# Design Patterns

# Introduction

# Reusable software design is hard

- Designing object-oriented software is hard, and designing reusable object-oriented software is even harder.
- It includes:
  - Finding appropriate objects.
  - Factoring them into classes at the right granularity - not to broad not to specific.
  - Defining class interfaces and inheritance hierarchies, and establish key relationships among them.
- Design should be specific to the problem at hand but also general enough to address future problems and requirements.
- You also want to avoid redesign, or at least minimize it.
- It is almost impossible to get right the first time.

# Use design patterns

- Experienced designers use solutions that have worked for them in the past.
- When they find a good solution, they use it again and again.
- You'll find recurring patterns of classes and communicating objects in many object-oriented systems.
- These patterns solve specific design problems and make object-oriented designs more flexible, elegant, and ultimately reusable.
- A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them.

# What is a design pattern?

- Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice. - Christopher Alexander
- Each design pattern systematically names, explains, and evaluates an important and recurring design in object-oriented systems.
- The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities.

# Benefits of using design patterns

- Design patterns make it easier to reuse successful designs and architectures.
- Expressing proven techniques as design patterns makes them more accessible to developers of new systems.
- Design patterns help you choose design alternatives that make a system reusable and avoid alternatives that compromise reusability.
- Design patterns can even improve the documentation and maintenance of existing systems.

# Four elements of a design pattern

- Pattern name - lets us design at a higher level of abstraction.
- Problem - describes when to apply the pattern.
- Solution - describes the elements that make up the design.
- Consequences - the results and trade-offs of applying the pattern.

# Categories of design patterns (1)

- Based on purpose - what pattern does:
  - Creational
  - Structural
  - Behavioral
- Creational patterns concern the process of object creation.
- Structural patterns deal with the composition of classes or objects.
- Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.

# Categories of design patterns (2)

- Based on scope - whether the pattern applies primarily to classes or to objects:
  - Class
  - Object
- Class patterns deal with relationships between classes and their subclasses.
  - These relationships are established through inheritance, so they are static—fixed at compile-time.
- Object patterns deal with object relationships, which can be changed at run-time and are more dynamic.

# Catalog of design patterns

| | | Purpose | | |
|---|---|---|---|---|
| | | Creational | Structural | Behavioral |
| Scope | Class | Factory Method | Adapter | Interpreter, Template Method |
| | Object | Abstract Factory, Builder, Prototype, Singleton | Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy | Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Visitor |

# Design pattern categories - overview

- Creational class patterns defer some part of object creation to subclasses.
- Creational object patterns defer it to another object.
- Structural class patterns use inheritance to compose classes.
- Structural object patterns describe ways to assemble objects.
- Behavioral class patterns use inheritance to describe algorithms and flow of control.
- Behavioral object patterns describe how a group of objects cooperate to perform a task that no single object can carry out alone.

# Interrelationships of design patterns

- There are other ways to organize the patterns.
- Some patterns are often used together
  - E.g. Composite is often used with Iterator or Visitor.
- Some patterns are alternatives.
  - E.g. Prototype is often an alternative to Abstract Factory.
- Some patterns result in similar designs even though the patterns have different intents.
  - E.g. The structure diagrams of Composite and Decorator are similar.

# How design patterns solve design problems

# Which problems design patterns can solve?

- Finding appropriate objects
- Determining object granularity
- Specifying object interfaces
- Specifying object implementations
- Putting reuse mechanisms to work
- Designing for change

# Finding appropriate objects

# Finding appropriate objects

- Object-oriented programs are made up of **objects**.
- An object packages both data and the procedures that operate on that data.
- The procedures are typically called **methods** or **operations**.
- An object performs an operation when it receives a request (or message) from a **client**.

# Requests

- Requests are the only way to get an object to execute an operation.
- Operations are the only way to change an object's internal data.
- Because of these restrictions, the object's internal state is said to be encapsulated.
- It cannot be accessed directly, and its representation is invisible from outside the object.

# Different approaches

- Object-oriented design methodologies favor many different approaches.
- You can write a problem statement, single out the nouns and verbs, and create corresponding classes and operations.
- Or you can focus on the collaborations and responsibilities in your system.
- Or you can model the real world and translate the objects found during analysis into design.
- There will always be disagreement on which approach is best.

# Modeling real world entities

- Object-oriented designs often end up with classes that have no counterparts in the real world.
- Strict modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrow's.
- The abstractions that emerge during design are key to making a design flexible.
- Design patterns help identify less-obvious abstractions and the objects that can capture them.
- For example, objects that represent a process or algorithm don't occur in nature, yet they are a crucial part of flexible designs.

# Example: Food Ordering System

- Intuitively, we might try to model things strictly based on the real world, using classes like:
  - Restaurant
  - Customer
  - Order
- This seems logical, but what if a restaurant can handle different types of orders (e.g., online, phone, scheduled orders)?
- If we strictly follow the real world, we might lack the flexibility to add new order types in the future.

