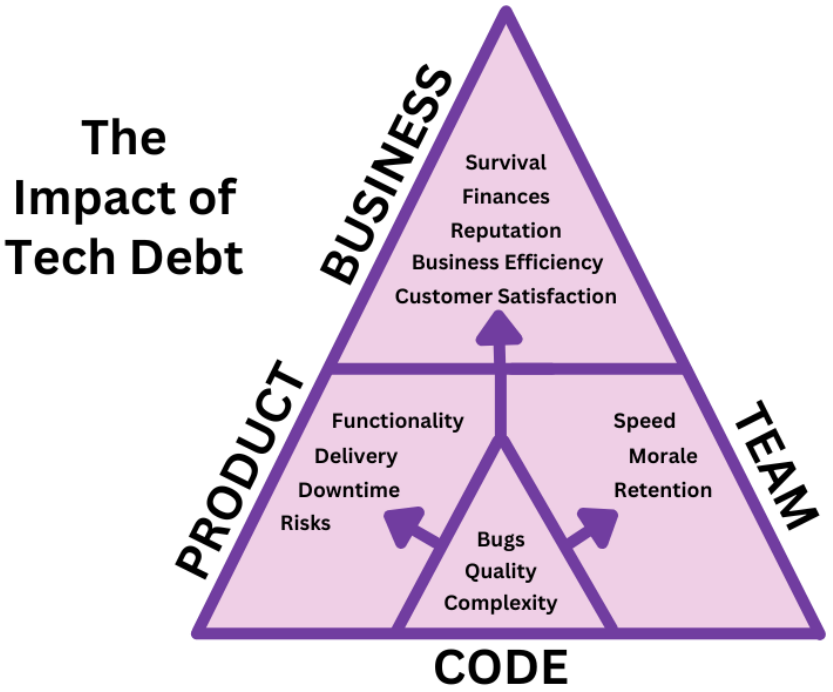
LESSON 1 ASD INTRODUCTION

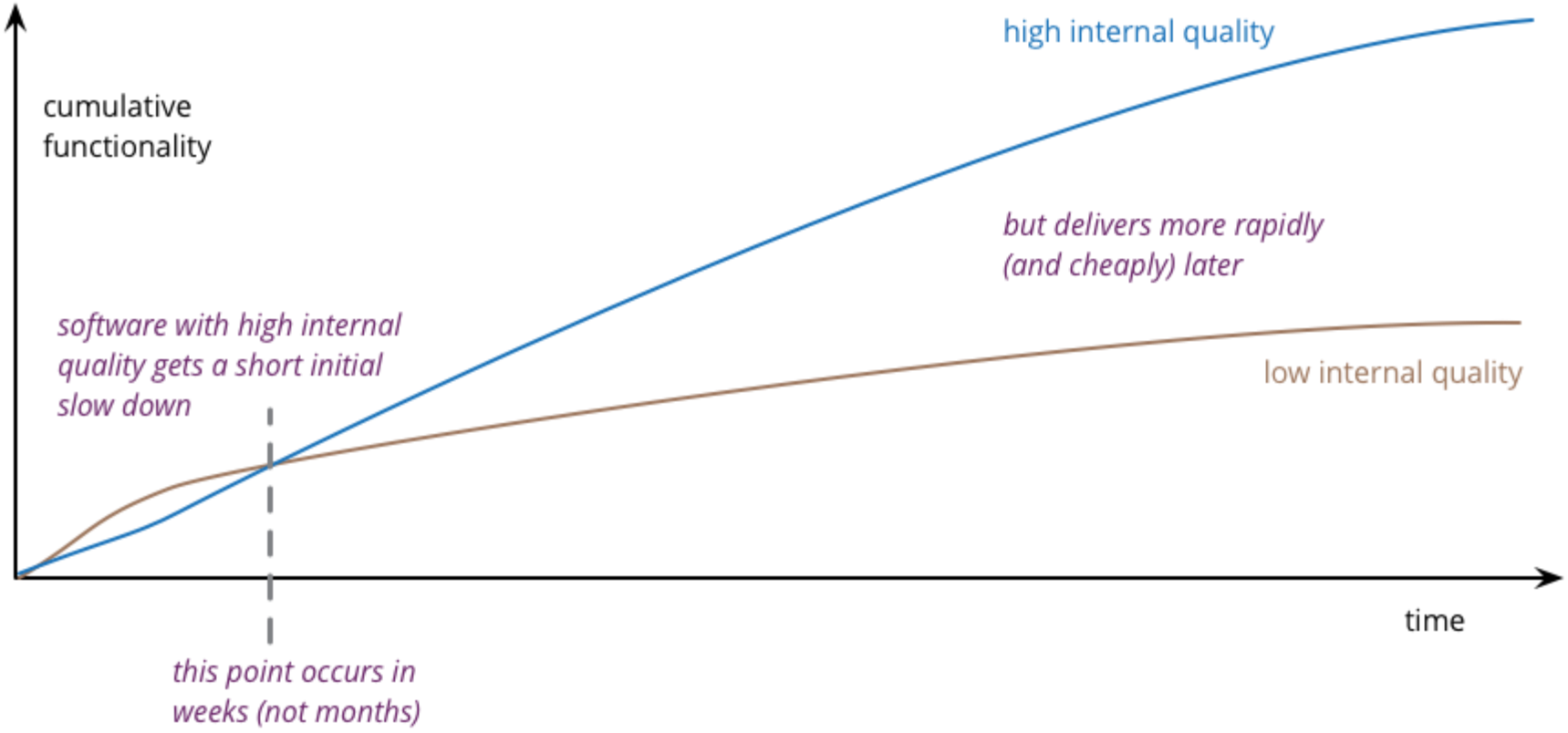
Advanced Software Development

- Principles and best practices of good software design
  - Design patterns
  - Frameworks
- Improve the quality of your design/code
  - Reduce technical debt

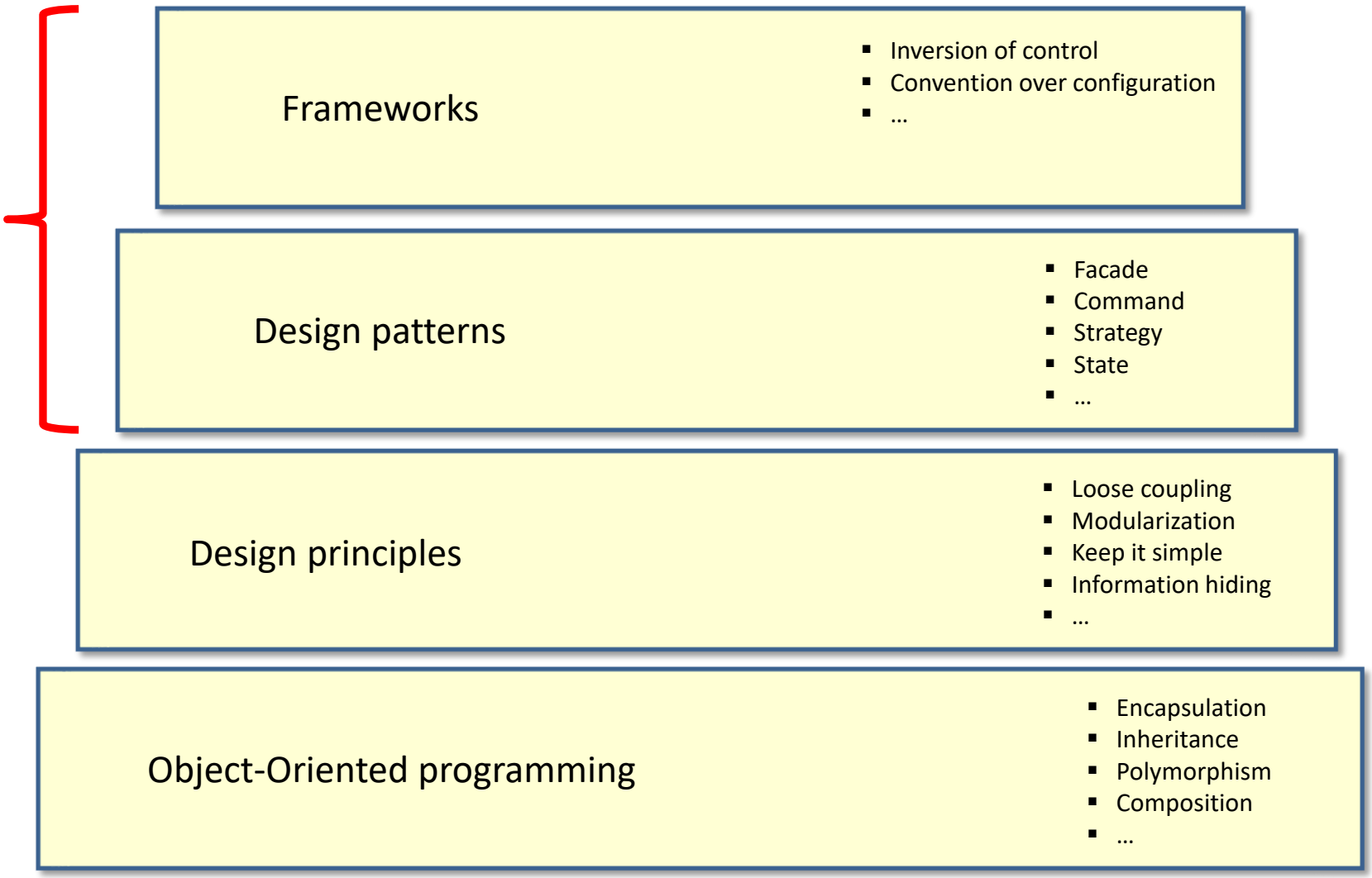
Technical debt



Why software quality matters?



ASD course

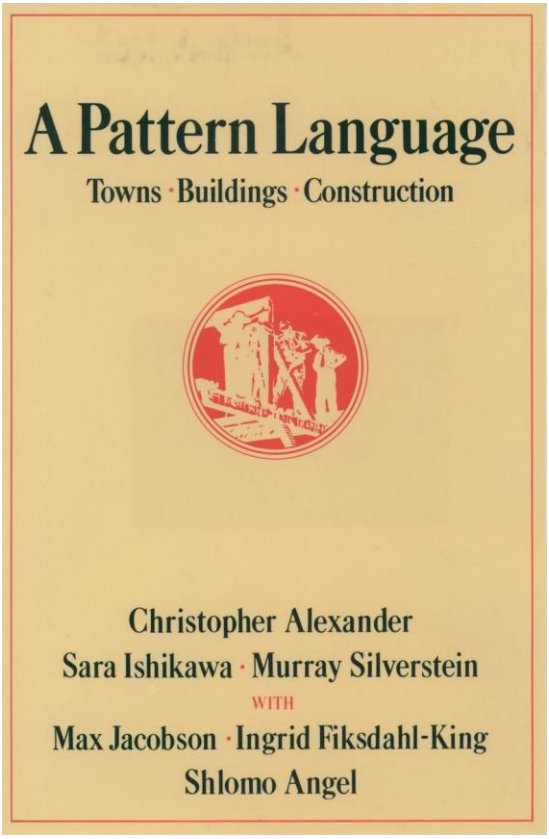


Lesson overview

Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
July 22 **Lesson 1** **Introduction**	July 23 **Lesson 2** **Command**	July 24 **Lesson 3** **Strategy** **Template method**	July 25 **Lesson 4** **Observer**	July 26 **Lesson 5** **State**	July 27 **Lesson 6** **Iterator** **Composite**	July 28
July 29 **Lesson 7** **COR** **Combining patterns**	July 30 **Lesson 8** **Adapter** **Mediator** **Proxy**	July 31 **Midterm Review**	August 1 **Midterm exam**	August 2 **Lesson 9** **Builder** **Factory** **Singleton**	August 3 **Lesson 10** **Decorator** **Visitor**	August 4
August 5 **Lesson 11** **Framework design**	August 6 **Lesson 12** **Framework design**	August 7 **Lesson 13** **Spring framework**	August 8 **Lesson 14** **Framework implementation**	August 9 **Final review** **Project**	August 10 **Project**	August 11
August 12 **Final exam**	August 13 **Project**	August 14 **Project**	August 15 **TBD**			

Pattern

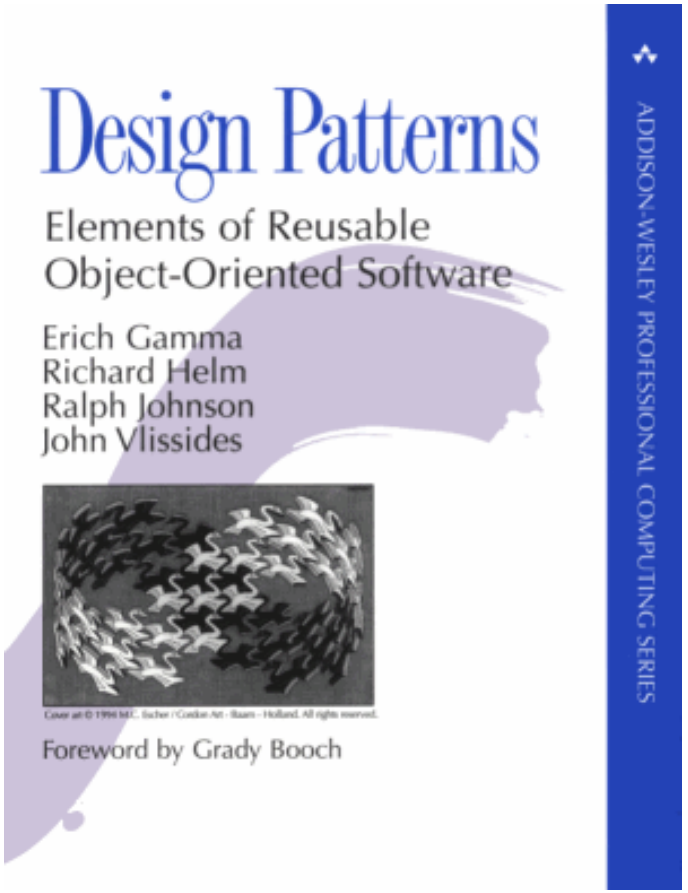
- Christopher Alexander



A pattern is a general, reusable solution to a commonly occurring problem within a given context

Design pattern

- A general repeatable solution to a commonly occurring problem in software design.
- Reuse of design (not code)
- GoF: Gang of Four



Patterns in software development

A pattern is a general, reusable solution to a commonly occurring problem within a given context

