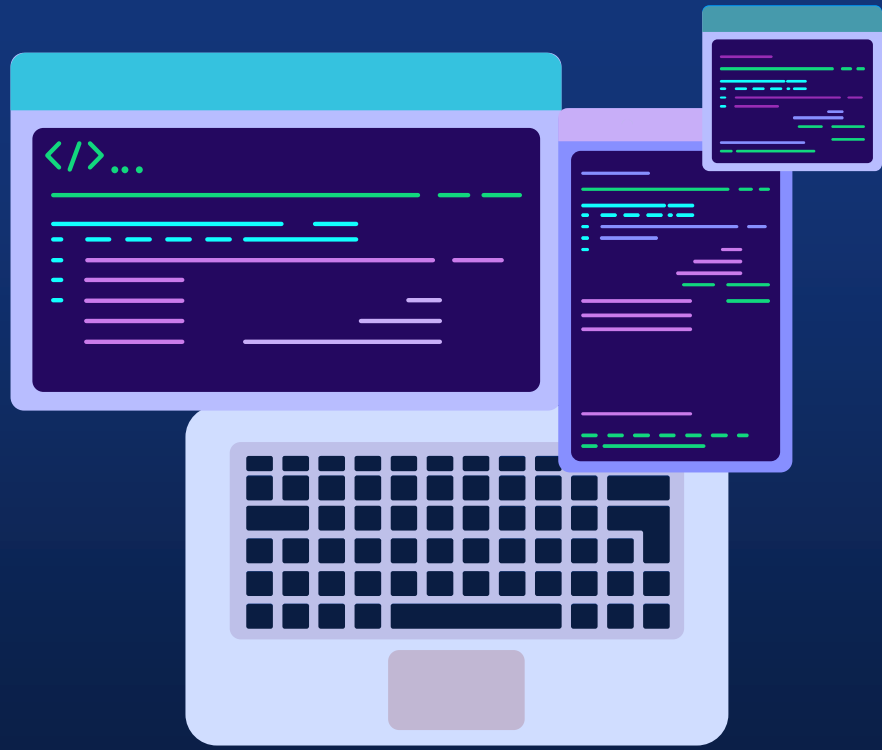
Software Design and Important concepts



Mentor: Einar Rocha

CONTENT

01
OOP Pillars

Inheritance, Polymorphism
Encapsulation, Abstraction

02
Clean Code

Meaningful Names,
Functions, Unit test
Code Smells...

03
SOLID

Single Responsiblity
Open closed
Liskov Substitution
Interface Segregation
Dependency Inversion

04
Design patterns

Singleton, Factory Method
Strategy, Observer
Builder...

04

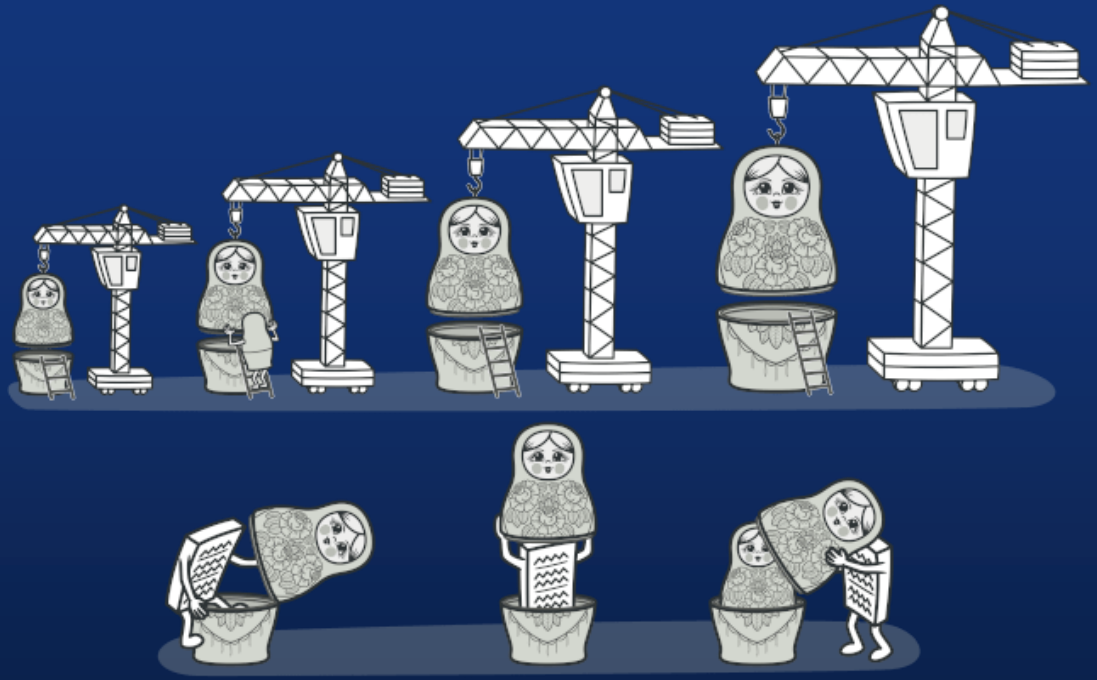
Design patterns

Agenda

Decorator — Example...