# Strictly modeling the real world can be problematic

```
class Order {
    String type; // "online", "phone"
    void processOrder() {
        if (type.equals("online")) {
            System.out.println("Processing an online order...");
        } else if (type.equals("phone")) {
            System.out.println("Processing a phone order...");
        }
    }
}
```

# Better approach: Introducing abstraction

- Problem with previous design: If we want to add a new order type tomorrow, we have to modify the Order class, making the system harder to maintain.
- We also break the open/closed principle.
- Instead of strictly following the real world, we introduce an abstract class OrderProcessor with different implementations (OnlineOrder, PhoneOrder, etc.).
- OrderProcessor doesn’t exist in real life, but it provides a better architecture.

# Design with abstract class

```
abstract class OrderProcessor {
    abstract void process();}

class OnlineOrder extends OrderProcessor {
    public void process()
        System.out.println("Processing an online order...");}

class PhoneOrder extends OrderProcessor{
    public void process()
        System.out.println("Processing a phone order...");}
```

# Expanding the design

- Now, we can easily add a new order type, such as ScheduledOrder, without modifying existing code:

```
class ScheduledOrder extends OrderProcessor {

    public void process() {

        System.out.println("Processing a scheduled order...");

    }

}
```

# Which design pattern was this?

- The example follows the Strategy design pattern.
- Each type of order (OnlineOrder, PhoneOrder, ScheduledOrder) has a different way of processing but shares the same interface.
- The system can dynamically select an appropriate strategy (order type) at runtime, making it flexible and easy to extend.

# Determining object granilarity

# Determining object granularity

- Objects can vary tremendously in size and number.
- They can represent everything down to the hardware or all the way up to entire applications.
- How do we decide what should be an object?
- Design patterns address this issue as well.

# Specifying object interfaces

# Object interface

- Every operation declared by an object specifies the operation's name, the objects it takes as parameters, and the operation's return value.
- This is known as the operation's signature.
- The set of all signatures defined by an object's operations is called the **interface to the object**.
- An object's interface characterizes the complete set of requests that can be sent to the object.
- Any request that matches a signature in the object's interface may be sent to the object.

# Type

- A type is a name used to denote a particular interface.
- We speak of an object as having the type "Window" if it accepts all requests for the operations defined in the interface named "Window."
- An object may have many types, and widely different objects can share a type.
- Part of an object's interface may be characterized by one type, and other parts by other types.
- Two objects of the same type need only share parts of their interfaces.
- Interfaces can contain other interfaces as subsets.

# Implementing interfaces

- Interfaces are fundamental in object-oriented systems.
- Objects are known only through their interfaces - because of eencapsulation.
- There is no way to know anything about an object or to ask it to do anything without going through its interface.
- An object's interface says nothing about its implementation—different objects are free to implement requests differently.
- That means two objects having completely different implementations can have identical interfaces.

# Issuing a request

- When a request is sent to an object, the particular operation that's performed depends on both the request and the receiving object.
- Different objects that support identical requests may have different implementations of the operations that fulfill that request.
- The run-time association of a request to an object and one of its operations is known as dynamic binding.
- Dynamic binding means that issuing a request doesn't commit you to a particular implementation until run-time.

# How design patterns help here?

- Design patterns help you define interfaces by identifying their key elements and the kinds of data that get sent across an interface.
- Design pattern might also tell you what not to put in the interface.
- Design patterns also specify relationships between interfaces. In particular, they often require some classes to have similar interfaces
- Also, they place constraints on the interfaces of some classes.

# Specifying object implementations

# Classes and instances

- An object's implementation is defined by its **class**.
- The class specifies the object's internal data and representation and defines the operations the object can perform.
- Objects are created by instantiating a class.
- The object is said to be an **instance** of the class.
- The process of instantiating a class allocates storage for the object's internal data (made up of instance variables) and associates the operations with these data.

# Class inheritance

- New classes can be defined in terms of existing classes using **class inheritance**.
- When a subclass inherits from a parent class, it includes the definitions of all the data and operations that the parent class defines.
- Objects that are instances of the subclass will contain all data defined by the subclass and its parent classes, and they'll be able to perform all operations defined by this subclass and its parents.

# Abstract class

- An **abstract class** is one whose main purpose is to define a common interface for its subclasses.
- An abstract class will defer some or all of its implementation to operations defined in subclasses; hence an abstract class cannot be instantiated.
- Classes that aren't abstract are called **concrete classes.**

# Class vs type

- An object's class defines how the object is implemented.
- The class defines the object's internal state and the implementation of its operations.
- In contrast, an object's type only refers to its interface—the set of requests to which it can respond.
- An object can have many types, and objects of different classes can have the same type.