Architecture patterns

- Client-server
- Layering
- Components
- Microservices
- Stream based
- …

Integration patterns

- Router
- Filter
- Point-to-point
- Transformer
- …

UI patterns

- Navigation
- Dealing with data
- Forms
- Menus
- …

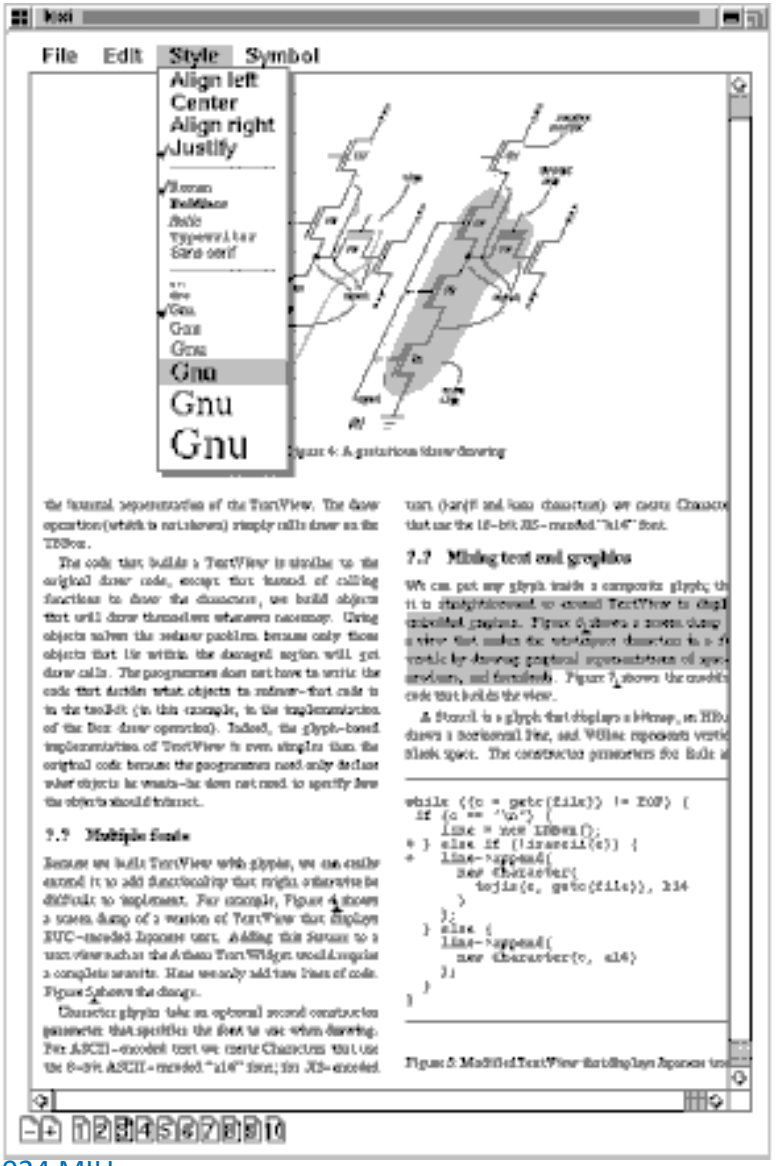
Data access patterns

- ORM
- Stored procedures
- SQL
- Lazy loading
- Id generation
- …

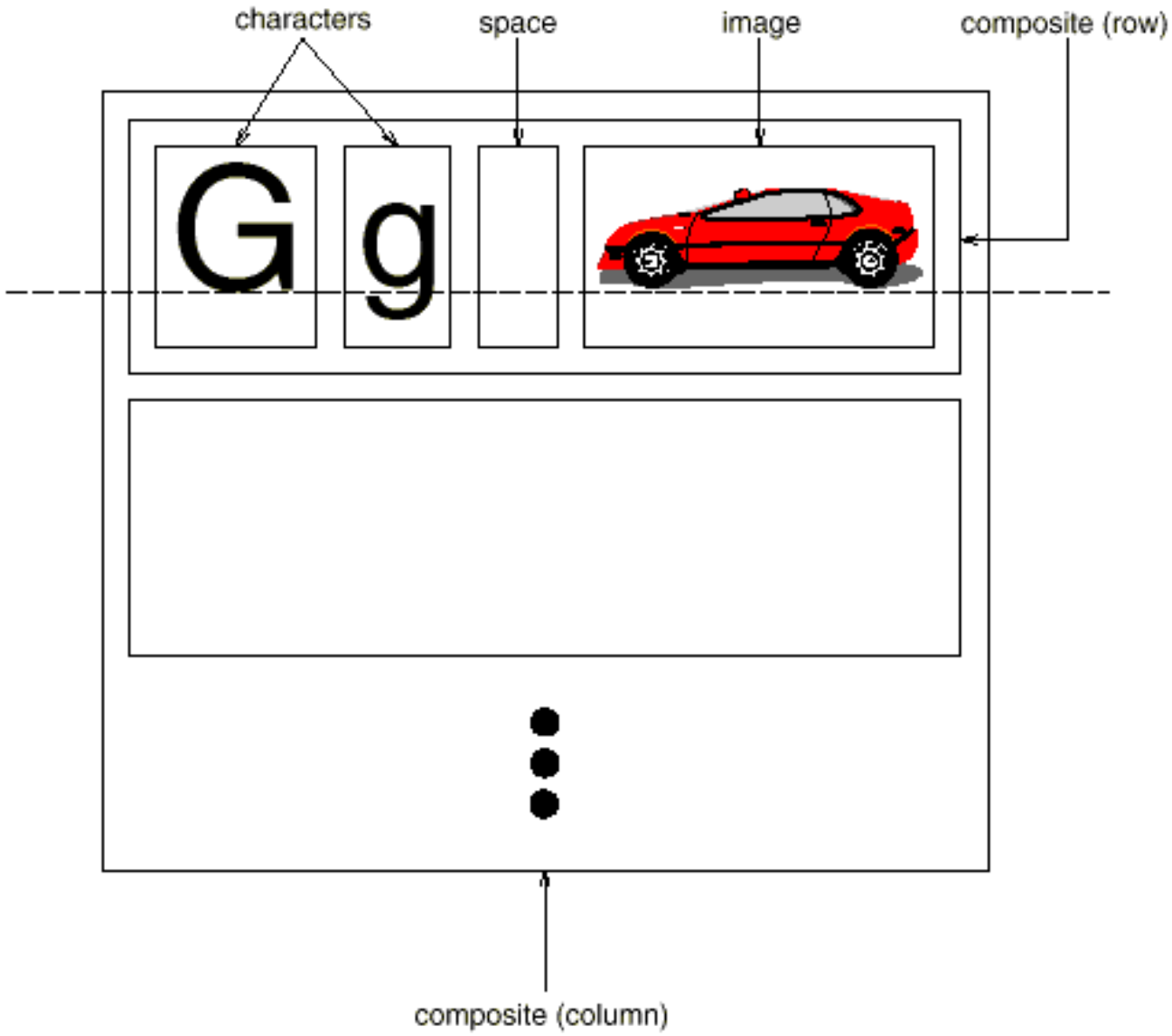
Design patterns

- Facade
- Command
- Strategy
- State
- …

Document editor

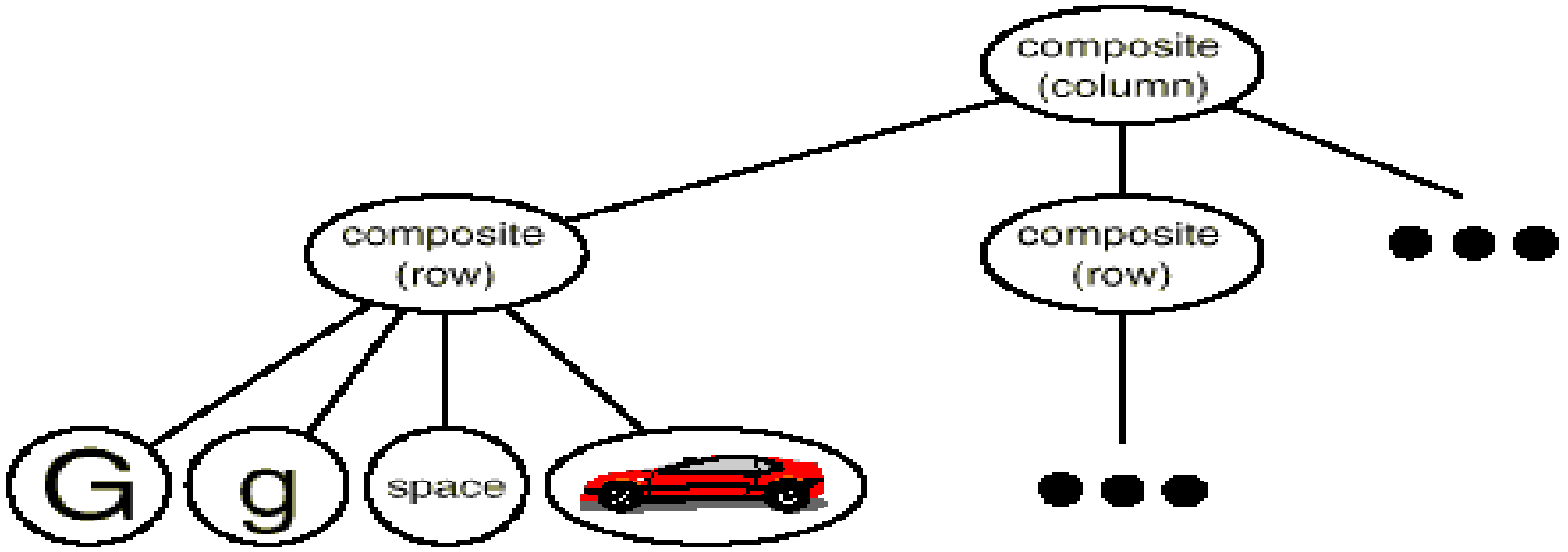


1. Document structure

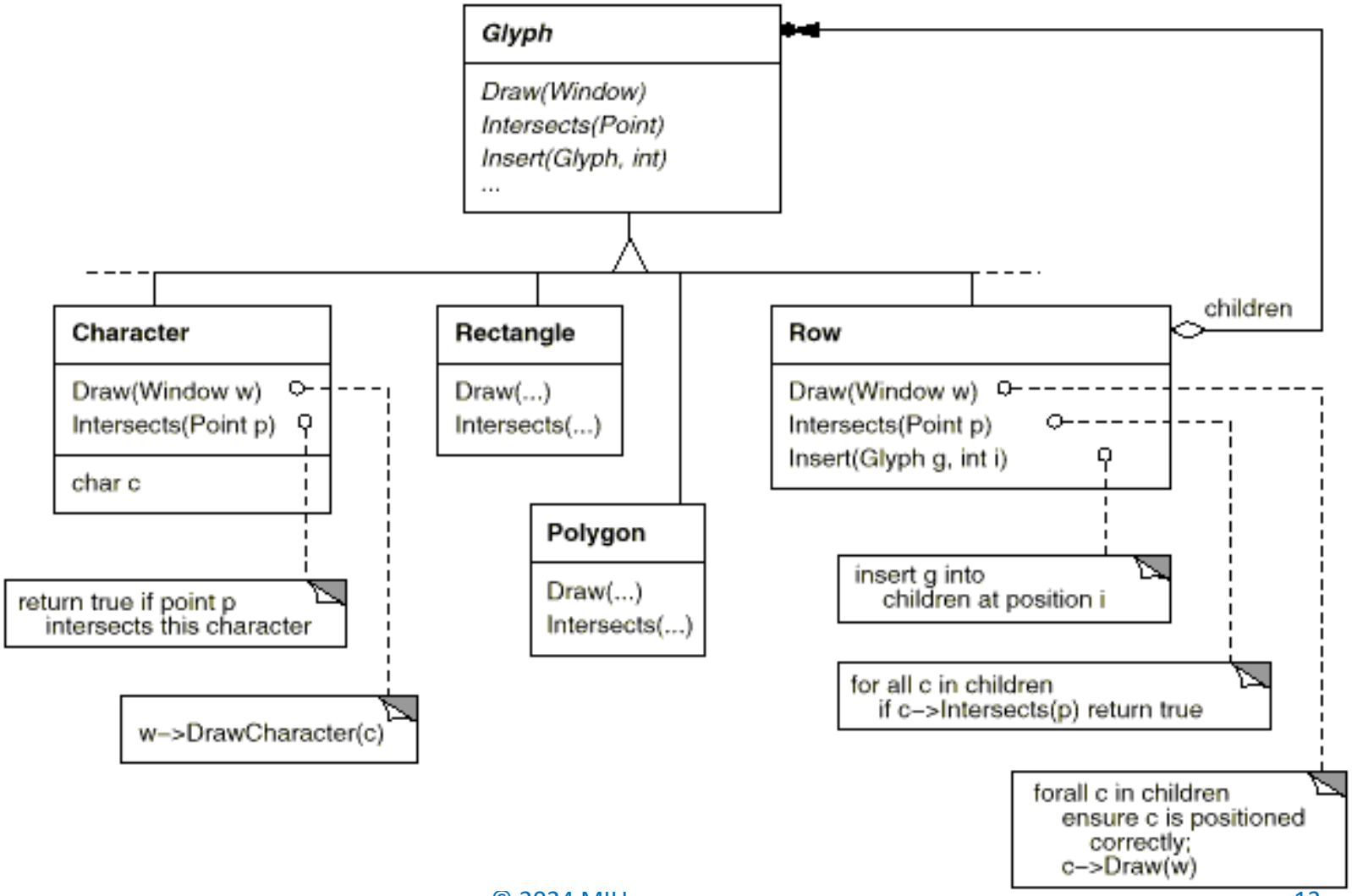


Tree structure

- We want to work with these tree elements in common way
  - Copy-paste
  - Drag & drop

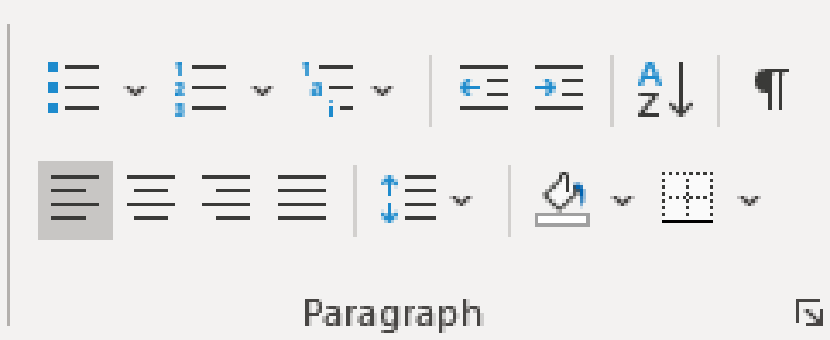


Composite pattern

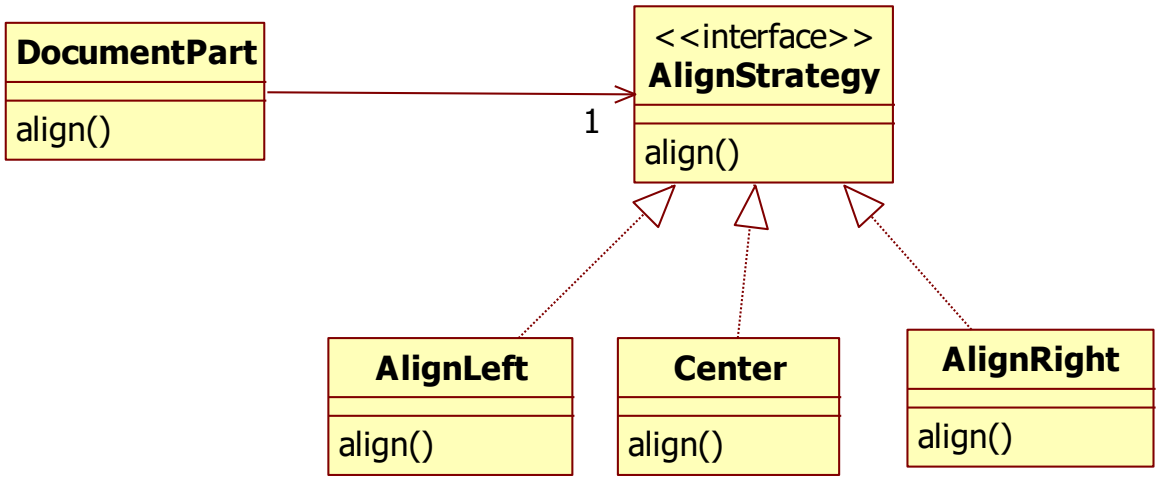


2. Formatting

- How can we format the structure of the document?
- We need to allow different ways to format.

