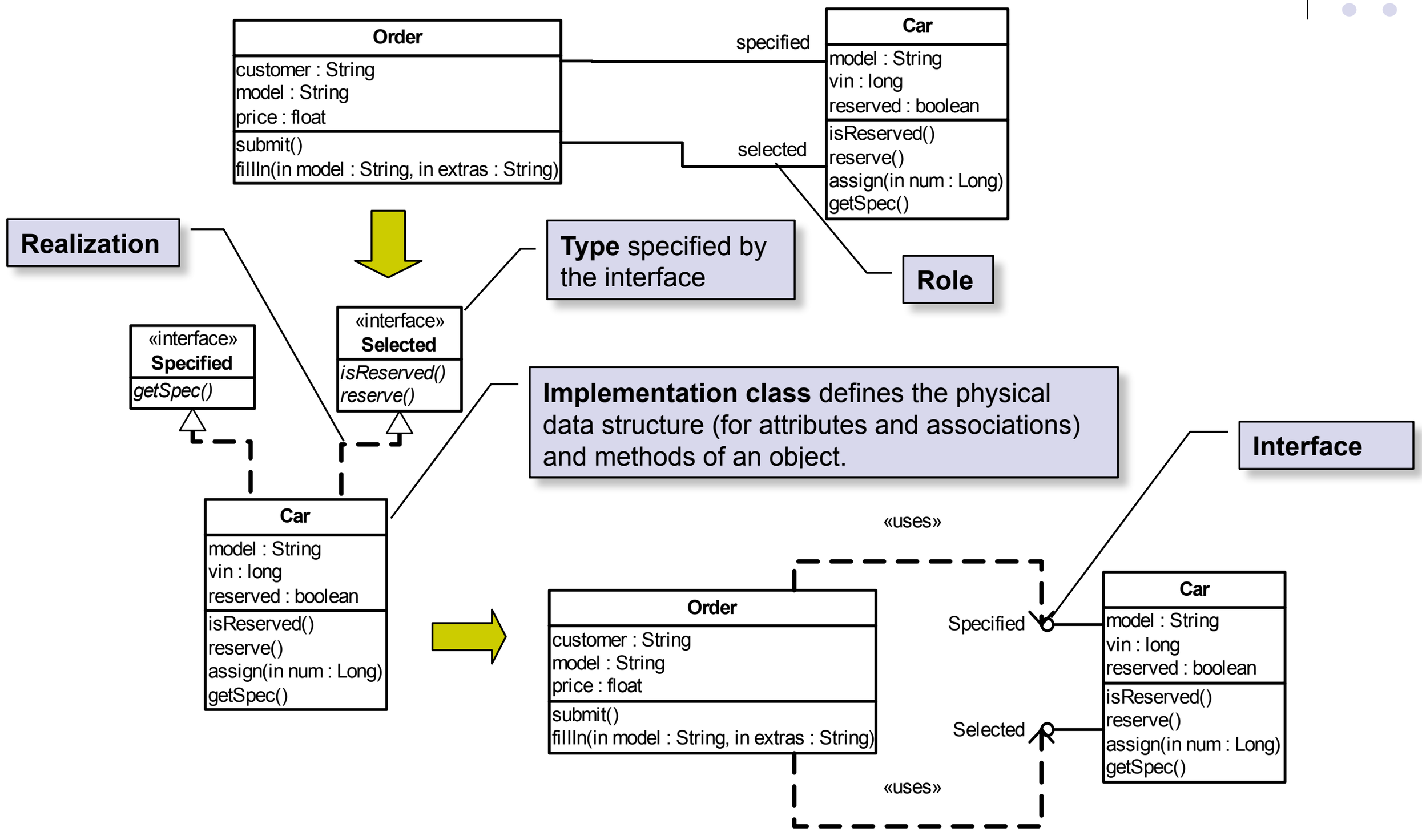

The client class uses the supplier to **create its instance**.
Order
customer : String
model : String
price : float
submit()
fillIn(in model : String, in extras : String)
<<instantiate>>
Car
model : String
vin : long
reserved : boolean
isReserved()
reserve()
assign(in num : Long)
getSpec()
*
Association with **navigation**
«uses»
Warehouse
searchFor(in c : Car)
1
The most common kind of **dependency relationship** is the connection between a class that only uses another class as a

Roles, Types and Interfaces

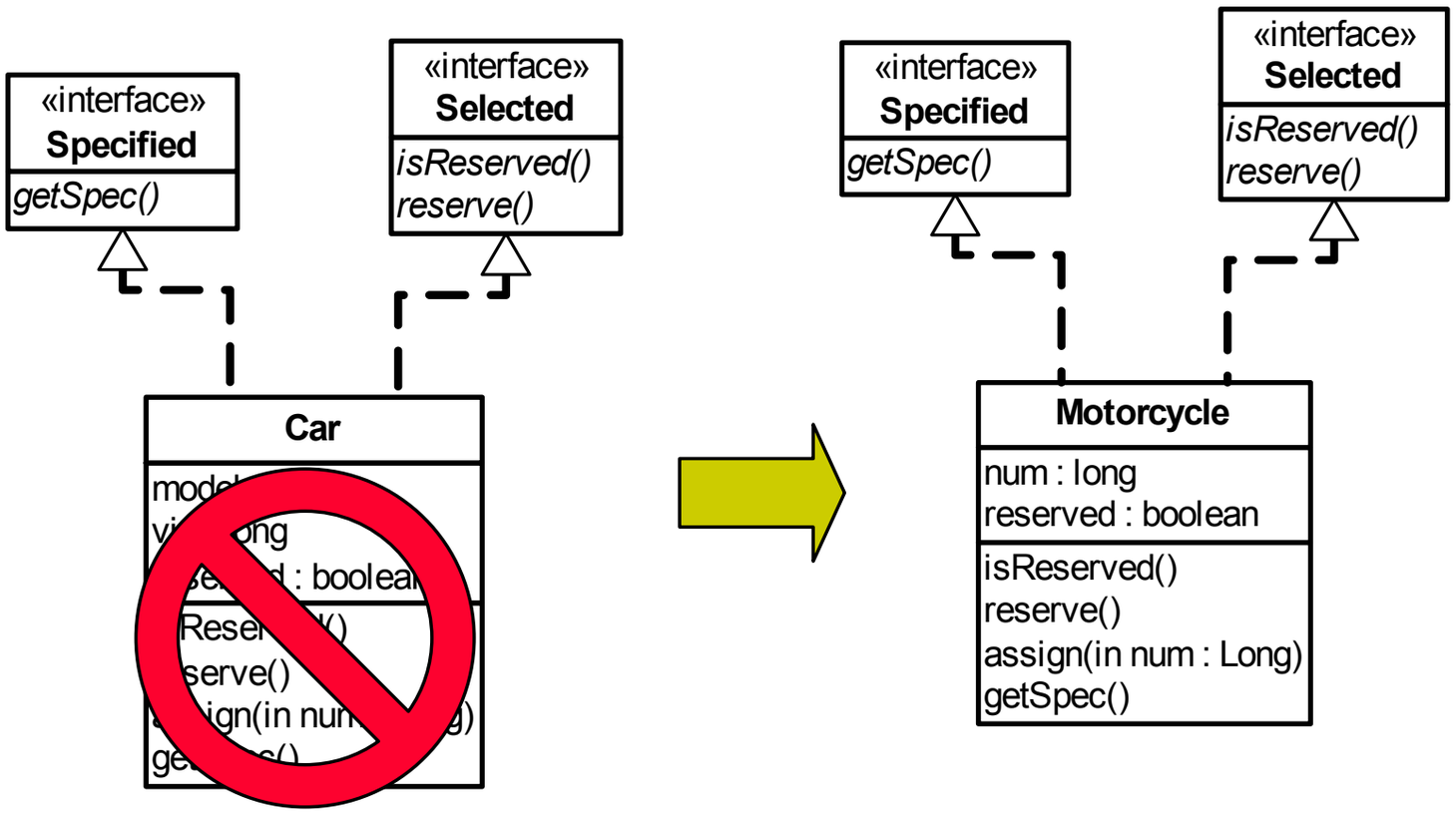


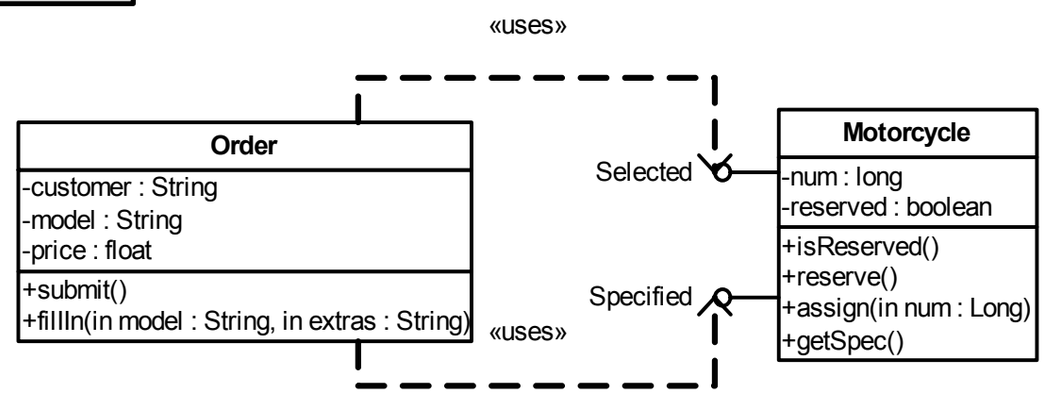
- Role is the named specific behavior of an object participating in a particular context (the face it presents to the world at a given moment).
- Type specifies a domain of objects together with the operations applicable to the objects (without defining the physical implementation of those objects).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

Type and Implementation Class



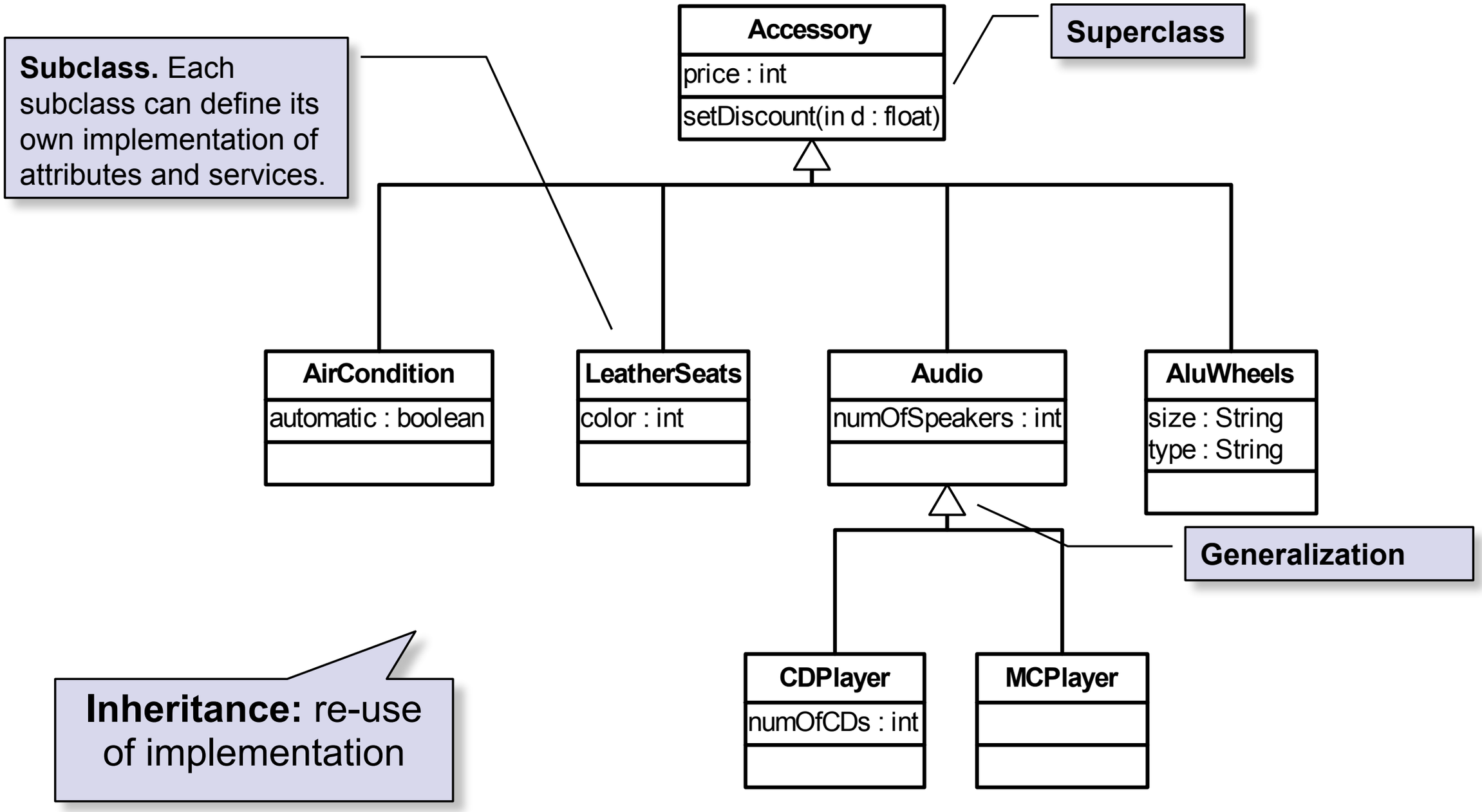
Implementation Independence





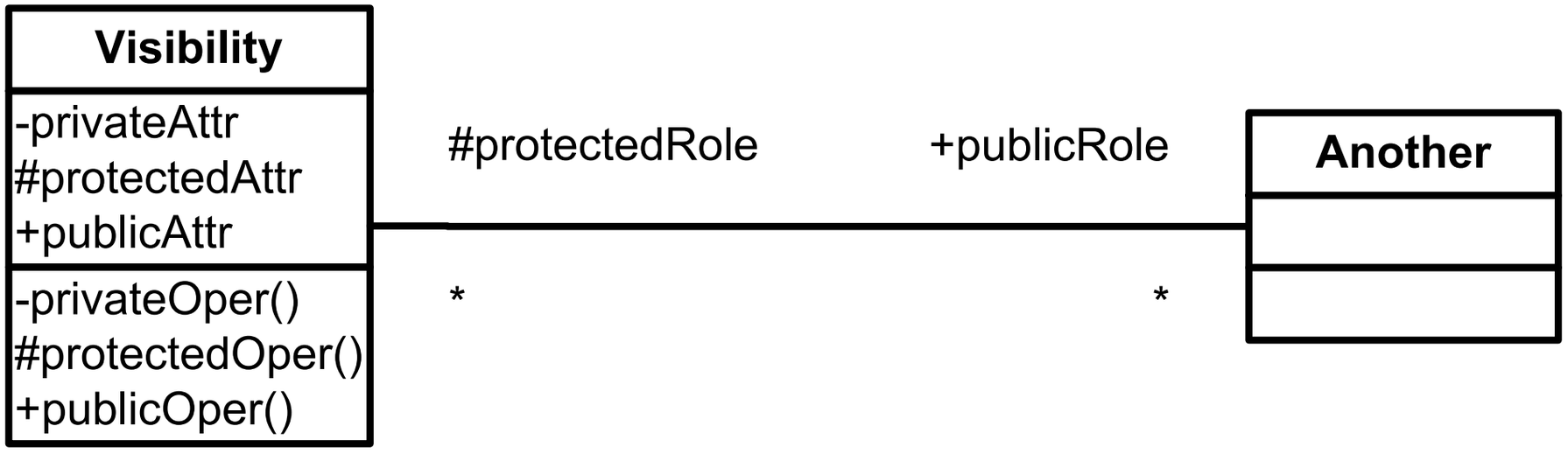
Interface for *Order* remains the same!

