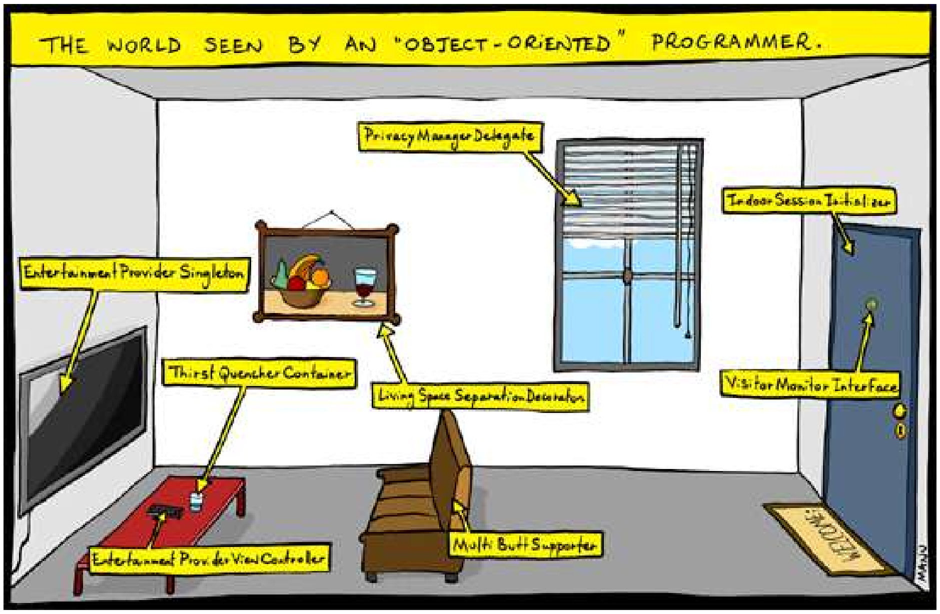

THE WORLD SEEN BY AN "OBJECT-ORIENTED" PROGRAMMER.
Privacy Manager Delegate
Indoor Session Initializer
Entertainment Provider Singleton
Thirst Quencher Container
Living Space Separation Decorator
Visitor Monitor Interface
Entertainment Provider View Controller
Multi Butt Supporter
WELCOME