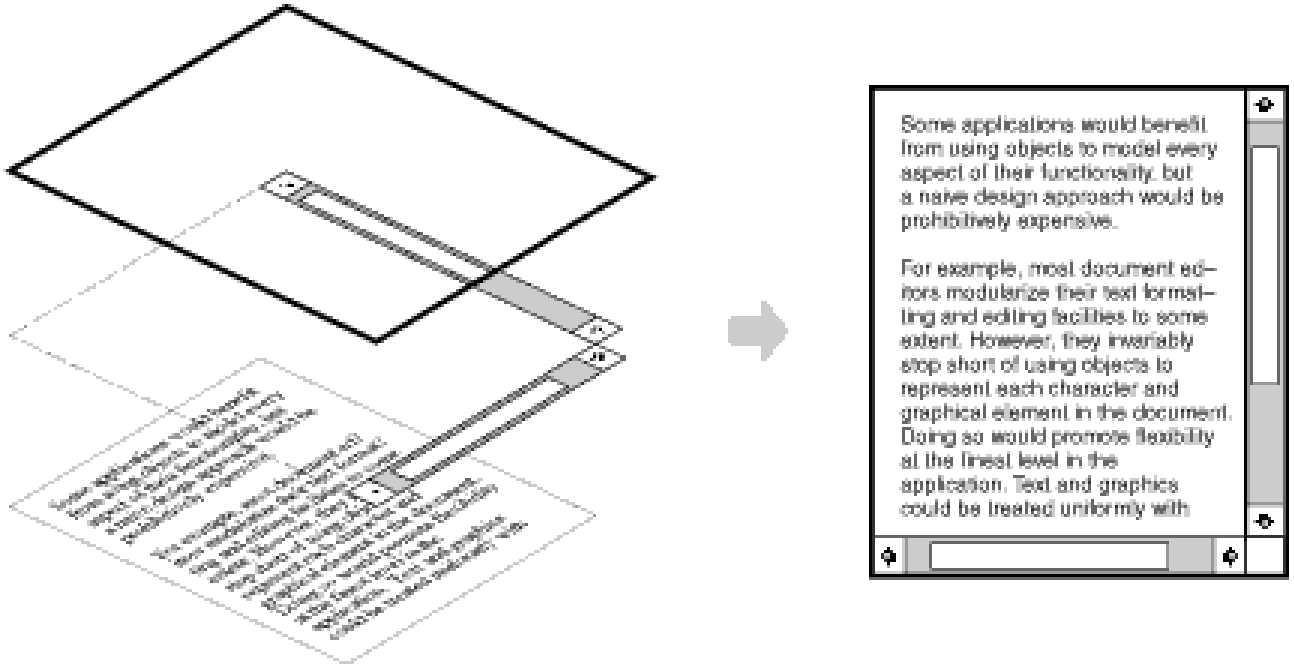


Strategy pattern

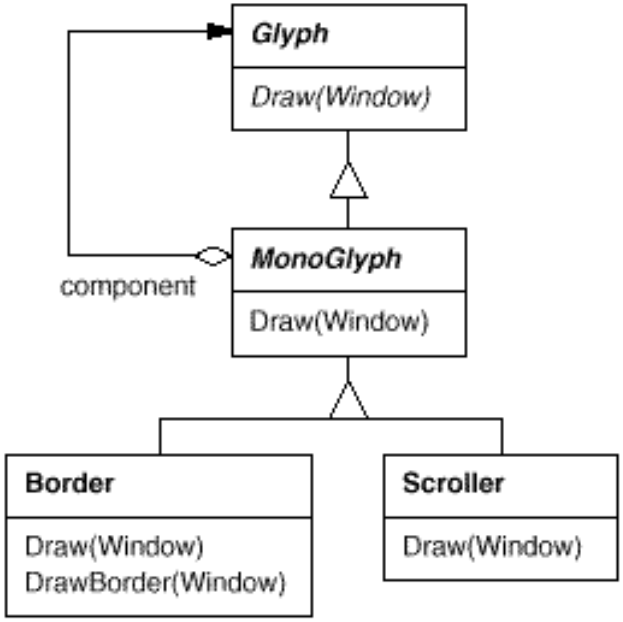


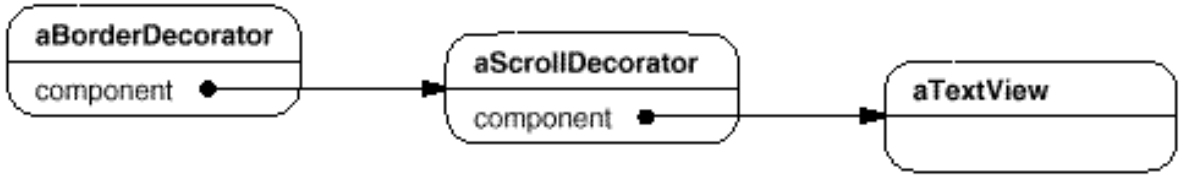
3. Embellishing the user interface

- Add a Border around the text editing area
- Add scroll bars



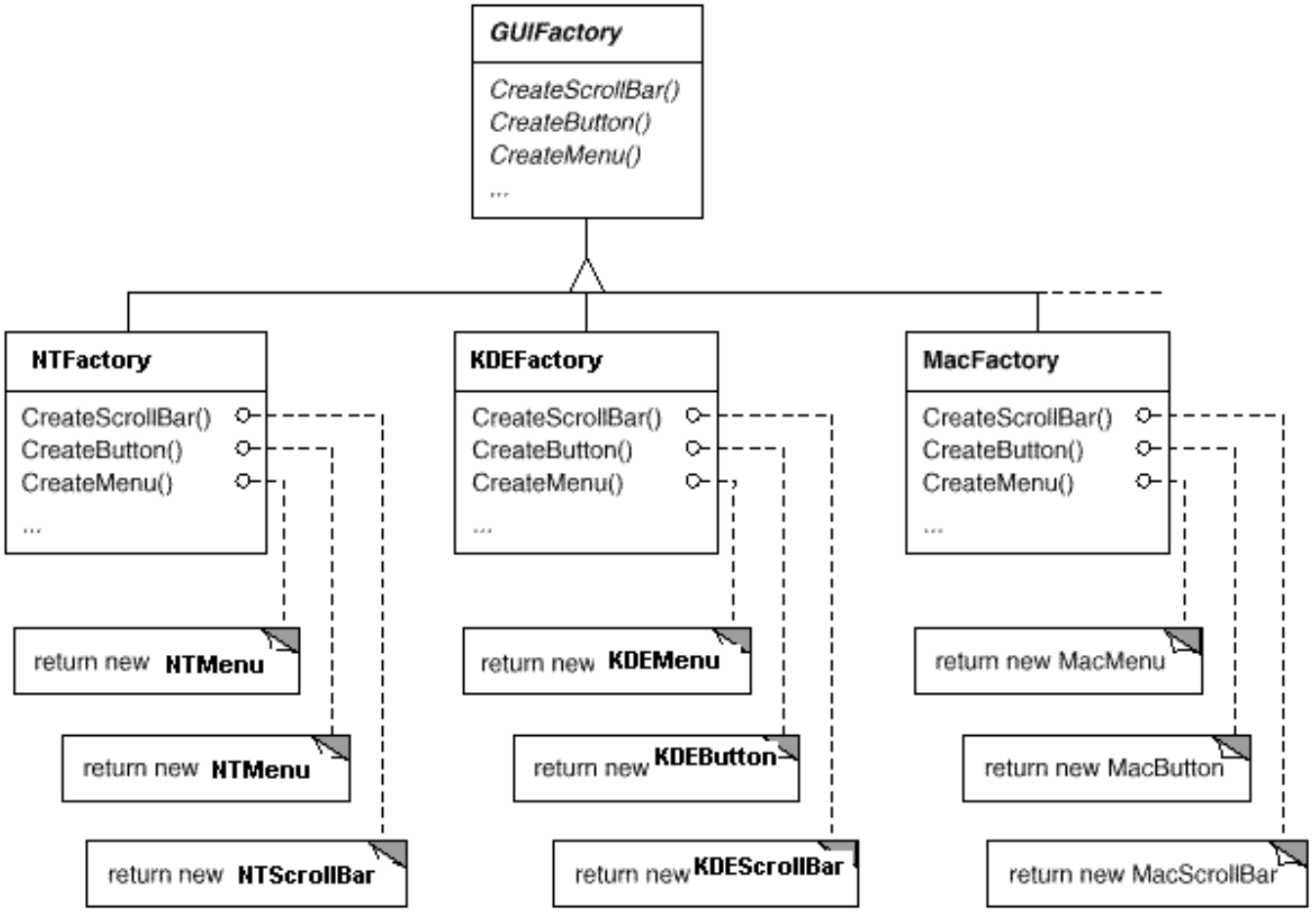
Decorator pattern





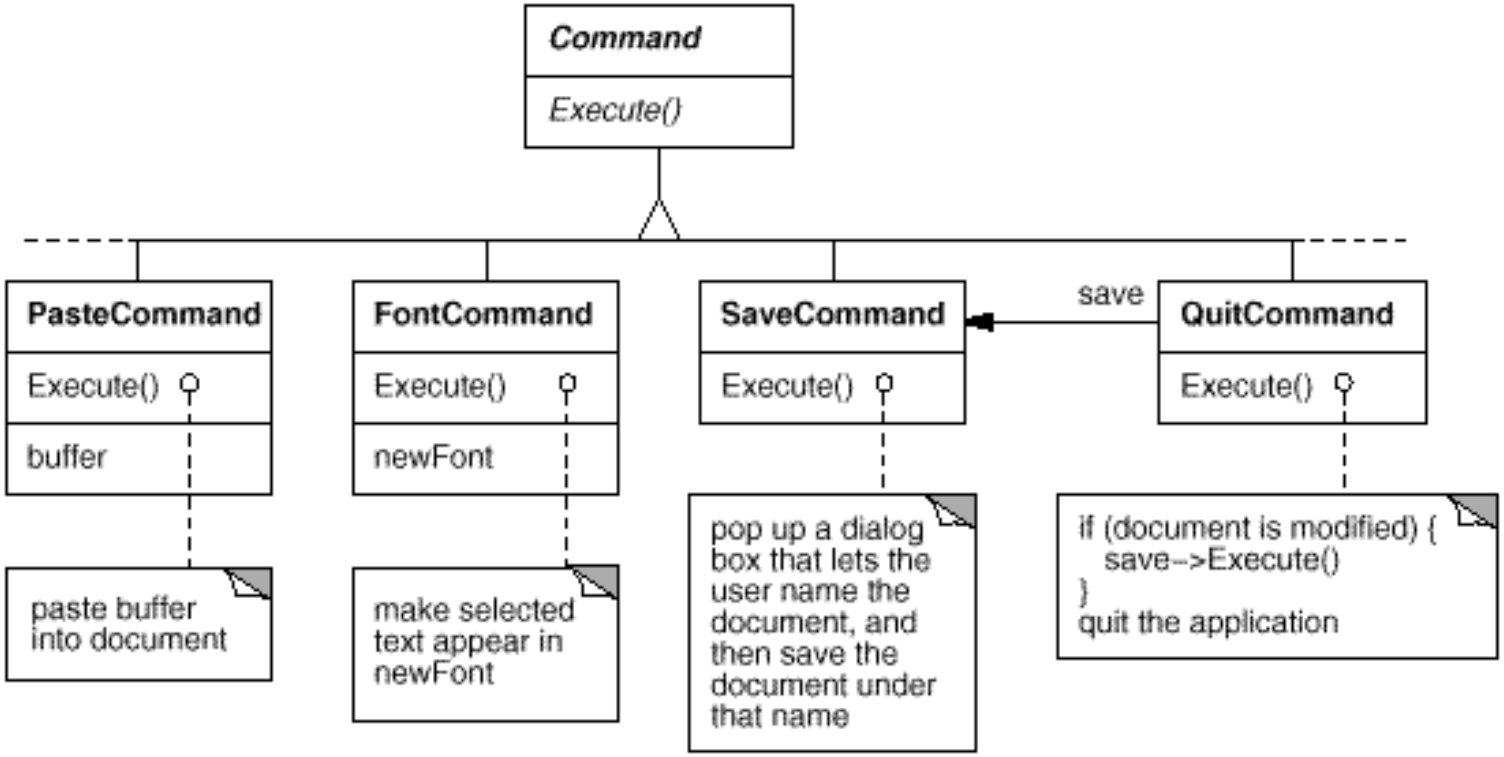
4. Supporting multiple look and feels

- Abstract factory pattern



5. User operations

- Should be independent of the UI
- We also want undo/redo
- Command pattern



Categories of patterns

Creational	Structural	Behavioral
▪ Factory method ▪ Abstract factory ▪ Builder ▪ Singleton	▪ Composite ▪ Decorator ▪ Adapter ▪ Façade ▪ Proxy	▪ Command ▪ Iterator ▪ Mediator ▪ Chain of responsibility ▪ Observer ▪ State ▪ Strategy ▪ Template method

Half baked

- A design pattern isn't a finished design that can be transformed directly into code.
- Design patterns are half baked
- You have to tailor them for your situation

How to become a good designer?

- By designing software
  - Class diagrams
  - Sequence diagrams
  - Code

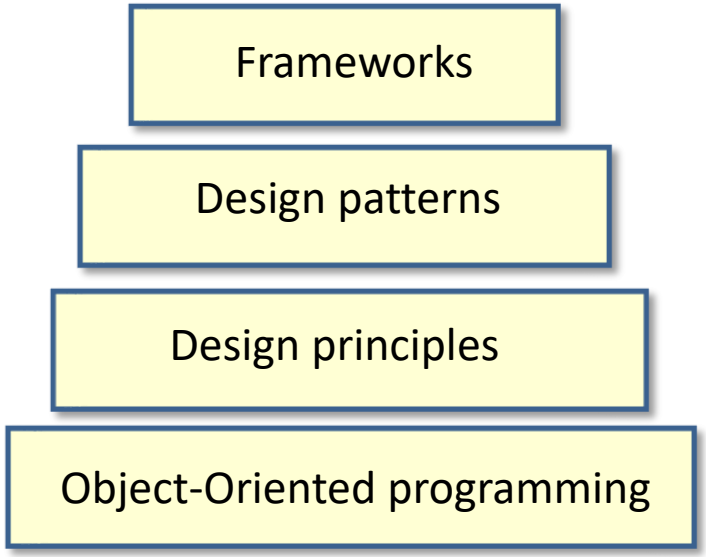
Main point

- A design pattern is a reusable solution for a generic design problem within a context
- The unified field is the field of all possibilities which contains the intelligence to solve all problems in the most optimal way.

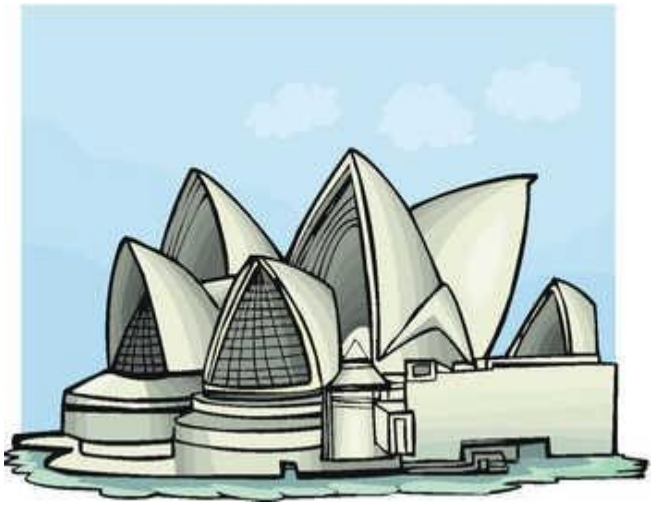
DESIGN PRINCIPLES

Design principles

- Keep it simple
- Keep it flexible
- Loose coupling
- Separation of concern
- Information hiding
- Principle of modularity
- DRY: Don’t repeat yourself
- Encapsulate what varies
- Solid
  - **S**ingle Responsibility Principle (SRP)
  - **O**pen-Closed Principle (OCP)
  - **L**iskov Substitution Principle (LSP)
  - **I**nterface Segregation Principle (ISP)
  - **D**ependency Inversion Principle (DIP)



Keep it simple

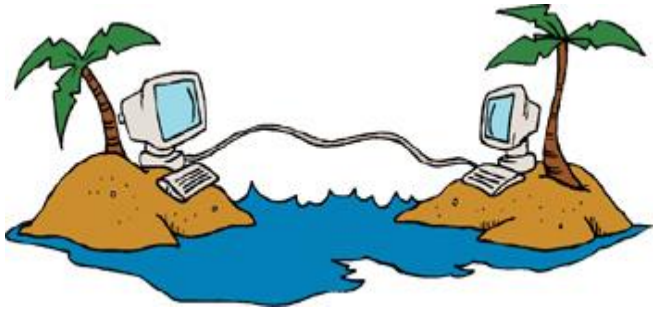


Keep it flexible

- Everthing changes
  - Business
  - Technical
- More flexibility leads to more complexity

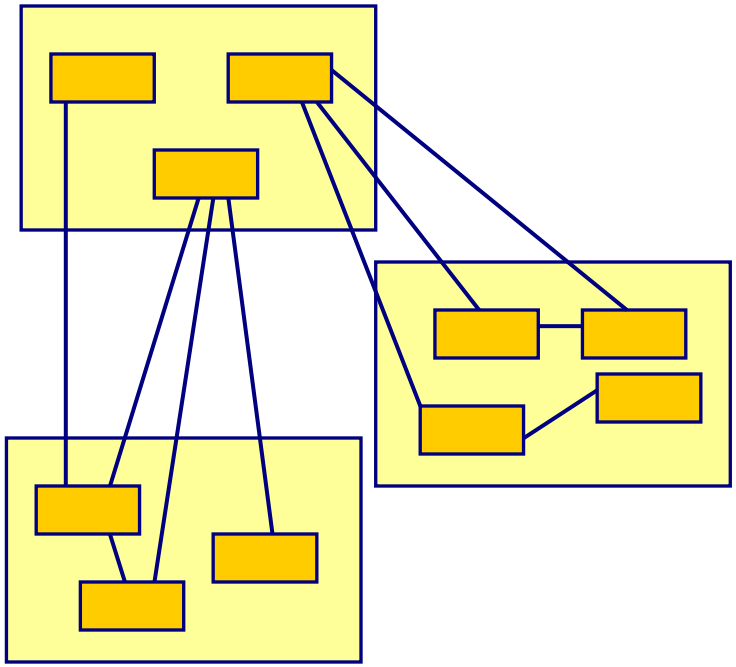
Loose coupling

- Different levels of coupling
  - Technology
  - Time
  - Location
  - Data structure
- You need coupling somewhere
  - Important is the level of coupling

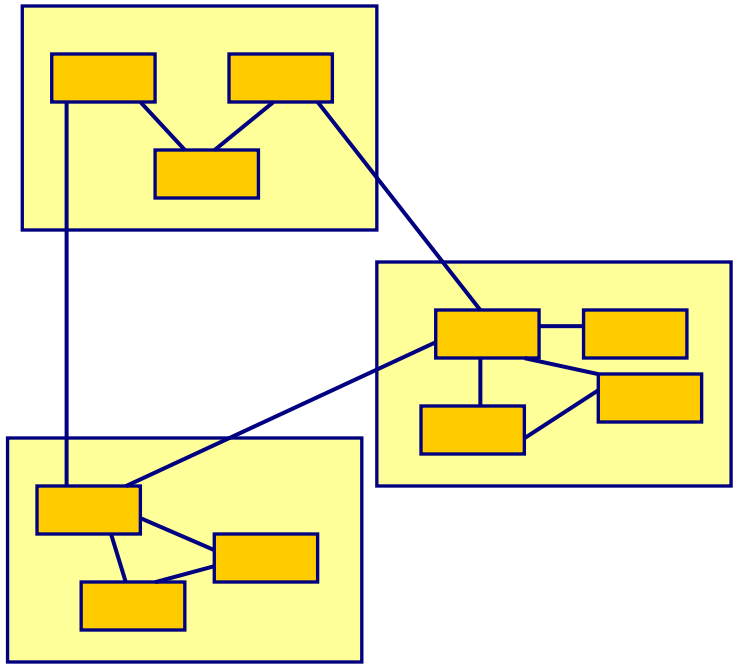


High cohesion, low coupling

- High coupling, low cohesion



- High cohesion, low coupling



Separation of concern

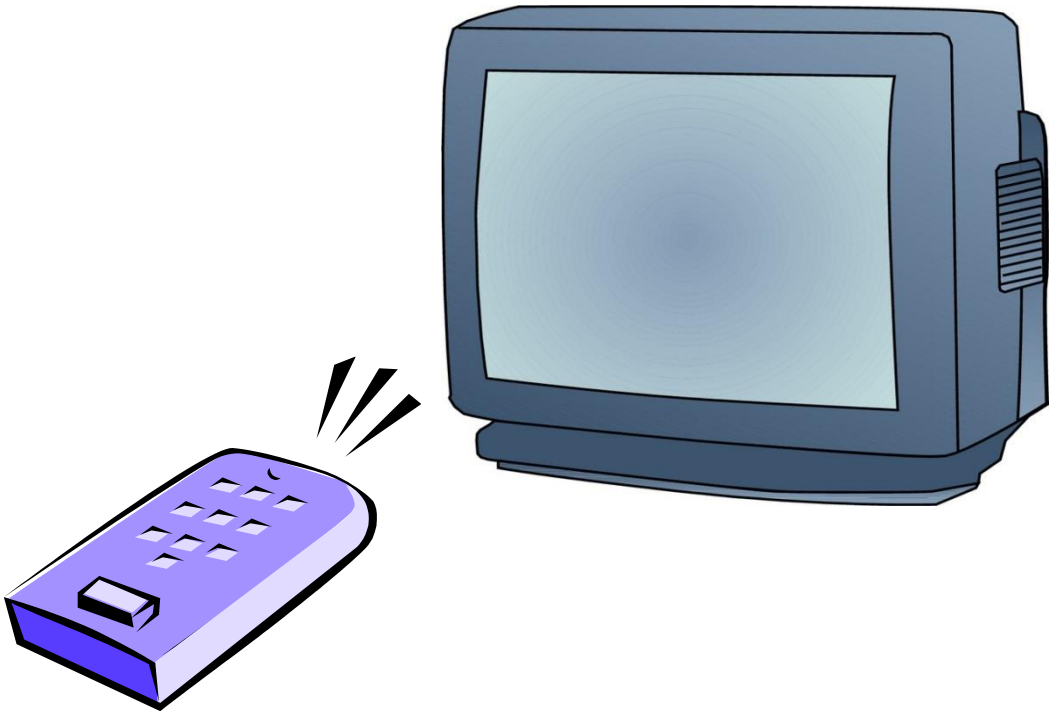
- Separate technology from business
- Separate stable things from changing things
- Separate things that need separate skills
- Separate business process from application logic
- Separate implementation from specification

Information hiding

- Black box principle
- Hide implementation behind an interface

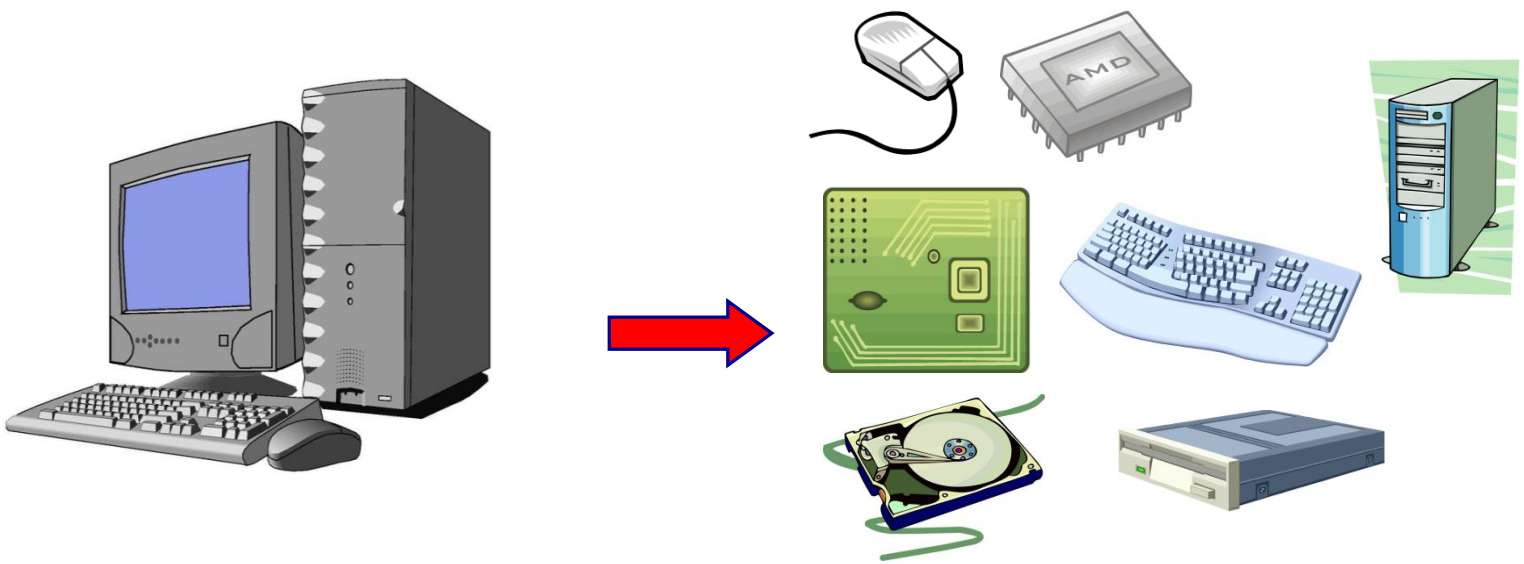
Program to an interface, not an implementation.

- Client app is decoupled from knowing the details of the concrete implementation.



Principle van modularity

- Decomposition
- Devide a big complex problem is smaller parts
- Use components that are
  - Better understandable
  - Independent
  - Reusable
- Leads to more flexibility
- Makes finding and solvings bugs easier

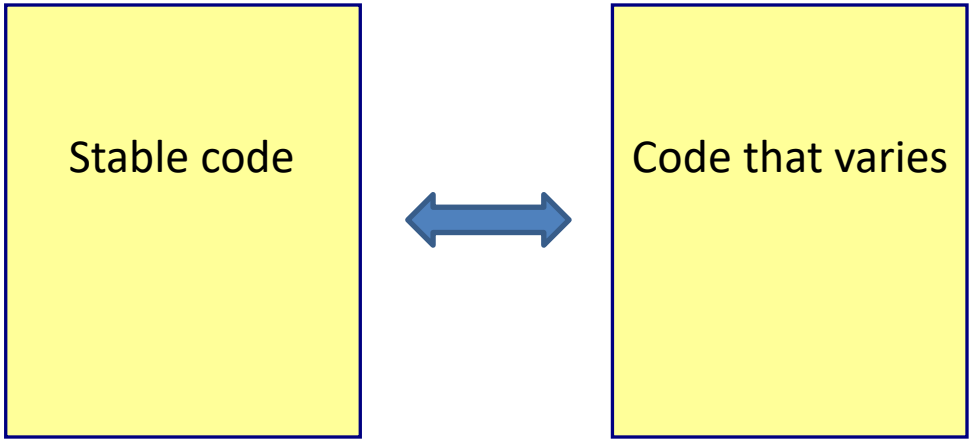


DRY: Don’t Repeat Yourself

- Write functionality at one place, and only at one place
- Avoid code scattering

Encapsulate what varies

- Take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting the parts that don't.