Facade — Example...

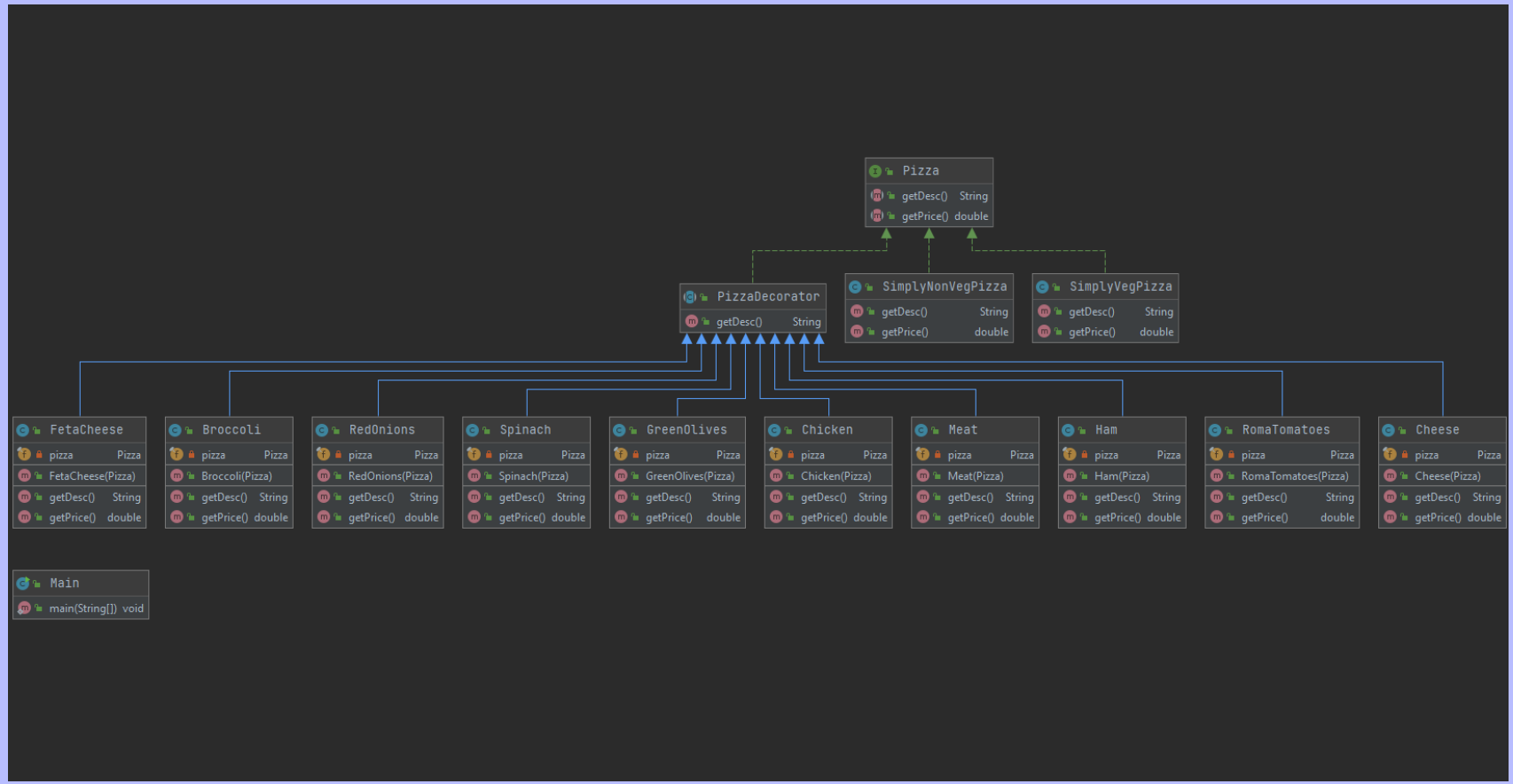
Decorator



When to use?