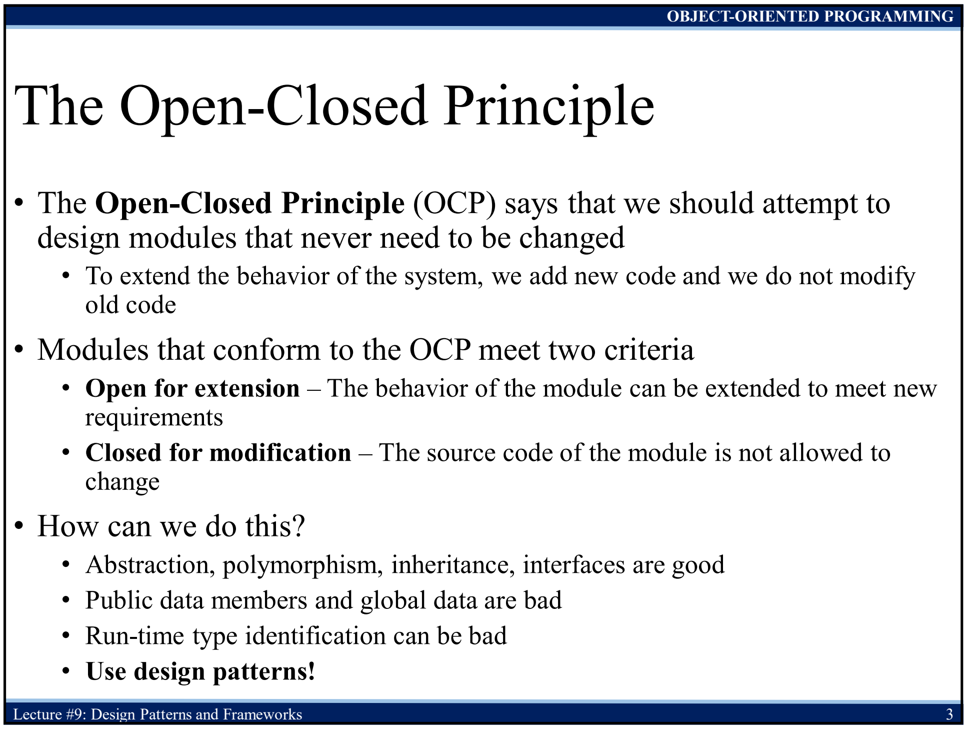
OBJECT-ORIENTED PROGRAMMING

Design Patterns and Frameworks

Lecture #9

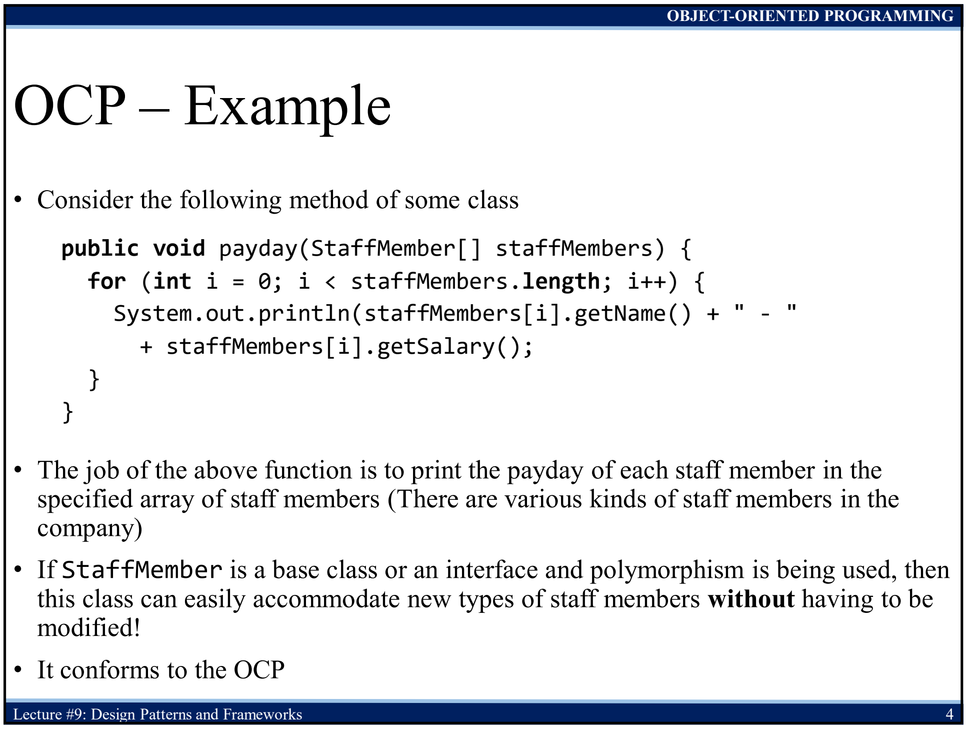
doc. Ing. Martin Tomášek, PhD.
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice

2024/2025



The Open-Closed Principle

- The **Open-Closed Principle** (OCP) says that we should attempt to design modules that never need to be changed
  - To extend the behavior of the system, we add new code and we do not modify old code
- Modules that conform to the OCP meet two criteria
  - **Open for extension** – The behavior of the module can be extended to meet new requirements
  - **Closed for modification** – The source code of the module is not allowed to change
- How can we do this?
  - Abstraction, polymorphism, inheritance, interfaces are good
  - Public data members and global data are bad
  - Run-time type identification can be bad
  - **Use design patterns!**

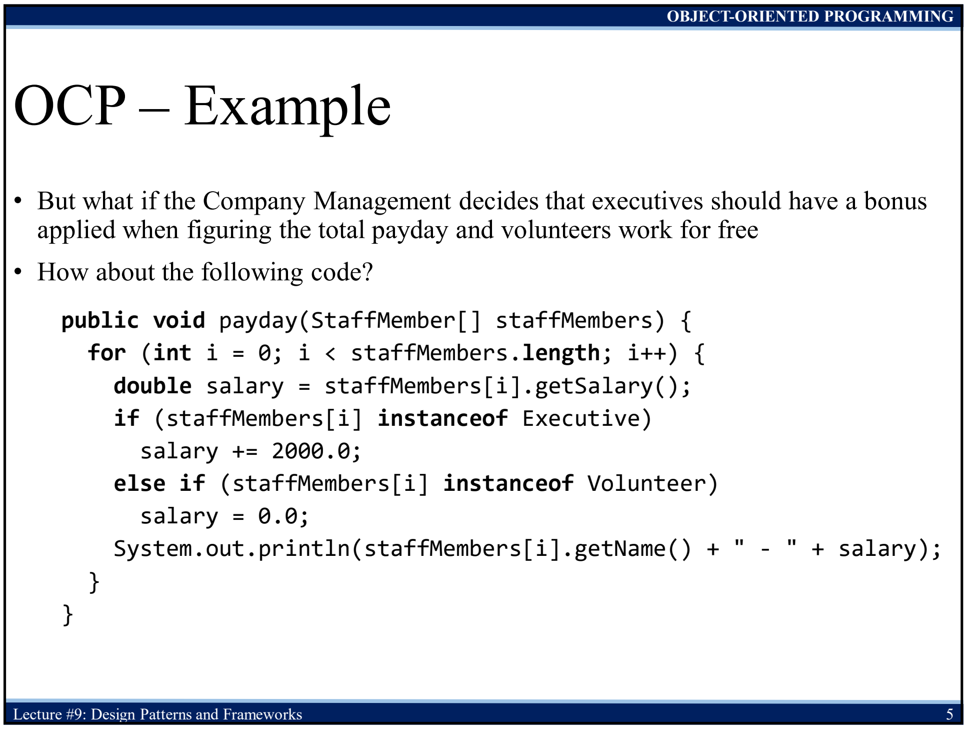


OCP – Example

- Consider the following method of some class

```
public void payday(StaffMember[] staffMembers) {
  for (int i = 0; i < staffMembers.length; i++) {
    System.out.println(staffMembers[i].getName() + " - "
      + staffMembers[i].getSalary();
  }
}
```

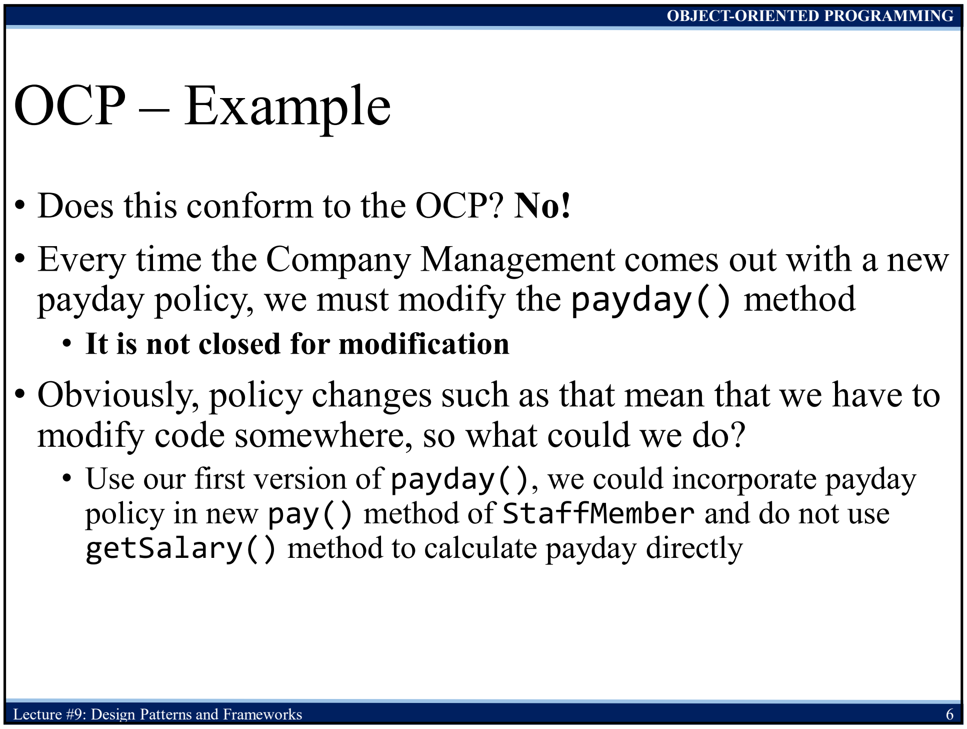
- The job of the above function is to print the payday of each staff member in the specified array of staff members (There are various kinds of staff members in the company)
- If `StaffMember` is a base class or an interface and polymorphism is being used, then this class can easily accommodate new types of staff members **without** having to be modified!
- It conforms to the OCP



