Lecture 9

Design Patterns (part 1)

Topics covered

- Introduction to design patterns
- Creational design patterns
  - Singleton
  - Factory, Abstract Factory
  - Builder
  - Prototype
- Structural design patterns
  - Adapter
  - Composite
  - Decorator
  - Proxy

Design patterns

Introduction to Design Patterns

What is design pattern?

✧ Solutions to general problems faced in software development

✧ Industry standard approach to solve recurring problems

✧ Programming-language-independent strategies for solving the common object-oriented design problems

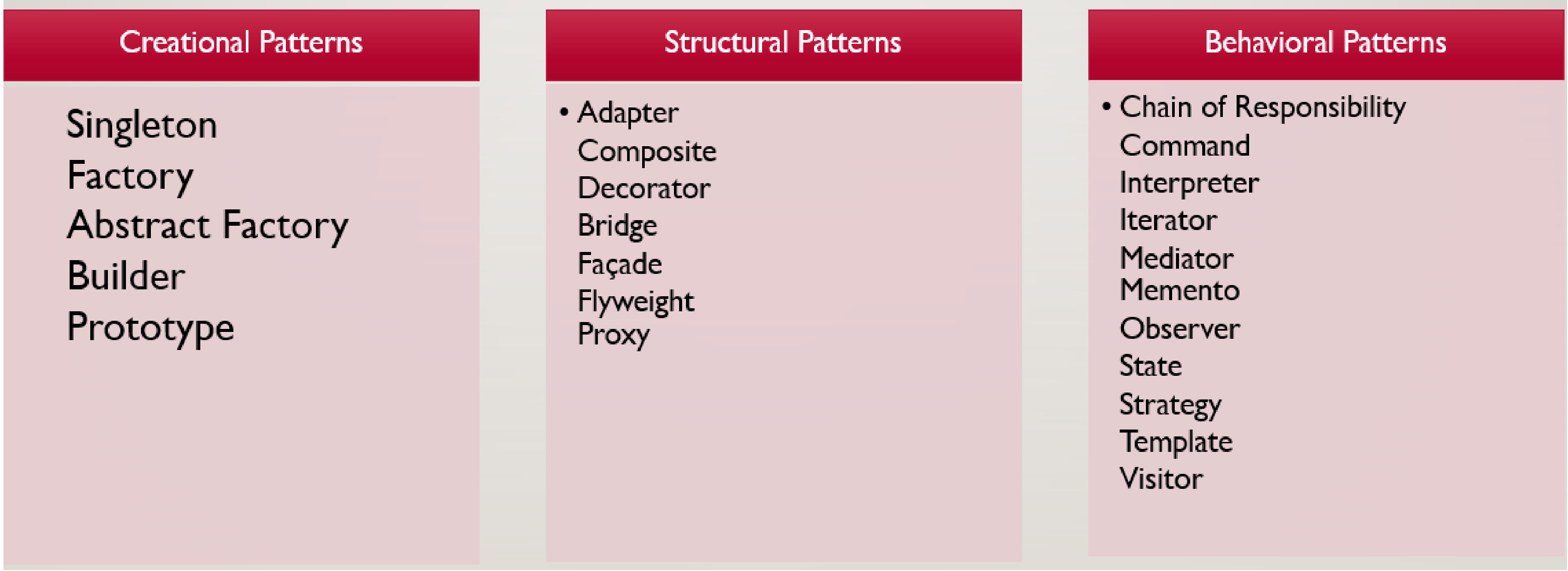
Why do we need design pattern?

- ✧ Promotes reusability: more robust and maintainable code
- ✧ Faster development, code easier to understand, debug
- ✧ Provides standard terminology
- ✧ Provides the solutions that help to define the system architecture
- ✧ Captures the software engineering experiences
- ✧ Provides transparency to the design of an application

When should we use design patterns?

- During the analysis & design phase of SDLC
- Design patterns ease the analysis & design phase of SDLC by providing information based on prior hands-on experiences

3 main types of design patterns



Creational Patterns

Singleton

Factory

Abstract Factory

Builder

Prototype

Structural Patterns

- Adapter

Composite

Decorator

Bridge

Façade

Flyweight

Proxy

Behavioral Patterns

- Chain of Responsibility

Command

Interpreter

Iterator

Mediator

Memento

Observer

State

Strategy

Template

Visitor

Creational Design Patterns

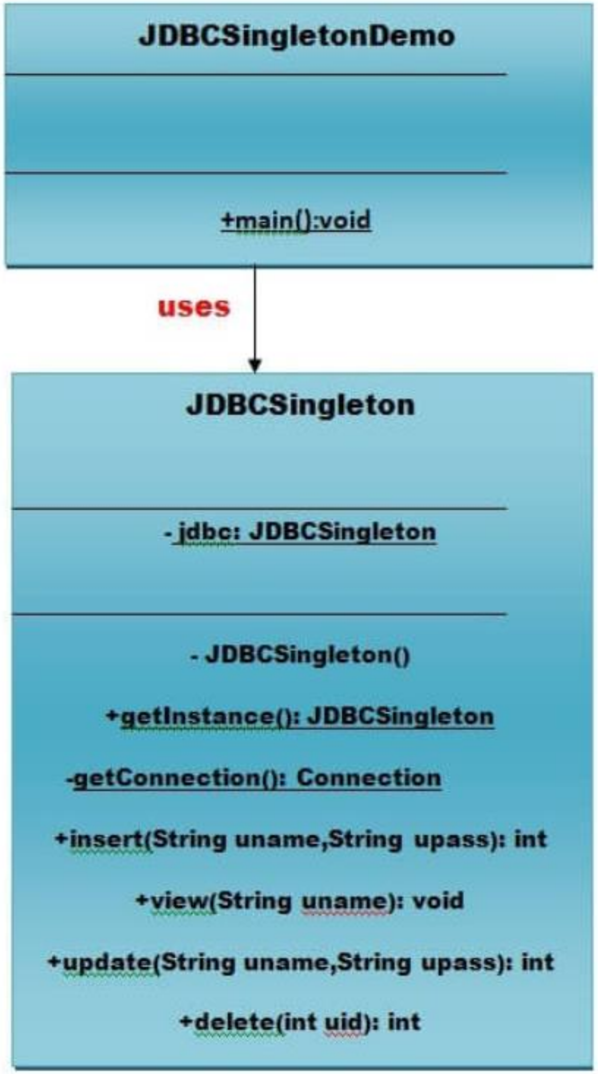
About creational patterns

- ✧ Are concerned with the way of creating objects
- ✧ Are used when a decision must be made at the time of instantiation of a class
- ✧ Way to create objects while hiding logic
- ✧ Do not use the `new` operator
- ✧ Offer more flexibility in deciding which objects need to be created for a given case
- ✧ Solution to instantiate an object in the best possible way for the situation

Singleton pattern

- Defines a class that has only *one instance* and provides a global point of access to it.
  - A class must ensure that only single instance should be created and single object can be used by all other classes.
- There are 2 main forms:
  - Early Instantiation: creation of instance at load time.
  - Lazy Instantiation: creation of instance when required.
- Usage:
  - Is mostly used in multi-threaded and database applications.
  - Is used in logging, caching, thread pools, configuration settings