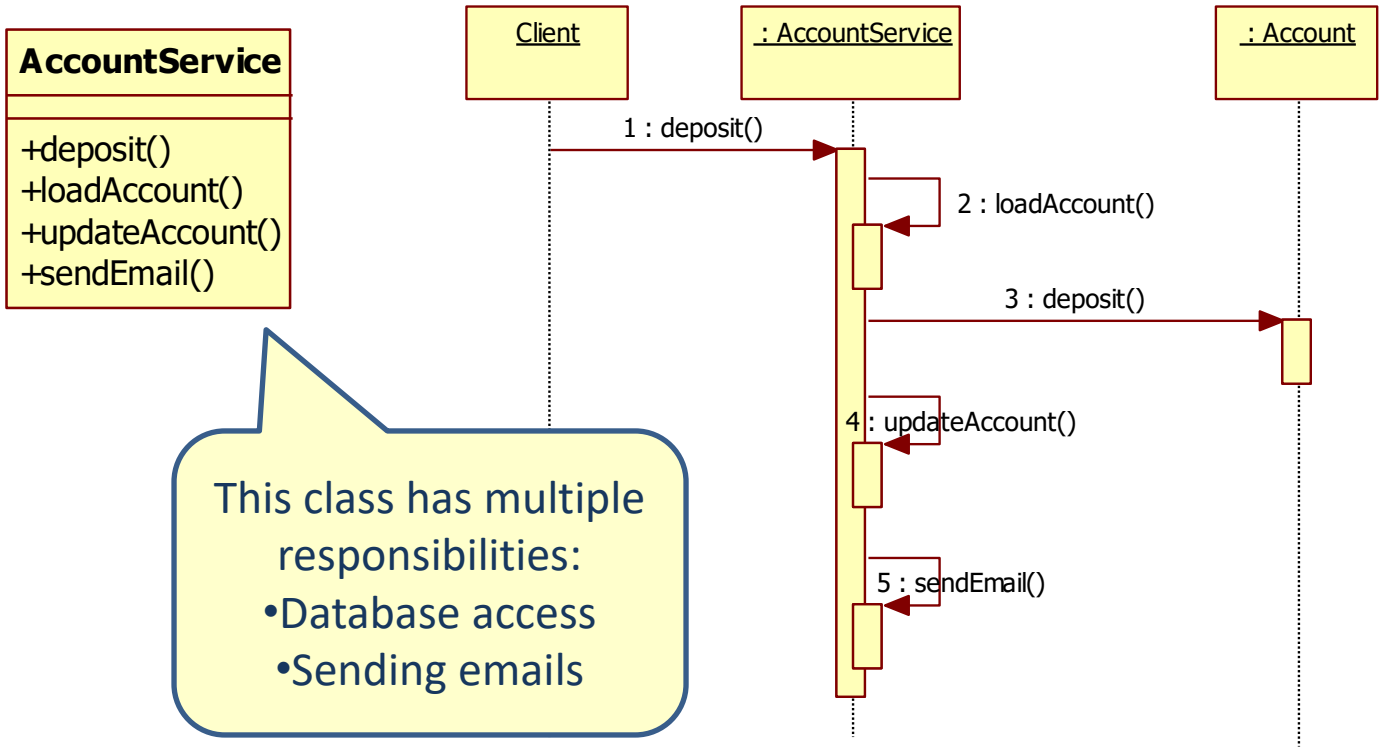


SOLID

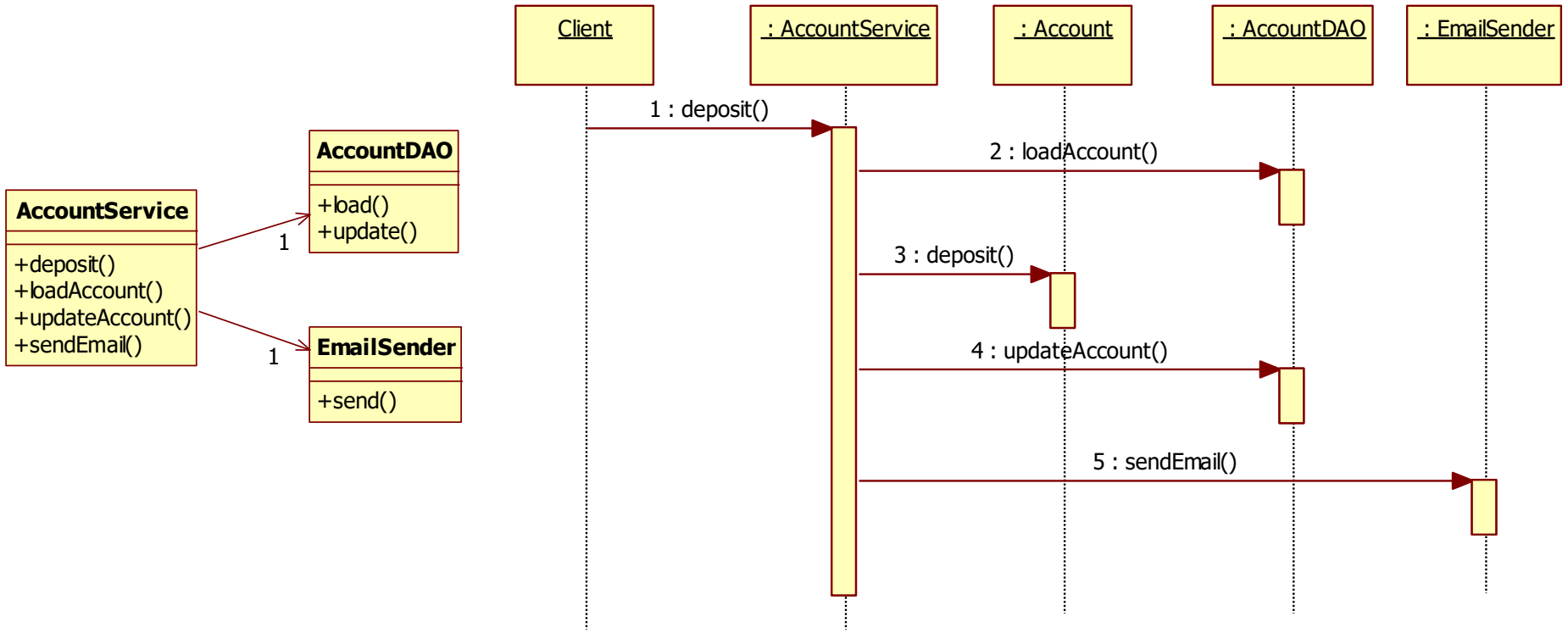
- **S**ingle Responsibility Principle (SRP)
- **O**pen-Closed Principle (OCP)
- **L**iskov Substitution Principle (LSP)
- **I**nterface Segregation Principle (ISP)
- **D**ependency Inversion Principle (DIP)

Single Responsibility Principle

- A class has only one responsibility
  - There should never be more than one reason for a class to change.

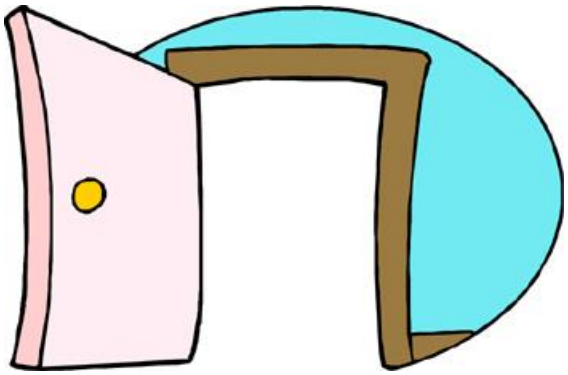


Single Responsibility Principle



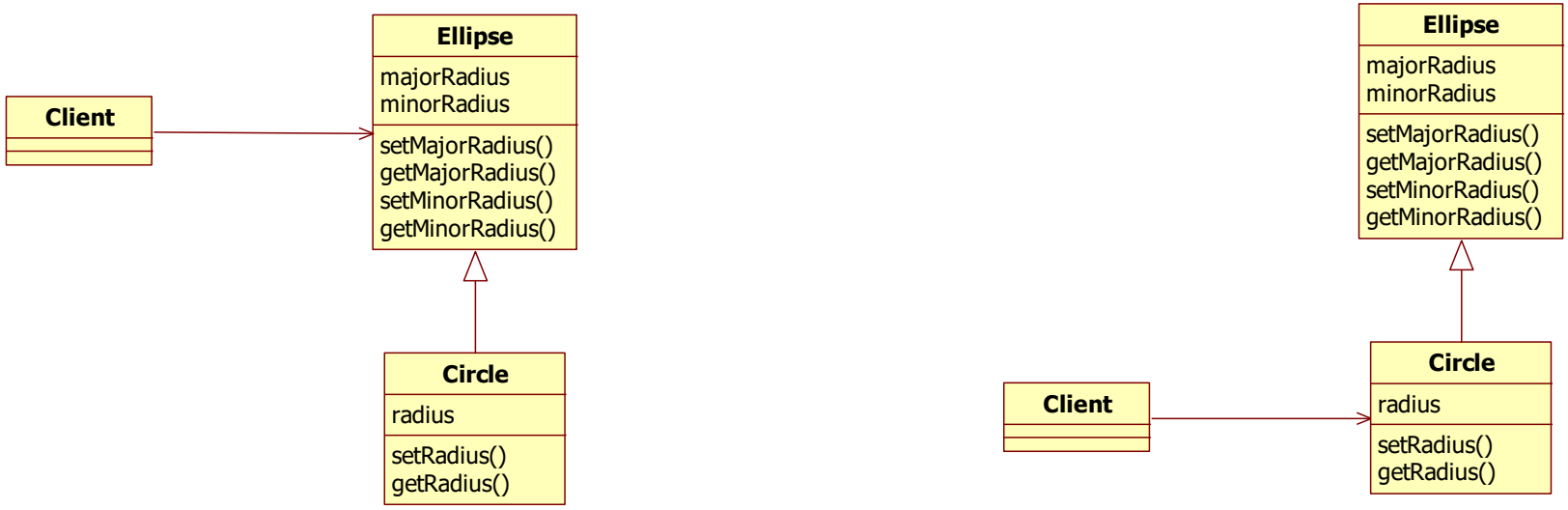
Open-closed principle (OCP)

- Your design should be open for extension, but closed for change
  - We want to add new code as much as possible, and we want to avoid changing working, and tested code

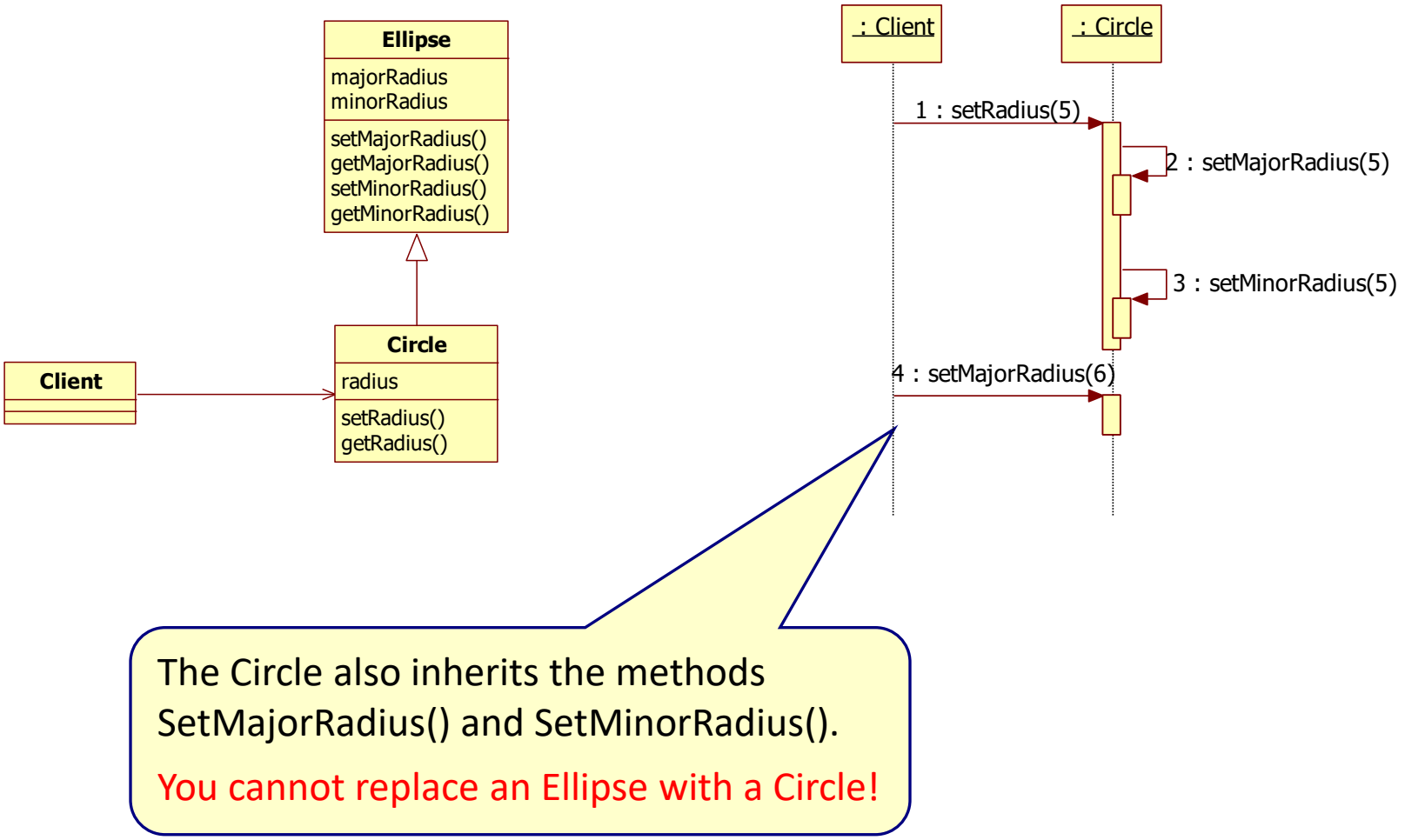


Liskov Substitution Principle

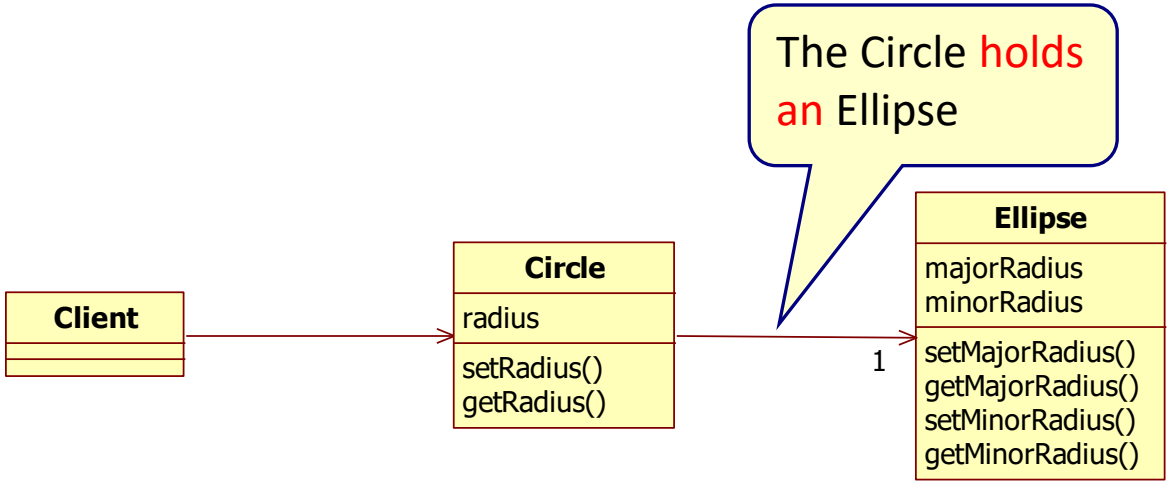
- It should always be possible to substitute a base class for a derived class without any change in behavior.

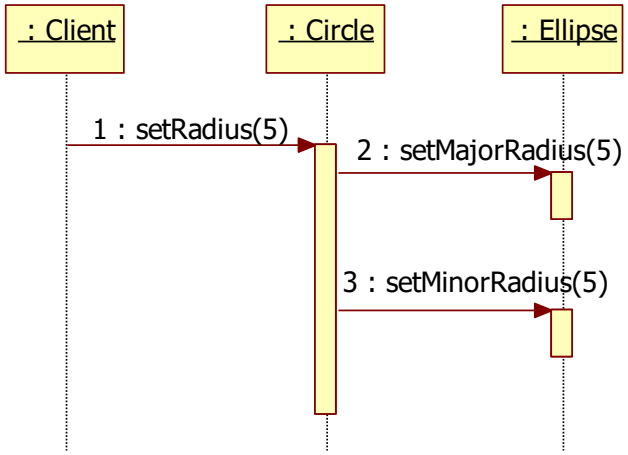


Liskov Substitution Principle Example



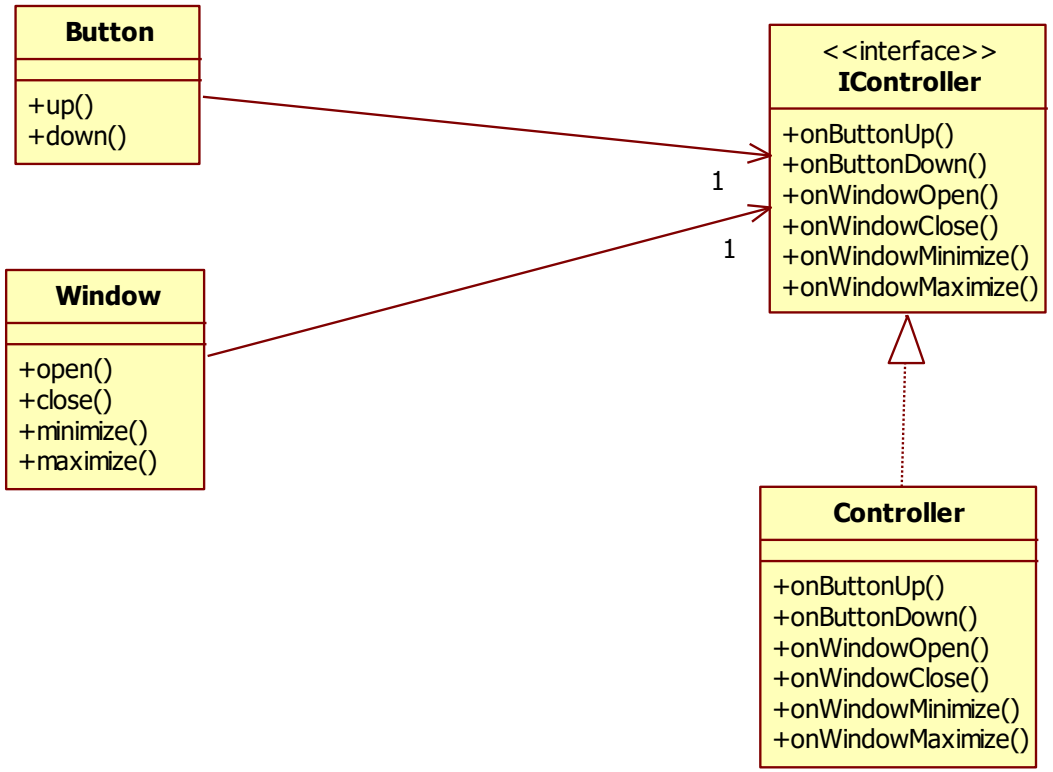
Solution: composition



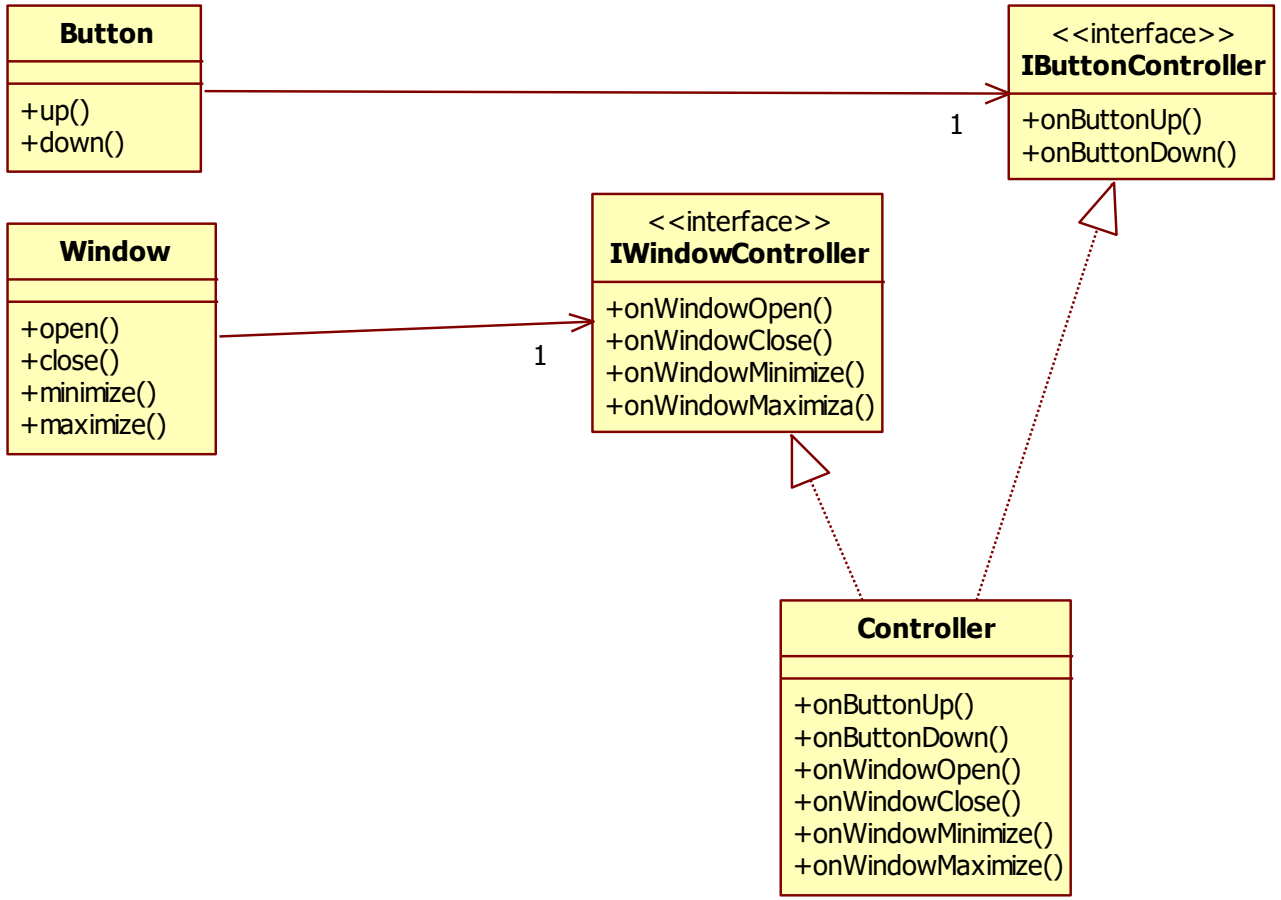


Interface Segregation Principle (ISP)