OCP – Example

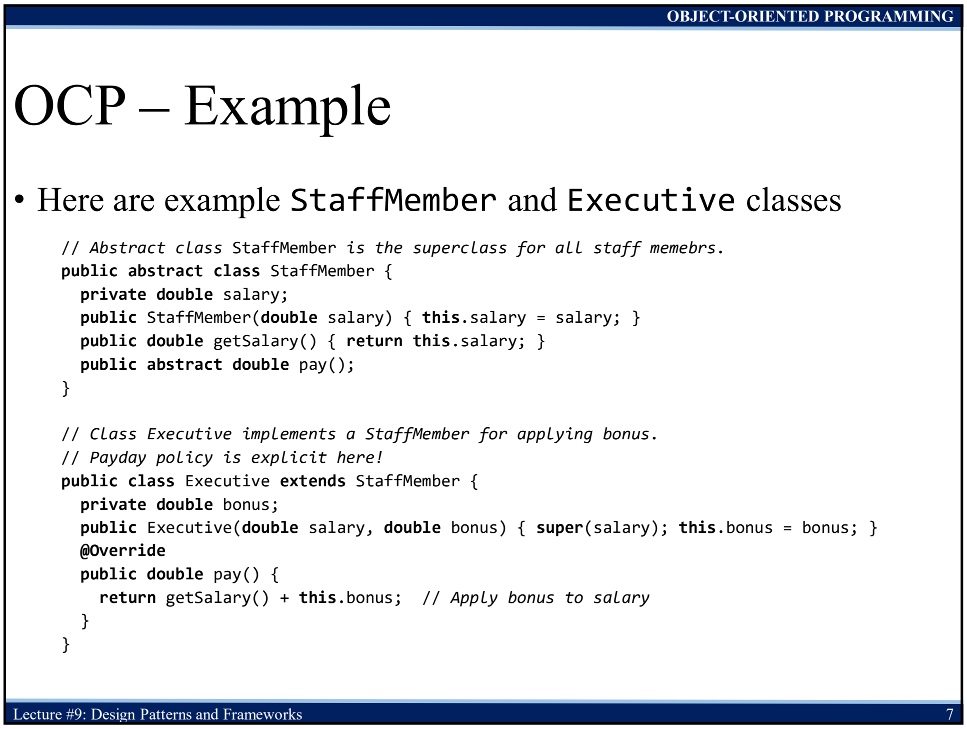
- But what if the Company Management decides that executives should have a bonus applied when figuring the total payday and volunteers work for free
- How about the following code?

```
public void payday(StaffMember[] staffMembers) {
  for (int i = 0; i < staffMembers.length; i++) {
    double salary = staffMembers[i].getSalary();
    if (staffMembers[i] instanceof Executive)
      salary += 2000.0;
    else if (staffMembers[i] instanceof Volunteer)
      salary = 0.0;
    System.out.println(staffMembers[i].getName() + " - " + salary);
  }
}
```



OCP – Example

- Does this conform to the OCP? **No!**
- Every time the Company Management comes out with a new payday policy, we must modify the `payday()` method
  - **It is not closed for modification**
- Obviously, policy changes such as that mean that we have to modify code somewhere, so what could we do?
  - Use our first version of `payday()`, we could incorporate payday policy in new `pay()` method of `StaffMember` and do not use `getSalary()` method to calculate payday directly

