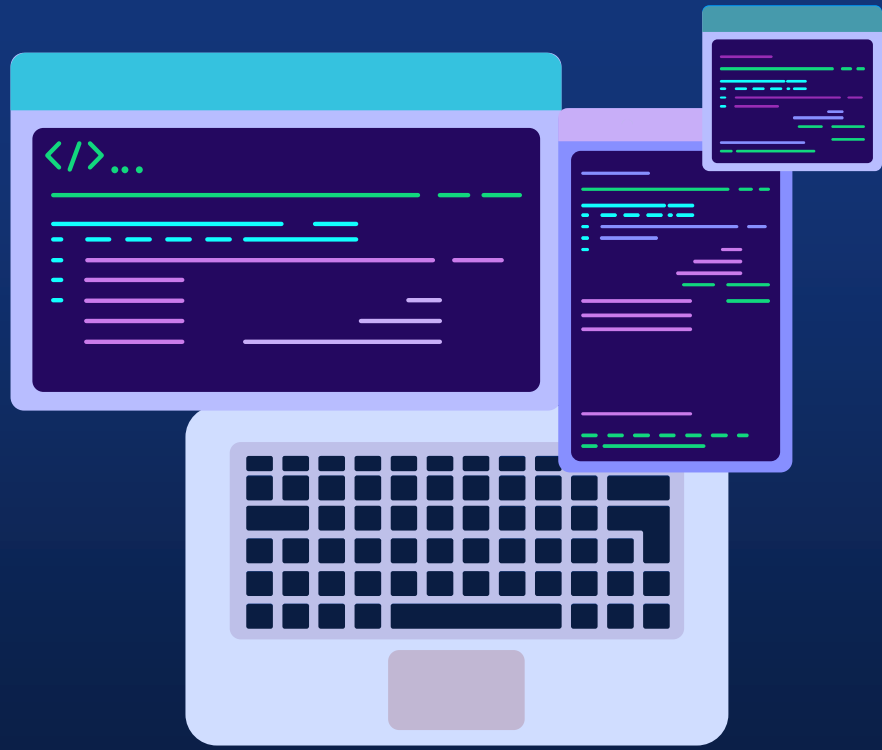
Software Design and Important concepts



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

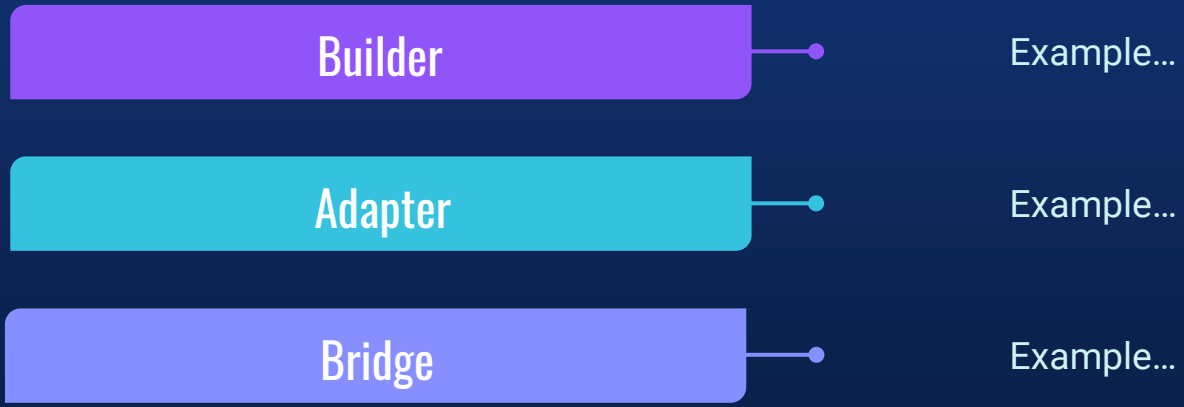
04
Design patterns

Singleton, Factory Method
Strategy, Observer
Builder...