# Class inheritance vs interface inheritance

- It's also important to understand the difference between class inheritance and interface inheritance (or subtyping).
- Class inheritance defines an object's implementation in terms of another object's implementation.
- In short, it's a mechanism for code and representation sharing.
- In contrast, interface inheritance (or subtyping) describes when an object can be used in place of another.
- It's easy to confuse these two concepts, because many languages don't make the distinction explicit.

# Benefits of writing code in terms of interfaces

- There are two benefits to manipulating objects solely in terms of the interface defined by abstract classes:
  - Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect.
  - Clients remain unaware of the classes that implement these objects. Clients only know about the abstract class(es) defining the interface.
- This leads to the following principle of reusable object-oriented design:
  - Don't declare variables to be instances of particular concrete classes. Instead, commit only to an interface defined by an abstract class.
- You have to instantiate concrete classes somewhere in your system, of course, and the creational patterns help with that.

# Designing for change

# Designing for change

- The key to maximizing reuse lies in anticipating new requirements and changes to existing requirements, and in designing your systems so that they can evolve accordingly.
- A design that doesn't take change into account risks major redesign in the future.
- Those changes might involve class redefinition and reimplementation, client modification, and retesting.
- Redesign affects many parts of the software system, and unanticipated changes are invariably expensive.
- Design patterns help to avoid this.

# Problem - Creating an object by specifying a class explicitly.

- Specifying a class name when you create an object commits you to a particular implementation instead of a particular interface.
- This commitment can complicate future changes.
- To avoid it, create objects indirectly.
- Design patterns: Abstract Factory, Factory Method, Prototype.

# Problem - Dependence on hardware and software platform

- External operating system interfaces and application programming interfaces (APIs) are different on different hardware and software platforms.
- Software that depends on a particular platform will be harder to port to other platforms.
- It may even be difficult to keep it up to date on its native platform.
- It's important therefore to design your system to limit its platform dependencies.
- Design patterns: Abstract Factory, Bridge.

# Problem - Dependence on object representations or implementations.

- Clients that know how an object is represented, stored, located, or implemented might need to be changed when the object changes.
- Hiding this information from clients keeps changes from cascading.
- Design patterns: Abstract Factory, Bridge, Memento, Proxy.

# Problem - Algorithmic dependence

- Algorithms are often extended, optimized, and replaced during development and reuse.
- Objects that depend on an algorithm will have to change when the algorithm changes.
- Therefore algorithms that are likely to change should be isolated.
- Design patterns: Builder, Iterator, Strategy, Template Method, Visitor.

# Problem - Tight coupling

- Classes that are tightly coupled are hard to reuse in isolation, since they depend on each other.
- Tight coupling leads to monolithic systems, where you can't change or remove a class without understanding and changing many other classes.
- Design patterns use techniques such as abstract coupling and layering to promote loosely coupled systems.
- Design patterns: Abstract Factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer.

# Problem - Inability to alter classes

- Sometimes you have to modify a class that can't be modified conveniently.
- Perhaps you need the source code and don't have it (as may be the case with a commercial class library).
- Or maybe any change would require modifying lots of existing subclasses.
- Design patterns offer ways to modify classes in such circumstances.
- Design patterns: Adapter, Decorator, Visitor.

# Kreacioni Paterni

# Uvod

- Kreacioni paterni su dizajnerski paterni koji se fokusiraju na stvaranje objekata na optimalan i fleksibilan način.
- Oni pomažu u izbjegavanju direktnog instanciranja klasa (korišćenjem new operatora), čime poboljšavaju održivost i proširivost koda.
- Ključni cilj je odvajanje logike kreiranja objekata od njihove upotrebe.
- Kreacioni paterni postaju važni kako sistemi evoluiraju i sve više se oslanjaju na komponovanje objekata umjesto na nasljeđivanje klasa.

# Vrste kreacionih paterna

- Singleton
- Factory Method
- Abstract Factory
- Builder
- Prototype

# Međusobni odnosi kreacionih paterna

- Ponekad su kreacioni paterni suprotstavljeni:
  - Na primjer, postoje slučajevi kada se ili Prototype ili Abstract Factory može koristiti.
- U drugim situacijama oni su komplementarni:
  - Builder može koristiti neki drugi patern da odredi koje komponente će biti kreirane.
  - Prototype može koristiti Singleton u svojoj implementaciji.

# Singleton

# Singleton

- Osigurava da samo jedna instanca klase postoji u sistemu.
- Omogućava globalnu tačku pristupa toj instanci.
- Sprječava kreiranje novih instanci iste klase.
- Koristi se za klase poput: Database Connection, Logger, Configuration Manager.

# Primjer upotrebe Singletona - Logger

- Jedan od najčešćih primjera upotrebe Singleton-a u realnim aplikacijama je Logger – sistem za zapisivanje logova u aplikaciji.
- Zamislimo da pravimo aplikaciju gdje različiti djelovi sistema zapisuju poruke u isti fajl (log.txt).
- Treba nam samo jedan Logger objekat kroz cijelu aplikaciju.
- Ako imamo više instanci Logger-a, može doći do problema (npr. istovremeno pisanje u fajl može izazvati korupciju podataka).
- Singleton obezbjeđuje da postoji samo jedna instanca Logger-a.