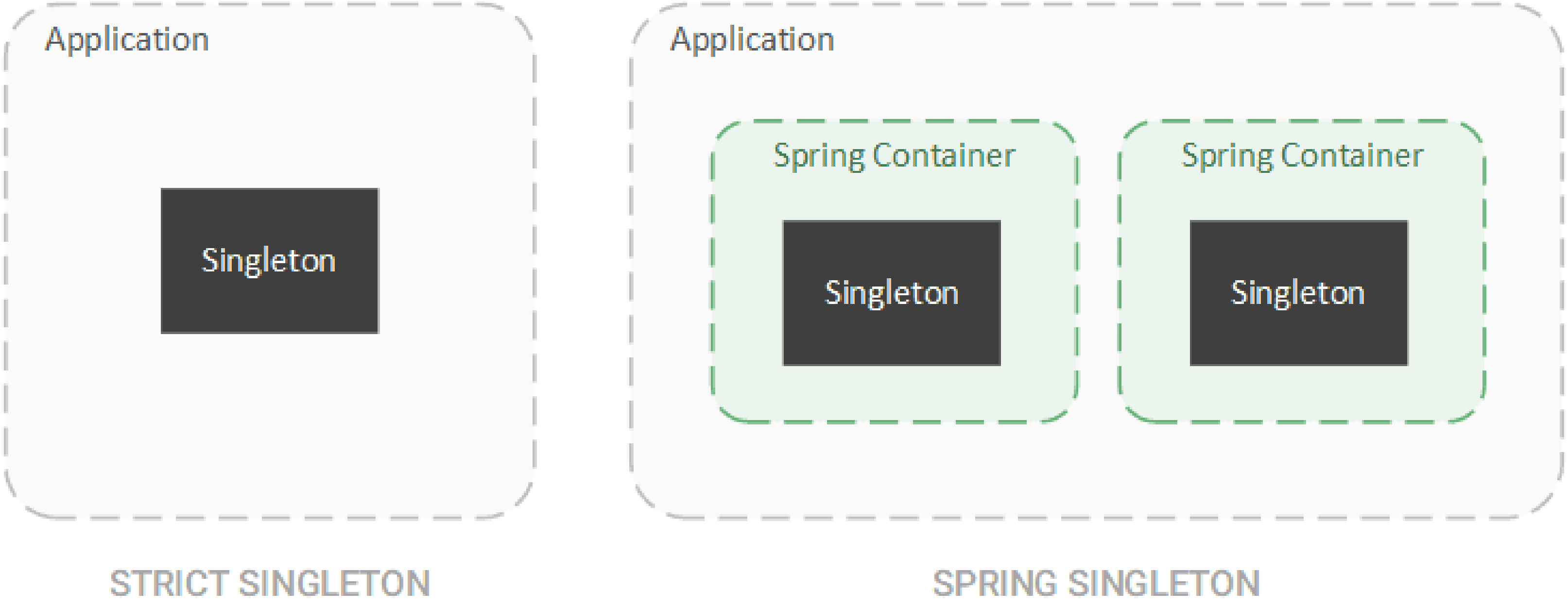
Singleton pattern example


JDBCSingletonDemo
+main():void
uses
JDBCSingleton
- jdbc: JDBCSingleton
- JDBCSingleton()
+getInstance(): JDBCSingleton
-getConnection(): Connection
+insert(String uname,String upass): int
+view(String uname): void
+update(String uname,String upass): int
+delete(int uid): int

Singleton pattern in Spring framework

✧ Spring Beans are singleton objects

- Spring restricts a singleton to one object per Spring IoC container
- This means Spring will only create one bean for each type per application context.



Spring Beans example

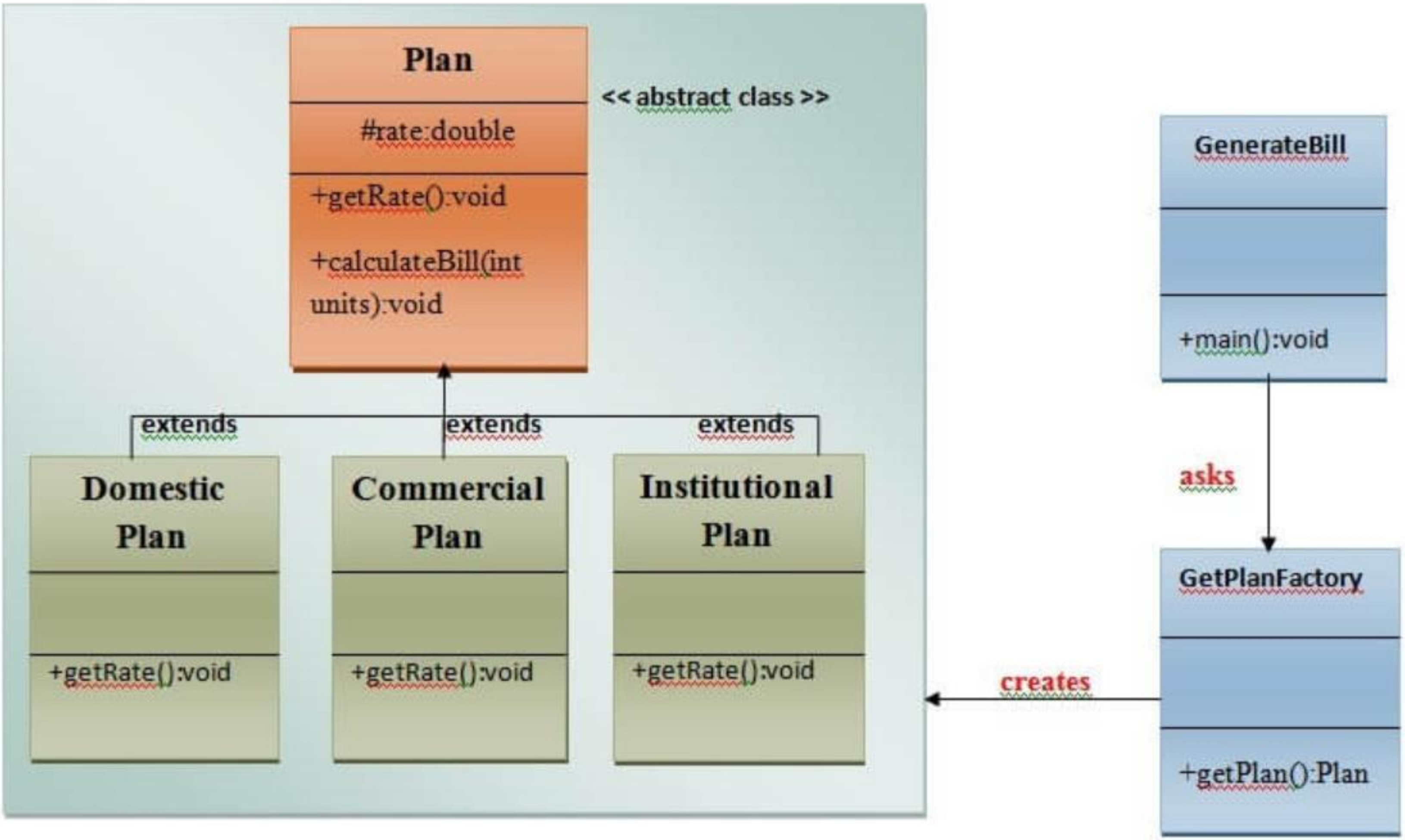
```
@RestController
public class LibraryController {
    @Autowired
    private BookRepository repository;
    @GetMapping("/count")
    public Long findCount() { return repository.count(); }
}
```

```
@RestController
public class BookController {
    @Autowired
    private BookRepository repository;
    @GetMapping("/book/{id}")
    public Book findById(@PathVariable long id) {
        return repository.findById(id).get();
    }
}
```