- Clients should not be forced to depend on methods they do not use

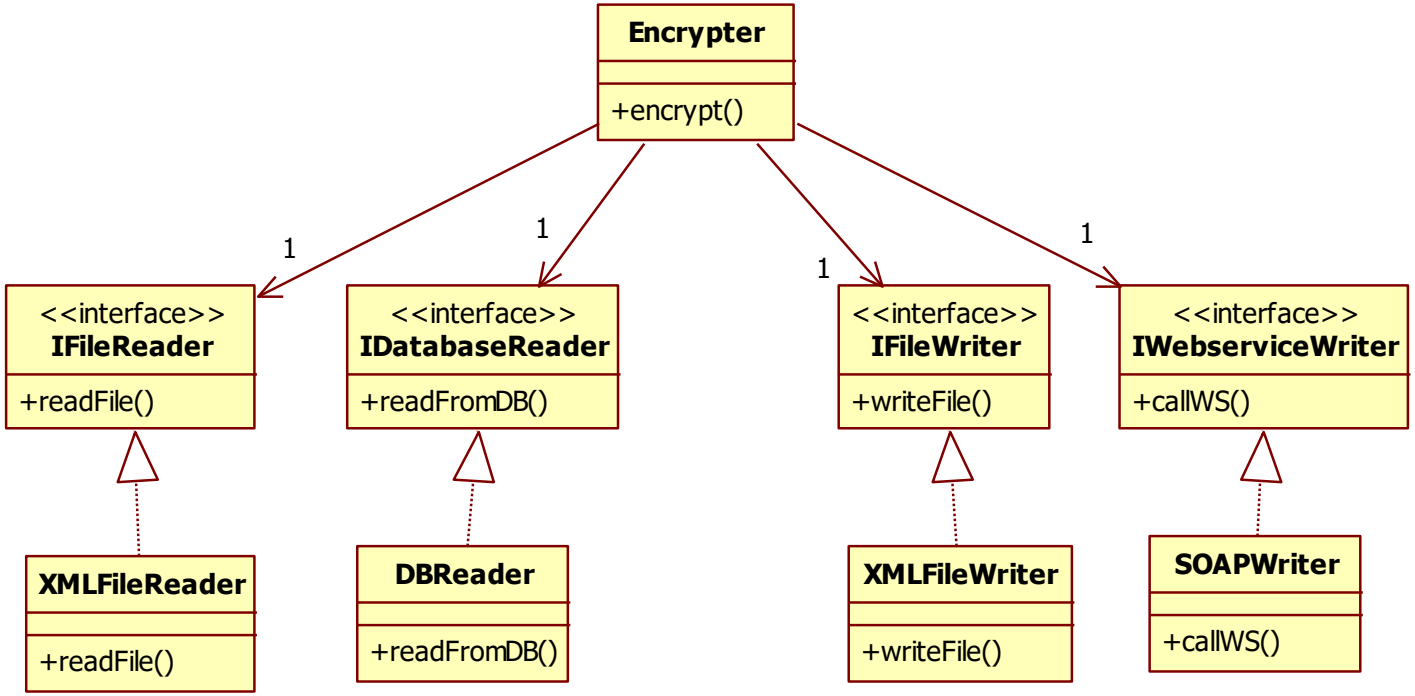


Interface Segregation Principle (ISP)

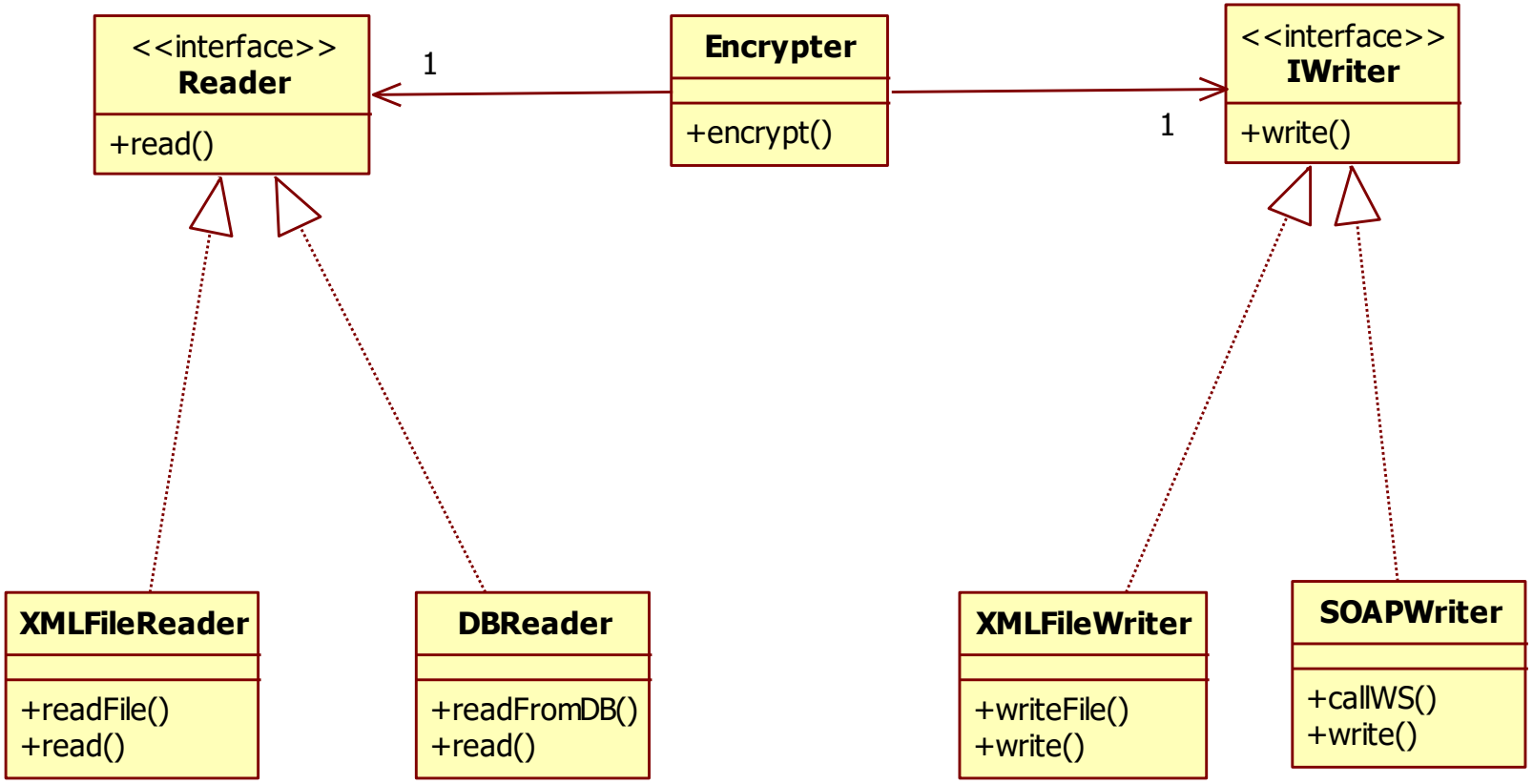


Dependency Inversion Principle (DIP)

- High-level modules should not depend on low-level modules. Both should depend on abstractions



Dependency Inversion Principle (DIP)



Design principles

- Keep it simple
- Keep it flexible
- Loose coupling
- Separation of concern
- Information hiding
- Principle of modularity
- DRY: Don’t repeat yourself
- Encapsulate what varies
- Solid
  - **S**ingle Responsibility Principle (SRP)
  - **O**pen-Closed Principle (OCP)
  - **L**iskov Substitution Principle (LSP)
  - **I**nterface Segregation Principle (ISP)
  - **D**ependency Inversion Principle (DIP)

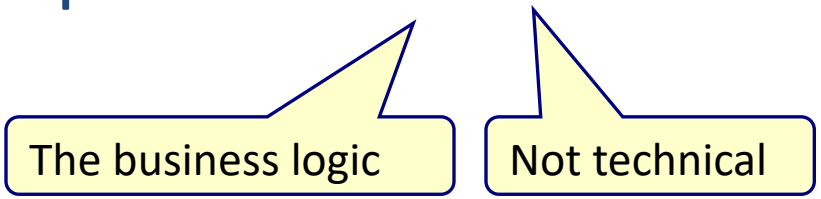
Main point

- A good designed system is often simple and easy to modify.
- The unified field is the underlying field at the basis of all relative creation.

OO DOMAIN MODELING

Domain modeling

- Create an OO model of the problem domain



Application layers

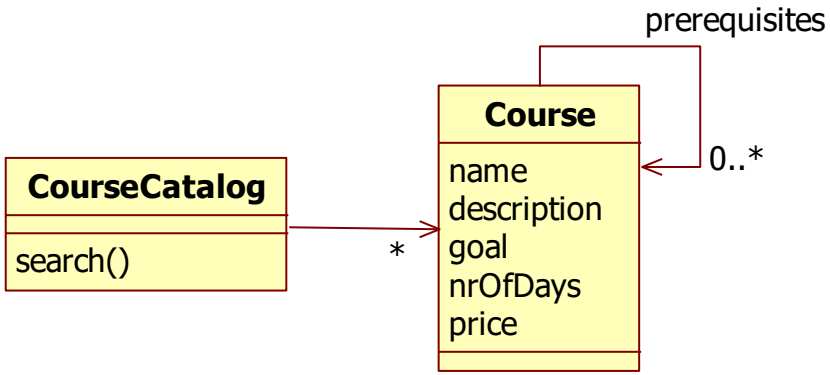
Presentation layer	Service layer	Domain/Business logic layer	Integration layer	Data access layer
		Separate the business logic from the technical code		

Course registration system

- The system should allow users to browse through the course catalog
- The system should allow us to search for courses in the course catalog
- For every course we need to record its name, a description, the goal of the course, one or more prerequisite courses, number of days and the price of the course.
- Courses are given a few times per year
- A course starts at a certain date.
- Students can subscribe to one or more courses.
- The system should record the student name, phone, email and street, city and zip of the student.
- A course takes at least one day, but can take any number of days
- Courses can be given on consecutive days, but courses can also be given on non-consecutive days. For example a 4 day course can be given on 4 Mondays in September.
- A course can be given by one instructor, but every single day can also be given by different instructors. For example, the first 2 days of a 4day course is taught by Bob and the last 2 days are taught by Mary.
- The system should record the instructors name, phone, email and street, city and zip of the instructor.
- A course can be given at one location, but every single day can also be given at a different location. For example, the first 2 days of a 4day course is given at a different location as the last 2 days.
- The system should record the locations name and street, city and zip of the location.
- The system should record all important information about a certain course:
  - Dates that this course is given
  - Instructor(s) for the different course dates
  - Location of the different course dates

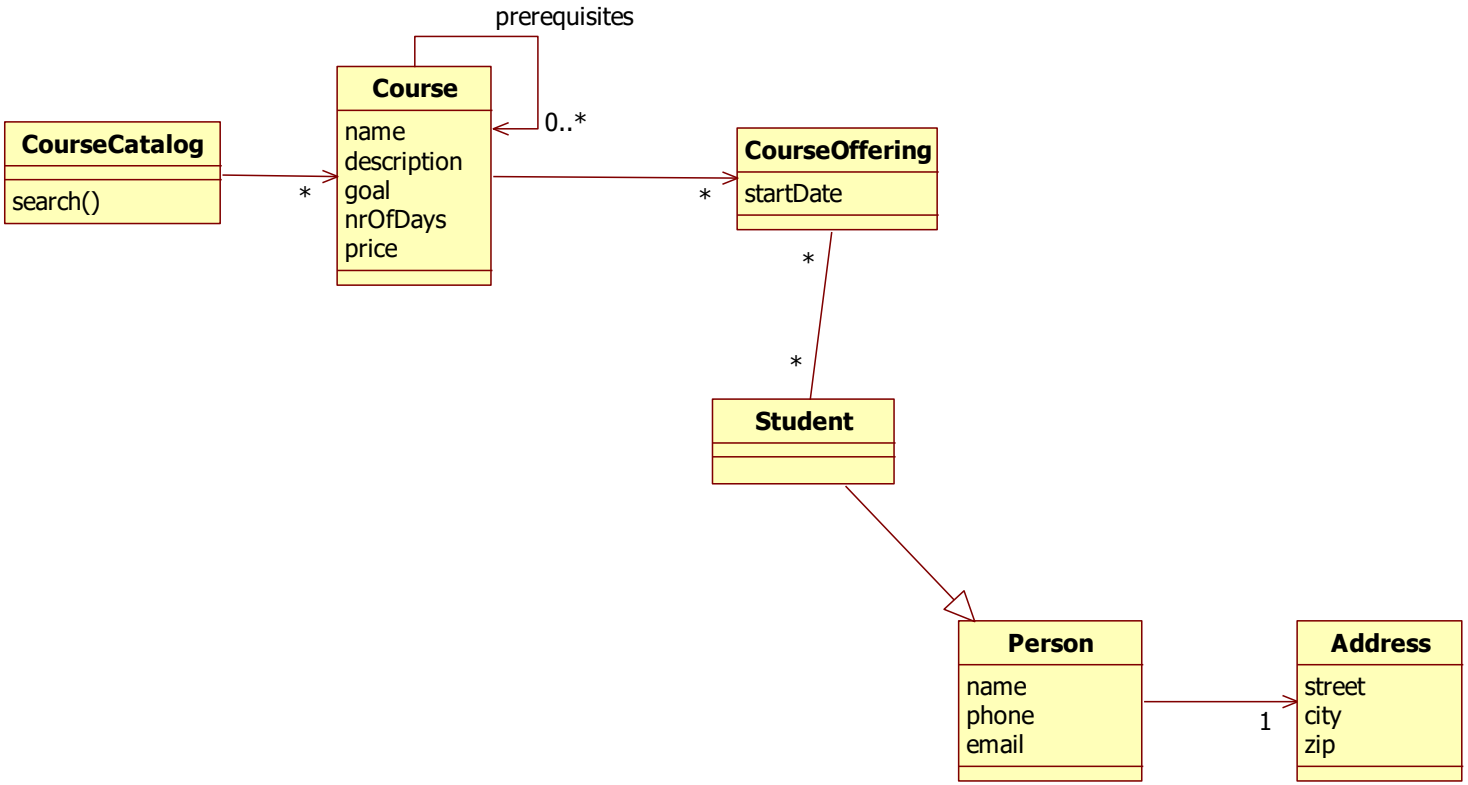
Course registration system

- The system should allow users to browse through the course catalog
- The system should allow users to search for courses in the course catalog
- For every course we need to record its name, a description, the goal of the course, one or more prerequisite courses, number of days and the price of the course.



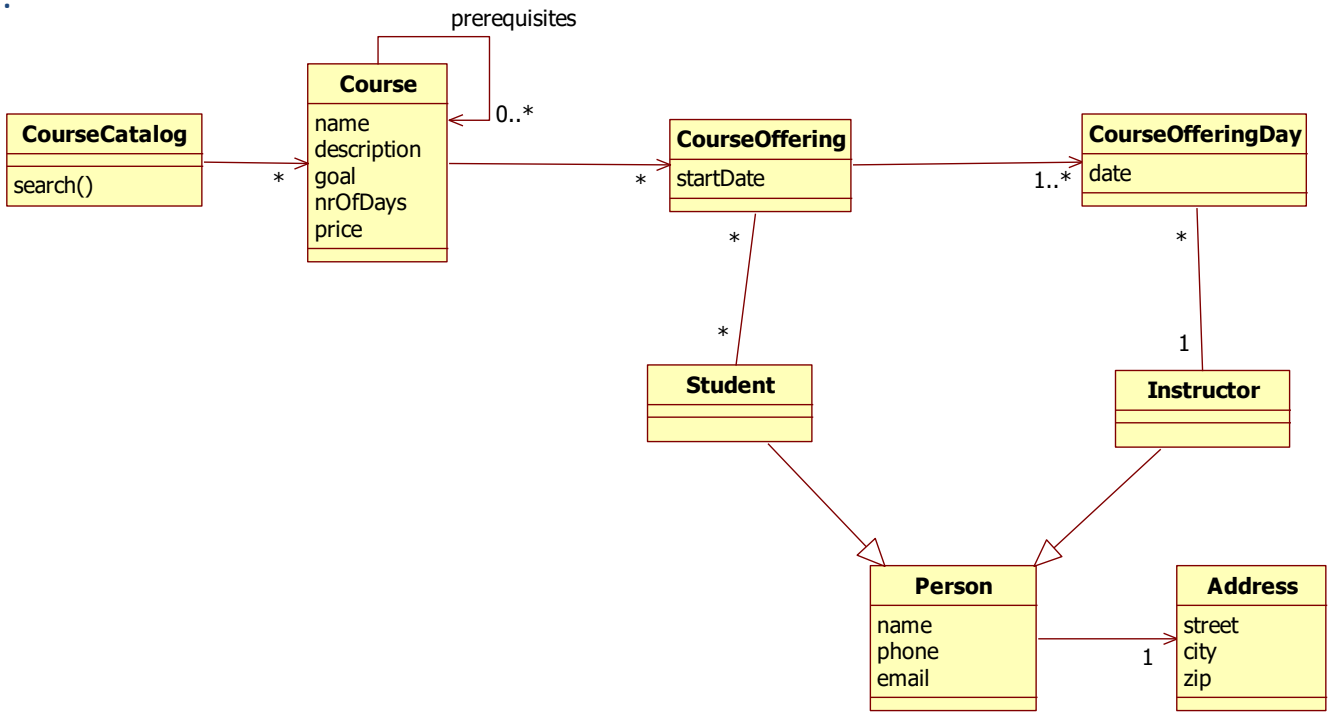
Course registration system

- Courses are given a few times per year
- A course starts at a certain date.
- Students can subscribe to one or more courses.
- The system should record the student name, phone, email and street, city and zip of the student.