# Druge upotrebe Singleton-a

- Database Connection – Samo jedna veza ka bazi podataka kroz aplikaciju.
- Configuration Manager – Centralizovano upravljanje konfiguracionim podacima.
- Thread Pool – Upravljanje nitima bez kreiranja nepotrebnih instanci.
- Cache System – Skladištenje podataka u memoriji da bi se izbegao višestruki upit prema bazi.

# Factory Method

# Factory Method

- Definiše interfejs za kreiranje objekata, ali prepušta podklasama da odluče koju konkretnu klasu će instancirati.
- Omogućava enkapsulaciju logike kreiranja objekata, tako da klijent ne mora da zna tačne detalje kreacije.
- Koristi se kada imamo različite tipove objekata koje želimo da kreiramo na osnovu nekih uslova.
- **Ključna ideja:** Umjesto da koristimo new za kreiranje objekata direktno, koristimo fabriku koja nam vraća odgovarajući objekat.

# Problem koji Factory Method rješava

- Zamislimo da pravimo sistem za naručivanje vozila.
- Možemo naručiti auto ili motor, ali ne želimo da klijent u kodu direktno koristi new Car() ili new Motorcycle(), jer bi tada:
  - Dodavanjem novih vozila u budućnosti bi se kršio OCP.
  - Klijent bi bio zavistan od konkretnih klasa (što smanjuje fleksibilnost).
  - Kod bi bio teži za održavanje i proširenje.
  - Ako se logika pravljenja vozila mijenja, mora se mijenjati svuda gdje se koristi new. Nema centralizovane kontrole.
  - Ne može se lako zamijeniti način pravljenja objekta (npr. iz fajla, baze, mreže...), jer je new direktno u kodu.
- Factory Method rješava ovaj problem tako što odvajamo kreaciju objekata u posebnu metodu.

# Primjena Factory metode

- Sakrivamo konkretne klase od klijenta.
- Omogućavamo lako dodavanje novih klasa bez mijenjanja postojeće logike.
- Centralizujemo logiku kreiranja objekata (keširanje, konfiguracija, logovanje).
- Omogućavamo dinamičko biranje tipa objekta u runtime-u (što new ne može).

- Međutim, klijent i dalje kreira konkretne fabrike.
- Da bismo u potpunosti iskoristili Factory Method, umjesto direktnog instanciranja fabrika (new CarFactory()), koristimo registraciju fabrika ili refleksiju, čime klijent uopšte ne zna koje konkretne klase postoje.

# Praktične upotrebe Factory Method-a

- GUI Frameworks – Kreiranje dugmadi (Button), prozora (Window), obaveštenja (Notification).
- Sistemi za baze podataka – Kreiranje različitih konekcija (MySQLConnection, PostgreSQLConnection).
- Igre – Generisanje različitih neprijatelja (Orc, Troll, Dragon).
- Logger sistemi – Kreiranje različitih logger-a (FileLogger, ConsoleLogger).

# Abstract Factory

# Abstract Factory

- Nadogradnja Factory Method-a za kreiranje grupa povezanih objekata.
- Koristimo ga kada treba da kreiramo familije povezanih objekata, a da ne zavisimo od konkretnih klasa.
- Zamislimo da pravimo sistem u kojem svako vozilo ima odgovarajuće djelove (npr. sjedište i motor).
  - Car koristi "CarSeat" i "CarEngine".
  - Motorcycle koristi "MotorcycleSeat" i "MotorcycleEngine".
- Ne želimo da pomiješamo djelove, zato koristimo Abstract Factory.

# Praktične primjene Abstract Factory paterna

- Biblioteke za rad sa bazama podataka kao što su JDBC, Hibernate koriste ovaj princip.
  - Omogućava kreiranje SQL i NoSQL konekcija bez mijenjanja klijentskog koda.
- Softveri za izvještaje (JasperReports, Crystal Reports).
  - Omogućava generisanje različitih tipova dokumenata iz istog sistema.
- U sistemima koji moraju podržavati više tipova API-ja.
  - Omogućava jednostavnu zamjenu REST/SOAP/GraphQL API klijenata.
  - Biblioteke kao što su Retrofit, Spring RestTemplate koriste ovaj princip.