Factory pattern

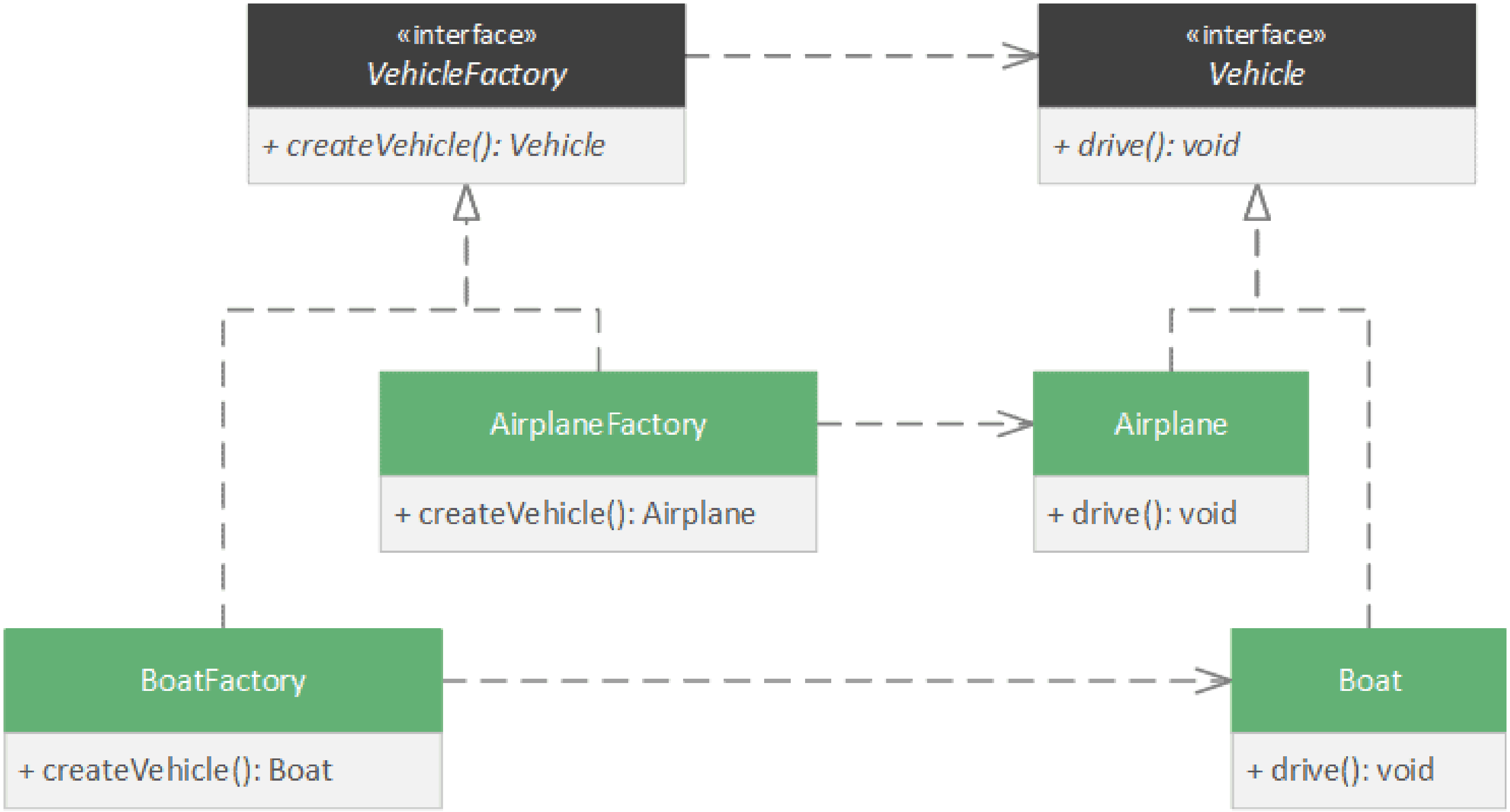
- ✧ Main idea: decouple the creation from the consumption
  - Allow more flexibility on what type to create, and how to create
  - Refer to newly created object using a common interface
  - Is used to define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate
- ✧ Usage:
  - When a class doesn't know what sub-classes to create
  - When a class wants that its sub-classes specify the objects to be created
  - When the parent classes choose the creation of objects to its sub-classes

Factory pattern example



Example: Electricity Bill

Factory pattern example



Example: Vehicle

Spring Application Context

- An Appllication Context in Spring is also a bean container, which is a factory that produces beans
  - Each of the `getBean` methods is considered a factory method

```
public interface BeanFactory {
    Object getBean(String name) throws BeansException;
    <T> T getBean(Class<T> requiredType) throws BeansException;
    boolean containsBean(String name);
    boolean isSingleton(String name) throws NoSuchBeanDefinitionException;
    boolean isPrototype(String name) throws NoSuchBeanDefinitionException;
}
```

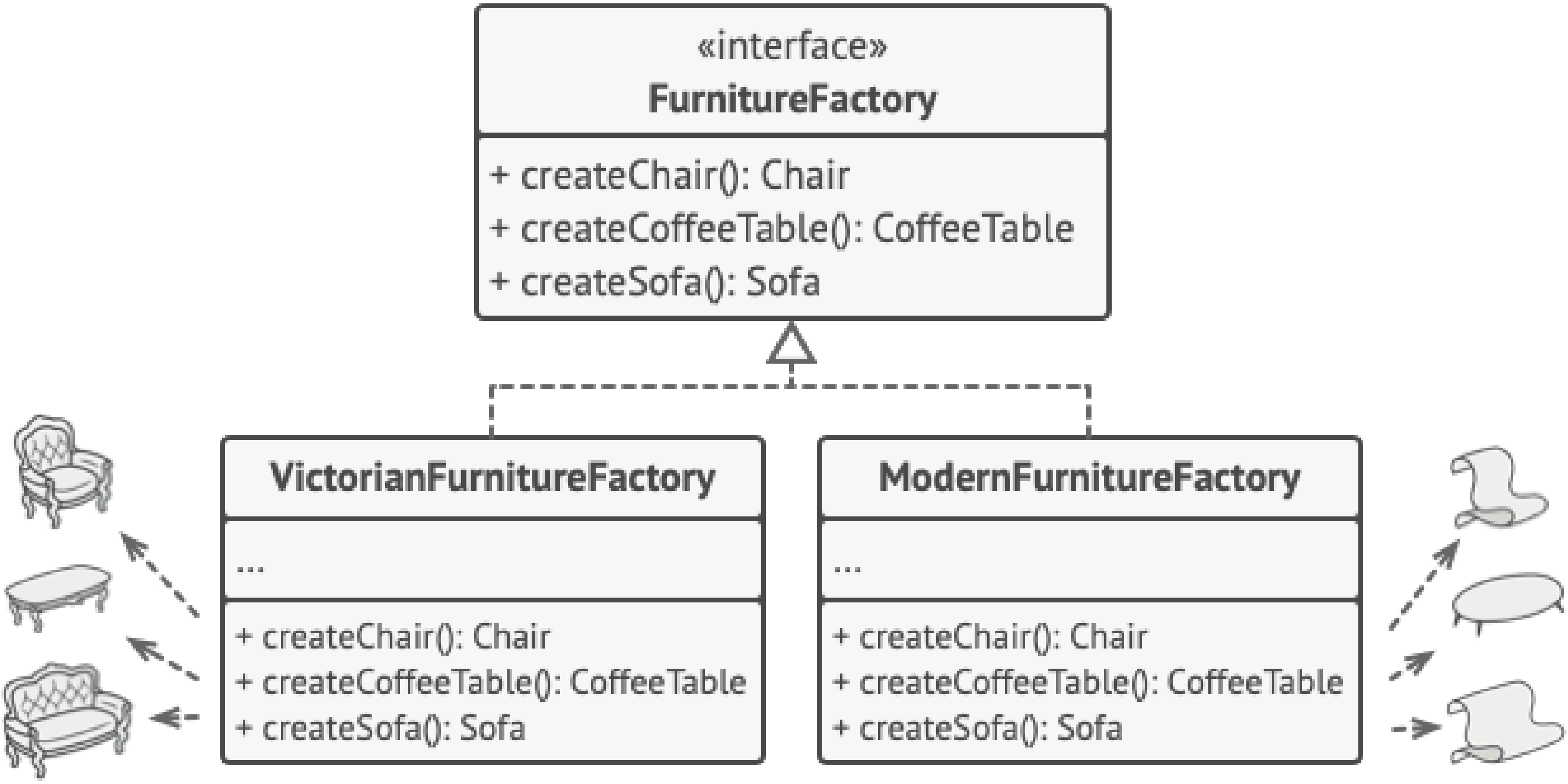
```
public interface ApplicationContext
        extends EnvironmentCapable, ListableBeanFactory,
        HierarchicalBeanFactory, MessageSource,
        ApplicationEventPublisher, ResourcePatternResolver {
    // ...

}
```