Generalization/Specialization

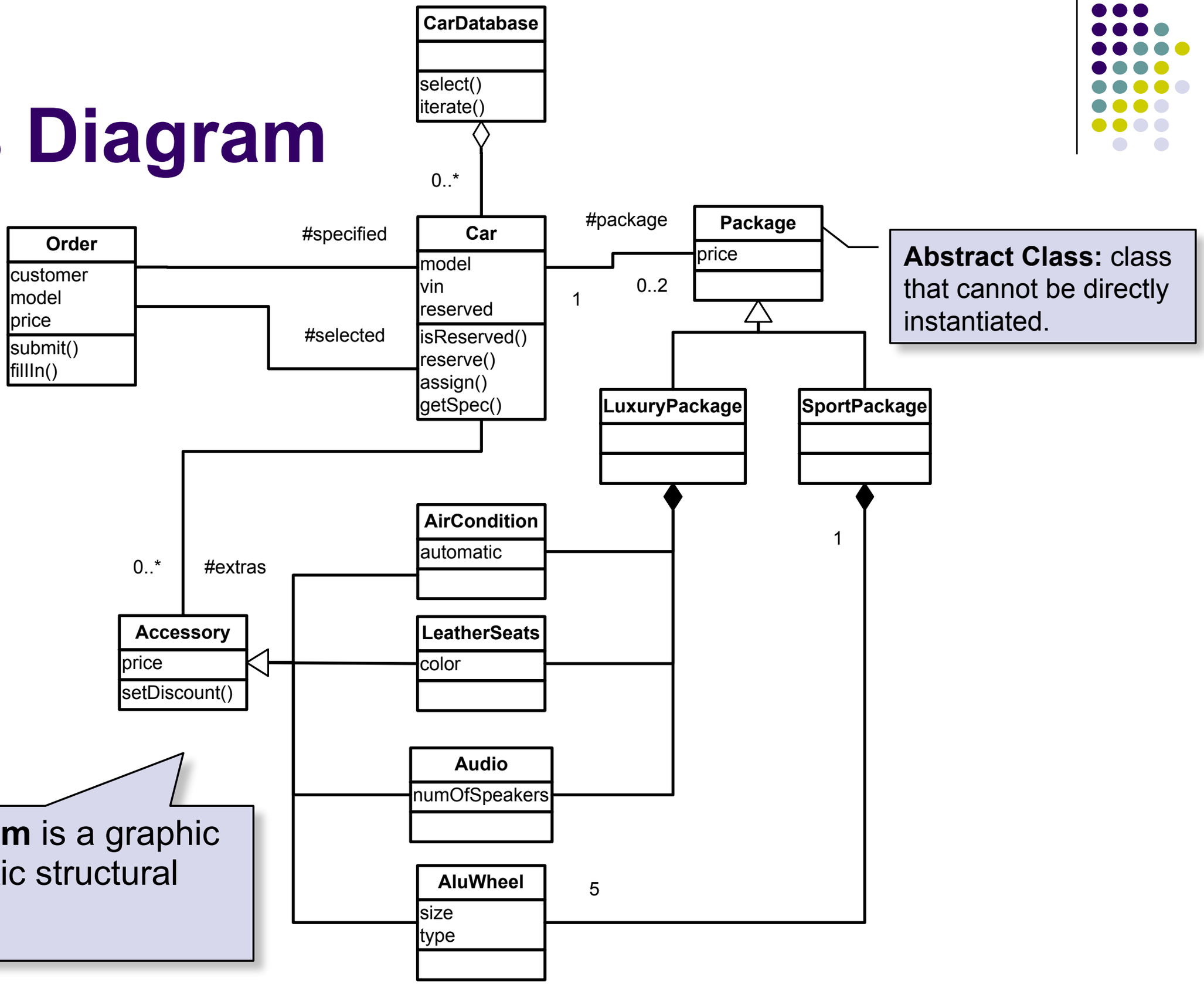


Visibility

- **Public** declaration is accessible to all clients.
- **Protected** declaration is accessible only to the class itself and its subclasses.
- **Private** declaration is accessible only to the class itself.



Class Diagram



A **class diagram** is a graphic view of the static structural model.

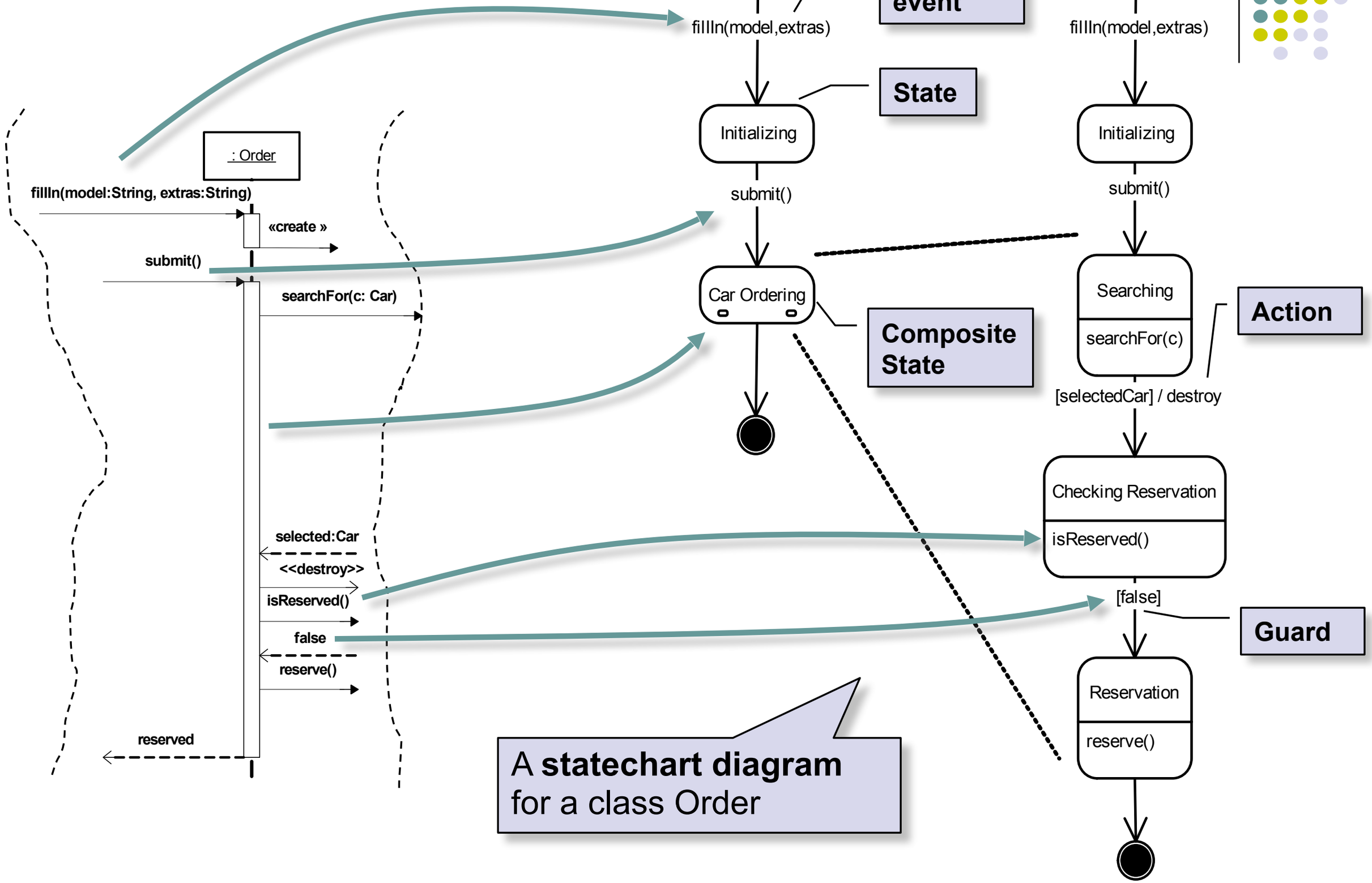
Exercises

- Identify the objects and their classes based on sequence diagram of lending process. Specify what **operations and attributes** these classes should implement.
- Apply **generalization/specialization relationship** to classes Videotape and DVD.
- Construct **class diagram** for video lending library.

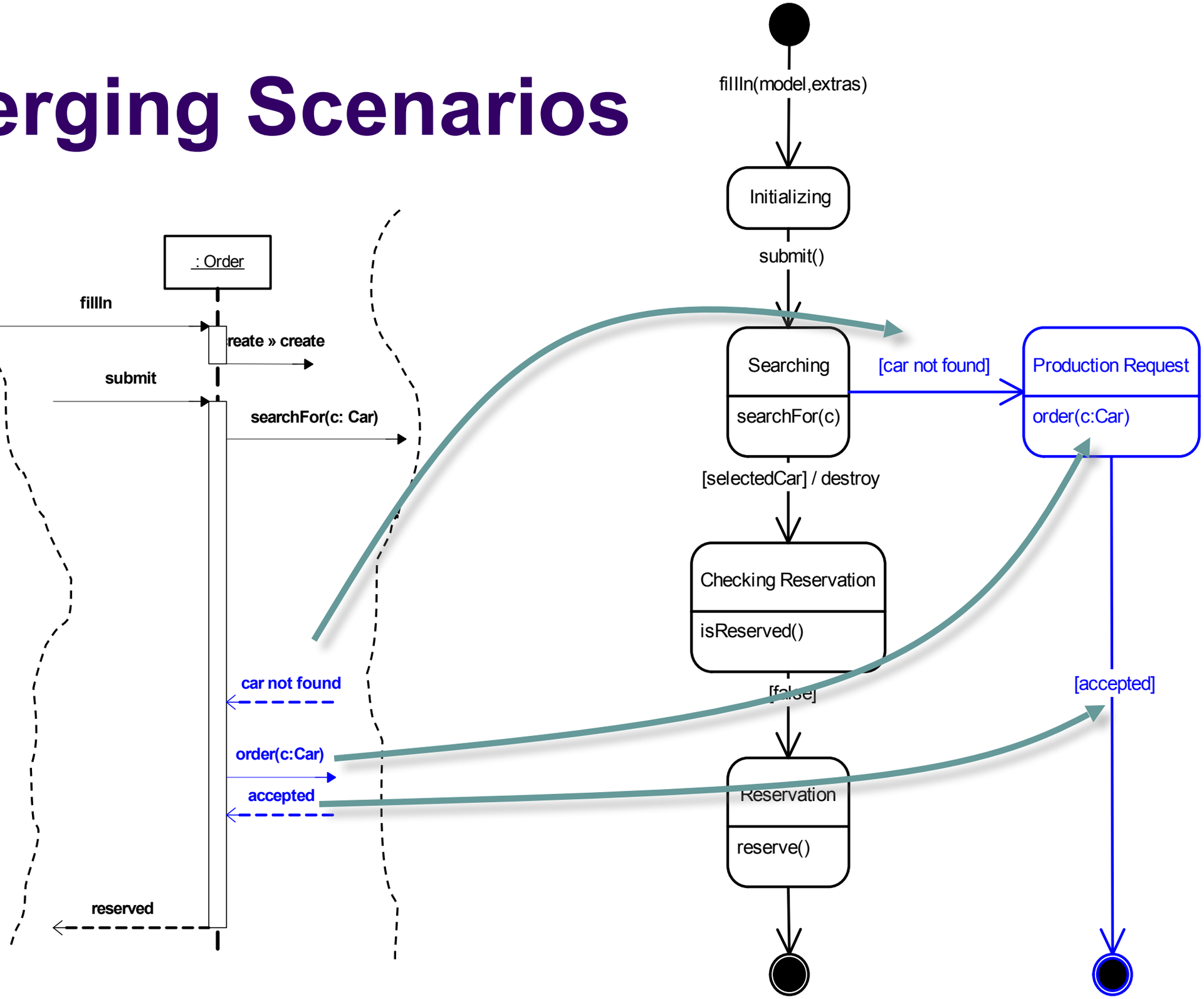
The Dynamic Behavior of an Object

- 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

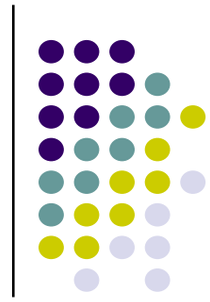
Statechart Diagram



Merging Scenarios

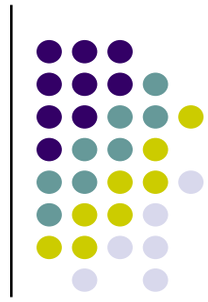


Exercises



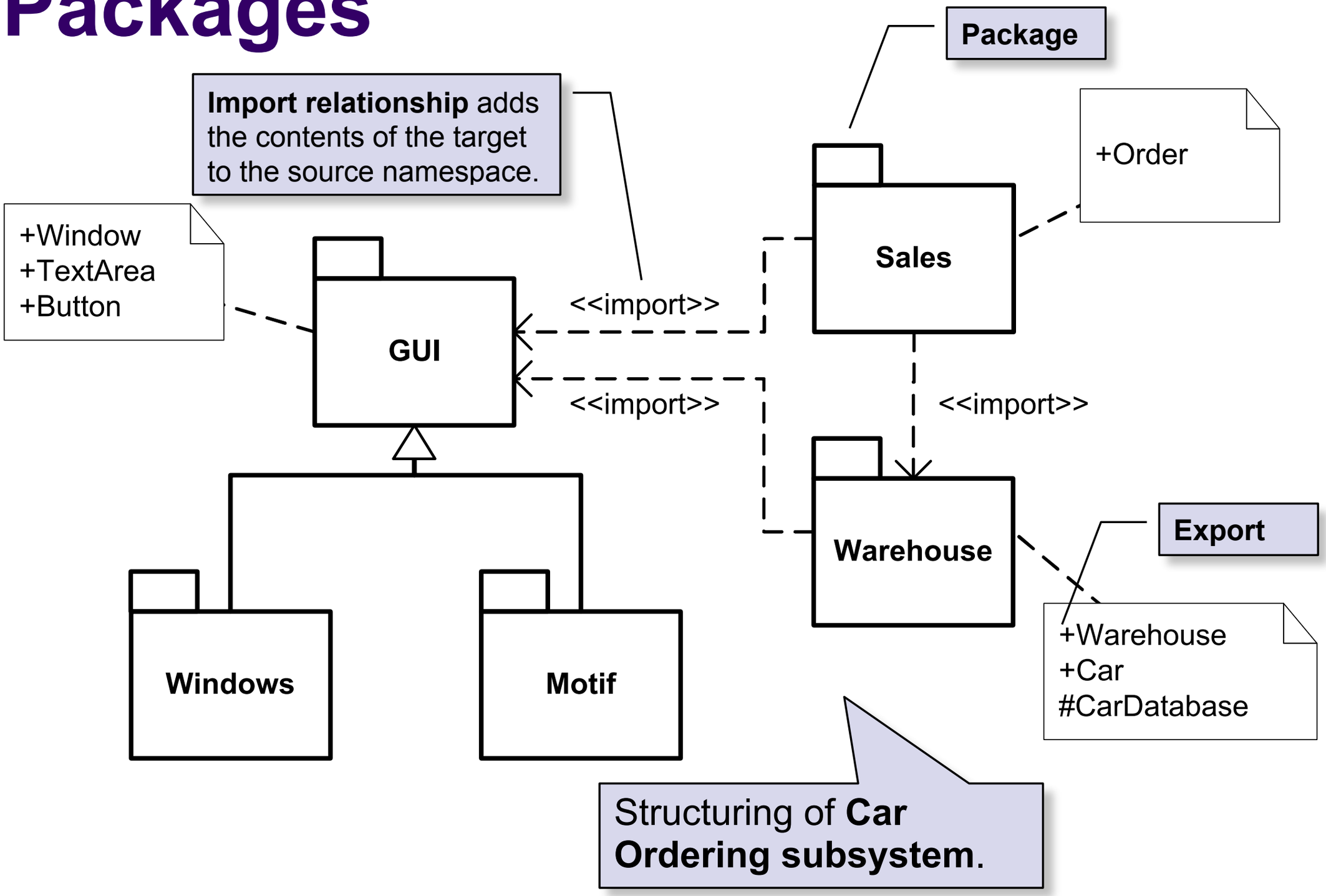
- Select two classes from video lending library class diagram and construct the **state diagram** for both of them.

Model Management Overview



- 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**.