# Ključna razlika između Factory Method i Abstract Factory

- Factory Method – Kreira jedan tip objekta.
- Abstract Factory – Kreira čitave familije (grupe) povezanih objekata.

| Osobina | Factory Method | Abstract Factory |
|---|---|---|
| Šta kreira? | Jedan tip objekta | Grupisane povezane objekte |
| Primjer | CarFactory kreira Car | CarFactory kreira CarSeat + CarEngine |
| Dodavanje novog proizvoda | Potrebna je nova fabrika za svaki novi tip | Potrebna je nova fabrika za svaku novu porodicu |
| Kada koristiti? | Kada imamo jedan proizvod | Kada imamo više povezanih proizvoda |

# Interfejs vs apstraktna klasa

- U Factory Method paternu često koristimo apstraktne klase jer:
  - Želimo da obijezbedimo osnovnu implementaciju (npr. zajedničke metode za sve podklase).
  - Podklase nasljeđuju logiku i mogu redefinisati samo ključne djelove.
  - Factory Method može imati konkretnu logiku u bazičnoj klasi, dok samo kreiranje objekta ostaje apstraktno.
- U Abstract Factory paternu često koristimo interfejse jer:
  - Želimo da definišemo skup metoda koje konkretne fabrike moraju implementirati.
  - Ne postoji zajednička implementacija koja bi zahtijevala apstraktnu klasu.
  - Interfejsi omogućavaju veću fleksibilnost (klase mogu nasljeđivati druge klase dok implementiraju interfejs).

# Builder

# Builder

- Razdvaja konstrukciju kompleksnih objekata od njihove reprezentacije.
- Kada imamo klasu sa mnogo atributa, posebno kada su neki opcioni.
- Kada želimo čitljiviji i fleksibilniji kod umjesto konstruktora sa 10+ parametara.

# Primjer

- Zamislimo da pravimo sistem za restoran u kojem korisnici mogu kreirati narudžbinu (Order), koja sadrži:
  - Glavno jelo (obavezno)
  - Piće (opciono)
  - Desert (opciono)
  - Dodatne priloge (lista priloga)
  - Napomena za kuvara (opciono)
- Upotrebom Builder paterna dobijamo
  - Čist i čitljiv kod – nema dugih lista parametara u konstruktoru.
  - Jednostavno dodavanje novih opcija (npr. "Dodaj sos") bez mijenjanja konstruktora.
  - Podršku za opcione atribute – ne moramo definisati nepotrebne null vrednosti.
  - Laku ekstenziju – možemo lako dodati još funkcionalnosti, npr. "Dodaj duplo meso".

# Primjene Builder paterna u realnim aplikacijama

- U bibliotekama za rad sa HTTP zahtjevima, kao što su OkHttp (Java/Kotlin) i Requests (Python).
  - Omogućava lakše konfigurisanje zahtjeva bez preopterećivanja konstruktora sa previše parametara.
  - Omogućava lakše dodavanje opcionalnih parametara (npr. header-a, HTTP metoda).
- U ORM alatima poput Hibernate-a ili Query DSL-ova.
  - Omogućava fleksibilno kreiranje složenih SQL upita bez ručnog sastavljanja stringova.
  - Omogućava dinamičko kreiranje SQL upita bez ručnog sastavljanja stringova.
  - Bez Builder-a, morali bismo da pišemo dugačke SQL stringove, što je podložno greškama.
- Kreiranje objekata u Android aplikacijama
  - U kreiranju Android AlertDialog-a i Notification objekata.

# Primjer korišćenja Builder paterna u OkHttp-u (Java/Kotlin)

```
OkHttpClient client = new OkHttpClient();


Request request = new Request.Builder()
        .url("https://api.example.com/data")
        .header("Authorization", "Bearer TOKEN")
        .get()
        .build();


Response response = client.newCall(request).execute();
```

# Kreiranje SQL upita (ORM biblioteke kao Hibernate)

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Employee> query = cb.createQuery(Employee.class);
Root<Employee> root = query.from(Employee.class);

query.select(root)
    .where(cb.equal(root.get("department"), "IT"))
    .orderBy(cb.desc(root.get("salary")));

List<Employee> results = entityManager.createQuery(query).getResultList();
```

# AlertDialog u Androidu