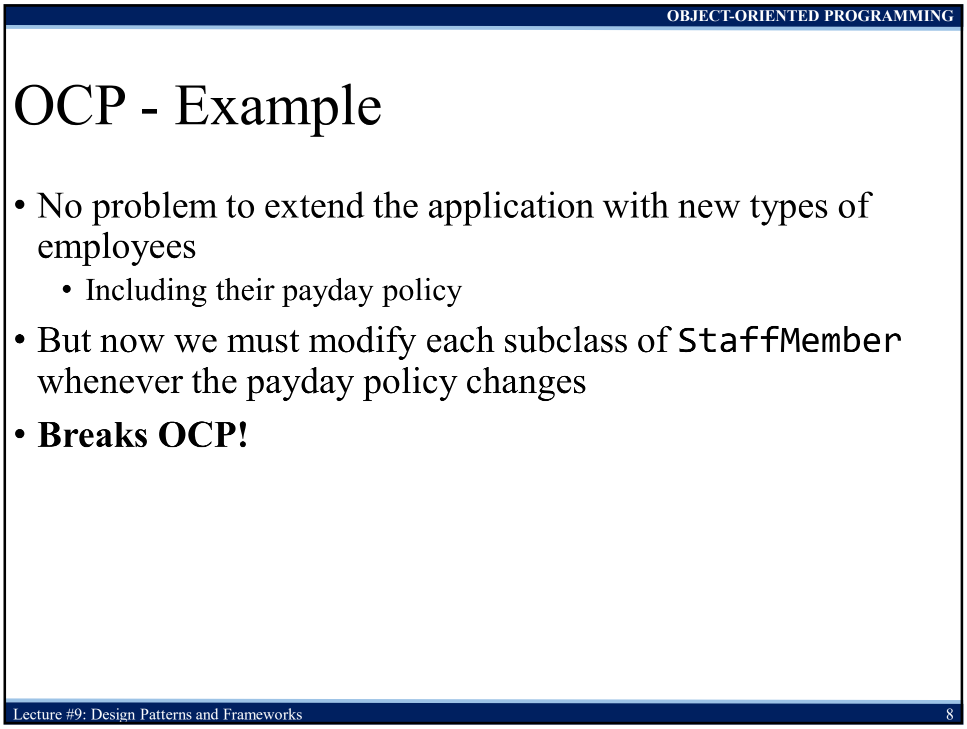


OCP – Example

- Here are example `StaffMember` and `Executive` classes

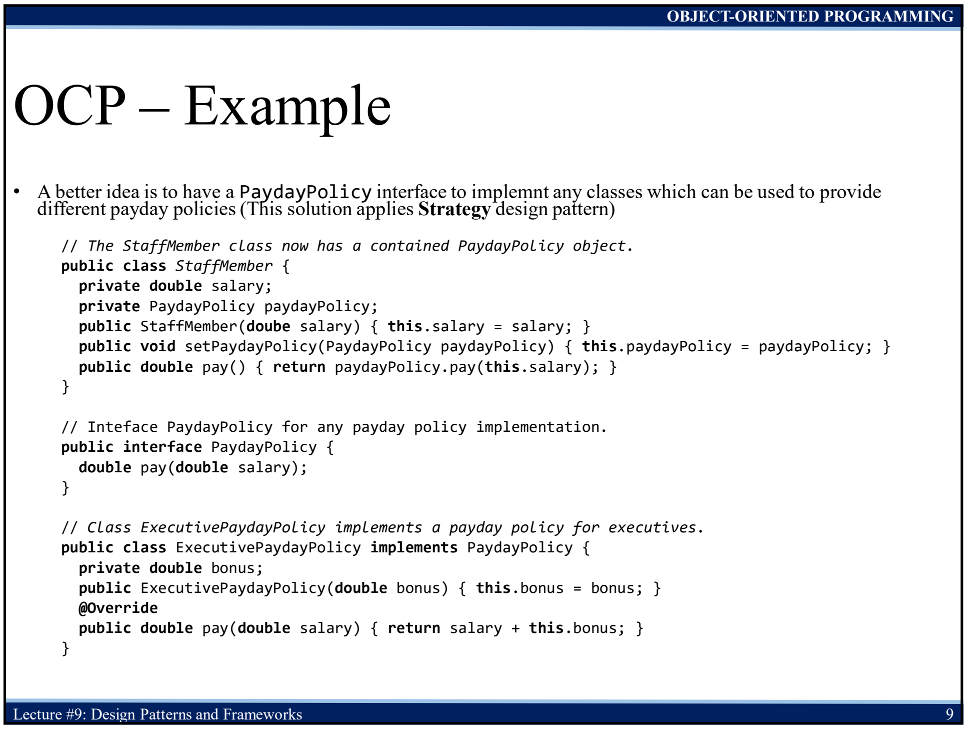
```
// Abstract class StaffMember is the superclass for all staff memebrs.
public abstract class StaffMember {
  private double salary;
  public StaffMember(double salary) { this.salary = salary; }
  public double getSalary() { return this.salary; }
  public abstract double pay();
}

// Class Executive implements a StaffMember for applying bonus.
// Payday policy is explicit here!
public class Executive extends StaffMember {
  private double bonus;
  public Executive(double salary, double bonus) { super(salary); this.bonus = bonus; }
  @Override
  public double pay() {
    return getSalary() + this.bonus;  // Apply bonus to salary
  }
}
```



OCP - Example

- No problem to extend the application with new types of employees
  - Including their payday policy
- But now we must modify each subclass of `StaffMember` whenever the payday policy changes
- **Breaks OCP!**