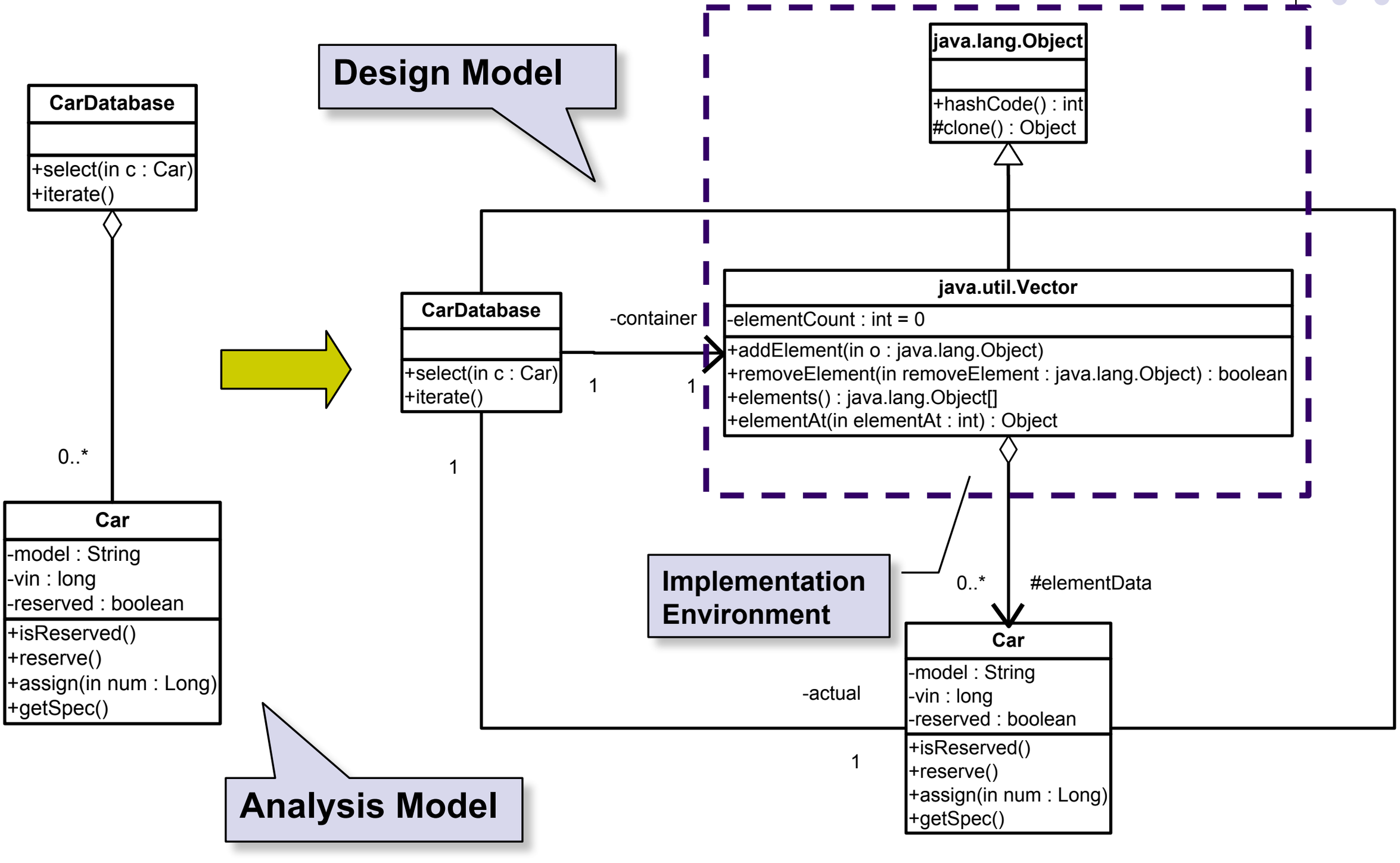
- In analysis **domain objects** are the primary focus.
- In design the **other layers** are added and refined
  - User Interface
  - Distribution
  - Persistence Mechanism

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

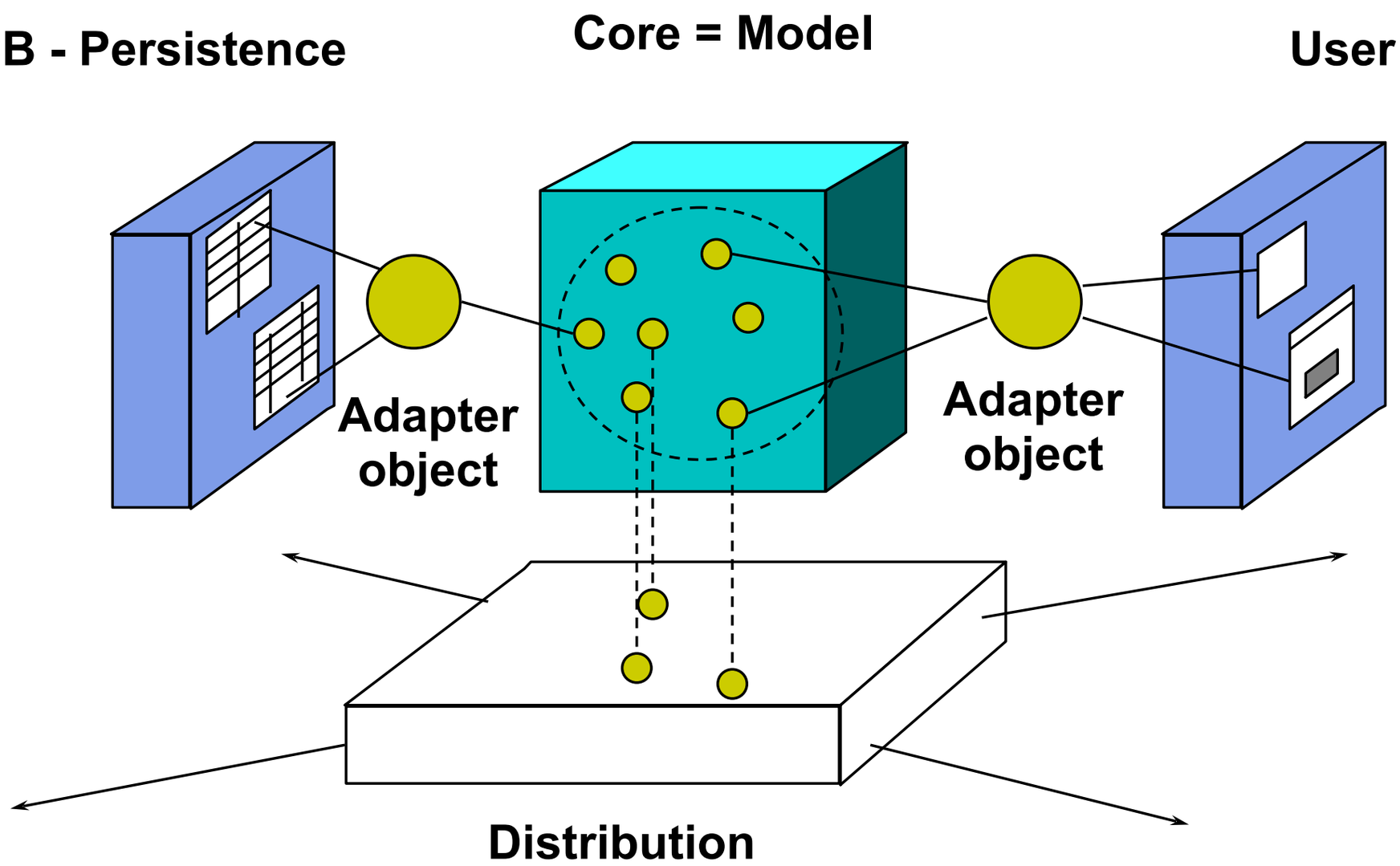
- Definition of the **system architecture**
- Identifying **design patterns** and **frameworks**
- Software **components** definition and re-use

Mapping into Software Components



System Architecture

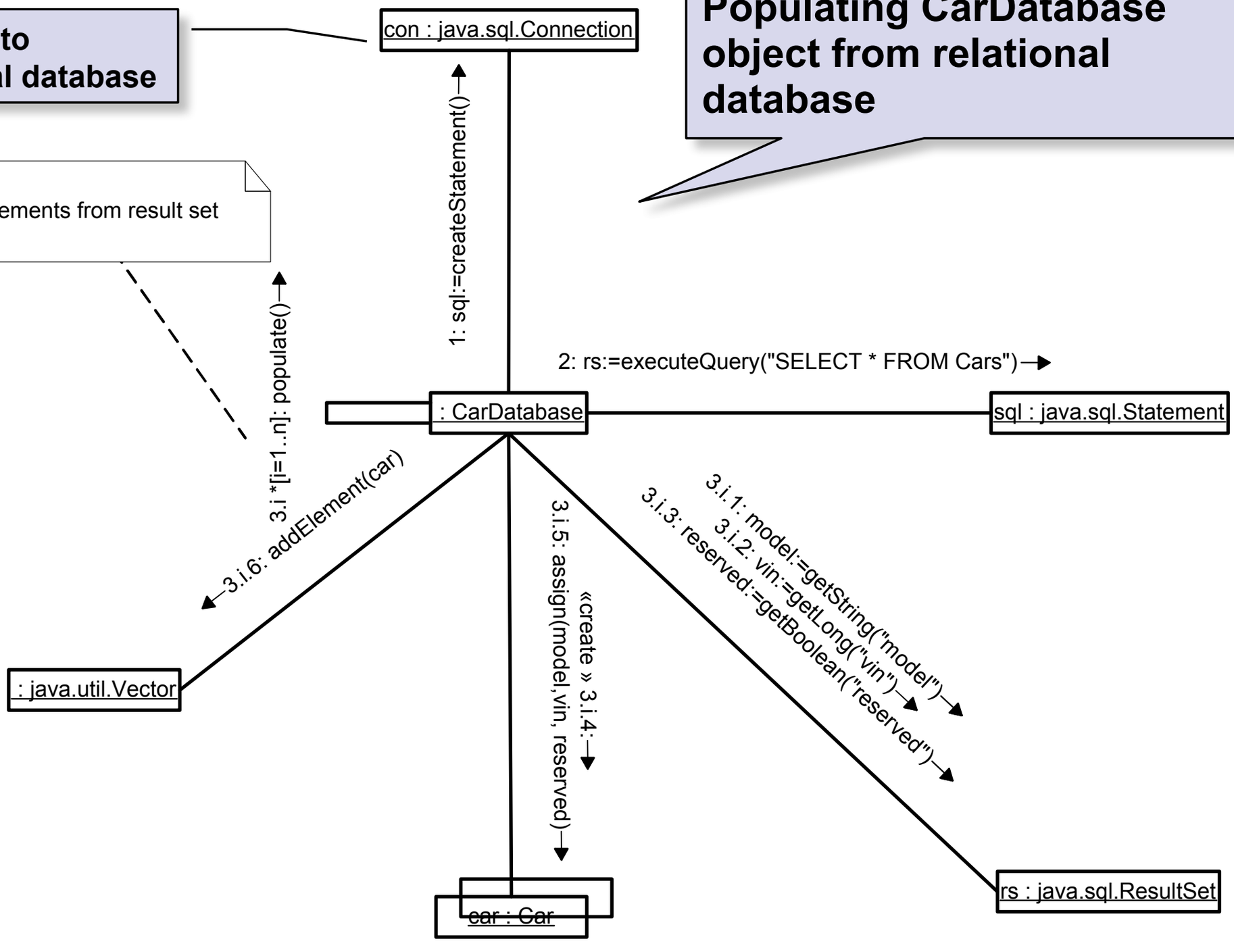
The **organizational structure** and associated behavior of a system.



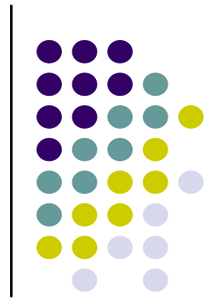
Interface to Database

Adapter to relational database

Populating CarDatabase object from relational database

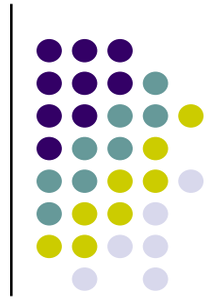


Design Patterns



- 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

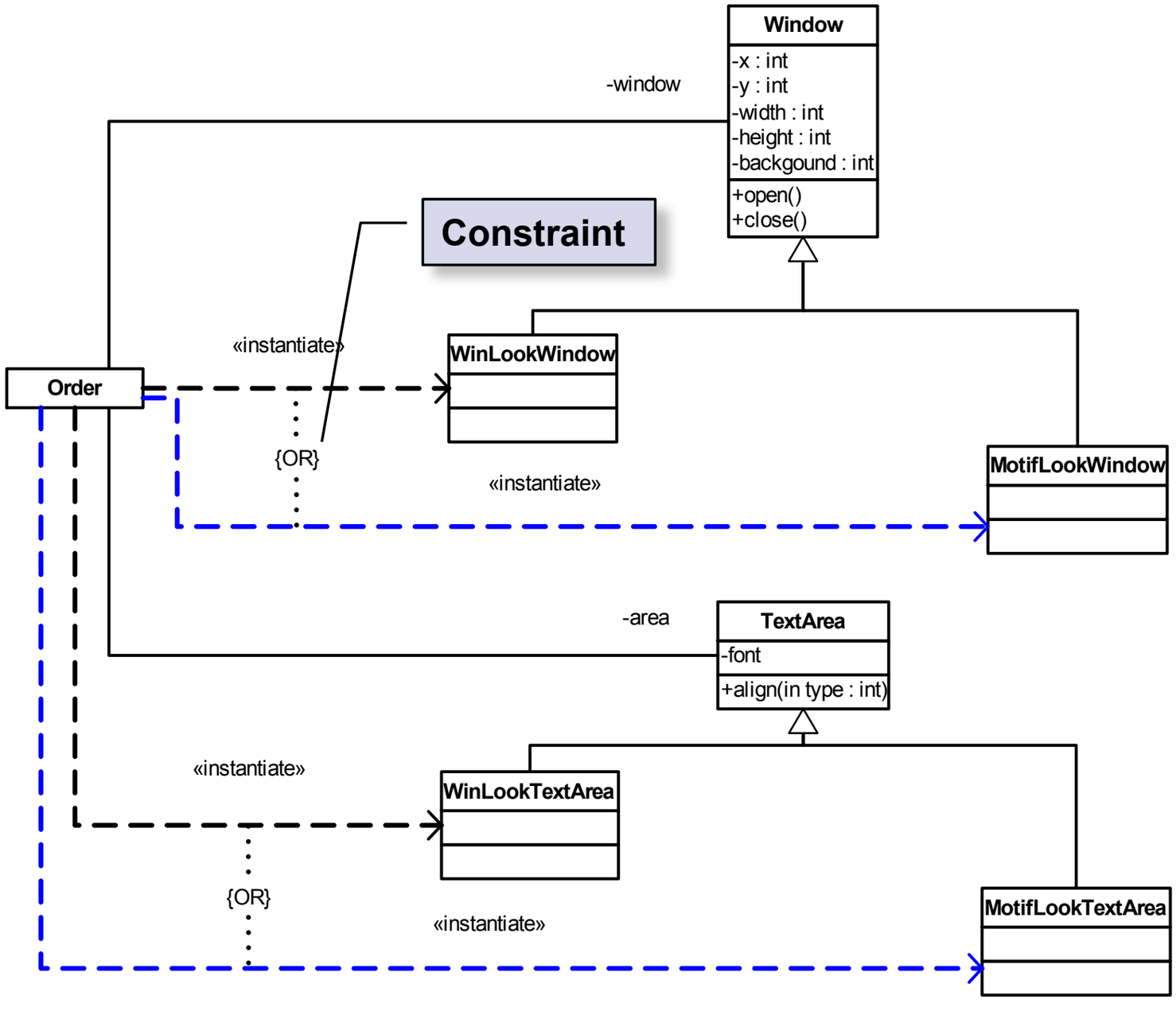


- **Creational patterns** defer some part of object creation to a subclass or another object.
- **Structural patterns** composes classes or objects.
- **Behavioral patterns** describe algorithms or cooperation of objects.