Abstract factory pattern

- ✧ Defines an interface or abstract class for creating families of related (or dependent) objects but without specifying their concrete sub-classes.
- ✧ Lets a class be a factory of a family of classes
- ✧ Usage:
  - When the system needs to be independent of how its object are created, composed, and represented.
  - When the family of related objects has to be used together, then this constraint needs to be enforced.
  - When you want to provide a library of objects that does not show implementations and only reveals interfaces.

Abstract factory pattern example



«interface»
FurnitureFactory

+ createChair(): Chair
+ createCoffeeTable(): CoffeeTable
+ createSofa(): Sofa

VictorianFurnitureFactory

...

+ createChair(): Chair
+ createCoffeeTable(): CoffeeTable
+ createSofa(): Sofa

ModernFurnitureFactory

...

+ createChair(): Chair
+ createCoffeeTable(): CoffeeTable
+ createSofa(): Sofa

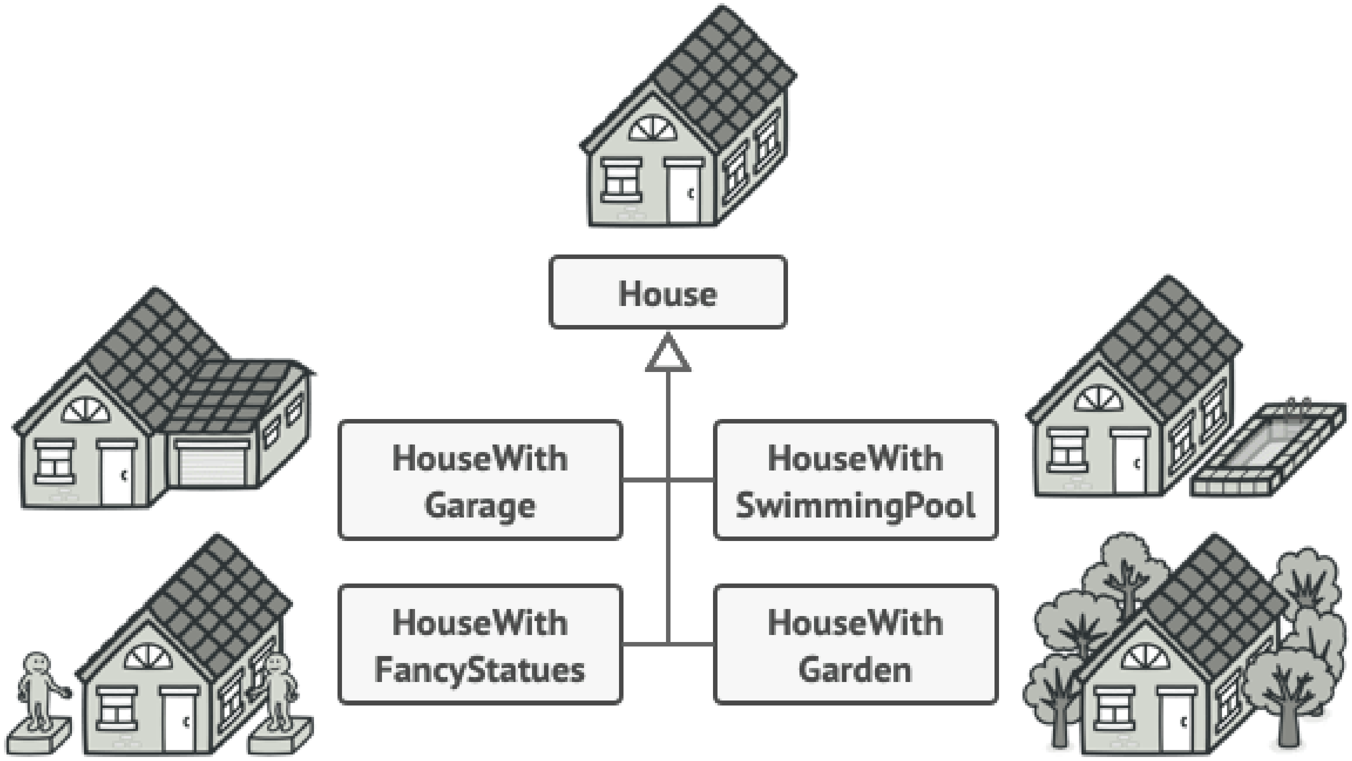
Example: Furniture

Builder pattern

- ✧ Constructs a complex object from simple objects using step-by-step approach
- ✧ Is mostly used when object can't be created in single step like in the de-serialization of a complex object
- ✧ Advantages:
  - Provides clear separation between the construction and representation of an object
  - Provides better control over construction process
  - Supports to change the internal representation of objects