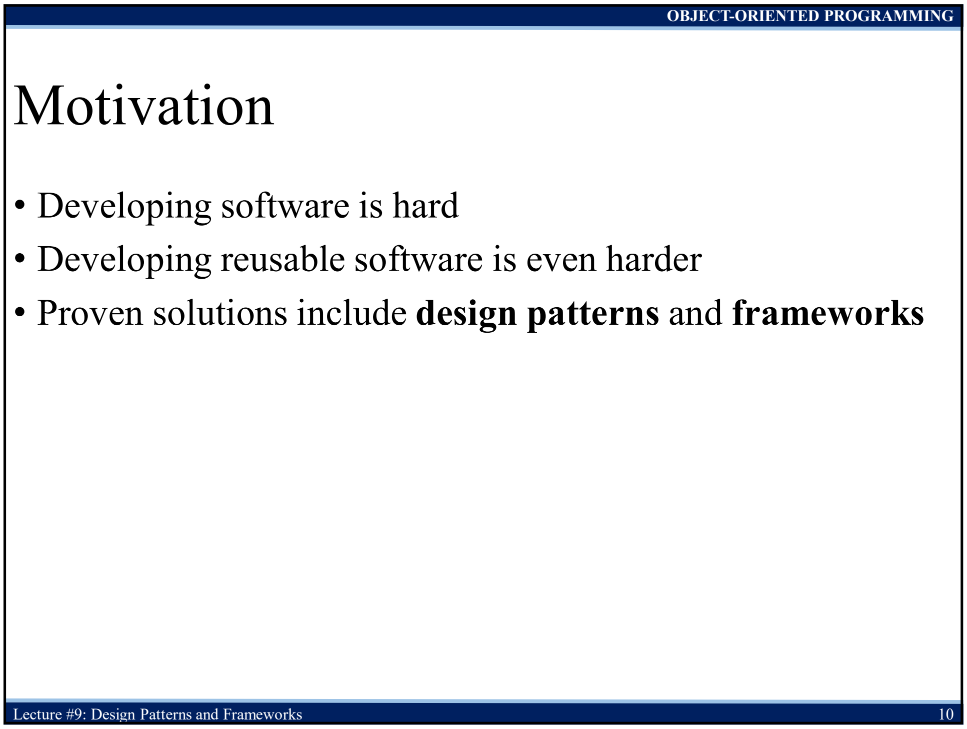
OCP – Example

- A better idea is to have a `PaydayPolicy` interface to implemnt any classes which can be used to provide different payday policies (This solution applies **Strategy** design pattern)

```
// The StaffMember class now has a contained PaydayPolicy object.
public class StaffMember {
  private double salary;
  private PaydayPolicy paydayPolicy;
  public StaffMember(doube salary) { this.salary = salary; }
  public void setPaydayPolicy(PaydayPolicy paydayPolicy) { this.paydayPolicy = paydayPolicy; }
  public double pay() { return paydayPolicy.pay(this.salary); }
}

// Inteface PaydayPolicy for any payday policy implementation.
public interface PaydayPolicy {
  double pay(double salary);
}

// Class ExecutivePaydayPolicy implements a payday policy for executives.
public class ExecutivePaydayPolicy implements PaydayPolicy {
  private double bonus;
  public ExecutivePaydayPolicy(double bonus) { this.bonus = bonus; }
  @Override
  public double pay(double salary) { return salary + this.bonus; }
}
```



Motivation

- Developing software is hard
- Developing reusable software is even harder
- Proven solutions include **design patterns** and **frameworks**

Overview

- What are design patterns?
  - Patterns support reuse of **software architecture** and **design**
  - Patterns capture the static and dynamic structures and collaborations of successful solutions to problems that arise when building applications in a domain
- What are frameworks?
  - Frameworks support reuse of **detailed design** and **code**
  - A framework is an integrated set of components that collaborate to provide a reusable architecture for a family of related applications
- Together, design patterns and frameworks help to improve **software quality** and **reduce development time**
  - Reuse, extensibility, modularity, performance, etc.

Patterns of learning

- Successful solutions to many areas of human endeavor are deeply rooted in patterns
  - An important goal of education is transmitting **patterns of learning** from generation to generation
- Let's see how patterns are used to learn chess
- Learning to develop good software is like learning to play good chess

Becoming a chess master

- **First learn the rules**
  - Names of pieces, legal movements, chess board geometry and orientation, etc.
- **Then learn the principles**
  - Relative value of certain pieces, strategic value of center squares, power of a threat, etc.
- **However, to become a master of chess, one must study the games of other masters**
  - These games contain **patterns** that must be understood, memorized, and applied repeatedly
- **There are hundreds of these patterns**

Becoming a software design master

- **First learn the rules**
  - The algorithms, data structures and languages of software
- **Then learn the principles**
  - Structured programming, modular programming, object-oriented programming, generic programming, etc.
- **However, to become a master of software design, one must study the designs of other masters**
  - These designs contain **patterns** that must be understood, memorized, and applied repeatedly
- **There are hundreds of these patterns**

Design patterns

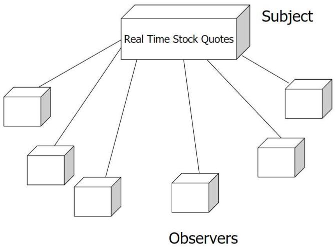
- Design patterns represent **solutions** to **problems** that arise when developing software within a context

  pattern = (problem, solution, context)

- Patterns capture the static and dynamic **structure** and **collaboration** among key **participants** in software designs
  - They are particularly useful for articulating how and why to resolve **non-functional** forces
- Patterns facilitate reuse of successful **software architectures** and **designs**

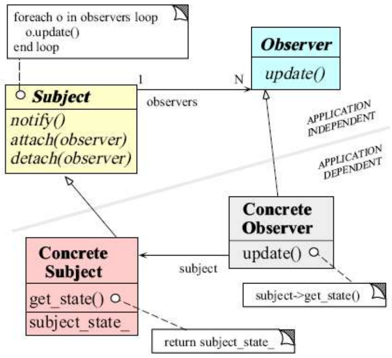
Example: Specification

- Application of Stock Quote Service
- Requirements
  - There may be many observers
  - Each observer may react differently to the same notification
  - The subject should be as decoupled as possible from the observers
    - Allow observers to change independently of the subject
- We will use **Observer** design pattern to design the system



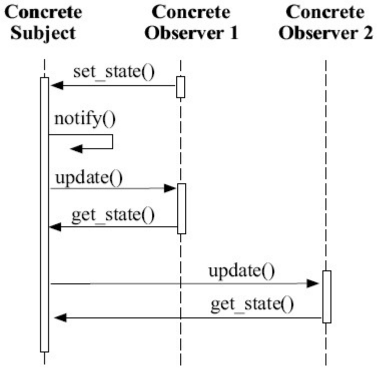
Example: Structure of the Observer design pattern

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



Example: Interaction in the Observer design pattern

- **Push architectures** combine control flow and data flow
- **Pull architectures** separate control flow from data flow



How to describe design patterns?