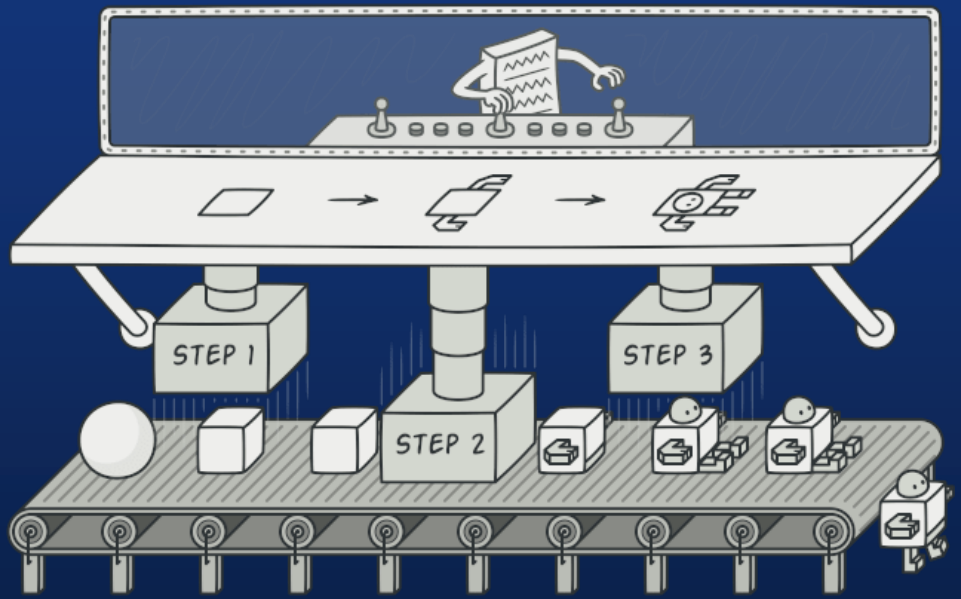
04

Design patterns

Agenda