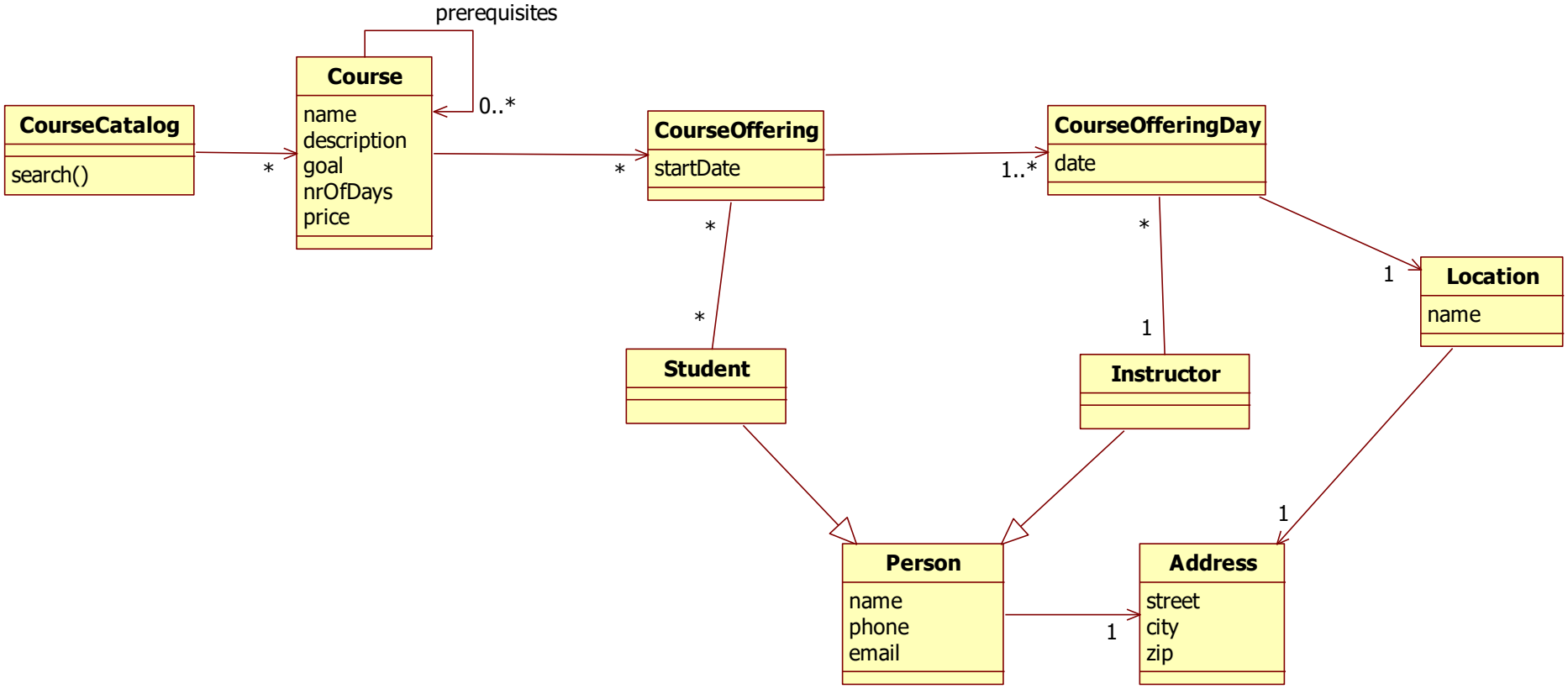
Course registration system

- A course takes at least one day, but can take any number of days
- Courses can be given on consecutive days, but courses can also be given on non-consecutive days. For example a 4 day course can be given on 4 Mondays in September.
- A course can be given by one instructor, but every single day can also be given by different instructors. For example, the first 2 days of a 4day course is taught by Bob and the last 2 days are taught by Mary.
- The system should record the instructors name, phone, email and street, city and zip of the instructor.

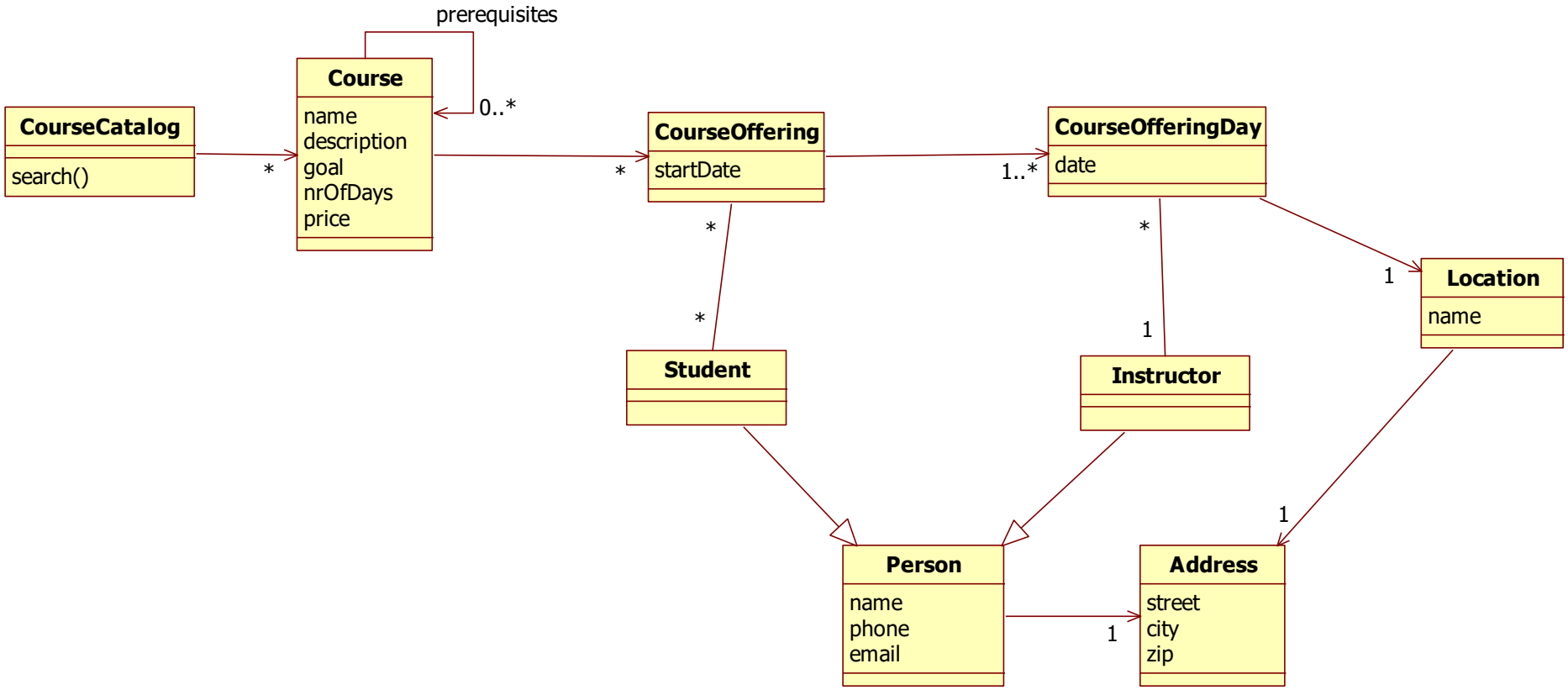


Course registration system

- A course can be given at one location, but every single day can also be given at a different location. For example, the first 2 days of a 4day course is given at a different location as the last 2 days.
- The system should record the locations name and street, city and zip of the location.
- The system should record all important information about a certain course:
  - Dates that this course is given
  - Instructor(s) for the different course dates
  - Location of the different course dates