```
AlertDialog dialog = new AlertDialog.Builder(context)
        .setTitle("Confirm")
        .setMessage("Are you sure you want to delete?")
        .setPositiveButton("Yes", (dialogInterface, i) -> { /* Handle Yes */ })
        .setNegativeButton("No", (dialogInterface, i) -> dialogInterface.dismiss())
        .create();
dialog.show();
```

# Builder vs setters

- Zašto koristiti Builder pattern umjesto setter metode u klasi?
- Validacija i kontrola
  - Builder omogućava validaciju podataka prije nego što objekat bude izgrađen.
  - Provjerava da li su svi obavezni atributi postavljeni i da li su podaci validni.
- Sprječavanje kreiranja "polu-kreiranih" objekata
  - Korišćenjem Builder-a sprječavamo rizik od "polu-kreiranih" objekata.
  - Samo potpuno formirani objekat može biti korišćen.
- Imutabilnost
  - Builder omogućava kreiranje imutabilnih objekata.
  - Nakon izgradnje objekta, njegovi atributi ne mogu biti mijenjani.

# Prototype

# Prototype

- Kreira nove objekte kloniranjem postojećih instanci umesto direktnim instanciranjem.
- Štedi resurse kada je kreiranje objekta skupo ili kompleksno.
- Omogućava pravljenje različitih verzija objekata iz postojećeg prototipa.
- Alternativa Factory-u kada želimo brzo kopiranje objekata.

# Kada koristiti Prototype patern?

- Kada je kreiranje objekta skupa operacije (npr. čitanje fajla, povezivanje sa bazom podataka).
- Kada imamo kompleksne objekte sa mnogo parametara.
- Kada želimo brzo i efikasno kopiranje objekata.

# Primjer

- Zamislimo da imamo klasu "Document" sa mnogo podataka:
  - Ima naziv, tip, veličinu i autora.
  - Može sadržati kompleksne objekte poput lista slika ili tabela.
- Ako želimo da napravimo kopiju dokumenta, mogli bismo kreirati novi objekat i ručno prepisati sve podatke, ali to može biti sporo i neefikasno.
- Prototype omogućava da jednostavno "kloniramo" postojeći objekat.

# Primjene Prototype paterna u realnim aplikacijama

- Sistemi za dokumente i grafikone – Kopiranje dokumenata, prezentacija, slika.
- Igre – Kloniranje neprijatelja, efekata, likova.
- Upravljanje memorijom – Brzo pravljenje kopija velikih objekata umesto rekreiranja.
- Konfiguracije sistema – Kreiranje više verzija istog osnovnog podešavanja.

# Kršenje SOLID principa bez upotrebe kreacionih paterna

- Single Responsibility Principle (SRP) - Princip jedne odgovornosti
  - Na primjer, ako neka klasa sama instancira new DatabaseConnection(), onda nije odgovorna samo za poslovnu logiku, već i za način na koji se konektuje na bazu.
- Open/Closed Principle (OCP) - Otvorenost za proširenja, zatvorenost za izmjene
  - Ako direktno koristimo new za kreiranje objekata, svaki put kada dodamo novi tip objekta, moramo mijenjati kod.
  - Na primjer, ako imamo klasu koja kreira MySQLDatabaseConnection, a kasnije treba da dodamo PostgreSQLDatabaseConnection, moramo mijenjati sve djelove koda gde je new MySQLDatabaseConnection().
- Liskov Substitution Principle (LSP) - Zamjena podklasa baznom klasom
  - Ako koristimo konkretne klase umjesto apstrakcija, ne možemo lako zamijeniti jednu implementaciju drugom.
  - Na primer, ako koristimo new MySQLDatabaseConnection() umjesto DatabaseConnection, ne možemo lako preći na PostgreSQL.

# Strukturni Paterni

# Uvod

- Strukturalni paterni opisuju kako da klase i objekti sarađuju i formiraju veće strukture koje su fleksibilne i lake za održavanje.
- Prednosti korišćenja strukturalnih paterna:
  - Jasnije odvajanje odgovornosti
  - Smanjena zavisnost između klasa
  - Lakša zamjena i proširivanje djelova sistema
  - Poboljšana čitljivost i organizacija koda

# Vrste strukturalnih paterna

- Facade
- Adapter
- Decorator
- Composite
- Proxy
- Bridge
- Flyweight

# Facade

# Facade

- Facade (fasada) je strukturni design pattern koji:
  - Pruža pojednostavljen interfejs ka kompleksnom sistemu,
  - Sakriva složenost iza jednog "ulaza" (interfejsa),
  - Čini sistem lakšim za korišćenje klijentima.
- Umjesto da klijent koristi mnogo klasa i metoda, koristi samo jednu Facade klasu koja poziva sve što treba "iza kulisa".

# Primjer – Smart home sistemi

- Imamo različite komponente:
  - Svijetla
  - TV
  - Zvučnici
  - Klima
- Bez Facade paterna, klijent mora da pozove sve ove klase direktno.
- Sa Facade paternom, sve ide preko jedne metode, npr. startMovieNight().

# Primjene Facade paterna u realnim aplikacijama

- Spring Framework – JdbcTemplate, RestTemplate
  - Sakrivaju složenost poziva baze ili HTTP komunikacije.
- Hibernate – Session API