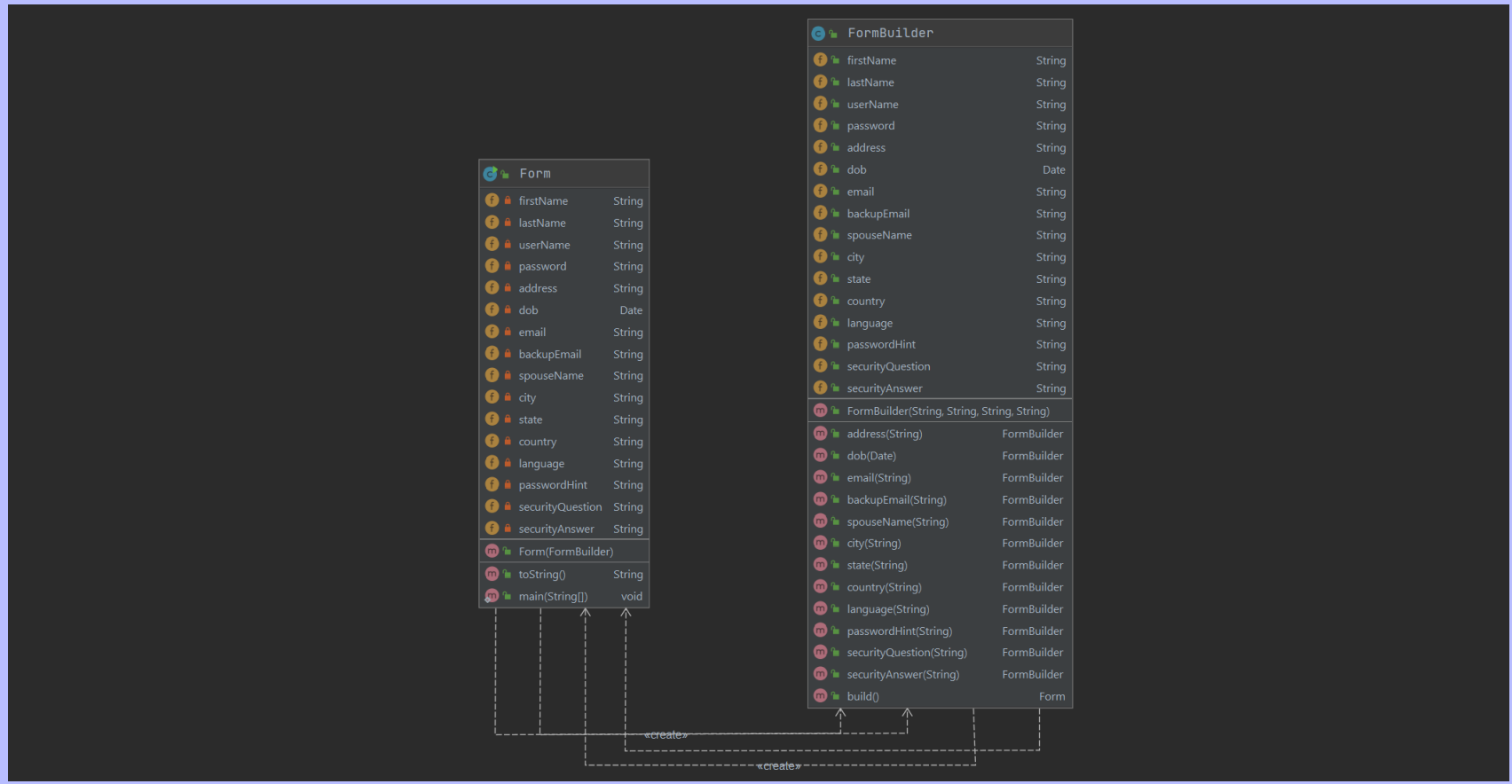
Builder



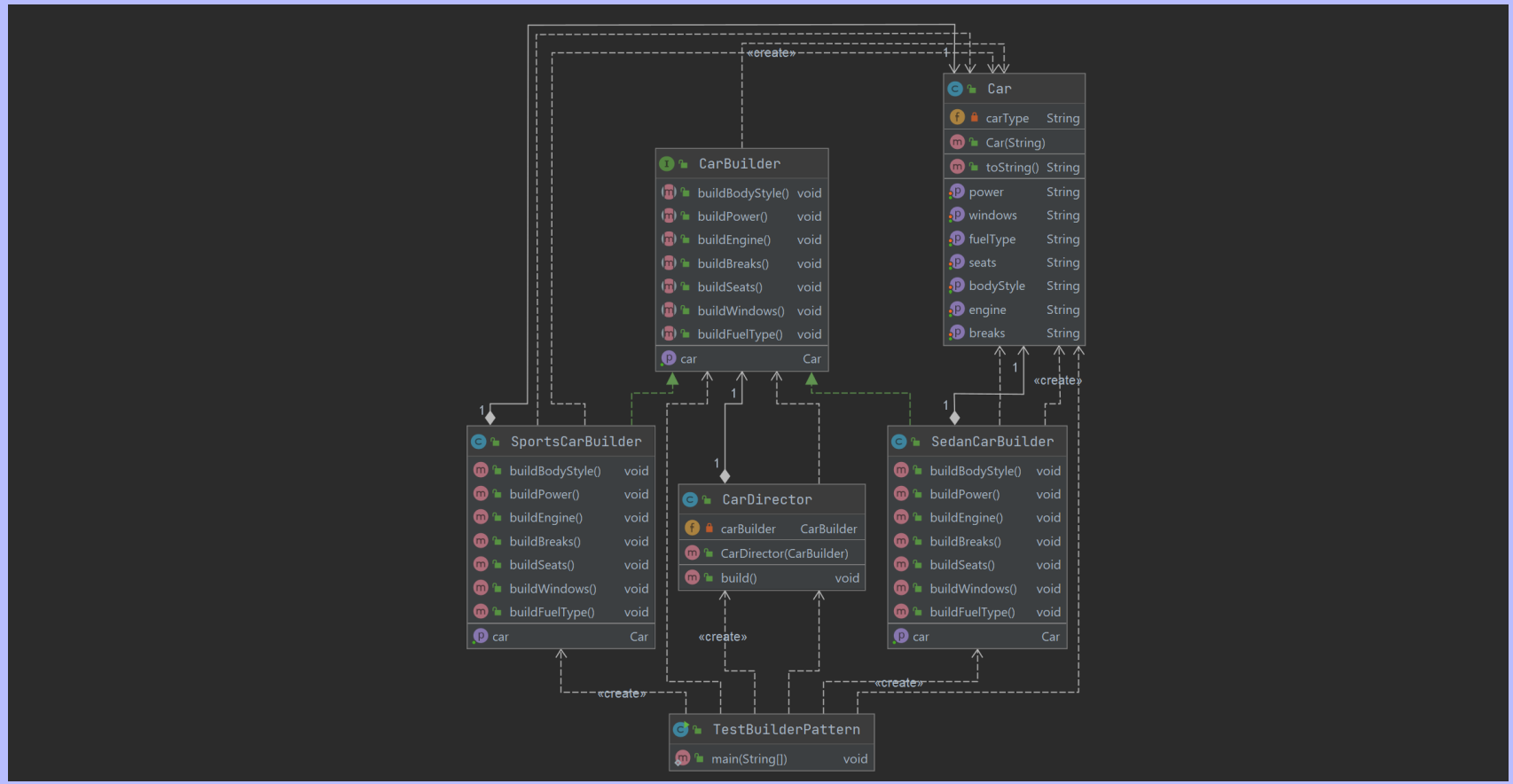
When to use?