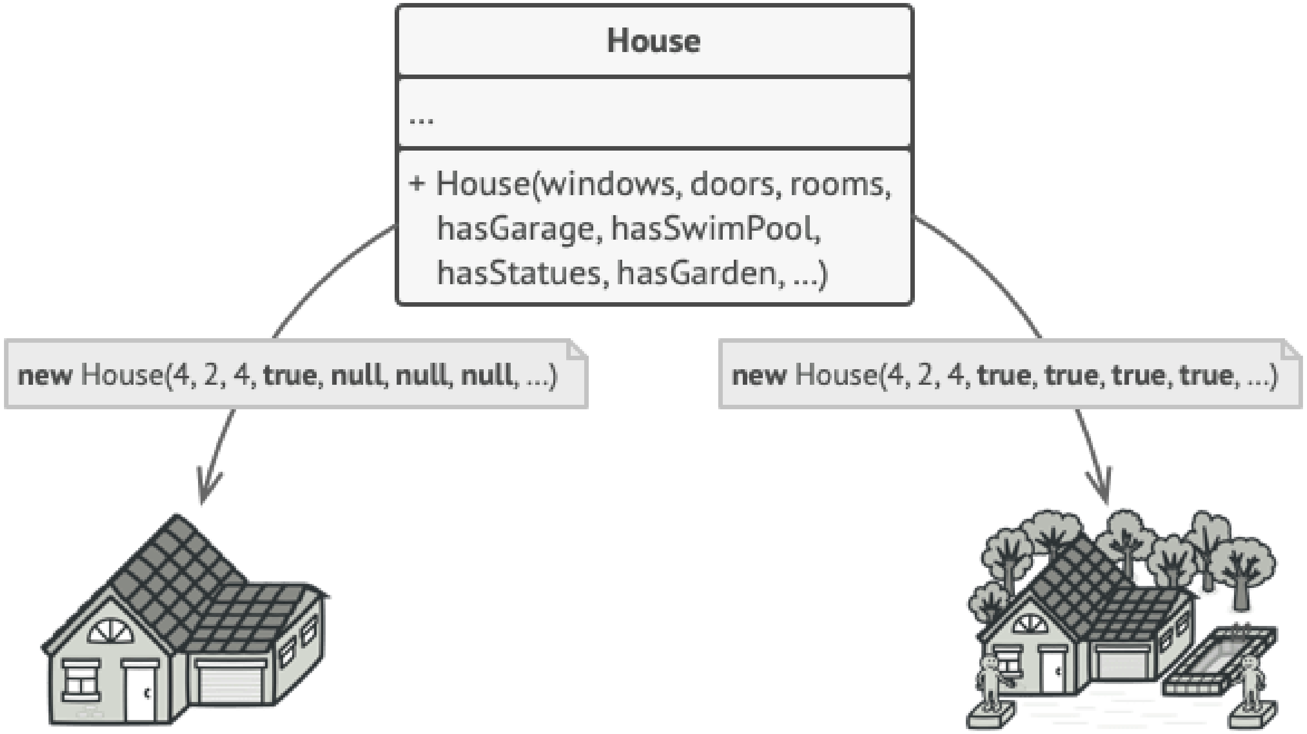
Builder pattern motivation

✧ Imagine having to create many houses with different parts



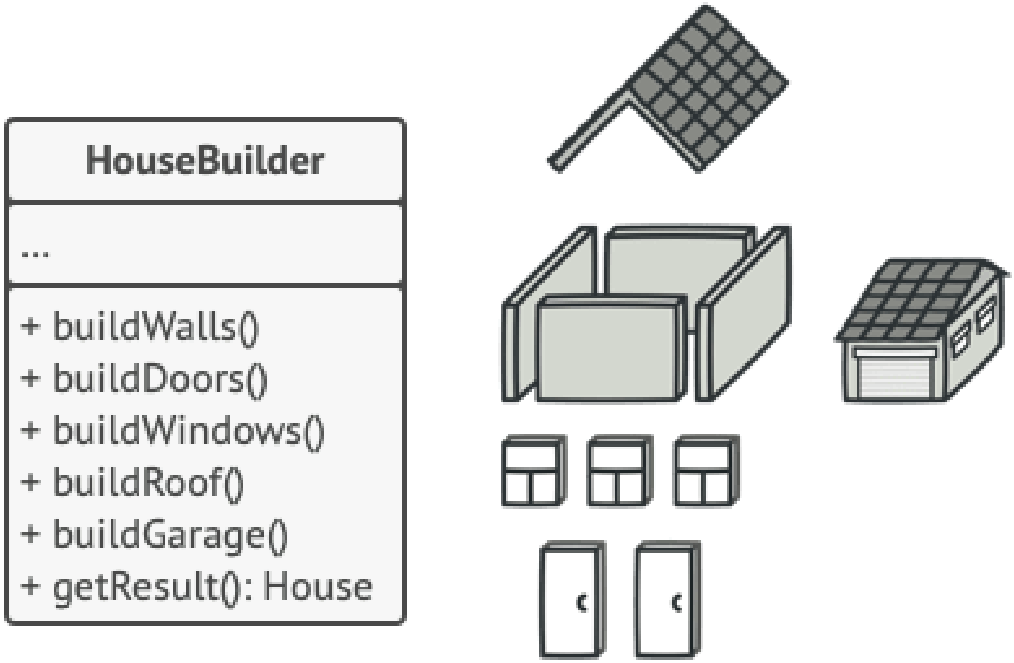
Builder pattern motivation

- The constructor with lots of parameters can be a bad idea, it makes the constructor difficult to use (and ugly)


House
...
+ House(windows, doors, rooms, hasGarage, hasSwimPool, hasStatues, hasGarden, ...)
new House(4, 2, 4, true, null, null, null, ...)
new House(4, 2, 4, true, true, true, true, ...)

Builder pattern example

- Builder pattern lets you construct complex objects in a *step-by-step* manner.



HouseBuilder

...

+ buildWalls()
+ buildDoors()
+ buildWindows()
+ buildRoof()
+ buildGarage()
+ getResult(): House

Example: HouseBuilder

Builder pattern in Spring framework

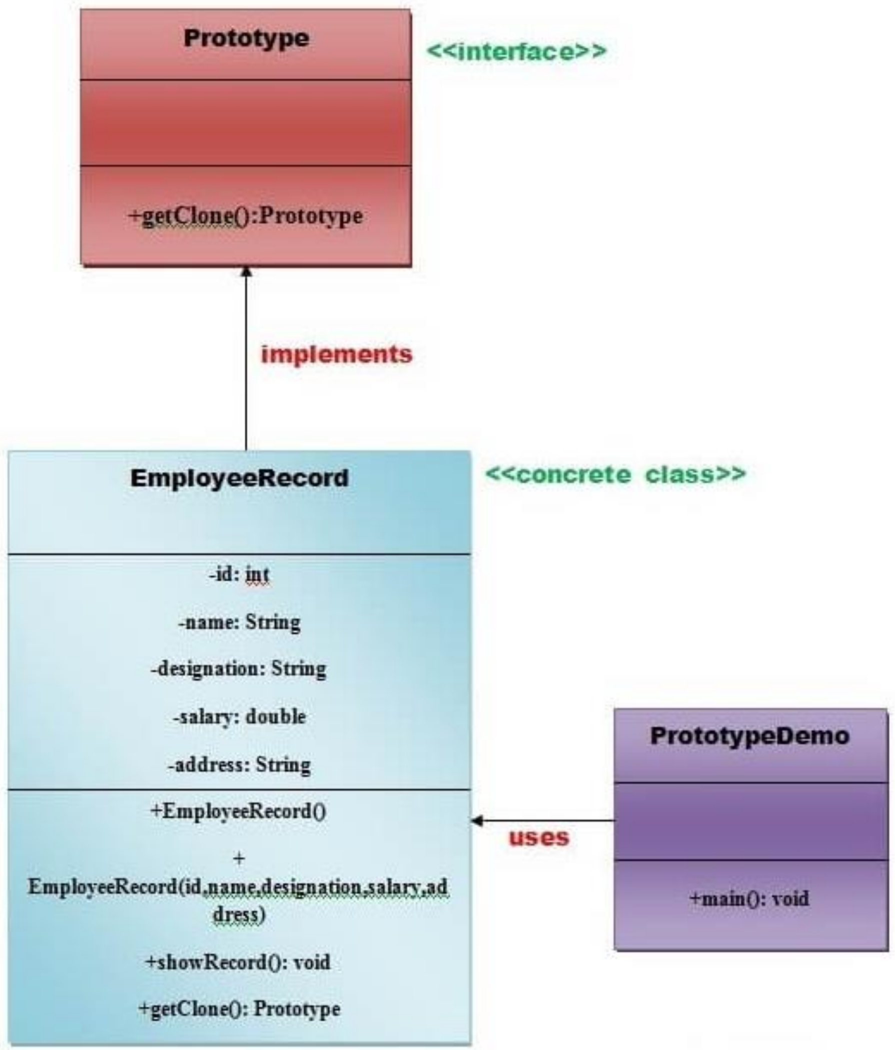
- A good example of builder pattern is the configuration of the `HttpSecurity` object in Spring Security.

```
http
.userDetailsService(myUserDetailsService)
.authorizeHttpRequests(req -> req
        .requestMatchers("/").permitAll()
        .requestMatchers("/register").permitAll()
        .anyRequest().authenticated()
)
.formLogin(Customizer.withDefaults())
.build();
```