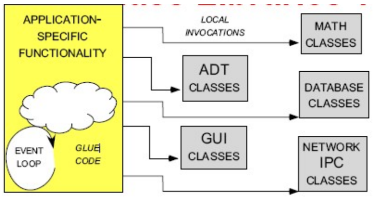
- Main parts
  - Name and intent
  - Problem and context
  - Force(s) addressed
  - Abstract description of structure and collaborations in solution
  - Positive and negative consequence(s) of use
  - Implementation guidelines and sample code
  - Known uses and related patterns
- Pattern descriptions are often independent of programming language or implementation details
  - Contrast with frameworks

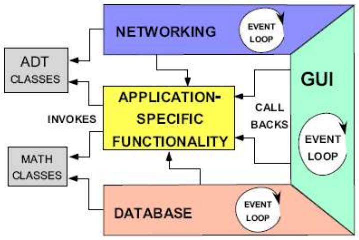
Frameworks

- Frameworks are **semi-complete applications**
  - Complete applications are developed by **inheriting** from, and **instantiating** parameterized framework components
- Frameworks provide **domain-specific functionality**
  - Business applications, telecommunication applications, window systems, databases, distributed applications, OS kernels, etc.
- Frameworks exhibit **inversion of control at run-time**
  - i.e., the framework determines which objects and methods to invoke in response to events

Class libraries vs. frameworks

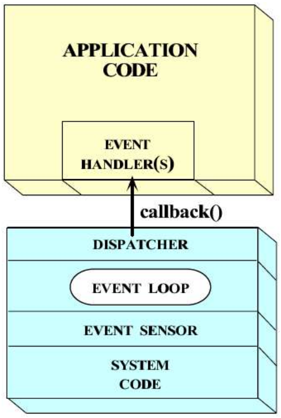
- Class libraries
  - Self-contained, **pluggable** ADTs
- Frameworks
  - Reusable, **semi-complete** applications





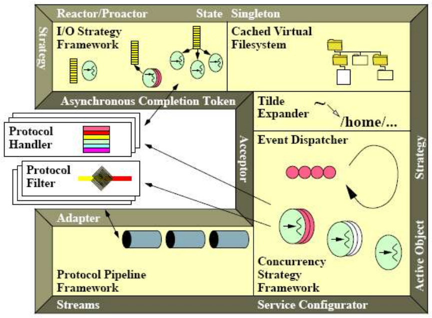
Component integration in framework

- Framework components are loosely coupled via **callbacks**
- Callbacks allow independently developed software components to be connected together
- Callbacks provide a connection-point where generic framework objects can communicate with application objects
  - The framework provides the common **template methods** and the application provides the variant hook methods



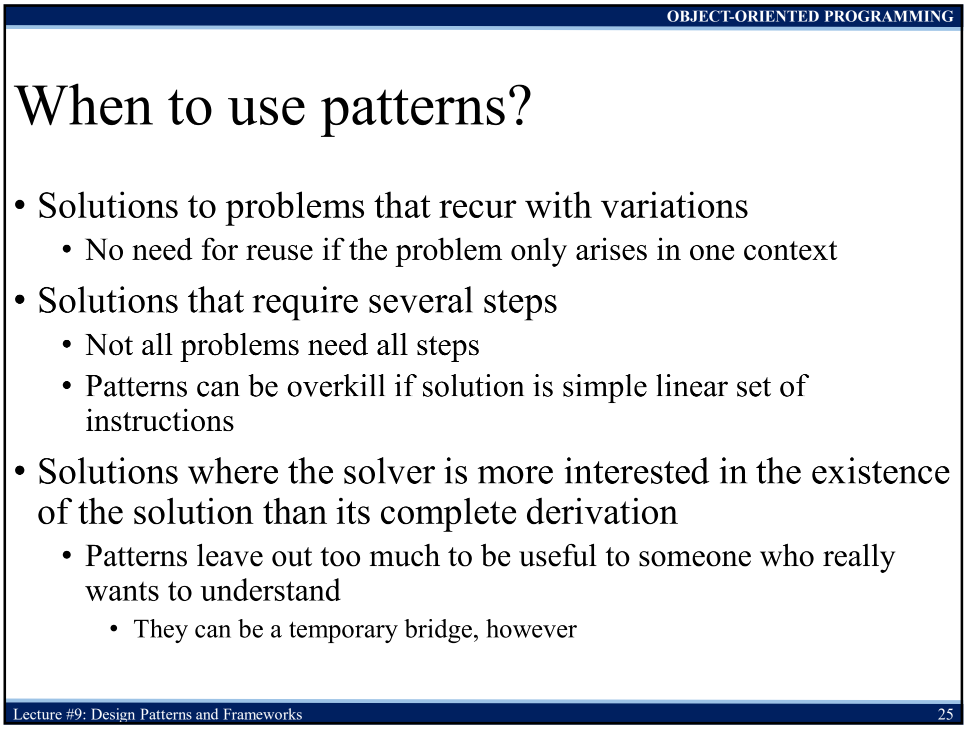
Comparing design patterns and frameworks

- Patterns and frameworks are highly synergistic
  - Neither is subordinate
- Patterns can be characterized as more abstract descriptions of frameworks, which are implemented in a language
- In general, sophisticated frameworks embody dozens of patterns and patterns are often used to document frameworks