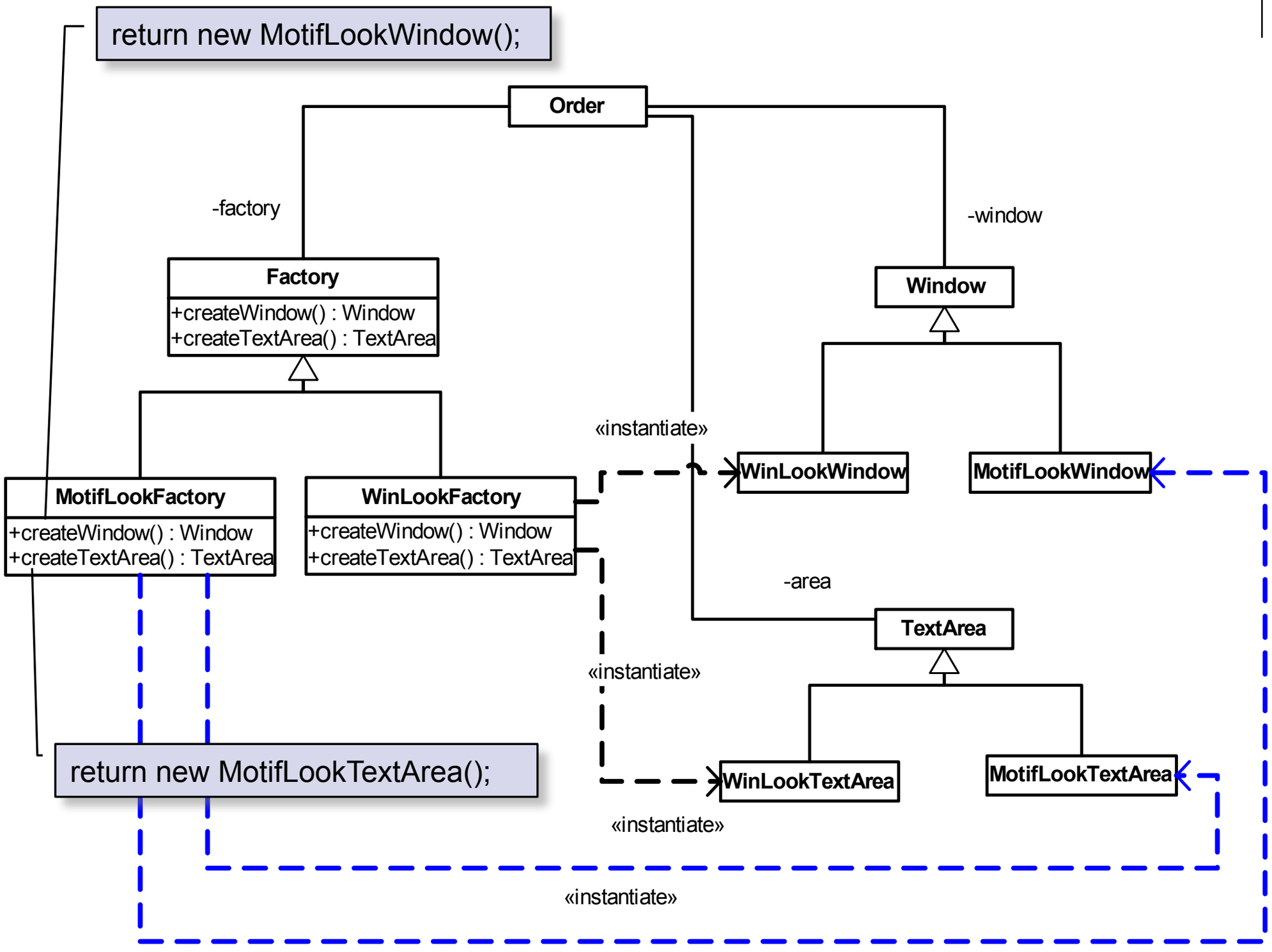
Factory – Creational Pattern

Intent - provide an **interface for creating families of related objects** without specifying their concrete classes.