- A client needs to build complex objects without knowing their implementation.
- A client needs to build complex objects that have multiple representations or implementations.

Builder - Optional



Builder


«create»
Car
carType String
Car(String)
toString() String
power String
windows String
fuelType String
seats String
bodyStyle String
engine String
breaks String
CarBuilder
buildBodyStyle() void
buildPower() void
buildEngine() void
buildBreaks() void
buildSeats() void
buildWindows() void
buildFuelType() void
car Car
1
1
1
1
«create»
SportsCarBuilder
buildBodyStyle() void
buildPower() void
buildEngine() void
buildBreaks() void
buildSeats() void
buildWindows() void
buildFuelType() void
car Car
CarDirector
carBuilder CarBuilder
CarDirector(CarBuilder)
build() void
SedanCarBuilder
buildBodyStyle() void
buildPower() void
buildEngine() void
buildBreaks() void
buildSeats() void
buildWindows() void
buildFuelType() void
car Car
«create»
«create»
«create»
TestBuilderPattern
main(String[]) void

CONS

The overall complexity of the code increases since the pattern requires creating multiple new classes.

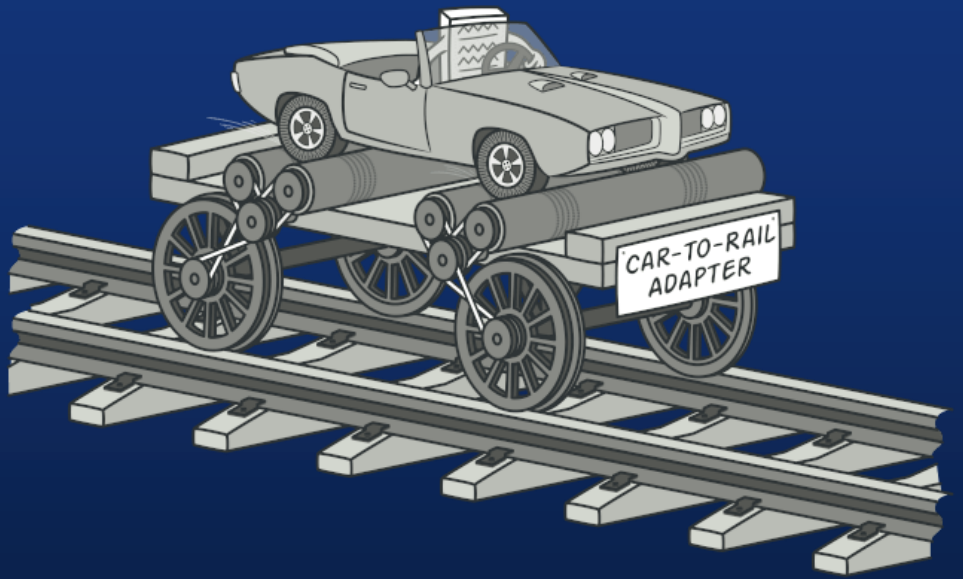
PROS

You can construct objects step-by-step.

You can reuse the same construction code when building various representations of products.

Single Responsibility Principle.