- To add responsibilities to individual objects dynamically and transparently, without affecting other objects.
- For responsibilities that can be withdrawn.
- When extension by sub-classing is impractical

Decorator


Pizza
getDesc() String
getPrice() double
PizzaDecorator
getDesc() String
SimplyNonVegPizza
getDesc() String
getPrice() double
SimplyVegPizza
getDesc() String
getPrice() double
FetaCheese
pizza Pizza
FetaCheese(Pizza)
getDesc() String
getPrice() double
Broccoli
pizza Pizza
Broccoli(Pizza)
getDesc() String
getPrice() double
RedOnions
pizza Pizza
RedOnions(Pizza)
getDesc() String
getPrice() double
Spinach
pizza Pizza
Spinach(Pizza)
getDesc() String
getPrice() double
GreenOlives
pizza Pizza
GreenOlives(Pizza)
getDesc() String
getPrice() double
Chicken
pizza Pizza
Chicken(Pizza)
getDesc() String
getPrice() double
Meat
pizza Pizza
Meat(Pizza)
getDesc() String
getPrice() double
Ham
pizza Pizza
Ham(Pizza)
getDesc() String
getPrice() double
RomaTomatoes
pizza Pizza
RomaTomatoes(Pizza)
getDesc() String
getPrice() double
Cheese
pizza Pizza
Cheese(Pizza)
getDesc() String
getPrice() double
Main
main(String[]) void

CONS

t’s hard to remove a specific wrapper from the wrappers stack.

It’s hard to implement a decorator in such a way that its behavior doesn’t depend on the order in the decorators stack.

The initial configuration code of layers might look pretty ugly.

PROS

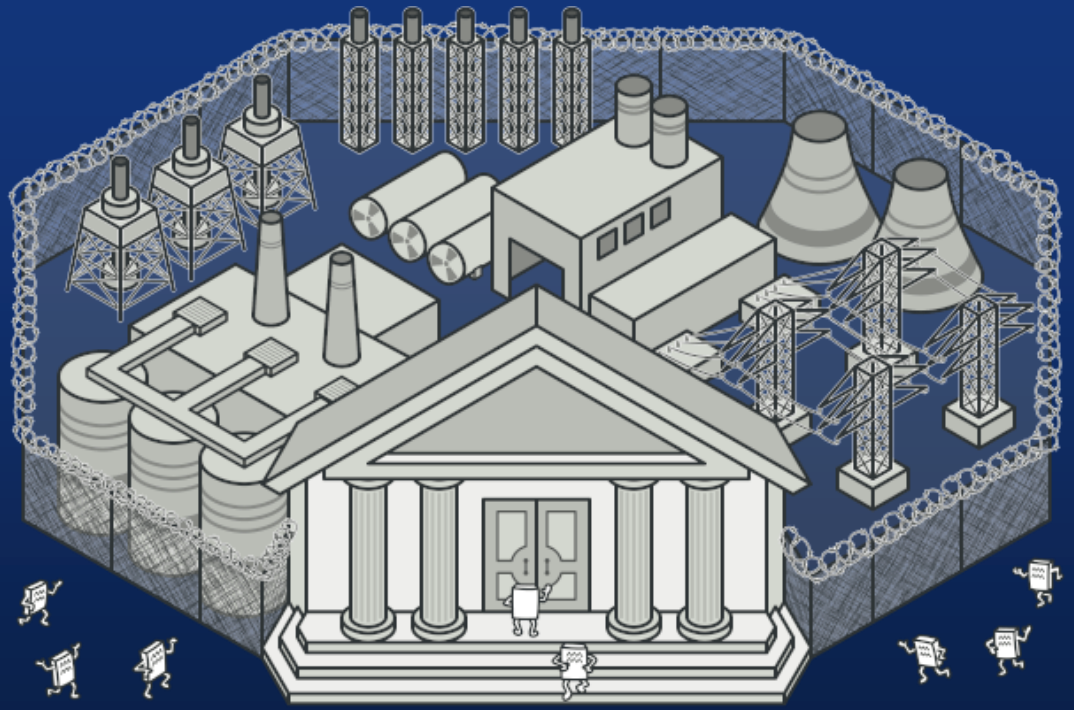
You can extend an object’s behavior without making a new subclass.

You can add or remove responsibilities from an object at runtime.

You can combine several behaviors by wrapping an object into multiple decorators.

Single Responsibility Principle. You can divide a monolithic class that implements many possible variants of behavior into several smaller classes.