  - Omogućava jednostavan interfejs za rad sa bazom podataka.
- Multimedija
  - Video plejeri, render engines (npr. start, pause, mute...).

# Adapter

# Adapter

- Adapter omogućava da dve klase koje inače nisu kompatibilne rade zajedno, tako što ih "spaja" kroz zajednički interfejs.
- Koristi se kada želite da koristite postojeću klasu, ali njen interfejs nije ono što vaš kod očekuje.
- Struktura:
  - Target – interfejs koji sistem očekuje
  - Adaptee – već postojeća klasa (nespojiva)
  - Adapter – most koji prevodi pozive sa Target na Adaptee

# Primjene Adapter paterna u realnim aplikacijama

- Prilikom korišćenja zastarjelih biblioteka koje se ne mogu mijenjati (legacy code).
- U API wrapperima kada treba koristi treći API čiji objekti nisu kompatibilni sa vašim sistemom.
- U GUI-u kada se koriste različite kontrole iz više biblioteka
- U testiranju kad pravite mock klase koje moraju imati isti interfejs kao pravi objekti.

# Primjer - nekompatiblini API

- Recimo da API vraća ovaj format:
  - { "full_name": "John Doe",  "email_address": "john@example.com"}
- Ali naša aplikacija koristi User model sa metodama getName() i getEmail().

# Decorator

# Decorator

- Decorator je strukturalni patern koji omogućava dodavanje funkcionalnosti objektu u runtime-u.
- Pomaže u poštovanju Open/Closed principa.
- Koristi se:
  - Kada želimo dodati ponašanje samo nekim objektima, ne svima iz klase
  - Kada želimo kombinovati više dodatnih osobina na fleksibilan način
  - Kada nasljeđivanje nije opcija ili bi bilo previše komplikovano

# Primjene Decorator paterna u realnim aplikacijama

- Java I/O - BufferedReader, InputStreamReader, itd.
- GUI elementi - dodavanje ponašanja komponentama.
- Loggeri - dodavanje različitih nivoa logovanja.
- Data validation - proširivanje validacije sloj po sloj.

# Composite

# Composite

- Composite je strukturalni dizajn paterni koji omogućava da se pojedinačni objekti i grupe objekata tretiraju na isti način.
- Koristi se za hijerarhije objekata gdje i "listovi" (npr. pojedinačni elementi) i "čvorovi" (npr. grupe elemenata) treba da se ponašaju isto.
- Pojednostavljuje klijentski kod – klijent ne mora da zna da li radi s listom ili čvorom.
- Odličan za hijerarhijske strukture: UI komponente, organizacioni dijagrami, fajl sistemi, meniji...

# Proxy

# Proxy

- Proxy je strukturalni dizajn patern koji pruža zamjenski objekat za neki drugi objekat.
- Koristi se kada želimo:
  - da kontrolišemo pristup pravom objektu
  - da dodamo lazy loading, caching, logovanje, autentikaciju, itd.
  - da olakšamo rad sa kompleksnim ili skupim objektima
- Elementi:
  - Subject – zajednički interfejs za RealSubject i Proxy
  - RealSubject – stvarna implementacija
  - Proxy – sadrži referencu na RealSubject i kontroliše pristup

# Vrste proxy-a

- Virtual proxy – učitavanje velikih slika ili fajlova tek kada su potrebni
- Protection proxy – za kontrolu pristupa
- Remote proxy – kod RMI (Remote Method Invocation), kad pristupamo objektu na drugoj mašini
- Smart reference – automatski dodaje logovanje, brojač referenci, itd.

# Primjeri

- Zamislimo da imamo dokument koji se može čitati, ali ne želimo da svi mogu da ga uređuju osim ako nisu admini. Proxy će to da kontroliše.
- Imamo veliki broj slika koje su teške za učitavanje (visoka rezolucija, mrežno preuzimanje itd.). Ne želimo da ih sve odmah učitamo – umjesto toga, koristimo proxy da ih predstavimo i učitamo tek kad korisnik klikne da ih vidi.

# Bihevioralni paterni

# Uvod

- Behavioralni (ponašajni) paterni fokusiraju se na komunikaciju i saradnju između objekata.
- Oni definišu kako objekti međusobno komuniciraju, kako prenose informacije i kako delegiraju odgovornosti – bez da su čvrsto povezani.

# Ciljevi bihevioralnih paterna

- Omogućiti fleksibilnu razmjenu poruka između objekata
- Smanjiti zavisnosti između klasa (low coupling)
- Promovisati odvajanje odgovornosti
- Olakšati promjene ponašanja bez izmjene postojećeg koda (Open/Closed princip)

# Vrste bihevioralnih paterna

- Strategy
- State
- Observer
- Command
- Visitor
- Template Method
- Iterator
- Mediator
- Interpreter
- Chain of Responsibility
- Memento

# Strategy

# Strategy

- Primjenjuje se kada ima više različitih algoritama ili ponašanja, a želimo da dinamički biramo koje ćemo koristiti, bez mijenjanja klijentskog koda.
- Definiše porodicu algoritama (strategija), enkapsulira ih, i učini ih zamjenjivim.
- Klijent koristi strategiju bez znanja kako ona radi.
- Elementi Strategy paterna:
  - Strategy (interfejs) - zajednički interfejs za sve algoritme
  - ConcreteStrategy - konkretna implementacija algoritma
  - Context - klasa koja koristi objekat strategije

# Primjene Strategy paterna

- Metode plaćanja
- Algoritmi sortiranja
- Validacije inputa
- Logovanje (npr. u fajl, konzolu, bazu...)
- Promjene UI ponašanja u zavisnosti od moda korisnika

# State

# State

- State pattern omogućava objektu da promijeni svoje ponašanje kada se njegovo unutrašnje stanje promijeni.
- Objekat će izgledati kao da mijenja svoju klasu.
- Umjesto da korišćenja if-else logike svuda, ponašanje se enkapsulira u state klase.

# Primjer

- Igrač koji može biti:
  - u normalnom stanju (NormalState)
  - nevidljiv (InvisibleState)
  - nepobjediv (InvincibleState)
- Svako stanje ima različito ponašanje za npr. move() i attack() metode.

# State vs Strategy

- State i Strategy paterni zgledaju slično na prvi pogled – oba koriste interfejse i enkapsuliraju ponašanje.
- Ali u suštini, imaju različitu svrhu i kontekst.

| Aspekt | State | Strategy |
|---|---|---|
| Svrha | Promjena ponašanja objekta **u zavisnosti od njegovog stanja** | Odabir algoritma/ponašanja **u zavisnosti od potrebe klijenta** |
| Promjena | Objekat **sam mijenja stanje** tokom života | Klijent **određuje strategiju** |
| Modelira | **Interno ponašanje** objekta | **Spoljno ponašanje** objekta (algoritam, način rada...) |
| Ko bira implementaciju? | Sam objekat dinamički | Klijent eksplicitno |

# Observer

# Observer

- Observer je behavioral dizajn patern koji definiše jedan-na-više odnos između objekata tako da kada se jedan objekat promijeni, svi koji ga posmatraju (prate) budu automatski obaviješteni i ažurirani.
- Koristi se:
  - Kada imamo subjekt (Subject) koji više drugih objekata mora pratiti.
  - Kada treba automatski obavještavati više komponenti o promeni stanja.
  - Prava stvar za event-sisteme, GUI, notifikacije, stock ticker-e...

# Elementi Observer paterna

- Subject  - objekat koji sadrži stanje. Ima metodu za prijavu, odjavu i obavještavanje posmatrača.
- Observer - interfejs koji implementiraju svi posmatrači.
- ConcreteObserver - klase koje reaguju na promene u Subject-u.
- ConcreteSubject  - klasa koja čuva stanje i šalje notifikacije kada se promijeni.

# Primjena Observer paterna u realnim aplikacijama

- GUI komponente: dugmad, scroll bar-ovi – svi se registruju za promene.
- Stock aplikacije: kada se promeni cijena, svi grafovi se ažuriraju.
- Event bus-ovi i Event-driven sistemi.
- News aplikacije (push notifikacije).

# Command

# Command

- Command Pattern je behavioralni patern koji omogućava da se sve akcije (komande) enkapsuliraju kao objekti.
- Time se omogućava da te komande budu prosleđivane, izvršavane, poništavane ili stavljene u red čekanja.
- Ovaj pattern je vrlo koristan u situacijama kada želimo da razdvojimo izvršenje određenih akcija od njihovog poziva, kao i u aplikacijama koje zahtevaju undo/redo funkcionalnost.

# Elementi Command Paterna

- Command - apstraktna komanda koja sadrži metodu execute(), koja se kasnije implementira u konkretnoj komandi.
- ConcreteCommand - konkretne implementacije komandi koje pozivaju odgovarajuće akcije na odgovarajućim objektima.
- Invoker - objekat koji poziva komande. To je objekat koji traži izvršenje određenih komandi.
- Receiver - objekat koji zapravo izvršava akciju. Receiver je objekat na kojem se izvršava konkretna logika (npr. neko dugme koje se pritisne).
- Client - objekat koji kreira komande i povezuje ih sa Invoker-om.

# Prednosti Command paterna

- Decentralizacija izvršenja
  - Invoker ne mora da zna ništa o implementaciji akcija, on samo poziva execute() metodu.
- Undo/Redo funkcionalnost
  - Moguće je implementirati mogućnost poništavanja ili ponovnog izvršenja akcija (ovo je vrlo popularno u editorima teksta, grafičkim alatima, itd.).
- Lako proširiv
  - Nove komande se lako dodaju bez promjena u postojećim klasama.

# Primjene Command paterna u realnim aplikacijama

- Daljinski upravljači
  - Daljinski upravljači mogu koristiti Command Pattern da enkapsuliraju različite akcije (turn on, turn off) kao komande.
- Sistemi za obradu narudžbina
  - U e-trgovini, kada korisnik naručuje proizvod, komande mogu predstavljati različite akcije (dodavanje u korpu, plaćanje, slanje).
- Undo/Redo u tekst editorima
  - Svaka akcija (npr. unos teksta, brisanje, formatiranje) može biti predstavljena kao komanda koja se može poništiti ili ponoviti.

# Visitor

# Iterator

# Template Method