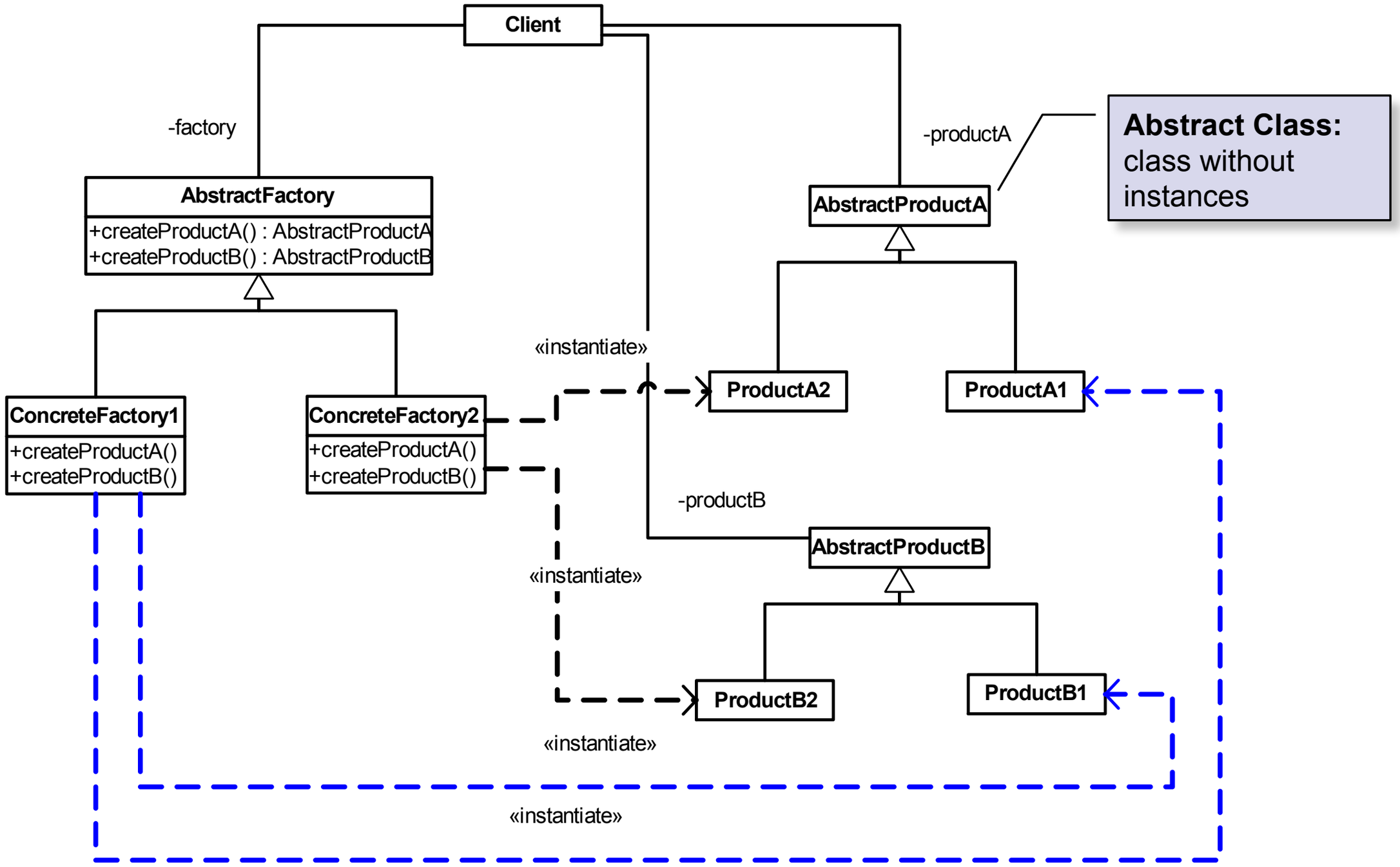
Motivation



Factory - Solution



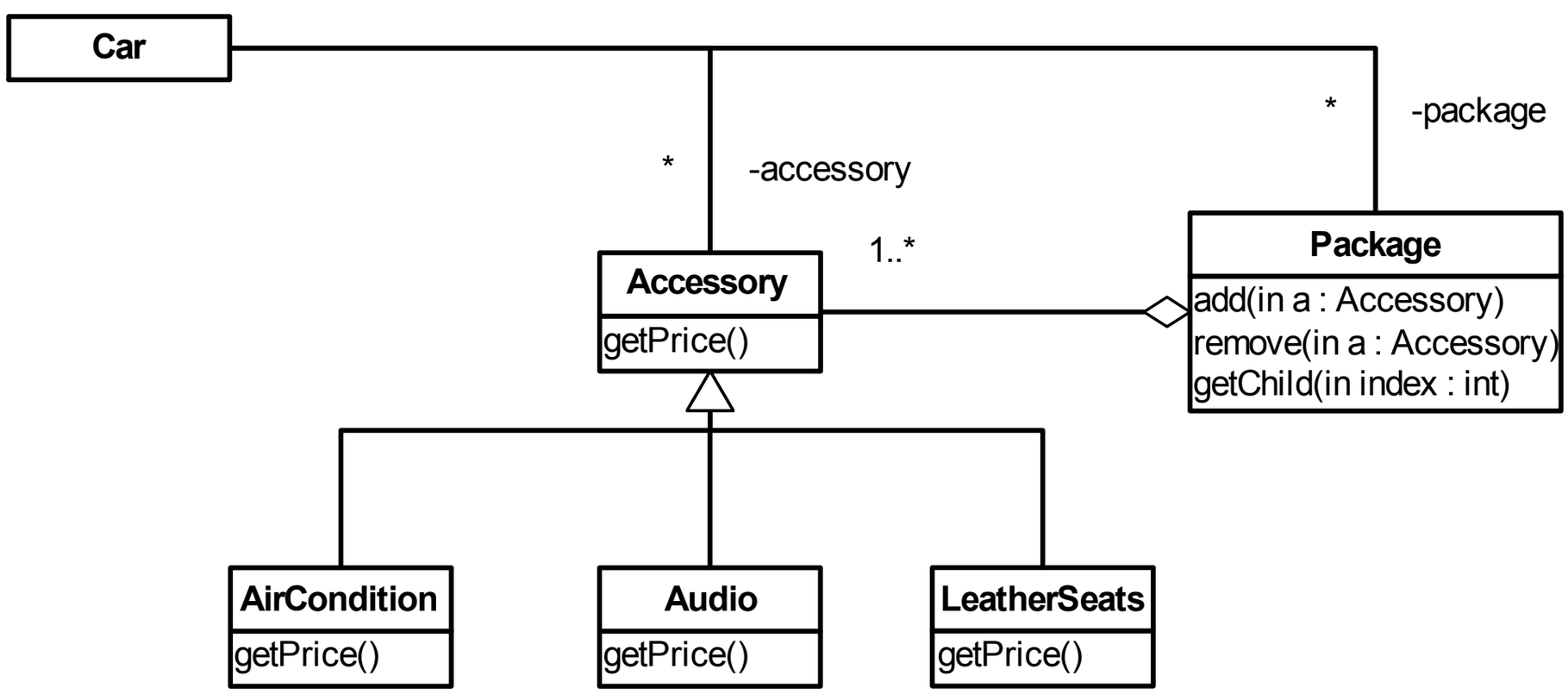
Factory - Abstraction



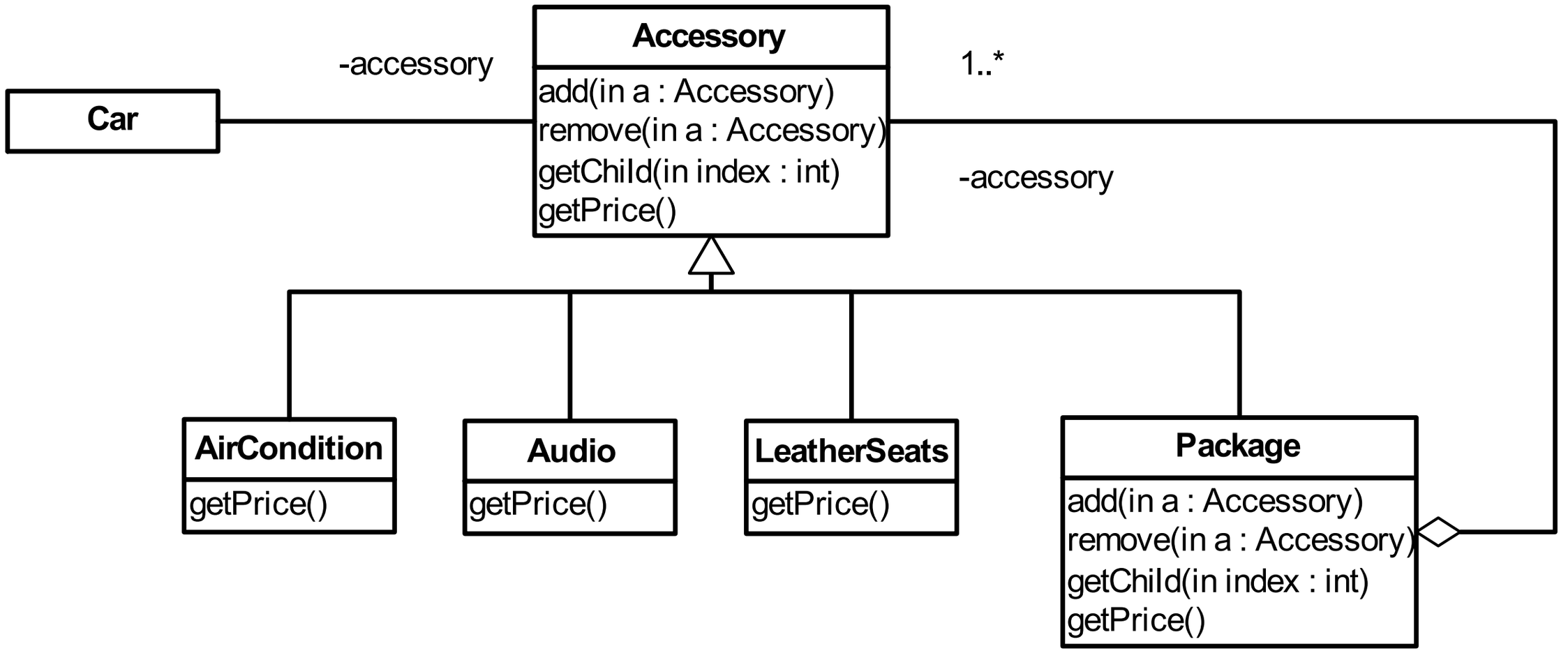
Composite – Structural Pattern

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

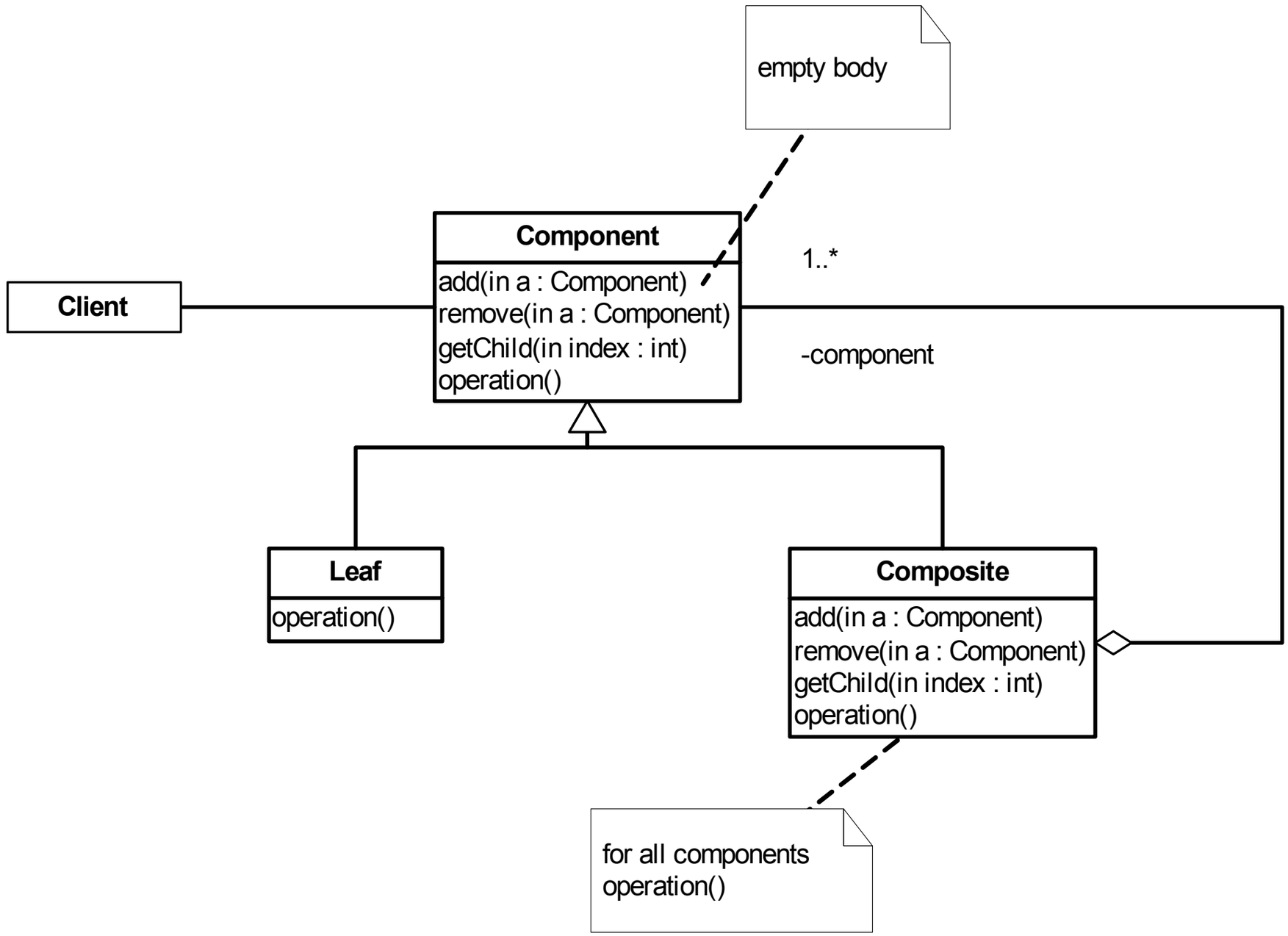
Motivation



Composite - Solution



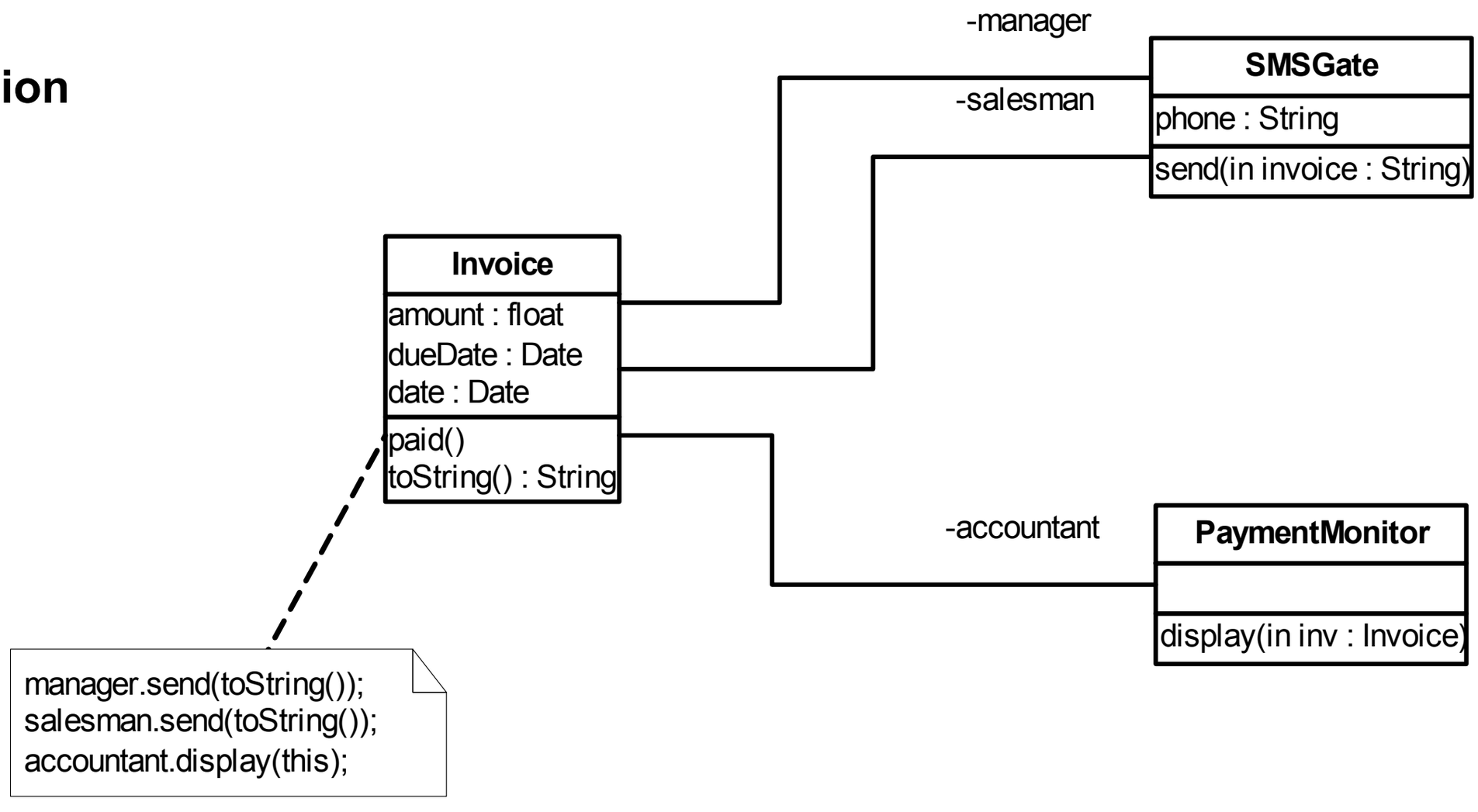
Composite - Abstraction



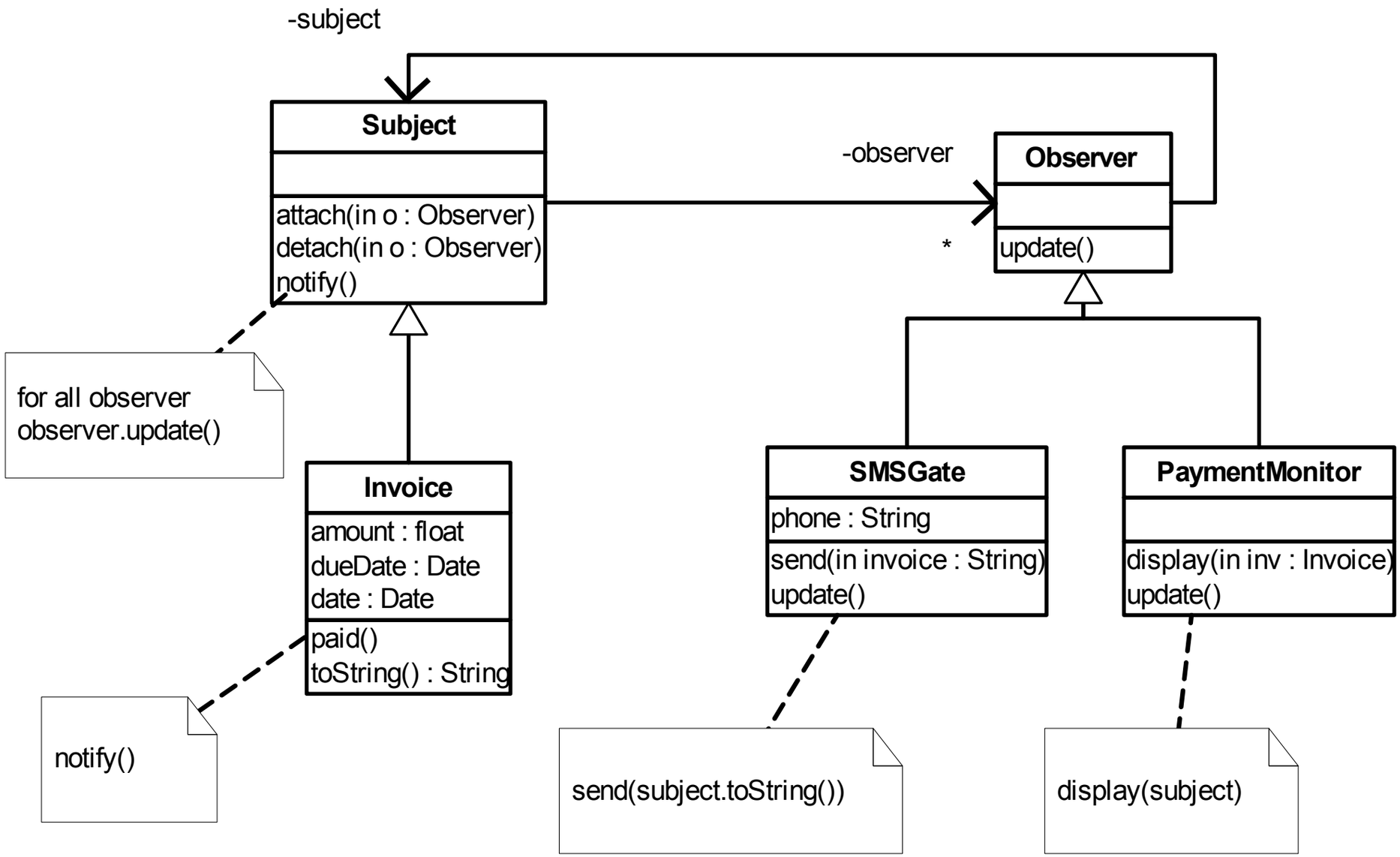
Observer – Behavioral Pattern

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

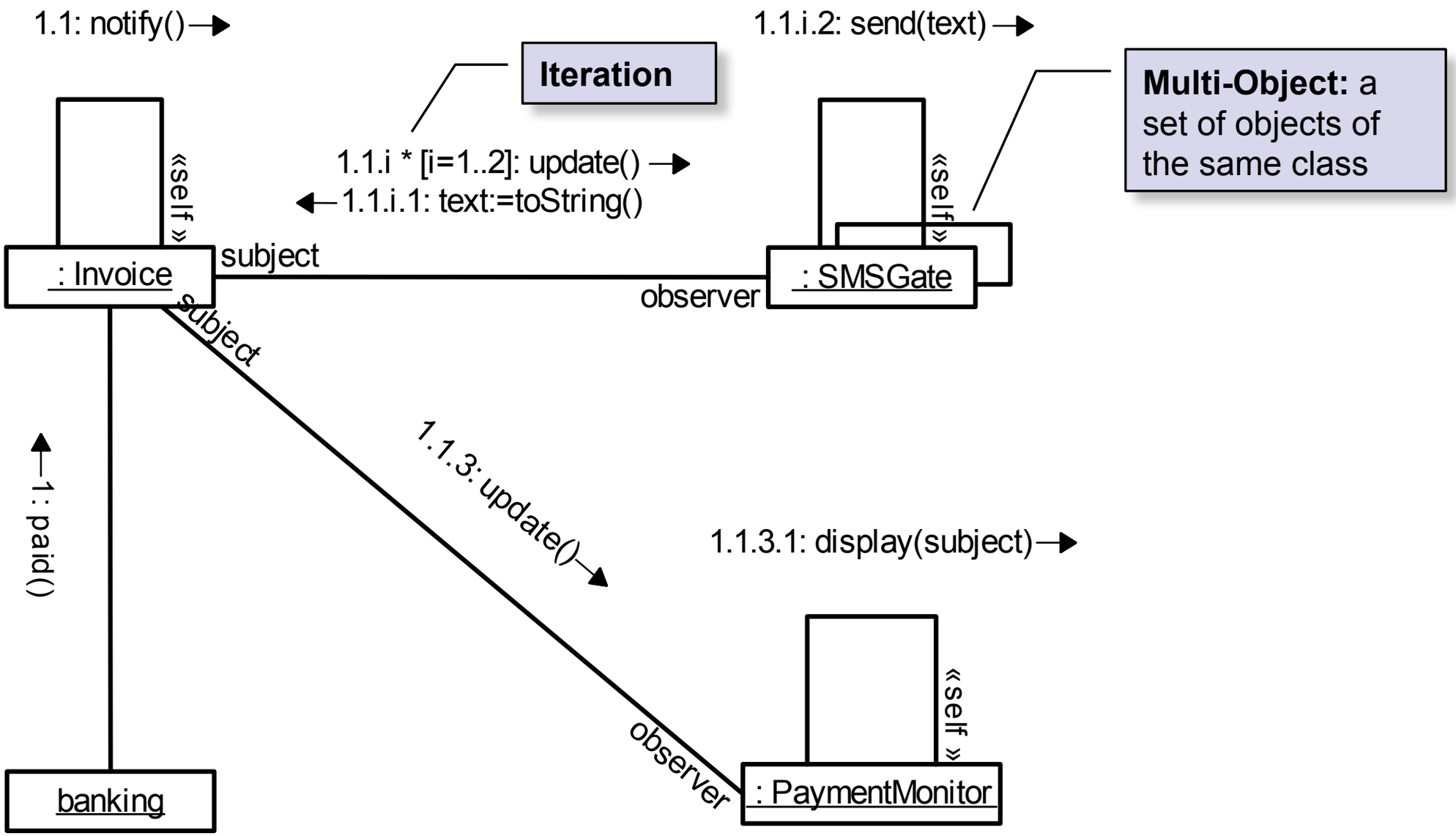
Motivation



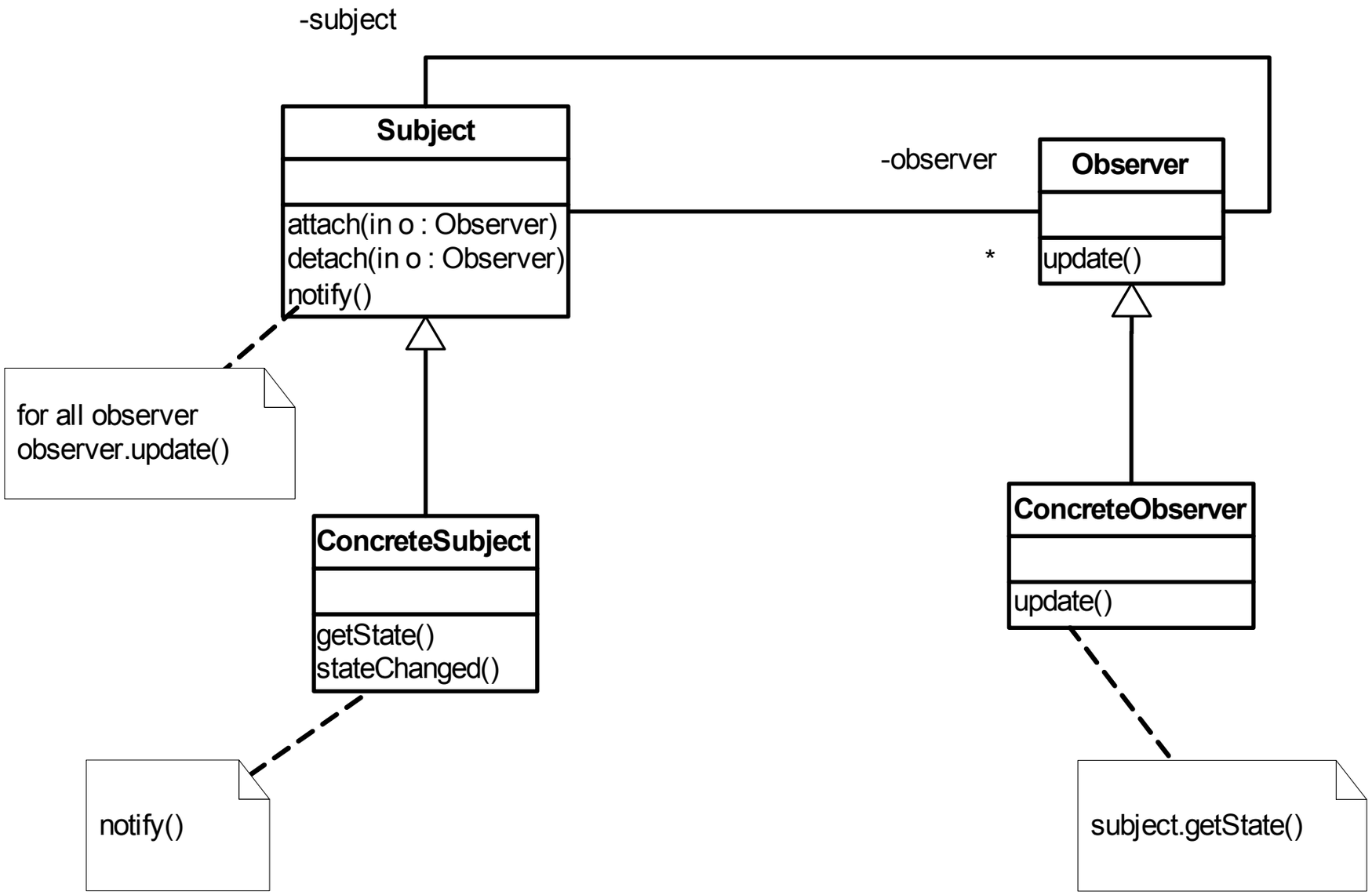
Observer - Solution



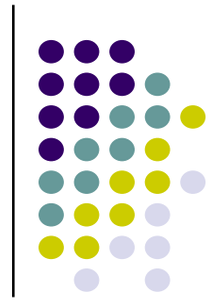
Observer – Solution (Collaboration)