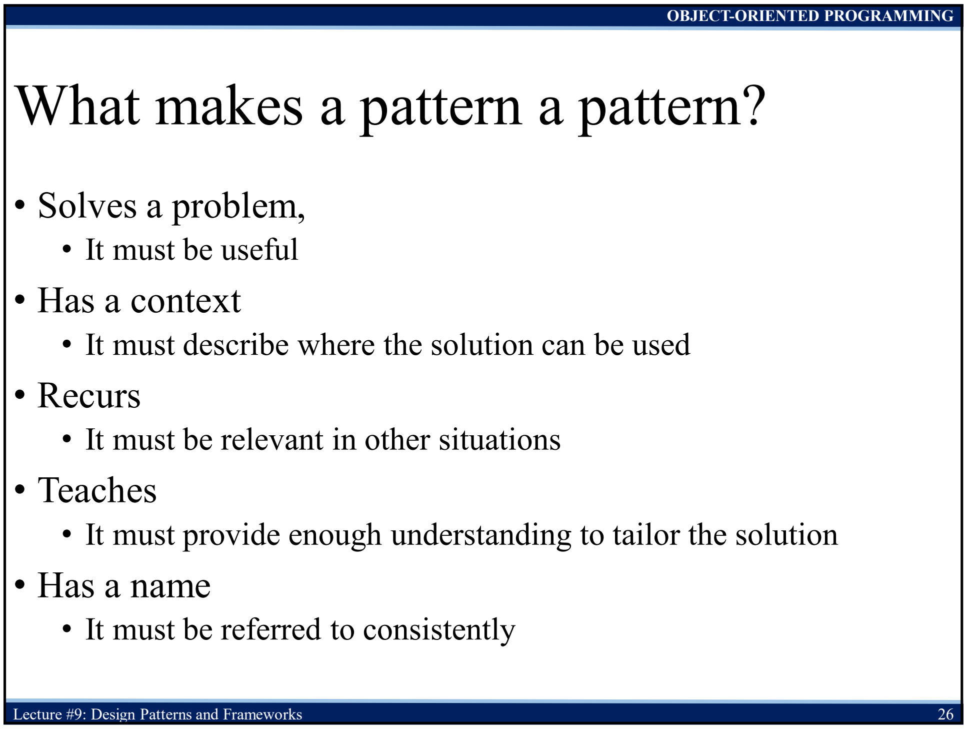
Design patterns classification

- **Creational patterns**
  - Deal with initializing and configuring classes and objects
  - Abstract factory, Builder, Factory method, Lazy initialization, Multiton, Object pool, Prototype, Resource acquisition is initialization, Singleton
- **Structural patterns**
  - Deal with decoupling interface and implementation of classes and objects
  - Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Front controller, Module, Proxy
- **Behavioral patterns**
  - Deal with dynamic interactions among societies of classes and objects
  - Blackboard, Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Null object, Observer, Servant, Specification, State, Strategy, Template method, Visitor



When to use patterns?

- Solutions to problems that recur with variations
  - No need for reuse if the problem only arises in one context
- Solutions that require several steps
  - Not all problems need all steps
  - Patterns can be overkill if solution is simple linear set of instructions
- Solutions where the solver is more interested in the existence of the solution than its complete derivation
  - Patterns leave out too much to be useful to someone who really wants to understand
    - They can be a temporary bridge, however



What makes a pattern a pattern?

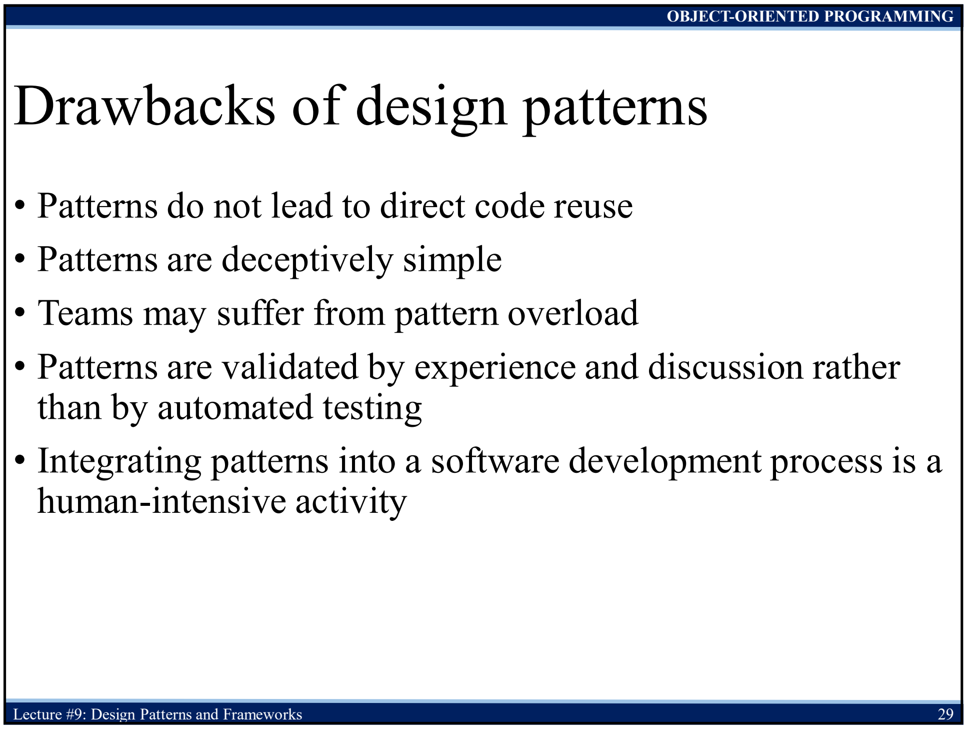
- Solves a problem,
  - It must be useful
- Has a context
  - It must describe where the solution can be used
- Recurs
  - It must be relevant in other situations
- Teaches
  - It must provide enough understanding to tailor the solution
- Has a name
  - It must be referred to consistently

Key principles

- Successful design patterns and frameworks can be boiled down to a few key principles
  - Separate interface from implementation
  - Determine what is **common** (stable) and what is **variable** with an interface and an implementation
  - Allow substitution of **variable** implementations via a **common** interface
- Dividing **commonality** from **variability** should be goal-oriented rather than exhaustive
- Frameworks often represent the distinction between commonality and variability via **template methods** and **hook methods**, respectively