Prototype pattern

- ✧ Make clones of an existing object instead of creating new ones
  - ▪ Objects can also be customized as per the requirement
- ✧ Should be followed, if the cost of creating a new object is expensive and resource intensive
- ✧ Usage:
  - ▪ When the classes are instantiated at runtime
  - ▪ When the cost of creating an object is expensive or complicated
  - ▪ When you want to keep the number of classes in an application minimum
  - ▪ When the client application needs to be unaware of object creation and representation

Prototype pattern example



Example: Employee

Structural Design Patterns

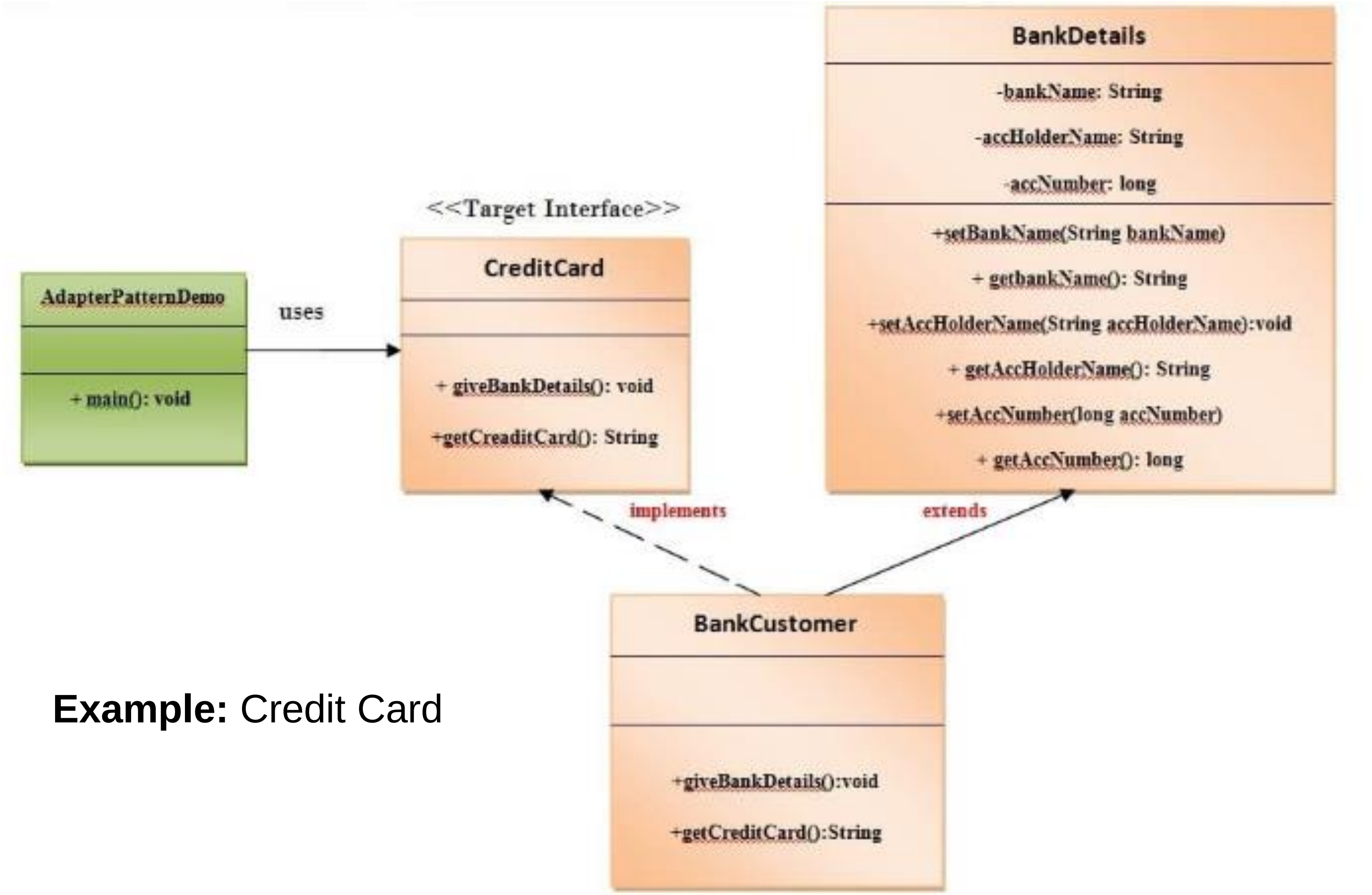
About structural patterns

- ✧ Provides different ways to create a class structure
- ✧ Uses inheritance to compose interfaces & define ways to create objects to get new functionality
- ✧ Is concerned with how classes and objects can be composed, to form larger structures
- ✧ Simplifies the structure by identifying the relationships
- ✧ Focuses on how the classes inherit from each other and how they are composed from other classes

Adapter pattern (a.k.a Wrapper)

- ✧ Converts the interface of a class into another interface that a client wants
- ✧ Provides the interface according to client requirement while using the services of a class with a different interface
- ✧ Usage:
    - ▪ When an object needs to utilize an existing class with an incompatible interface.
    - ▪ When you want to create a reusable class that cooperates with classes which don't have compatible interfaces.