Observer - Abstraction



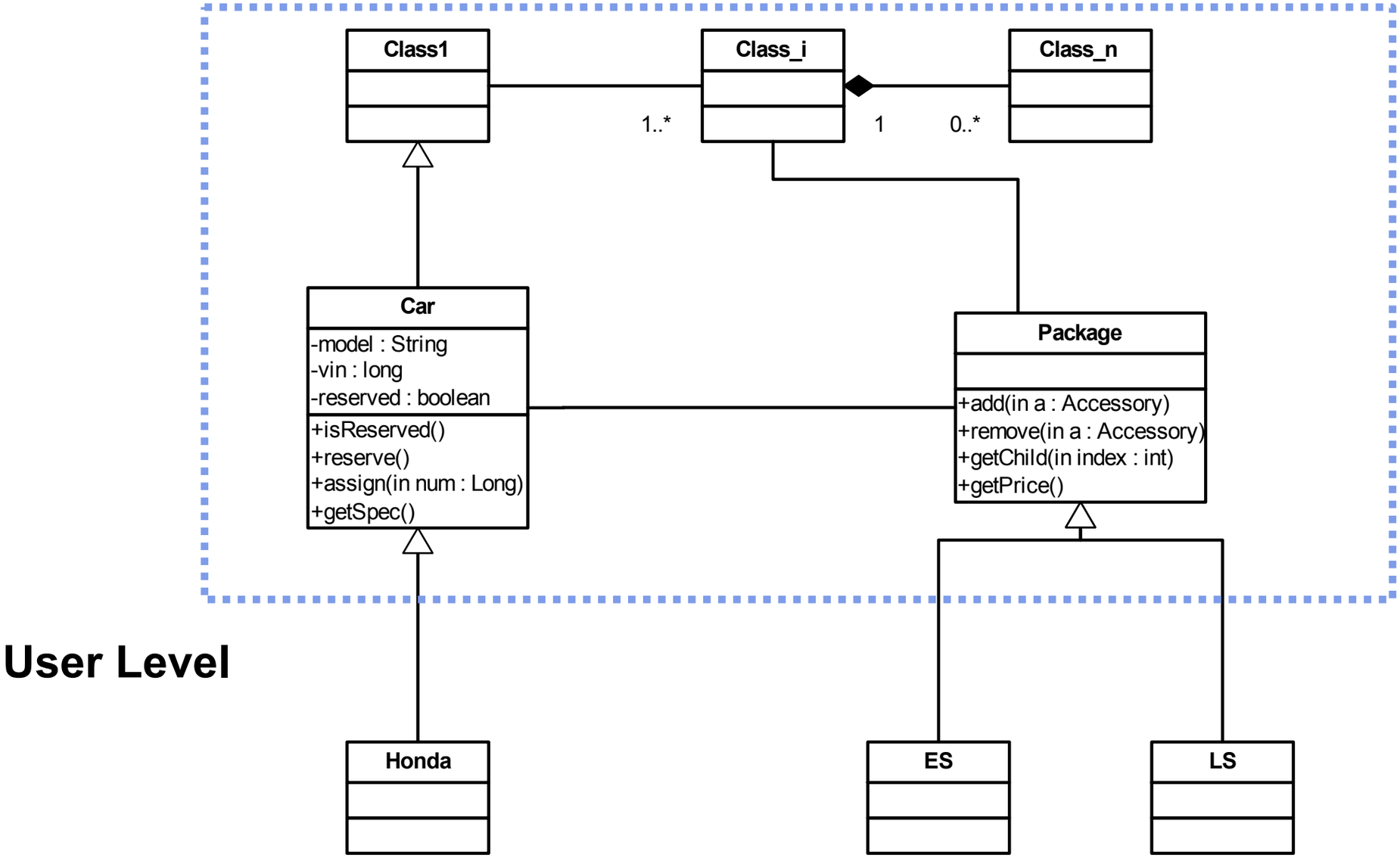
Exercises



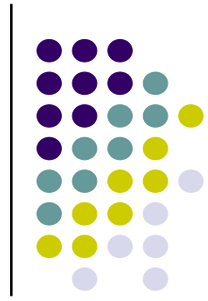
- Describe the design pattern **Observer**.
- Try to **integrate desing pattern Observer** into the design of video lending library.

Frameworks

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

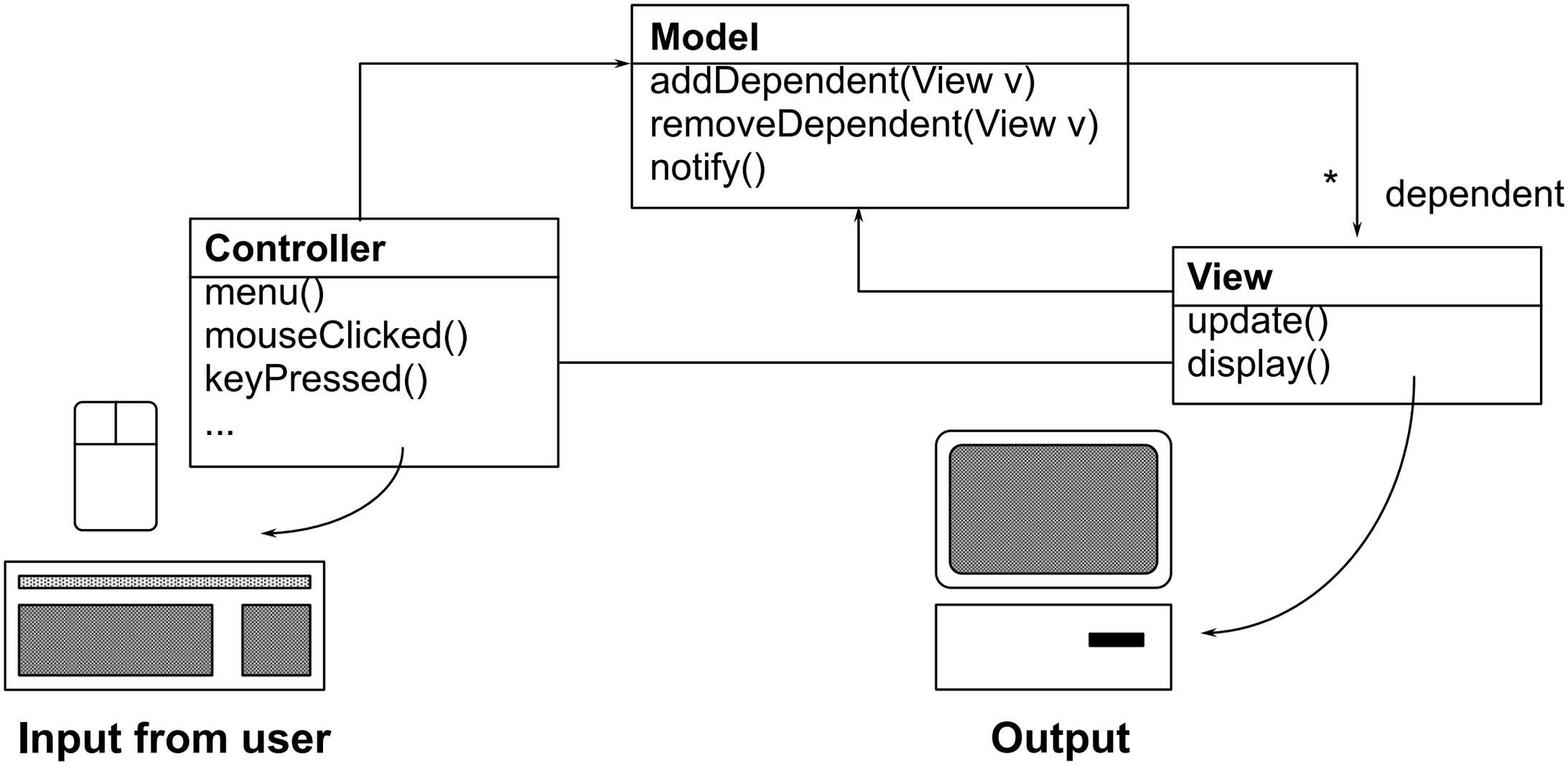


MVC – Application Framework

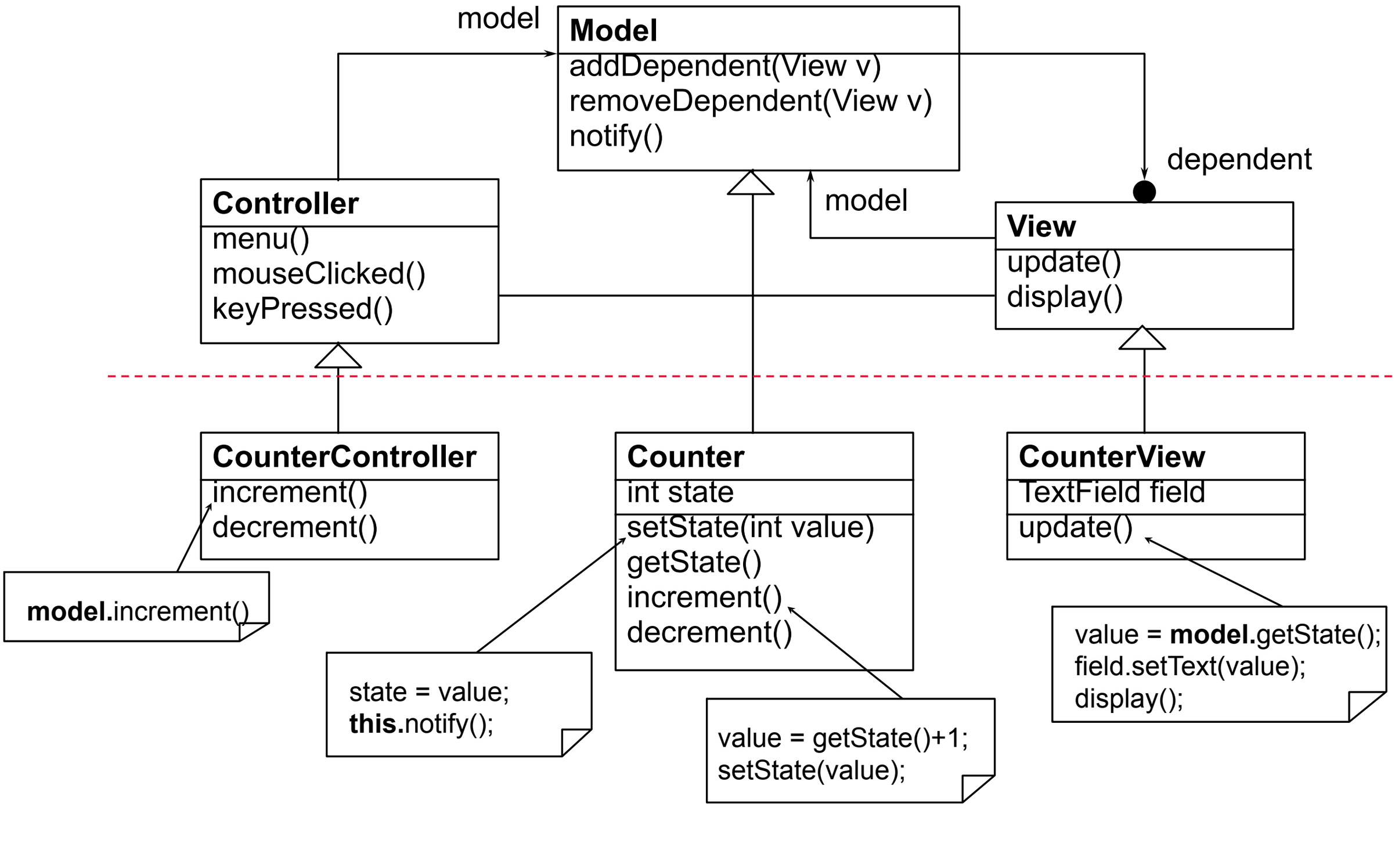


- Any application can be divided into two parts:
  - The **domain model**, which handles data storage and processing.
  - The **user interface**, which handles input and output.
- **Model-View-Controller** Architecture
  - **Model**: domain state and behavior (dynamic object).
  - **View**: display current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

MVC Classes


Model
addDependent(View v)
removeDependent(View v)
notify()
*
dependent
Controller
menu()
mouseClicked()
keyPressed()
...
View
update()
display()
Input from user
Output

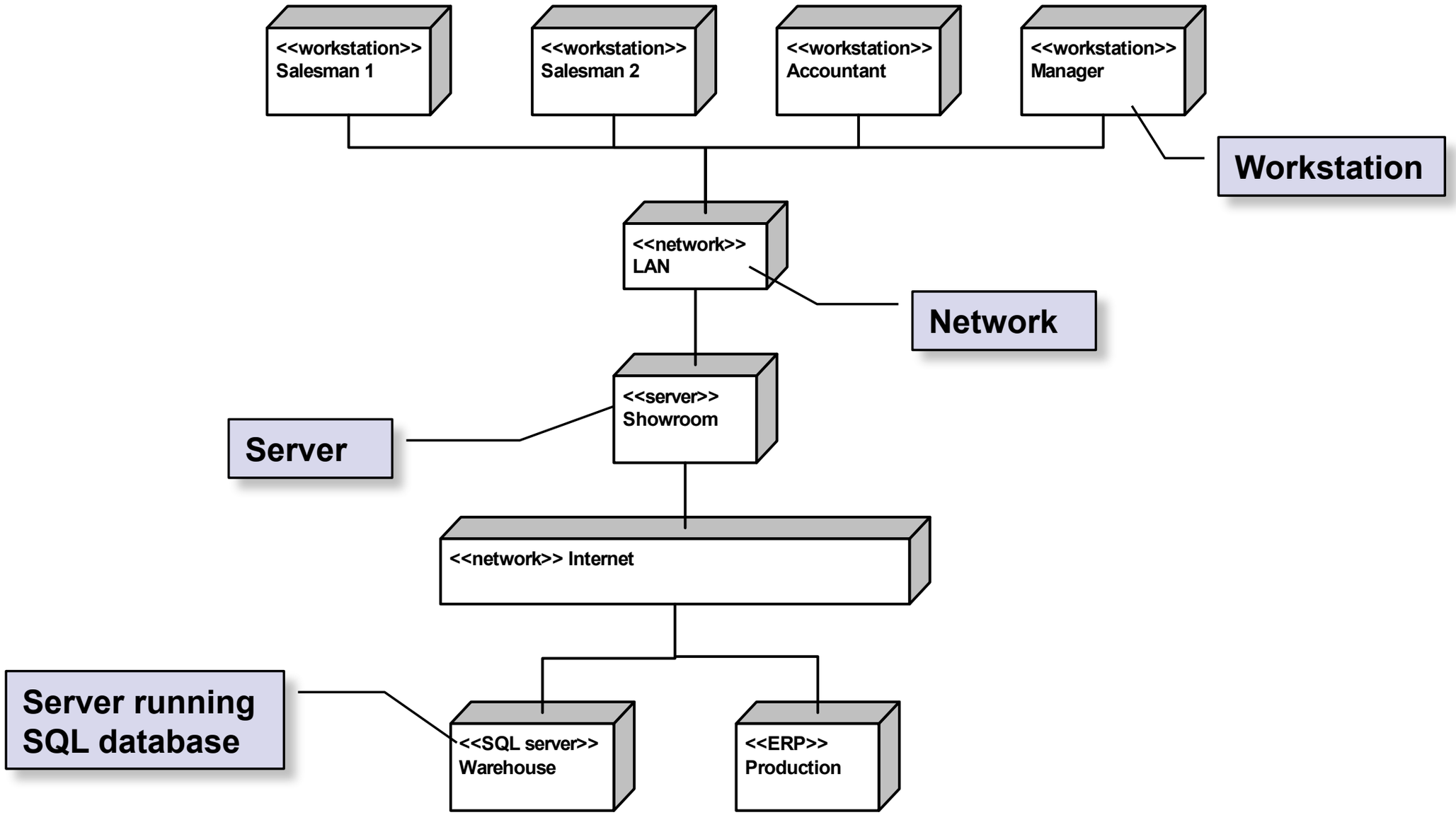
MVC Example


model
Model
addDependent(View v)
removeDependent(View v)
notify()
dependent
model
Controller
menu()
mouseClicked()
keyPressed()
View
update()
display()
CounterController
increment()
decrement()
Counter
int state
setState(int value)
getState()
increment()
decrement()
CounterView
TextField field
update()
model.increment()
state = value;
this.notify();
value = getState()+1;
setState(value);
value = model.getState();
field.setText(value);
display();

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

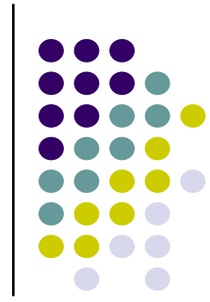


Implementation

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).