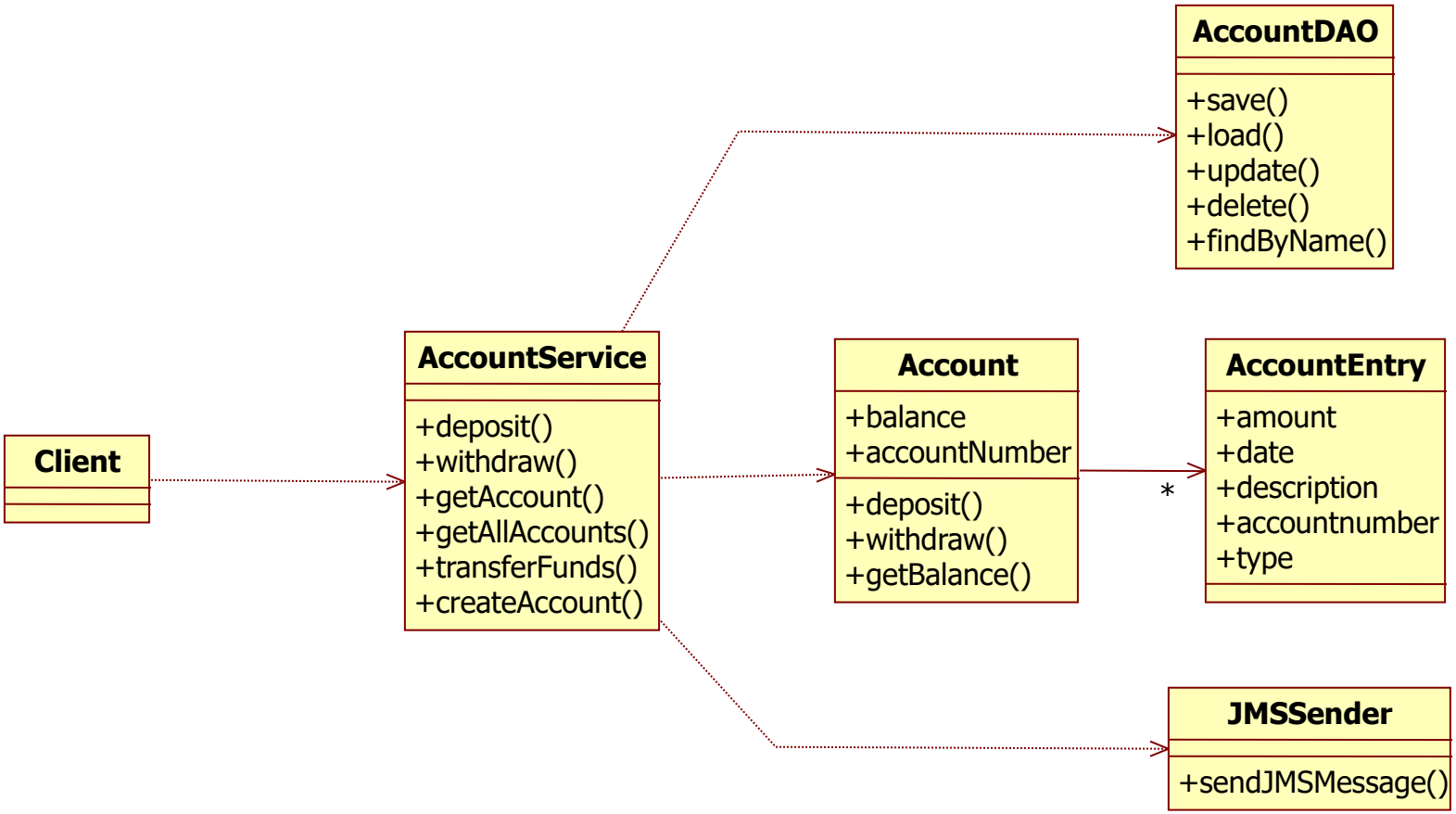


Domain model

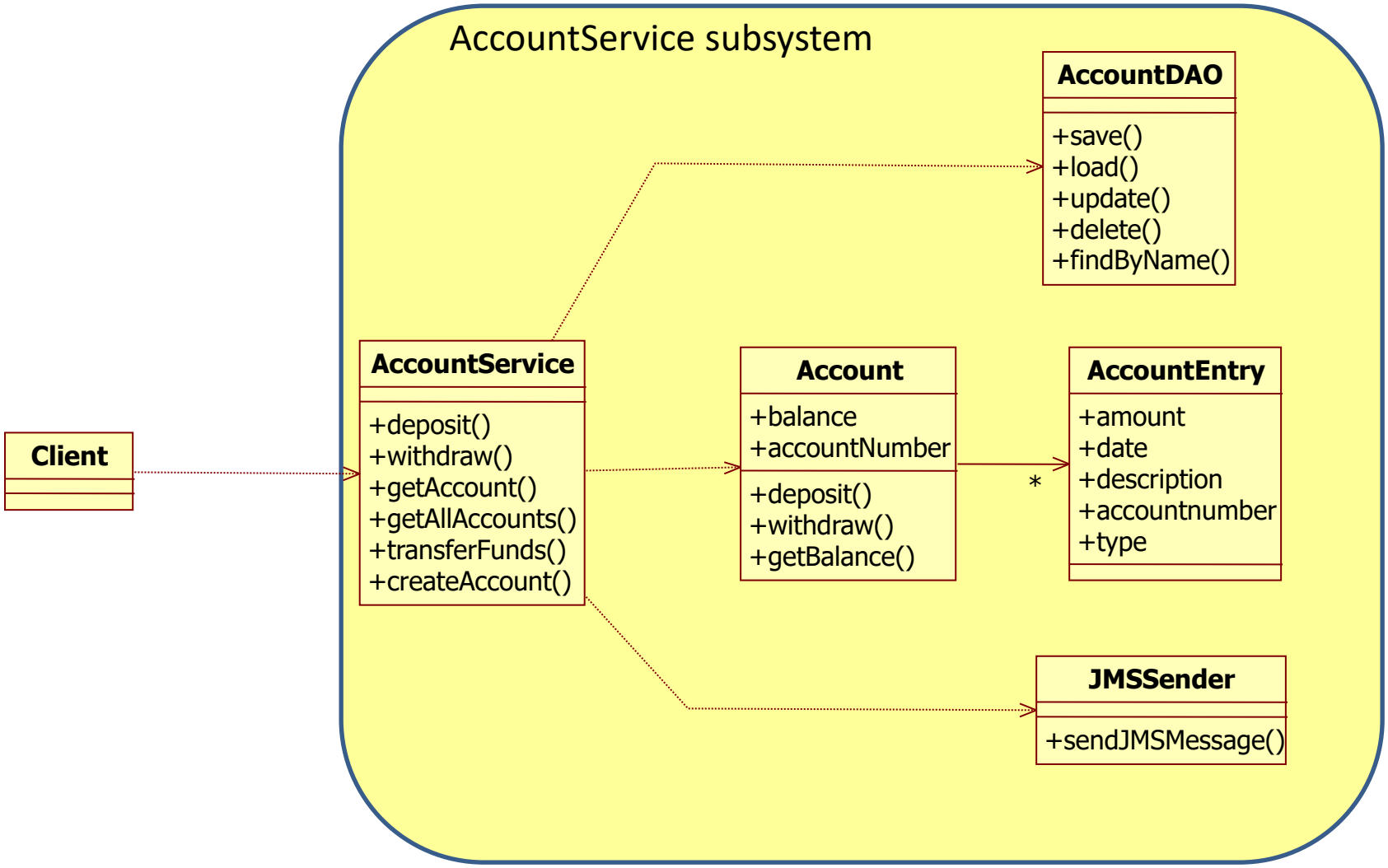


SERVICE CLASS FACADE

Service Class



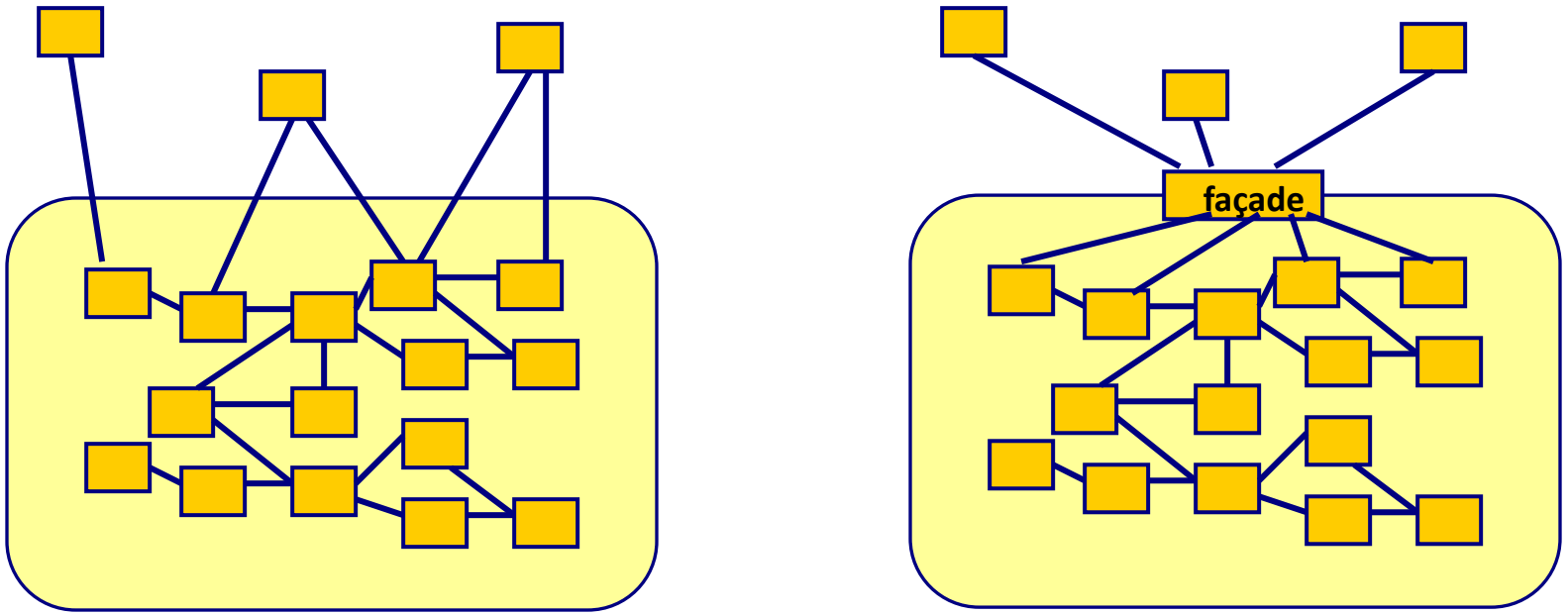
Entry of a complex subsystem



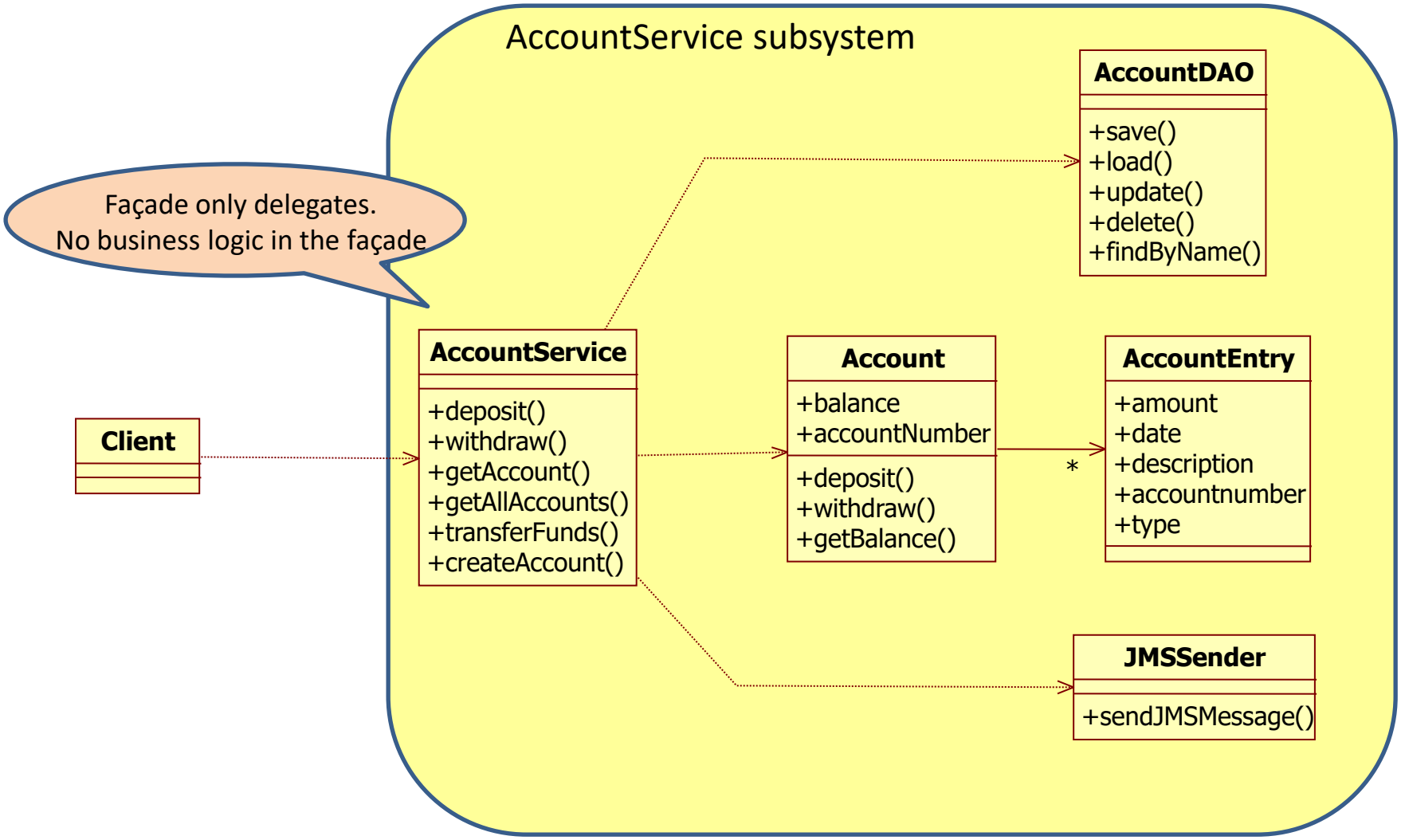
Facade pattern

- Provide a unified interface to a complex set of classes

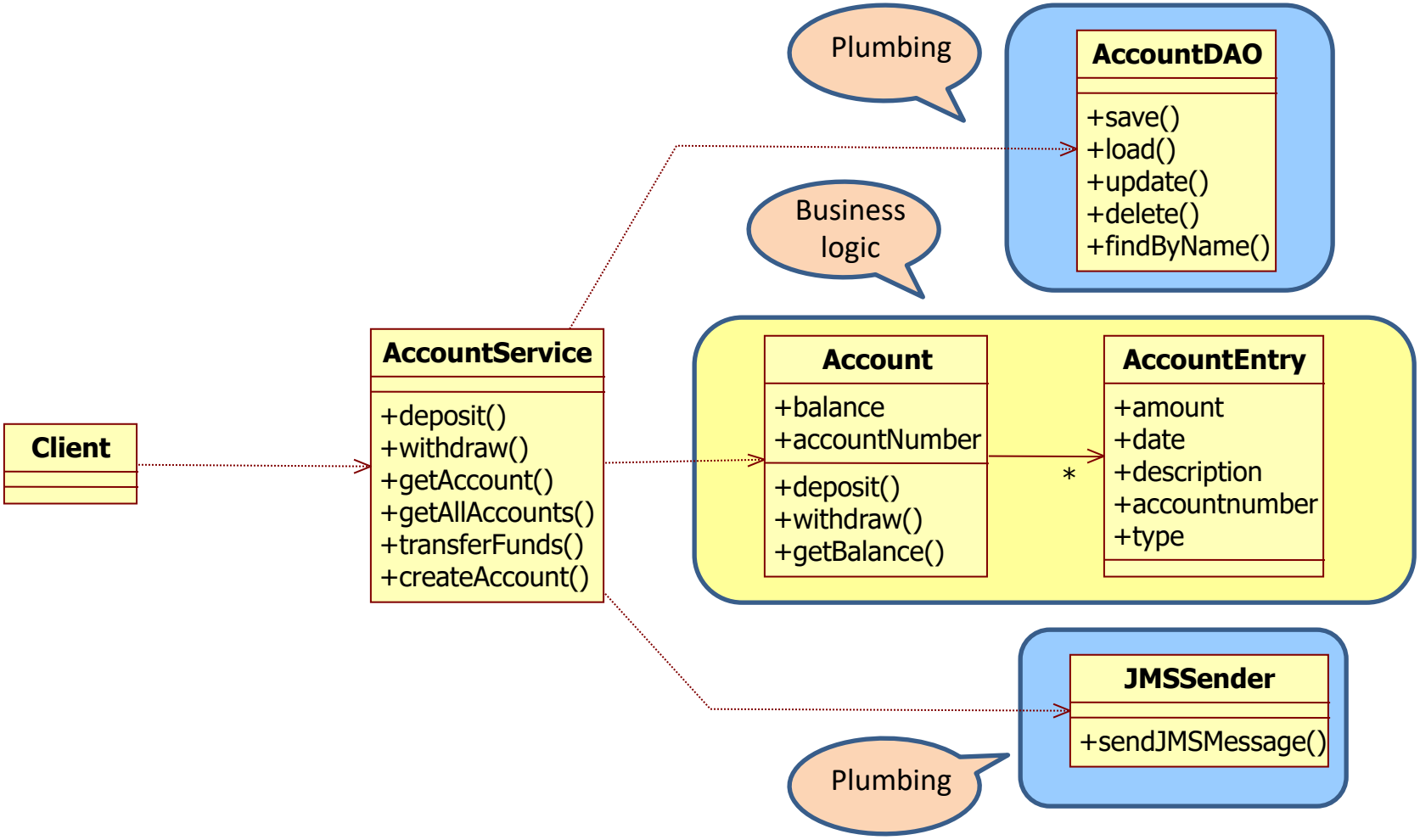
Facade pattern



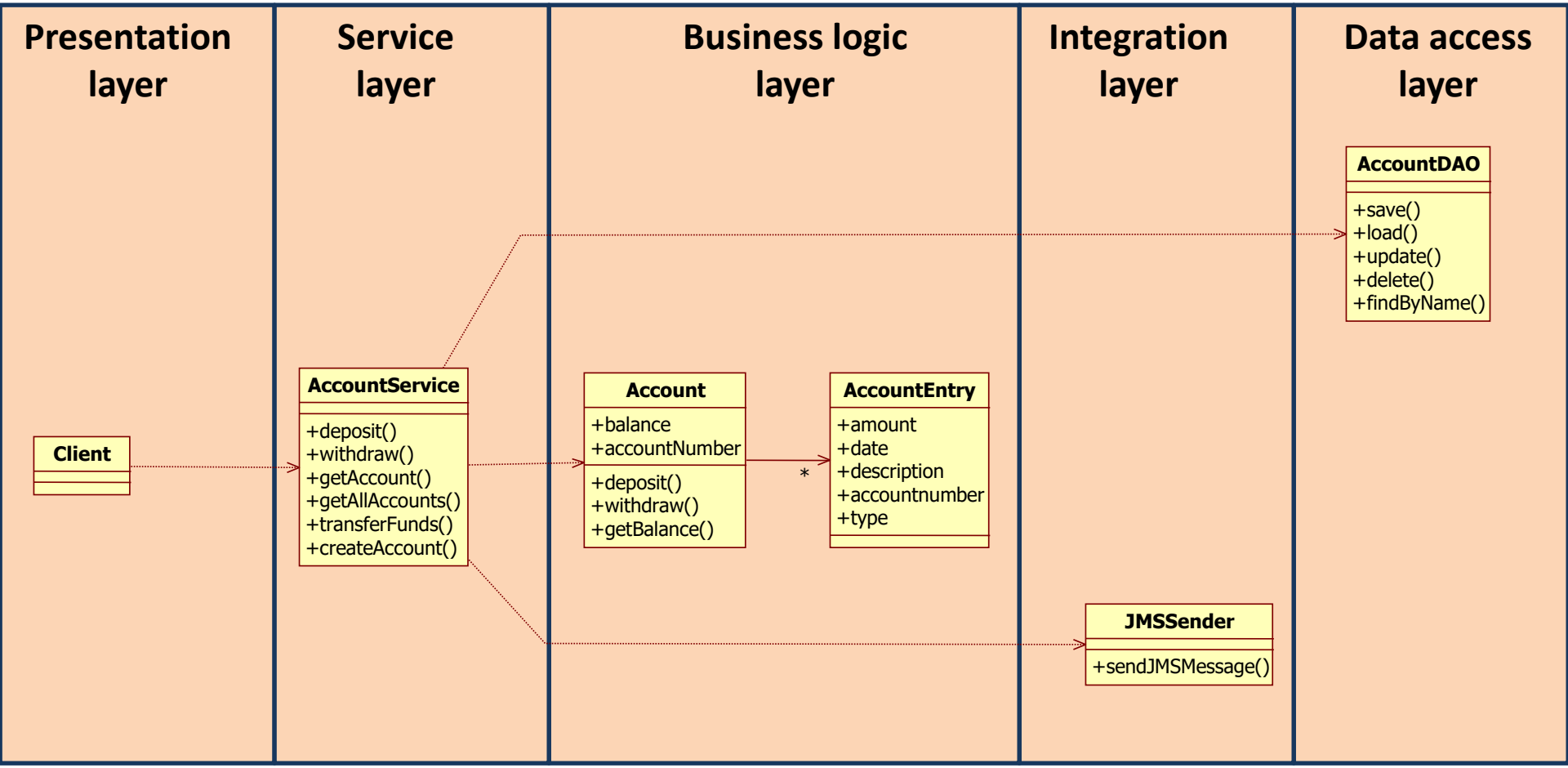
Entry of a complex subsystem



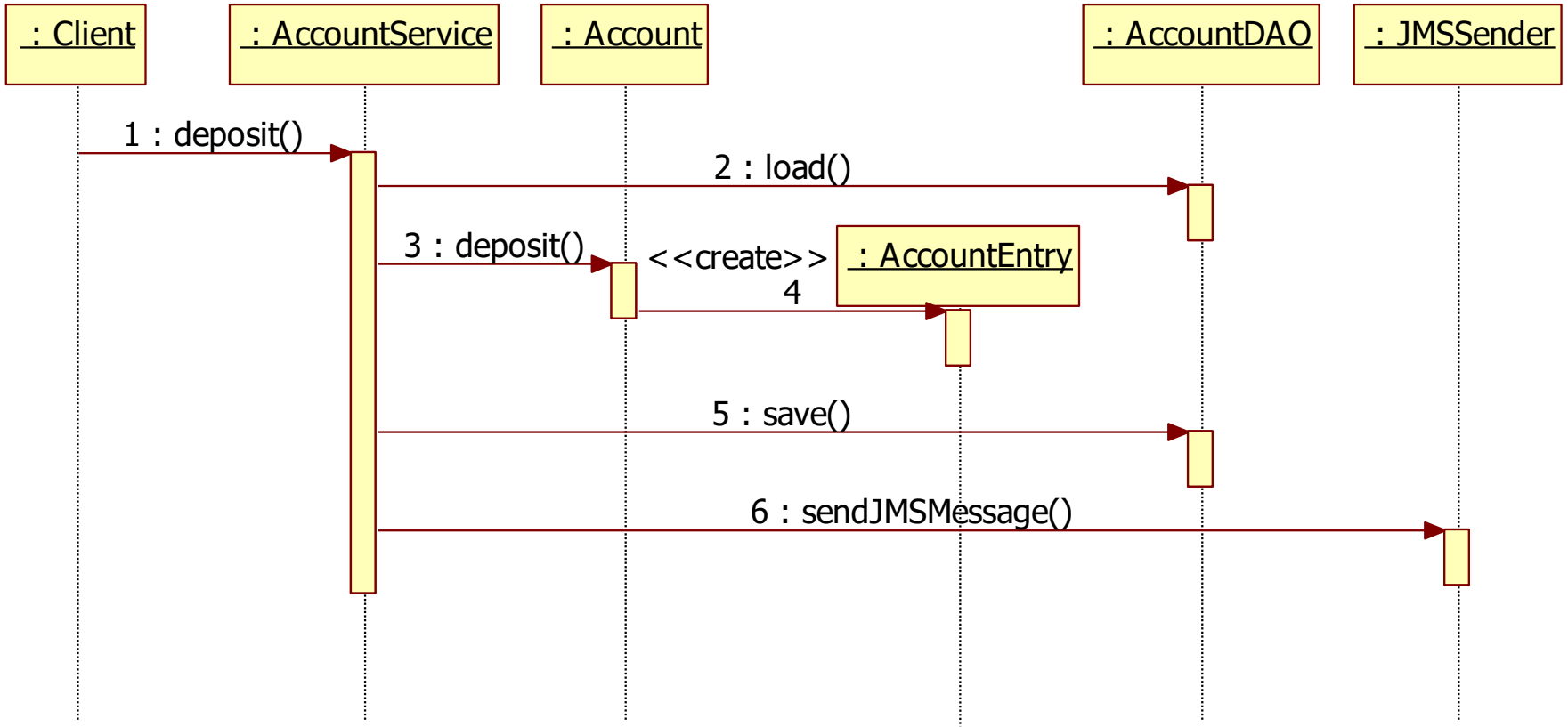
Separation of concern



Application layers



Service object

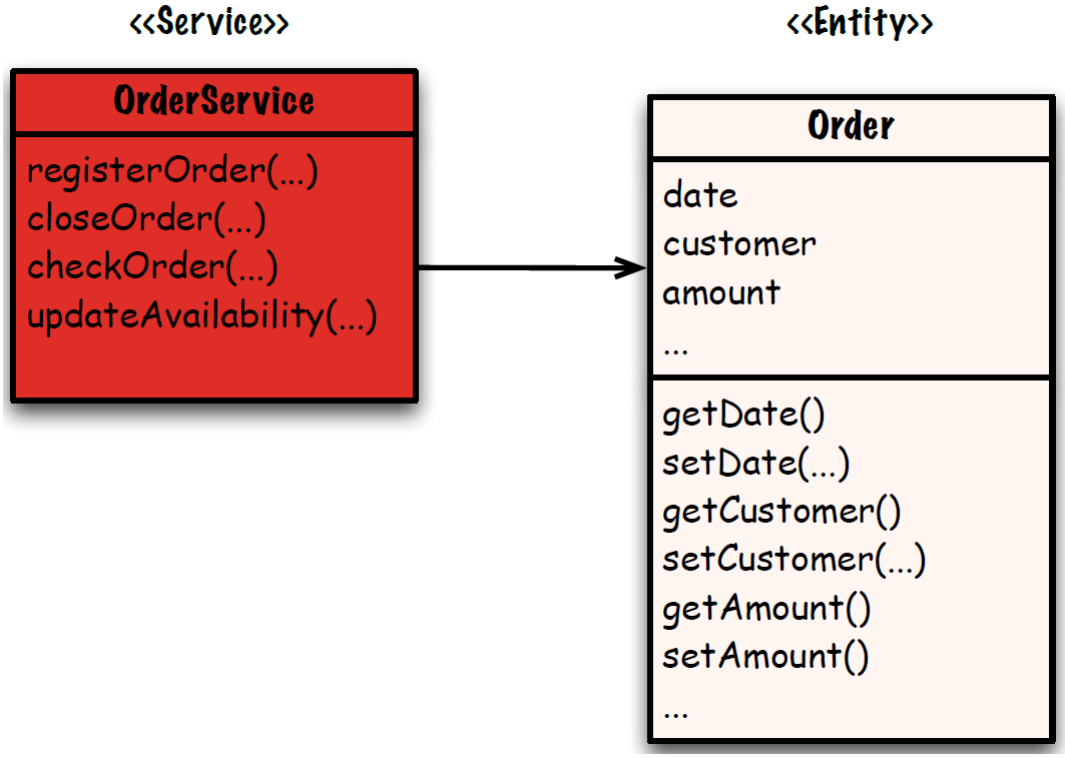


ANEMIC AND RICH DOMAIN MODEL

Anemic domain model

- Classes in the model have no business logic

NOT OK



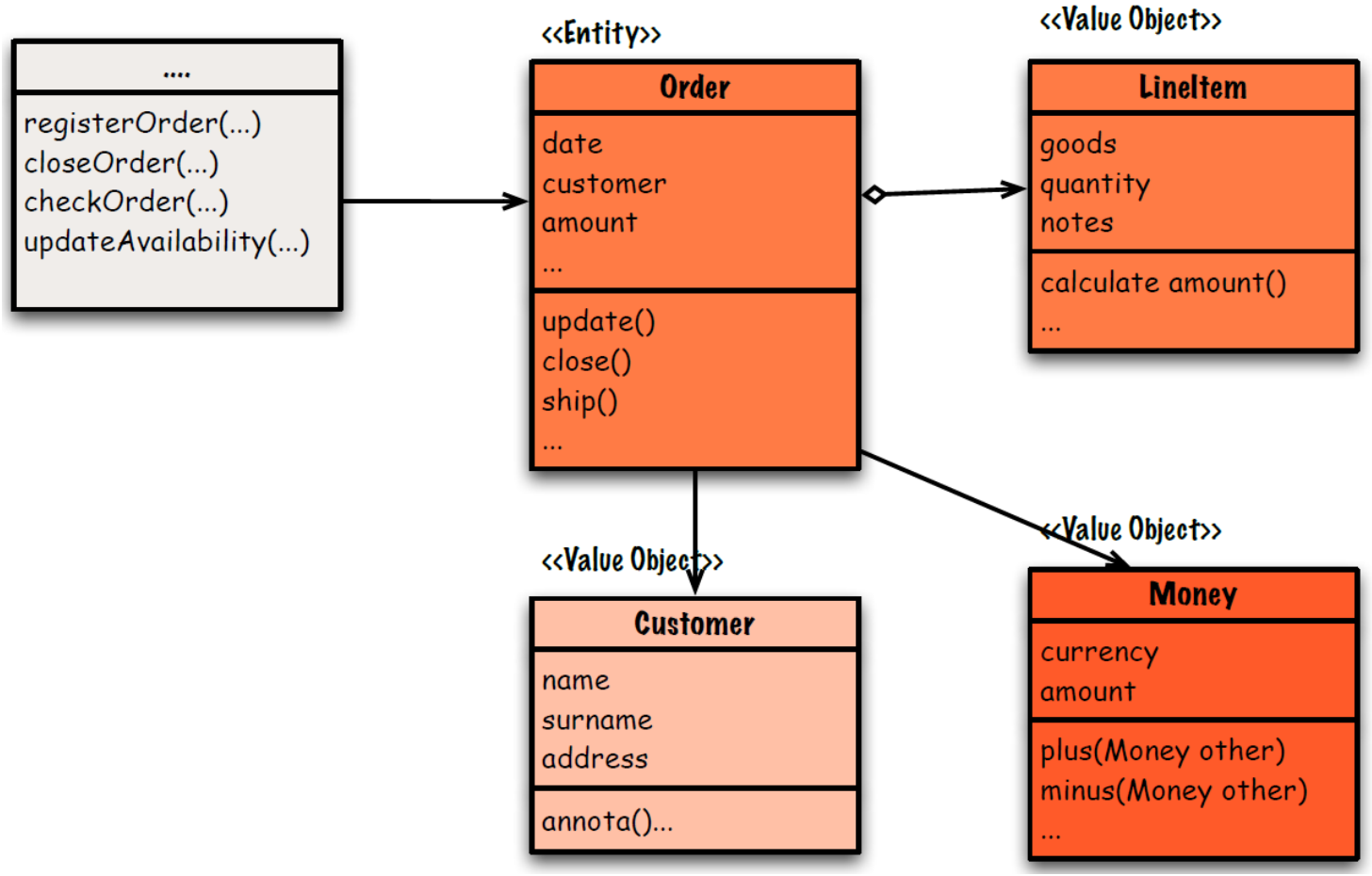
Disadvantages anemic domain model

- You do not use the powerful OO techniques to organize complex logic.
- Business logic (rules) is hard to find, understand, reuse, modify.
- The software reflects the data structure of the business, but not the behavioral organization
- The service classes become too complex
  - No single responsibility
  - No separation of concern