Adapter pattern example



Example: Credit Card

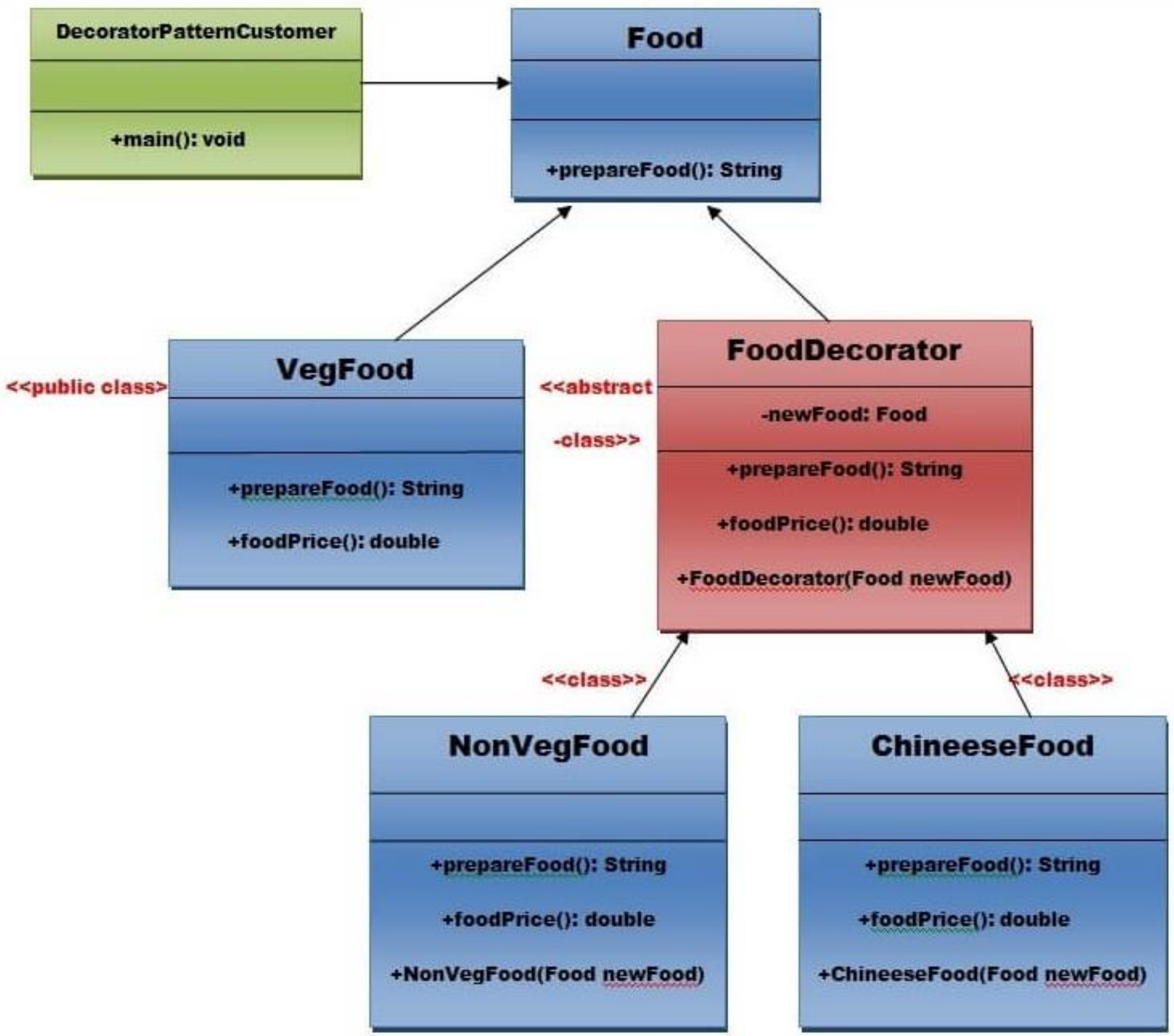
Composite pattern

- Allows clients to operate in generic manner on objects that may or may not represent a hierarchy of objects
- Advantages:
  - Defines class hierarchies that contain primitive objects.
  - Makes easier to you to add new kinds of components.
- Usage:
  - When you want to represent a full or partial hierarchy of objects.
  - When the responsibilities are needed to be added dynamically to the individual objects without affecting other objects.

Decorator pattern

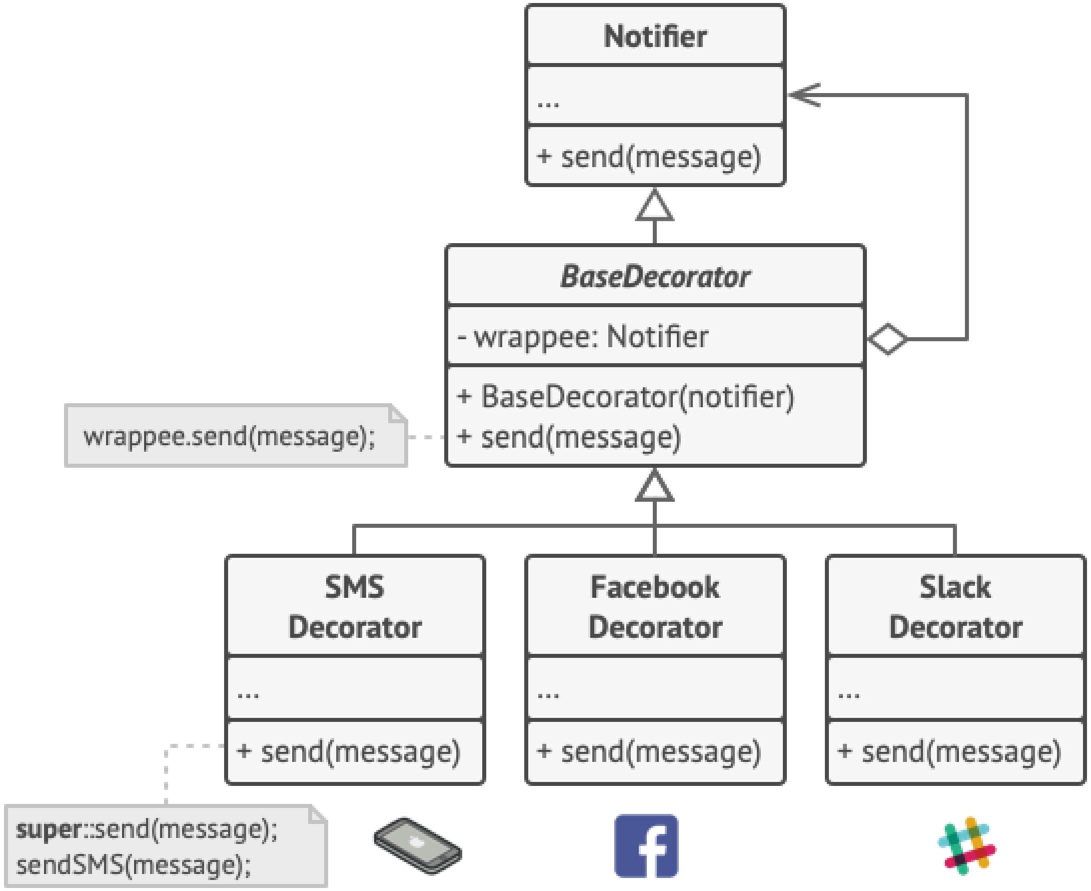
- ✧ Attaches flexible additional responsibilities to an object dynamically
- ✧ Uses composition instead of inheritance to extend the functionality of an object at runtime.
- ✧ Usage:
  - ▪ When you want to transparently and dynamically add responsibilities to objects without affecting other objects.
  - ▪ When you want to add responsibilities to an object that you may want to change in future.
  - ▪ Extending functionality by sub-classing is no longer practical.