UML Diagrams for Implementation

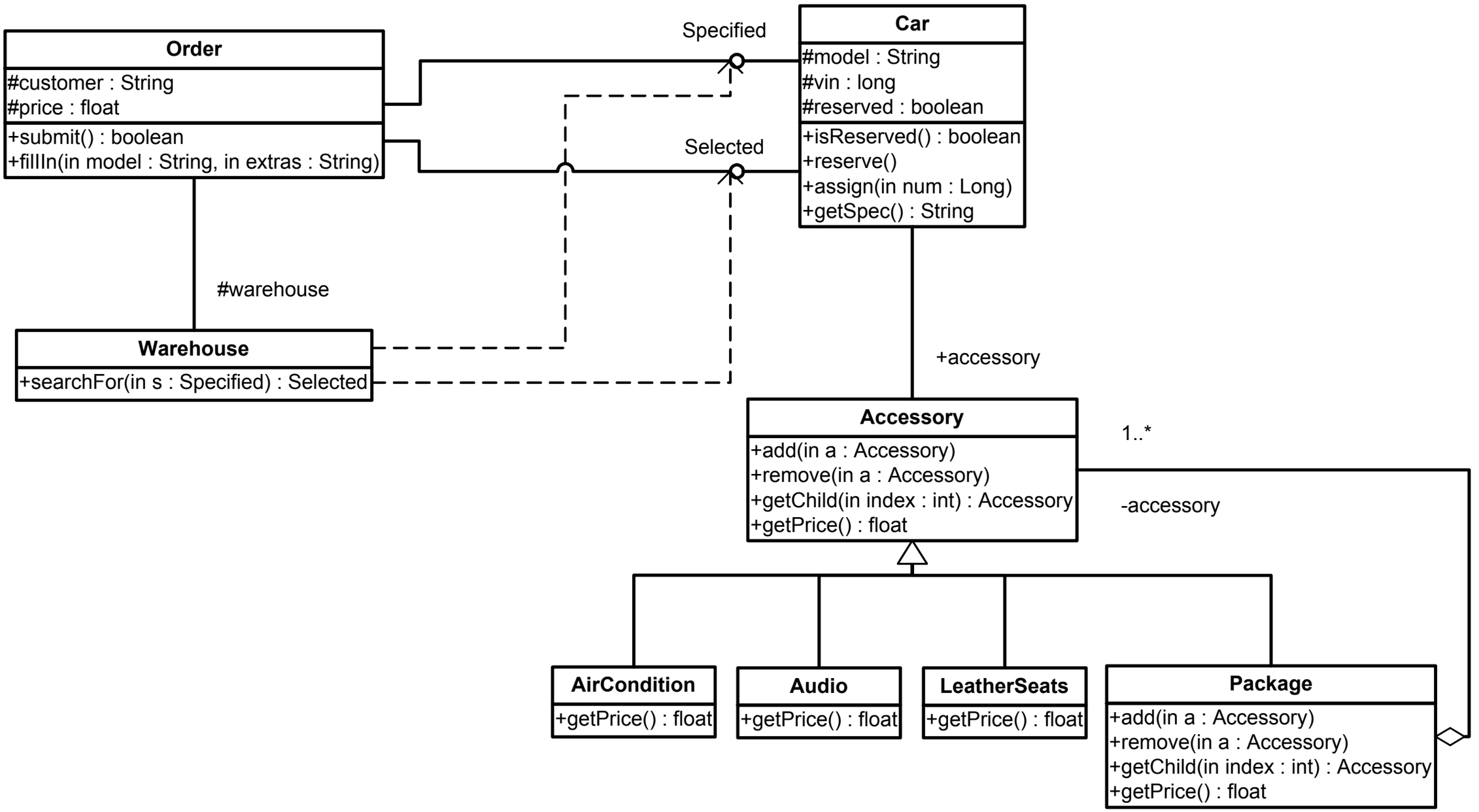


- **Component Diagram** illustrates the organization and dependencies among software components – **physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces**. A component may be
  - A source code component
  - A binary or executable component
  - 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.

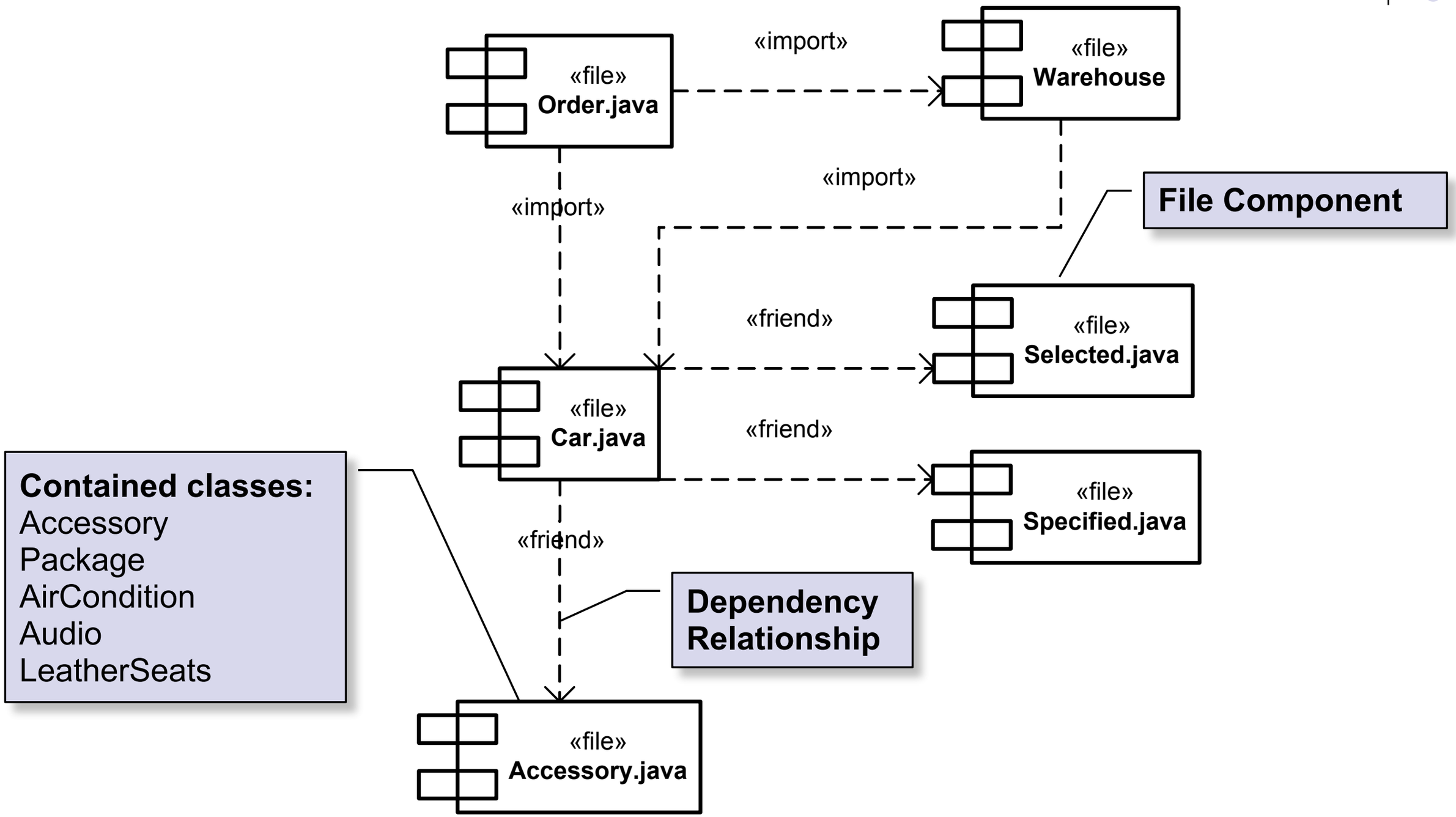
Mapping

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

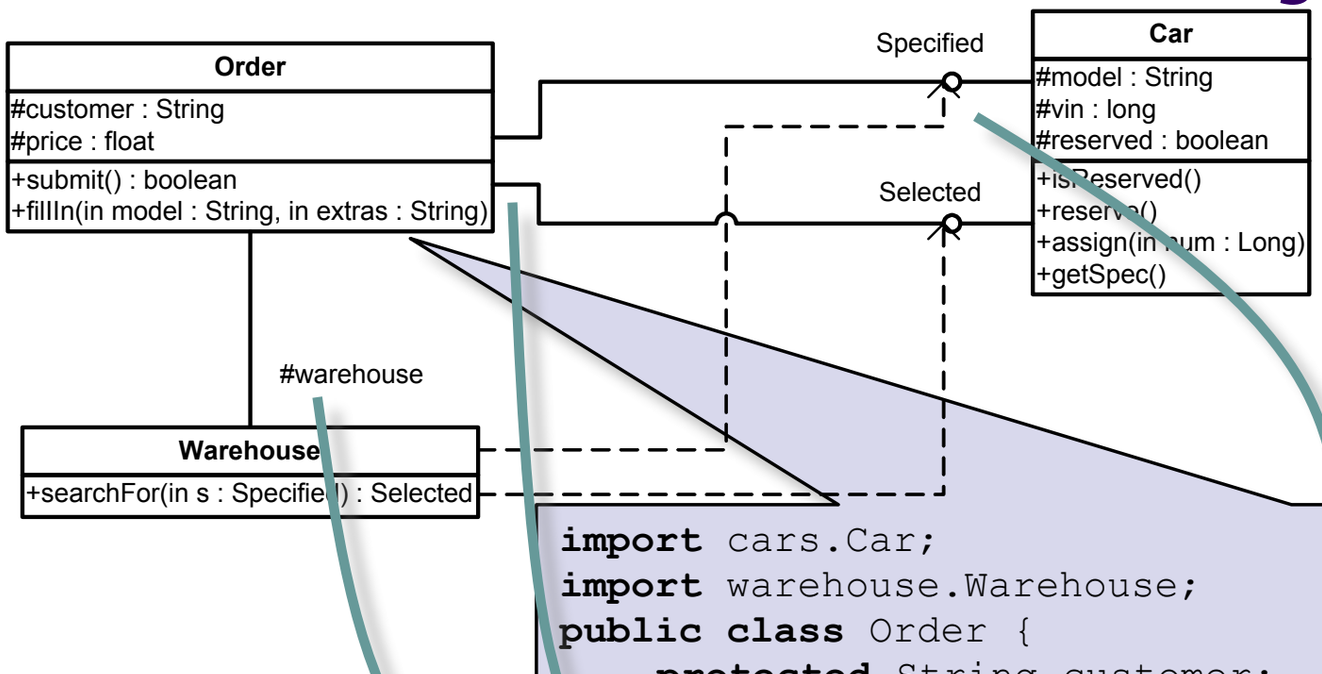
Design Results



Component Diagram: Source Code



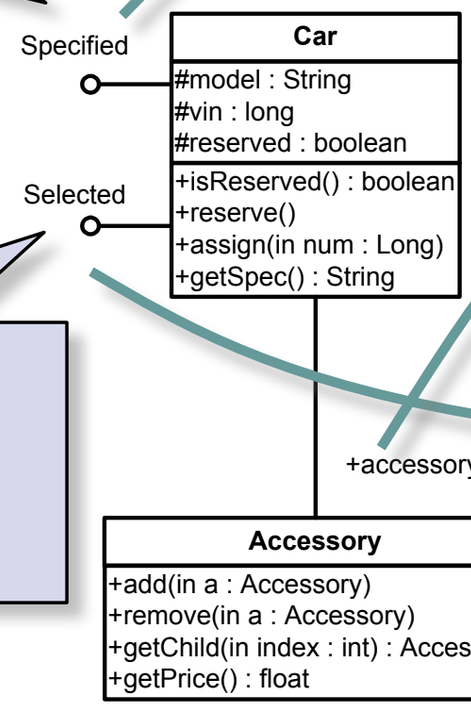
Source Code: Order.java




```
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

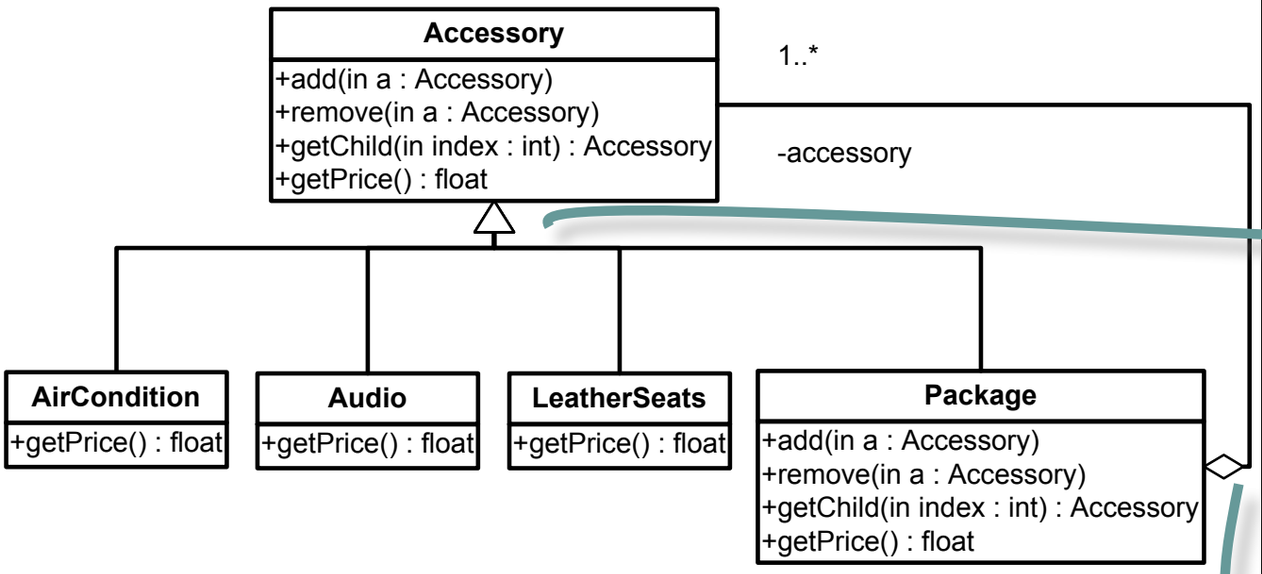
```
package cars;
public interface Specified {
    String getSpec();
}
```




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

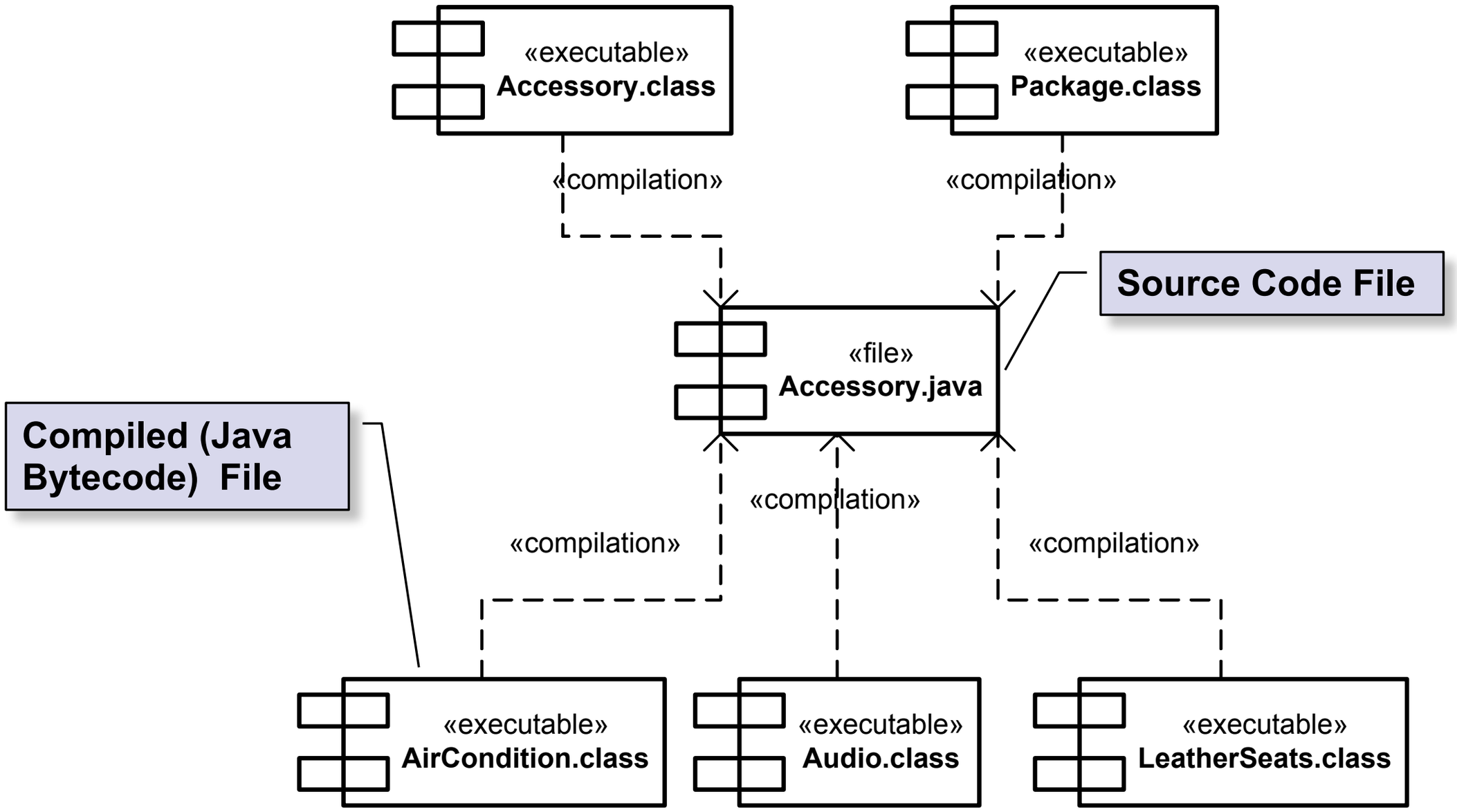
```
package cars;
public class Car implements Specified, Selected {
    protected String model;
    protected long vin;
    protected boolean reserved;
    public Accessory accessory;