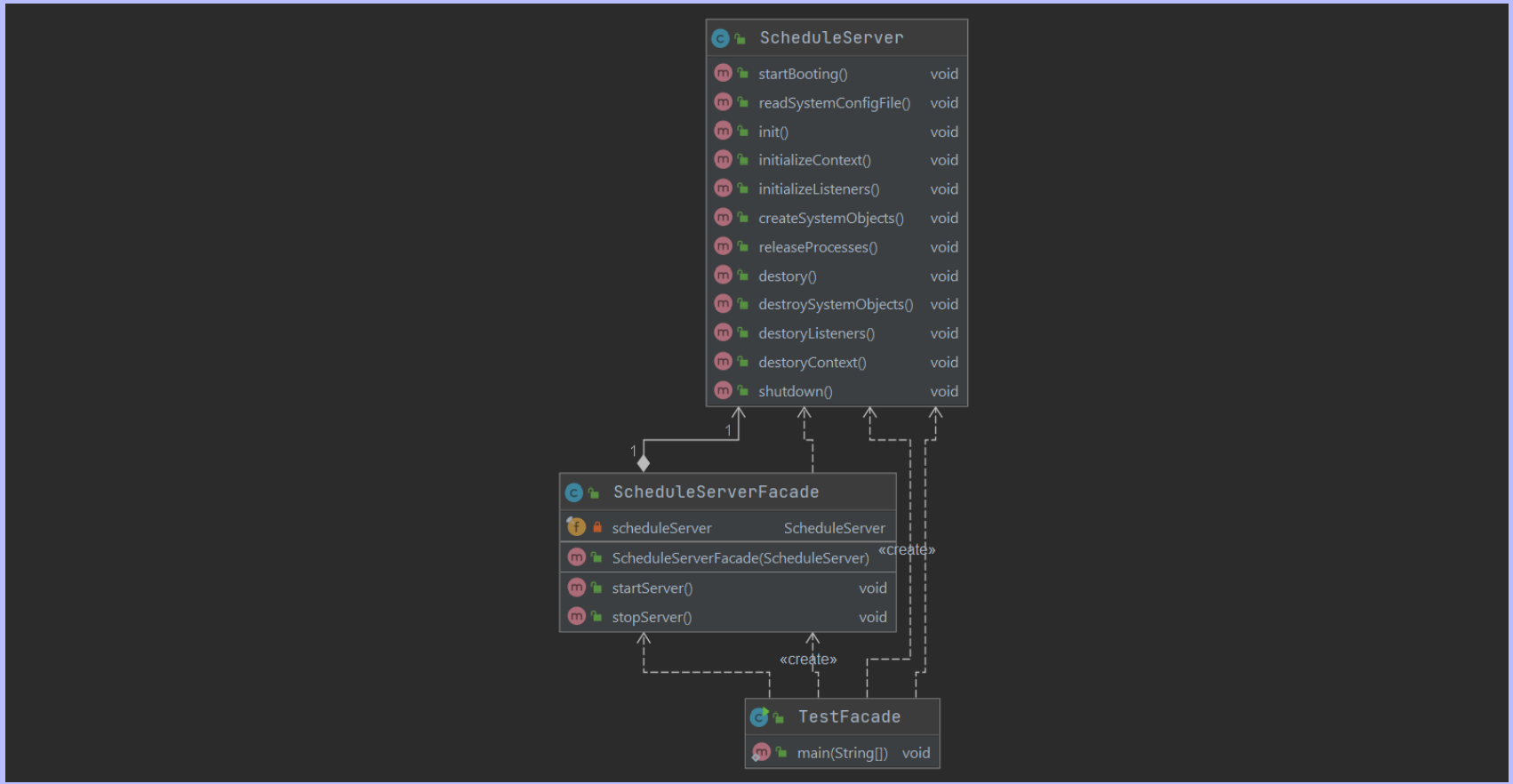
Facade



When to use?

- You want to provide a simple interface to a complex subsystem.
- There are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from clients and other subsystems.
- You can layer your subsystems. Use a facade to define an entry point to each subsystem level.

Facade


ScheduleServer
startBooting() void
readSystemConfigFile() void
init() void
initializeContext() void
initializeListeners() void
createSystemObjects() void
releaseProcesses() void
destory() void
destroySystemObjects() void
destoryListeners() void
destoryContext() void
shutdown() void
1
1
ScheduleServerFacade
scheduleServer ScheduleServer
ScheduleServerFacade(ScheduleServer)
startServer() void
stopServer() void
«create»
«create»
TestFacade
main(String[]) void

CONS

A facade can become a god object coupled to all classes of an app.

PROS

You can isolate your code from the complexity of a subsystem.