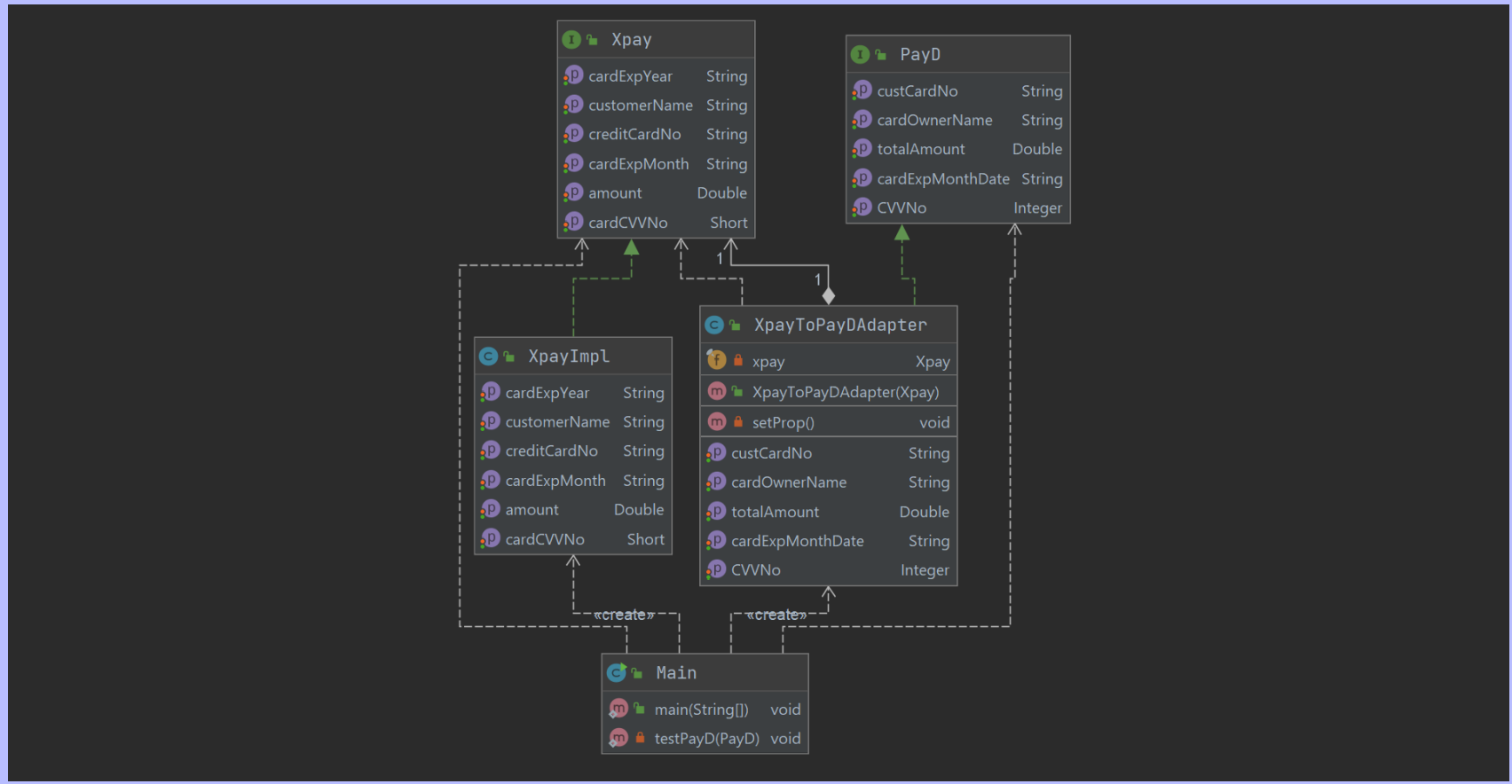
Adapter



When to use?

- To integrate into the system an object whose interface does not correspond to the interface required inside this system.
- To provide multiple interfaces to an object in its design stage.

Adapter


Xpay
cardExpYear String
customerName String
creditCardNo String
cardExpMonth String
amount Double
cardCVVNo Short
PayD
custCardNo String
cardOwnerName String
totalAmount Double
cardExpMonthDate String
CVVNo Integer
1
1
XpayImpl
cardExpYear String
customerName String
creditCardNo String
cardExpMonth String
amount Double
cardCVVNo Short
XpayToPayDAdapter
xpay Xpay
XpayToPayDAdapter(Xpay)
setProp() void
custCardNo String
cardOwnerName String
totalAmount Double
cardExpMonthDate String
CVVNo Integer
«create»
«create»
Main
main(String[]) void
testPayD(PayD) void

CONS

The overall complexity of the code increases because you need to introduce a set of new interfaces and classes.

PROS

Single Responsibility Principle.

Open/Closed Principle. You can introduce new types of adapters into the program without breaking the existing client code.