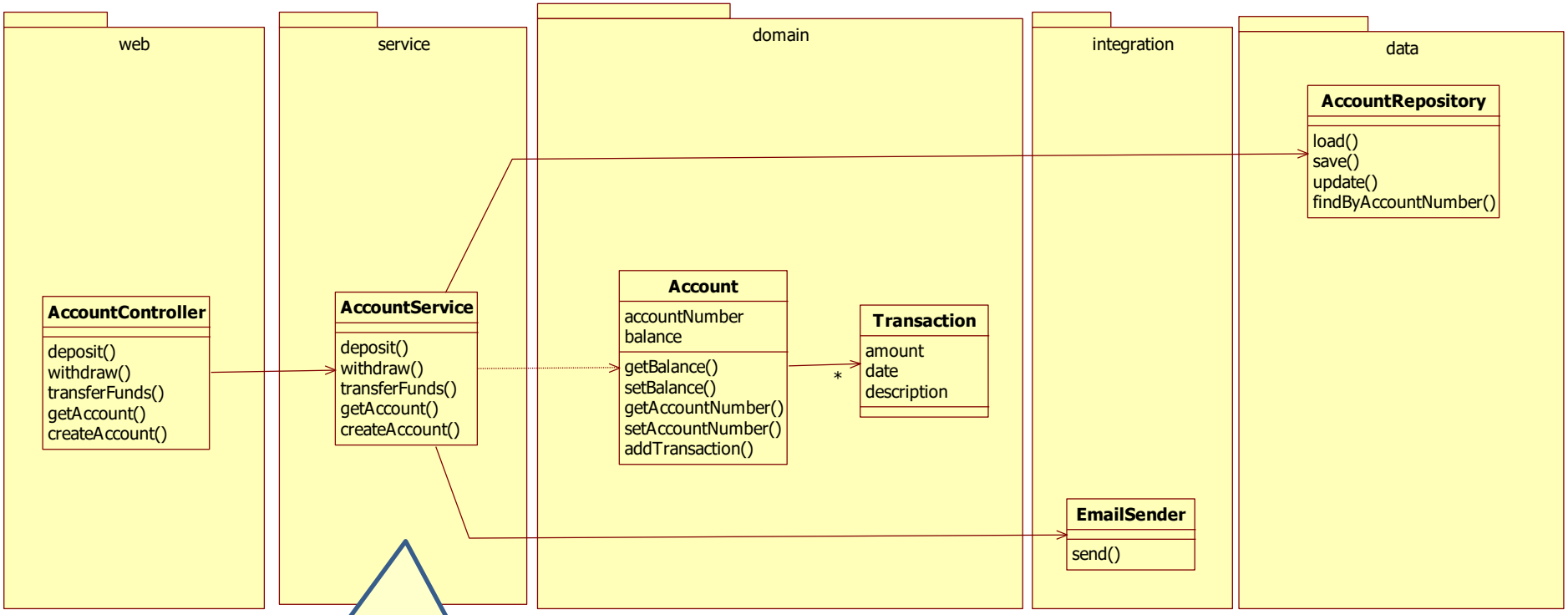
Rich domain model

- Classes with business logic

OK

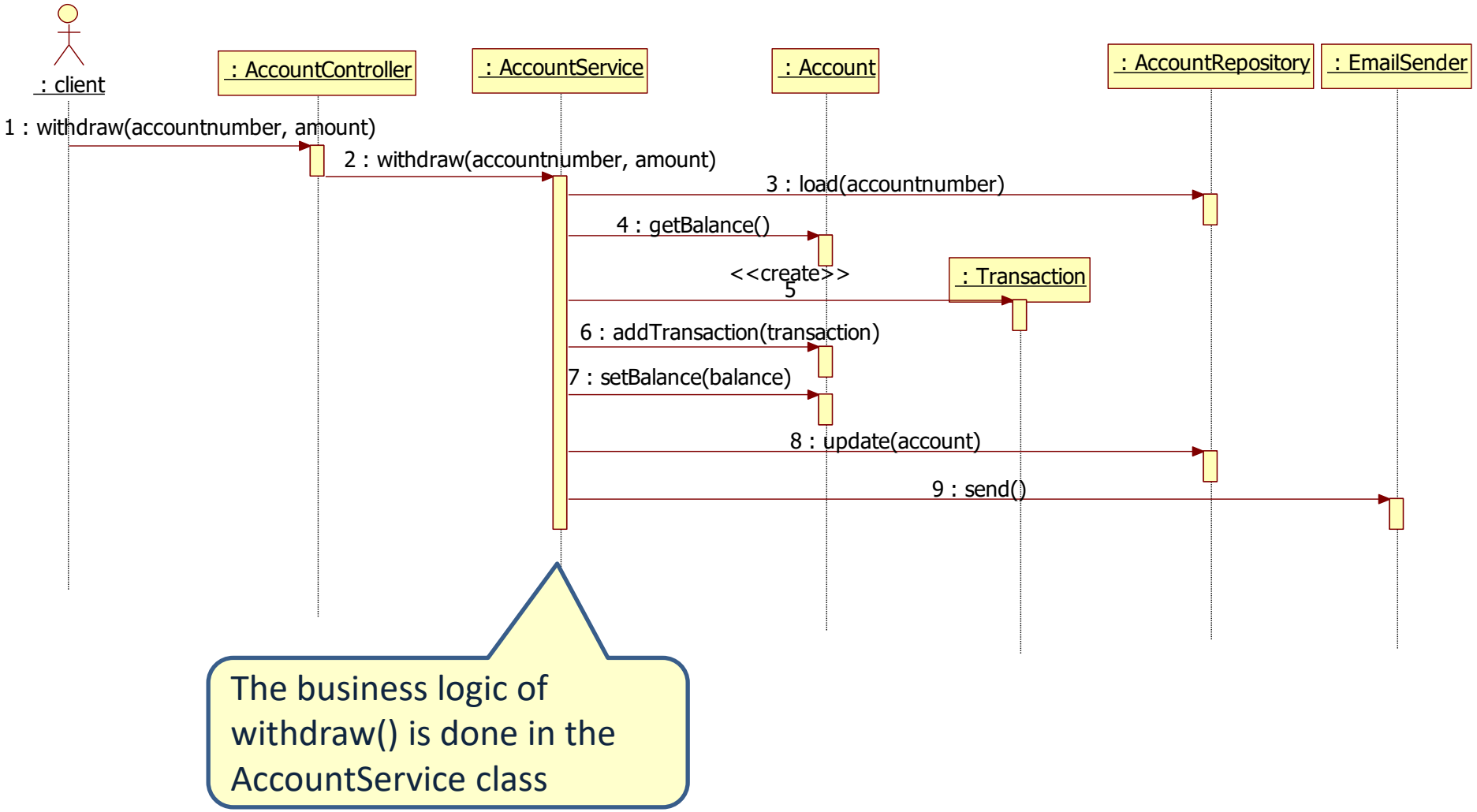


Anemic domain model example

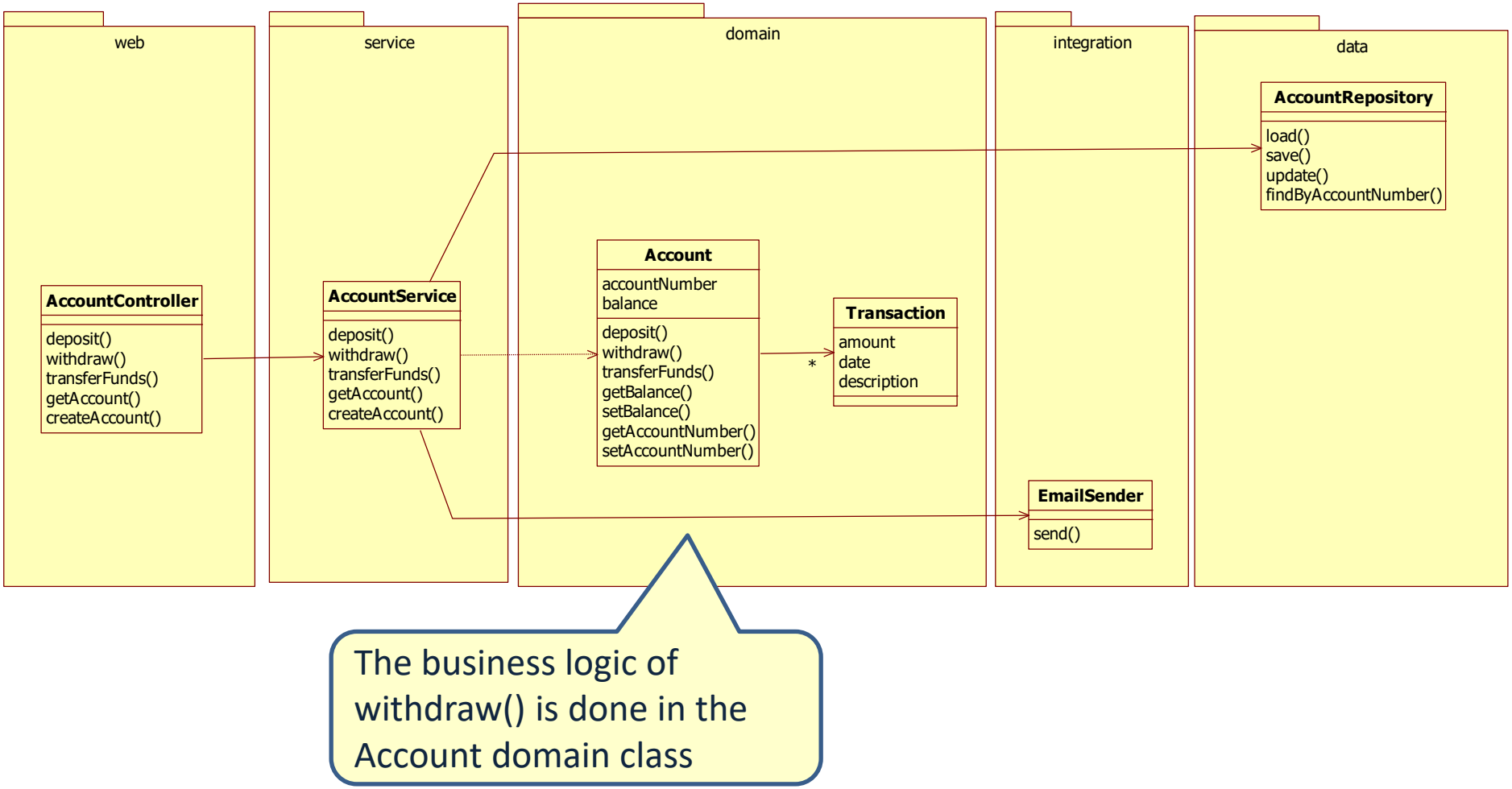


The business logic of withdraw() is done in the AccountService class

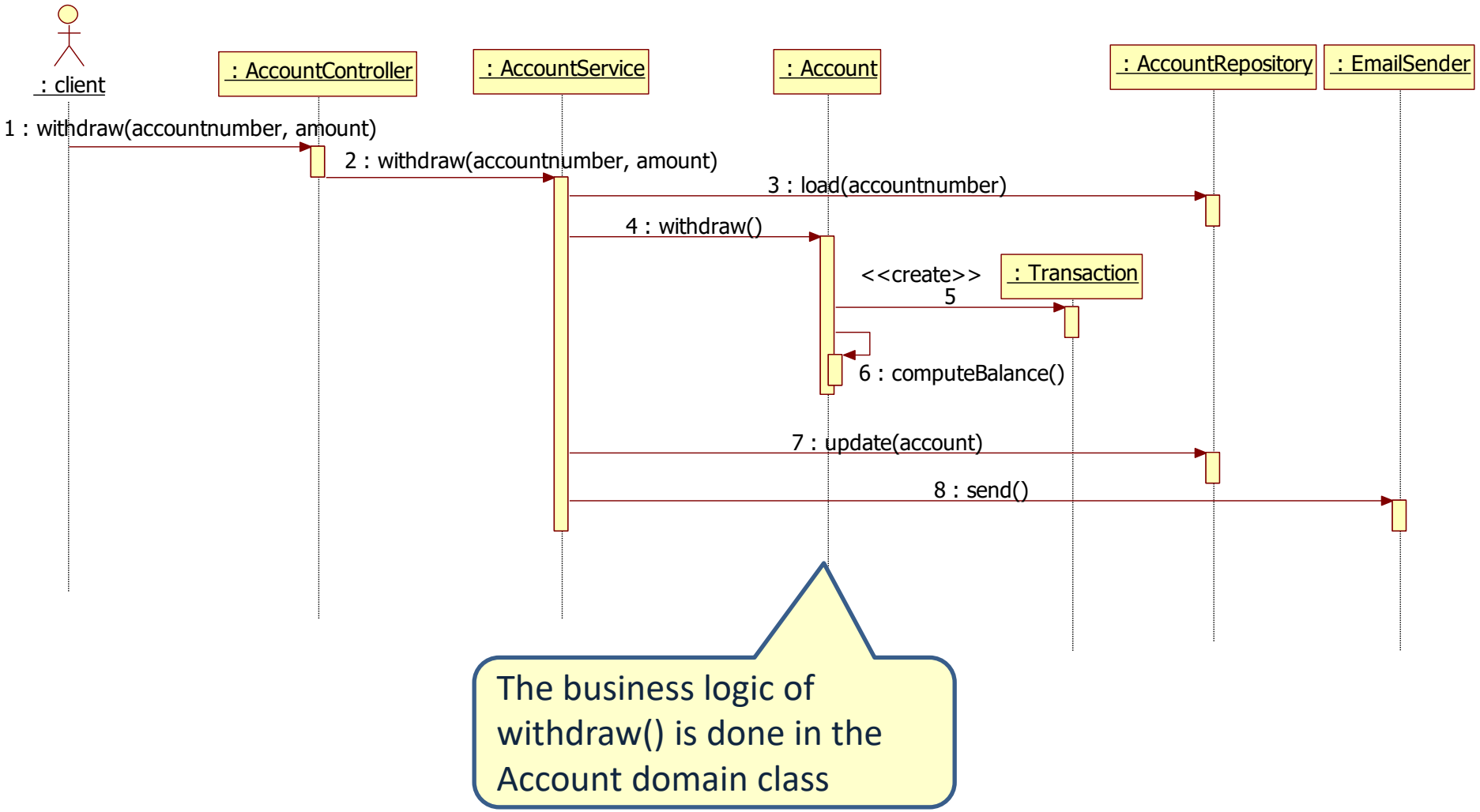
Anemic domain model example



Rich domain model example



Rich domain model example



ORCHESTRATION & CHOREOGRAPHY

Orchestration vs. choreography

- Orchestration
  - One central brain

Easy to follow the process

Does not work well in large and or complex applications

- Choreography
  - No central brain

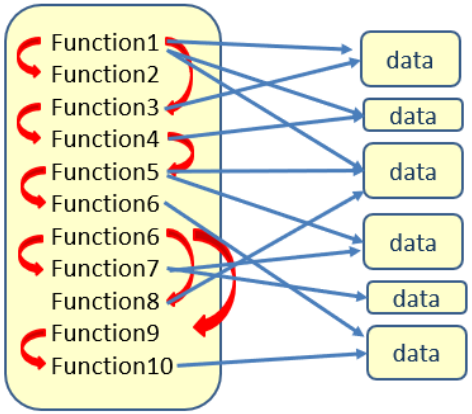
Hard to follow the process

Does work well in large and or complex applications

Orchestration vs. choreography

- Orchestration
  - One central brain

Procedural programming
(C, Pascal, Algol, Cobol)

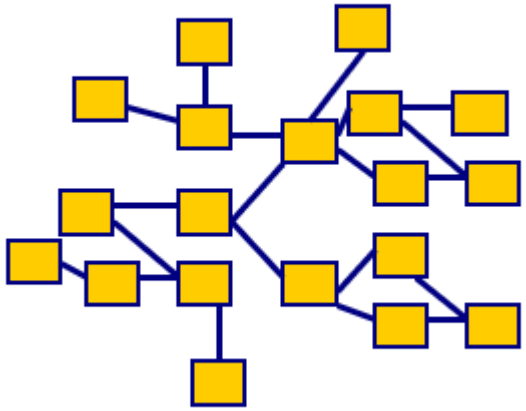


Easy to follow the process

Does not work well in large and or complex applications

- Choreography
  - No central brain

Object-Oriented programming
(Java, C#, Python, C++)



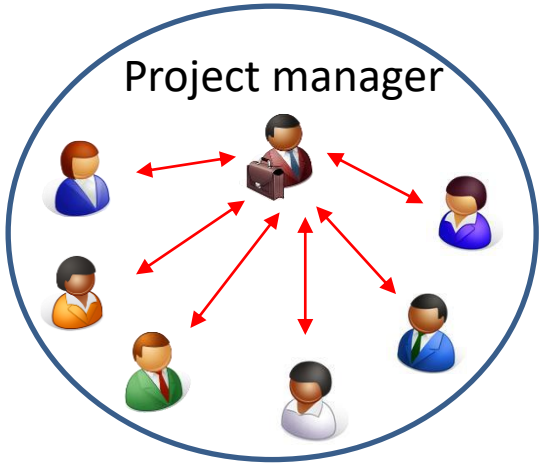
Hard to follow the process

Does work well in large and or complex applications

Orchestration vs. choreography

- Orchestration
  - One central brain

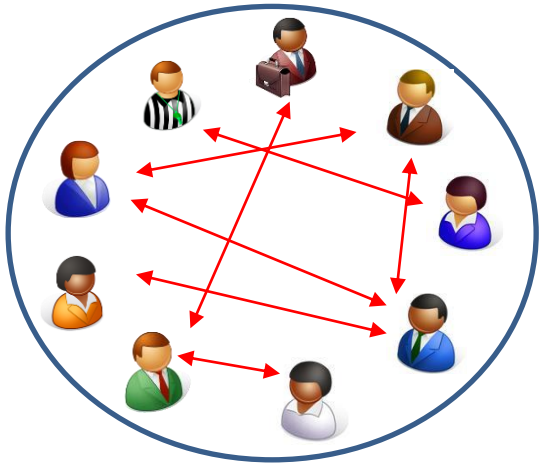
Waterfall



Does not work well complex projects

- Choreography
  - No central brain

Agile

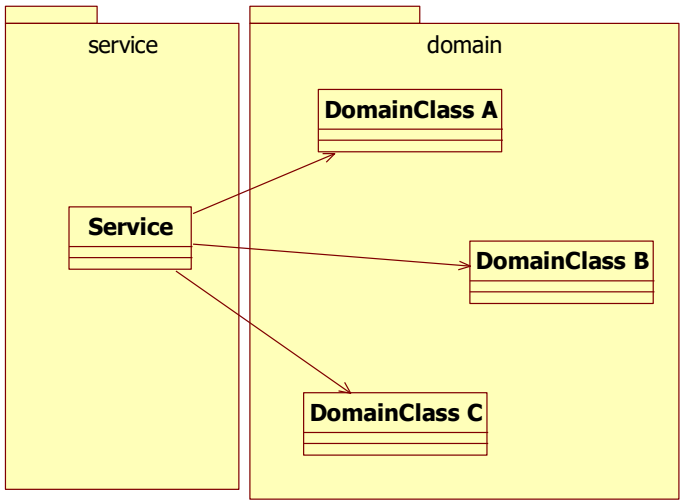


Does work well in complex projects

Orchestration vs. choreography

- Orchestration
  - One central brain

Anemic domain model

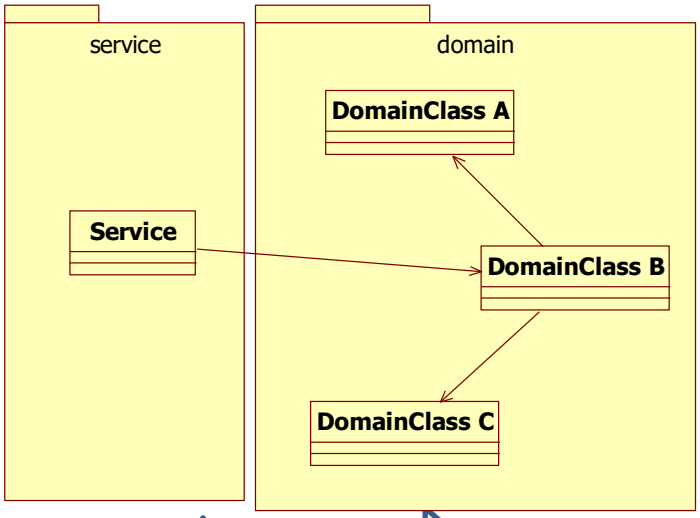


Easy to follow the process

Does not work well in large and or complex applications

- Choreography
  - No central brain

Rich domain model



Hard to follow the process

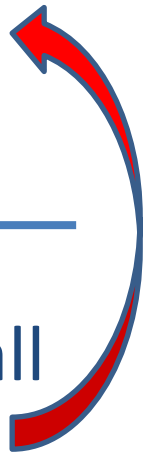
Does work well in large and or complex applications

Main point

- The façade pattern provides a unified interface to a complex set of classes. It hides the complexity from the client(s).
- Pure Consciousness provides a unified interface to all aspects of creation, and the daily experience of Pure Consciousness makes life much more enjoyable.

Connecting the parts of knowledge with the wholeness of knowledge

1. Good software design is based on design principles that improve the quality of an application.
2. Design patterns are solutions to certain design problems within a certain context.



3. **Transcendental consciousness** is the home of all the laws of nature.
4. **Wholeness moving within itself:** In Unity Consciousness, one experiences that everything is an expression of ones own Self.