Decorator pattern example



Example: Food

Decorator pattern example

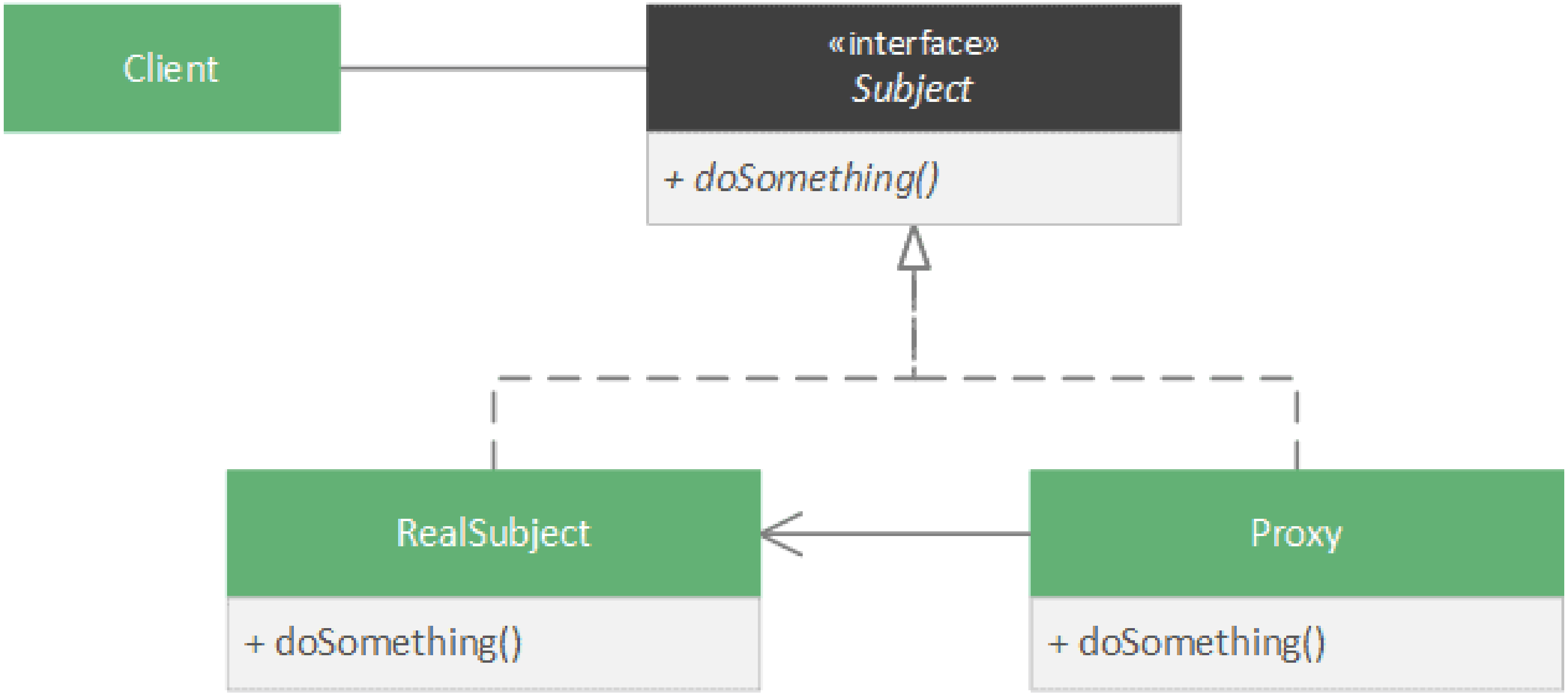


Example: Notifiers

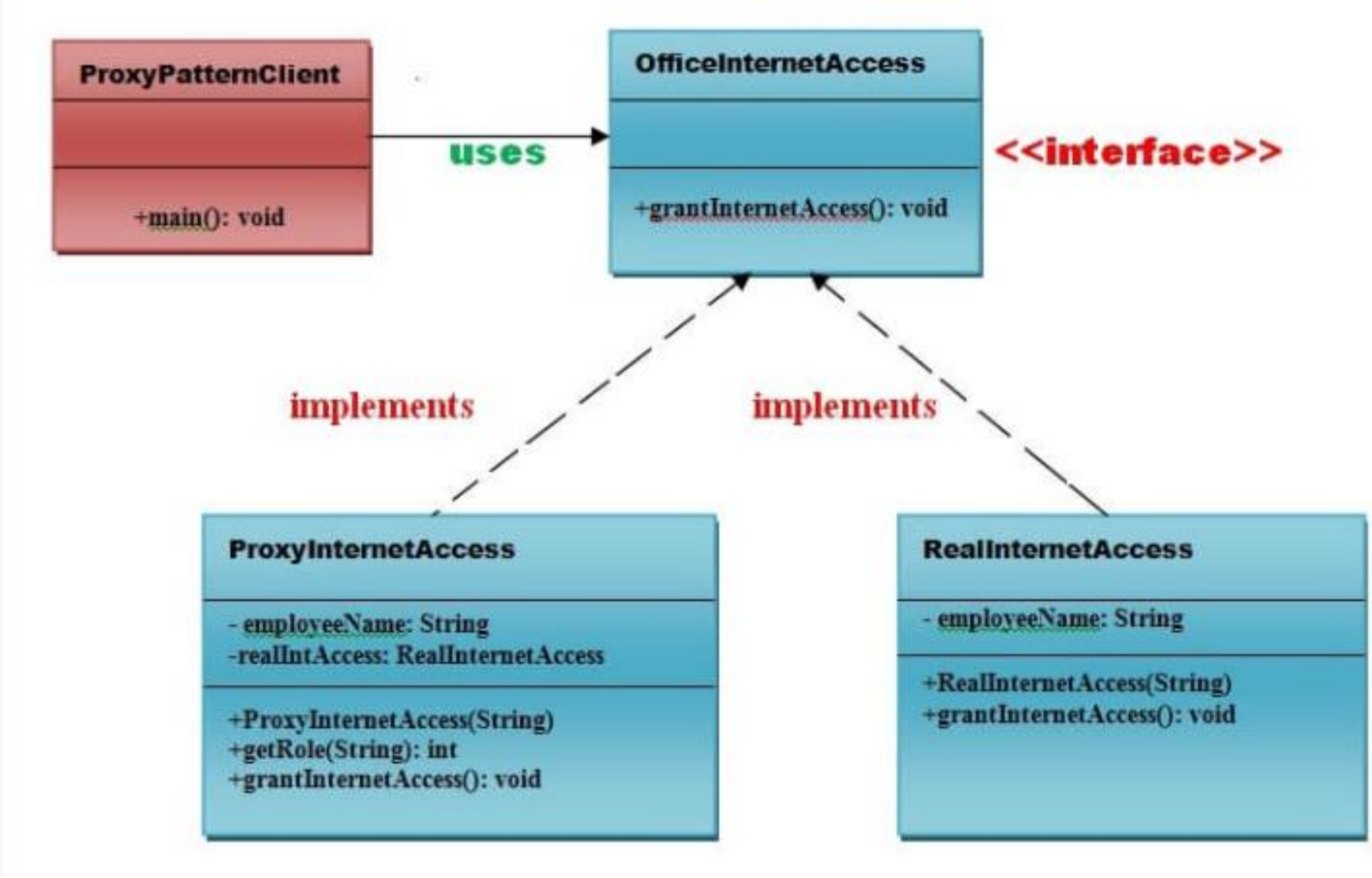
Proxy pattern

- ✧ Provides the means for controlling the original (underlying) object
- ✧ We can perform many operations like hiding the information of original object, on demand loading etc.
- ✧ Usage:
  - Virtual Proxy scenario
  - Protective Proxy scenario
  - Remove Proxy scenario
  - Smart Proxy scenario

Proxy pattern example



Proxy pattern example



Example: InternetAccess

Proxy pattern in Spring framework

```
@Service
public class BookManager {
    @Autowired
    private BookRepository repository;
    @Transactional
    public Book create(String author) {
        System.out.println(repository.getClass().getName());
        return repository.create(author);
    }
}
```

Proxy pattern in Spring framework

```
public class MyUserDetails implements UserDetails {
    private User user; // underlying object

    // proxy method
    @Override
    public String getPassword() {
        return user.getPassword();
    }
}
```