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

Accessory.java

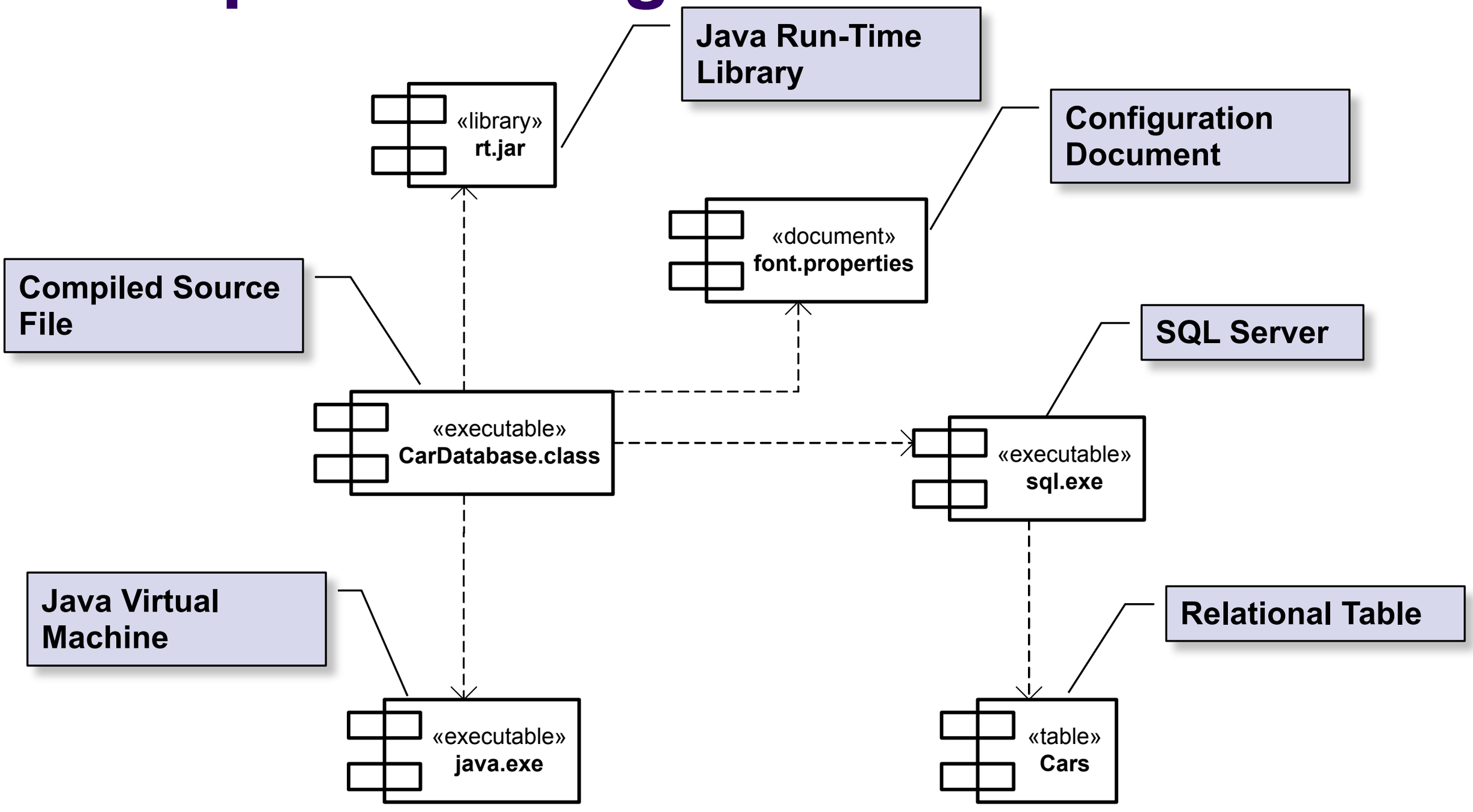



```
package cars;
public abstract class Accessory {
    public void add(Accessory a) {}
    public void remove(Accessory a) {}
    public Accessory getChild(int index) {
        return null;
    }
    public abstract float getPrice();
}

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;
    }
}
```

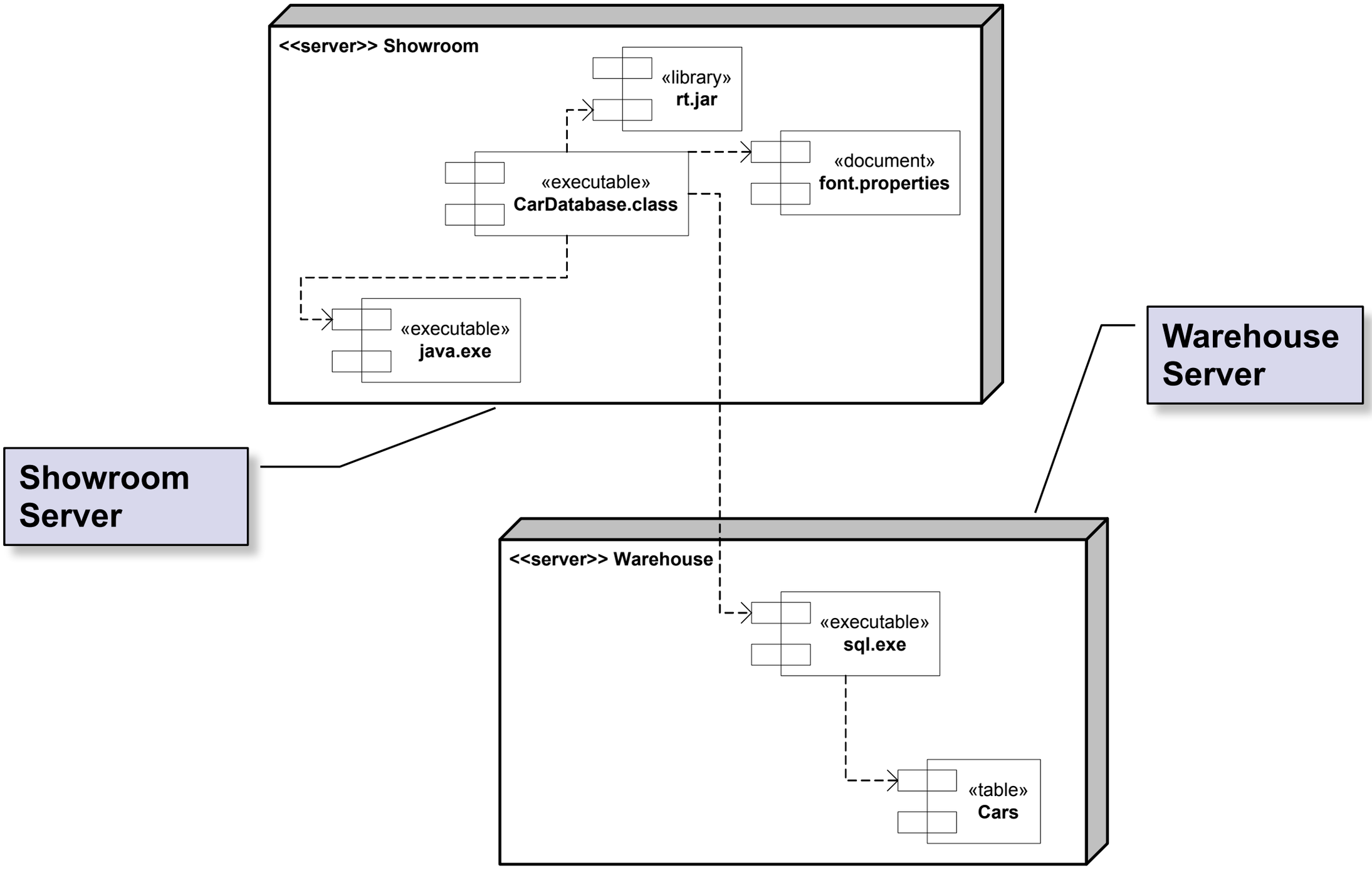
Component Diagram: Binaries



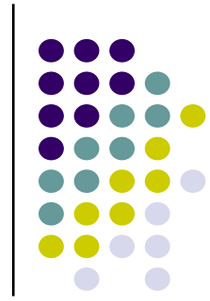
Component Diagram: Run-Time



Refined Deployment Diagram



Unit Tests



The purpose of performing a unit test is to test the implemented components as individual units. The following types of unit testing are done:

- **Specification testing**, or "black-box testing" verifies the unit's externally observable behavior
- **Structure testing**, or "white-box testing", verifies the unit's internal implementation
- **Integration and system tests** must be done to ensure that several components behave correctly when integrated
- Other tests of **performance**, **memory usage** and load

Exercises

- Based on class diagram of video lending library define the source **component diagram**.
- For the two classes with state diagram specified write the **source code in Java programming language**.

Test

Tests are carried out along three quality dimensions reliability, functionality, and system performance. Testing is related to all models and their diagrams!!!

The purposes of testing are:

- Plan the tests required in each iteration, including **integration tests and system tests**. Integration tests are required for every build within the iteration, whereas system tests are required only at the end of the iteration.
- Design and implement the tests by creating **test cases** that specify what to test, creating **test procedures** that specify how to perform the tests, and creating executable **test components** to automate the tests if possible
- Perform the various tests and systematically handle the results of each test. **Builds found to have defects** are retested and possibly **sent back to other core workflows**, such as design and implementation, so that the significant defects can be fixed.

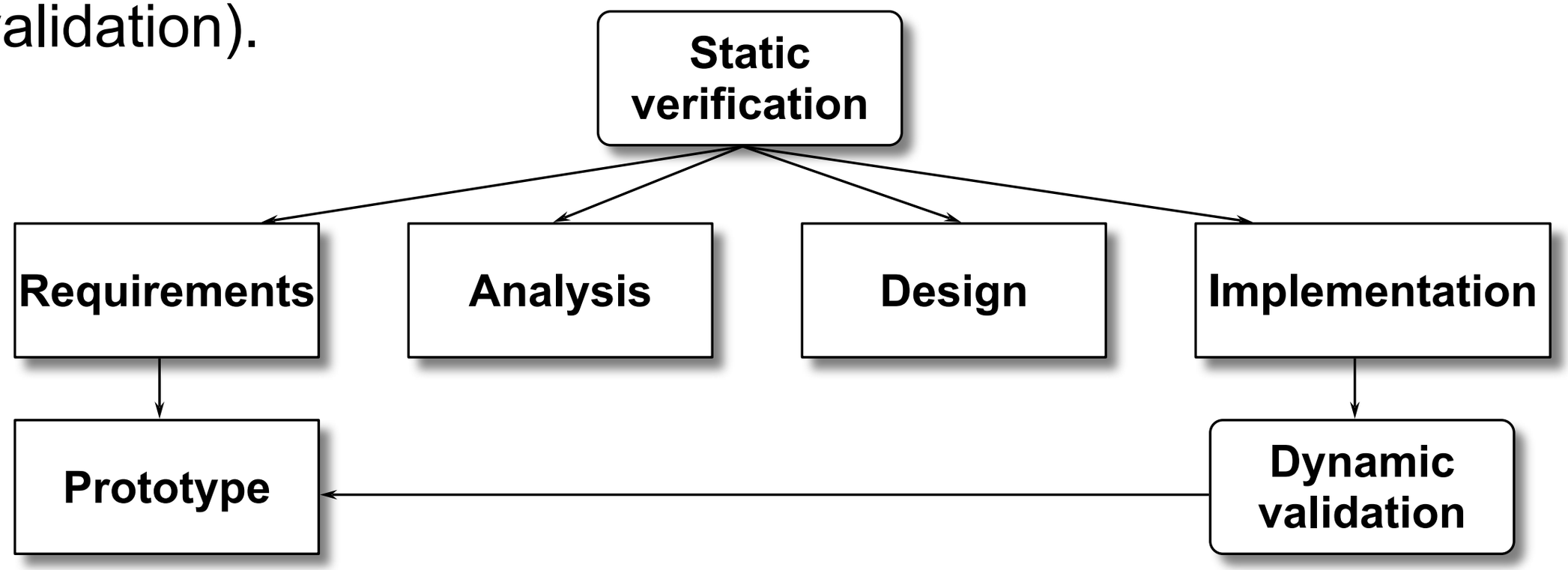
Verification and Validation

Verification and validation (V&V) is the term for checking processes which ensure that the software meets its requirements and that the requirements meet the needs of software procurer.

- **Verification:** Are we building the product right? => The discovery of defects in the system.
- **Validation:** Are we building the right product? => The assessment of whether or not the system is usable in an operational situation.

Static and Dynamic V&V

- **Static techniques** are concerned with the analysis of the system representation such as the requirements, analysis, design and program listing. ST can only check the correspondence between a program and its specification (verification).
- **Dynamic techniques** involve exercising an implementation. DT can demonstrate that the software operation is useful (validation).



Test Model

Test Model is a collection of:

- **Test cases**, which specify what to test in the system.
- **Test procedures**, which specify how to perform the test cases.
- **Test components**, which automate the test procedures

Test Case

Test case specifies one way of testing the system, including what to test with which input, and under which condition to test.

The following are common test cases:

- A test case that specifies how to test a use case or a specific scenario of a use case. Such a test case includes verifying the result of the interaction between the actor and the system (**"black-box" test**). Black-box test is the test of the **externally observable behavior** of the system
- A test case that specifies how to test a use case realization. Such a test case includes verifying the interaction between the components implementing the use case (**"white-box" test**). White-box test is the test of the **internal interaction between components** of the system

Test Procedure

Test procedure specifies how to perform one or several test cases or parts of them.

Test procedures can be based on the following:

- Instructions for an individual on how to perform a test case manually
- Description of how to create executable test components
- Specifications of how to interact with a test automation tool

Test Component

Test component automates one or several test procedures or parts of them. Test components are used to test the components in the implementation model by providing test inputs, controlling and monitoring the execution of the tested component, and possibly reporting the test results.

Test components can be developed using:

- Scripting language or a programming language
- Test automation tool

Deployment

The purpose of the deployment workflow is to **successfully produce product releases**, and deliver the software to its end users. It covers a wide range of activities including:

- Producing external releases of the software.
- Packaging the software.
- Distributing the software.
- Installing the software.
- Providing help and assistance to users.
- Planning and conduct of beta tests.
- Migration of existing software or data.