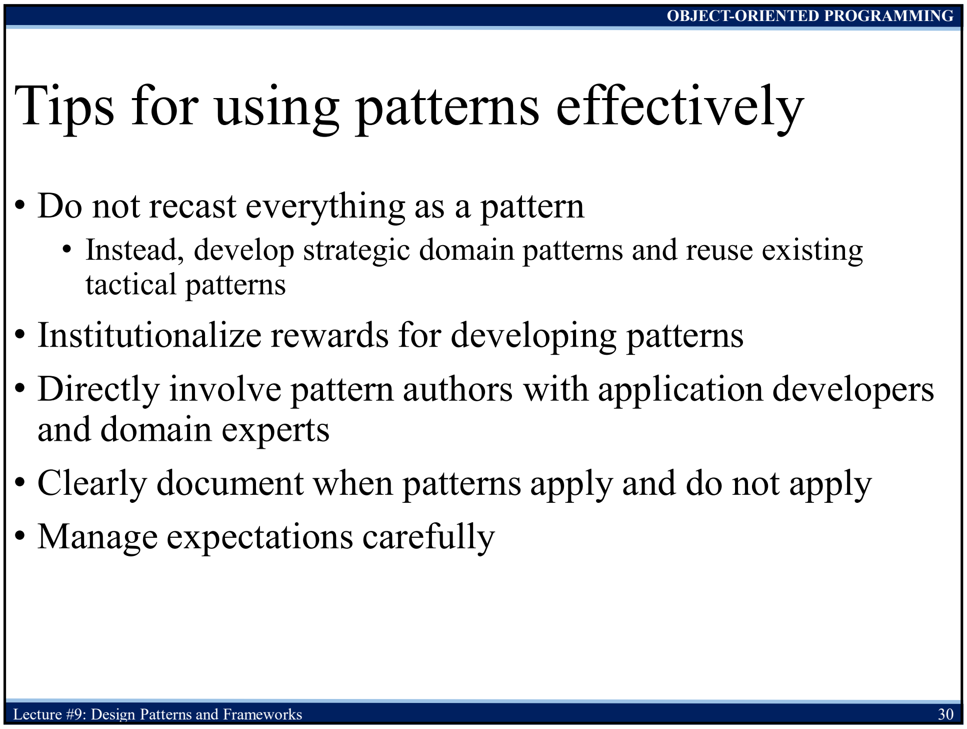
Benefits of design patterns

- Design patterns enable large-scale reuse of software architectures
  - They also help document systems to enhance understanding
- Patterns explicitly capture expert knowledge and design tradeoffs, and make this expertise more widely available
- Patterns help improve developer communication
  - Pattern names form a vocabulary
- Patterns help ease the transition to object-oriented technology



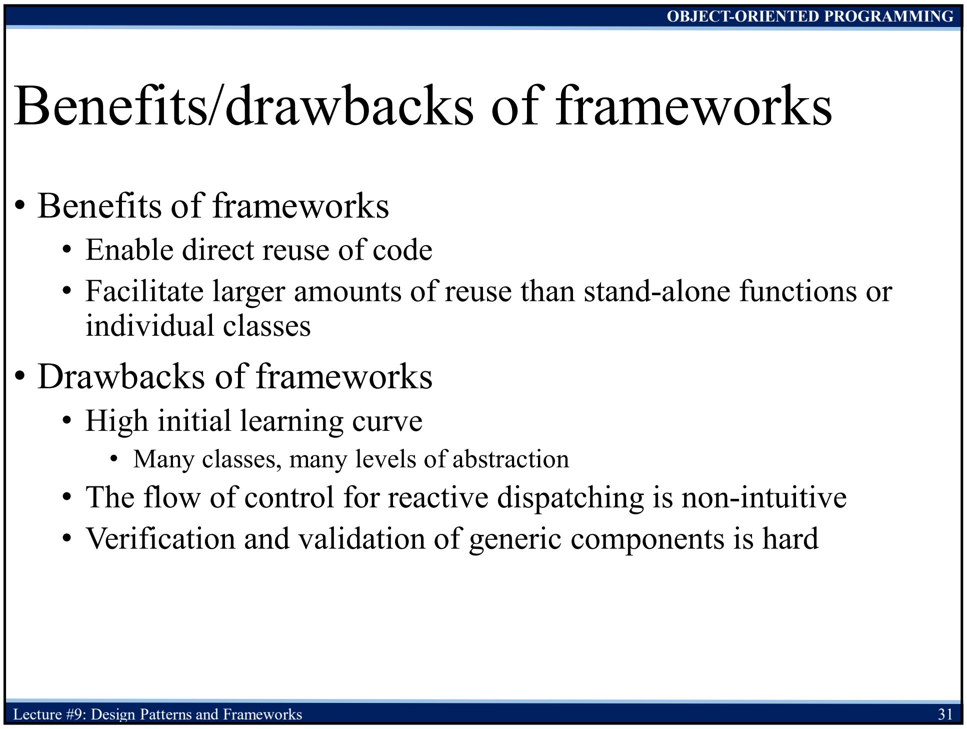
Drawbacks of design patterns

- Patterns do not lead to direct code reuse
- Patterns are deceptively simple
- Teams may suffer from pattern overload
- Patterns are validated by experience and discussion rather than by automated testing
- Integrating patterns into a software development process is a human-intensive activity



Tips for using patterns effectively

- Do not recast everything as a pattern
  - Instead, develop strategic domain patterns and reuse existing tactical patterns
- Institutionalize rewards for developing patterns
- Directly involve pattern authors with application developers and domain experts
- Clearly document when patterns apply and do not apply
- Manage expectations carefully



Benefits/drawbacks of frameworks

- Benefits of frameworks
  - Enable direct reuse of code
  - Facilitate larger amounts of reuse than stand-alone functions or individual classes
- Drawbacks of frameworks
  - High initial learning curve
    - Many classes, many levels of abstraction
  - The flow of control for reactive dispatching is non-intuitive
  - Verification and validation of generic components is hard