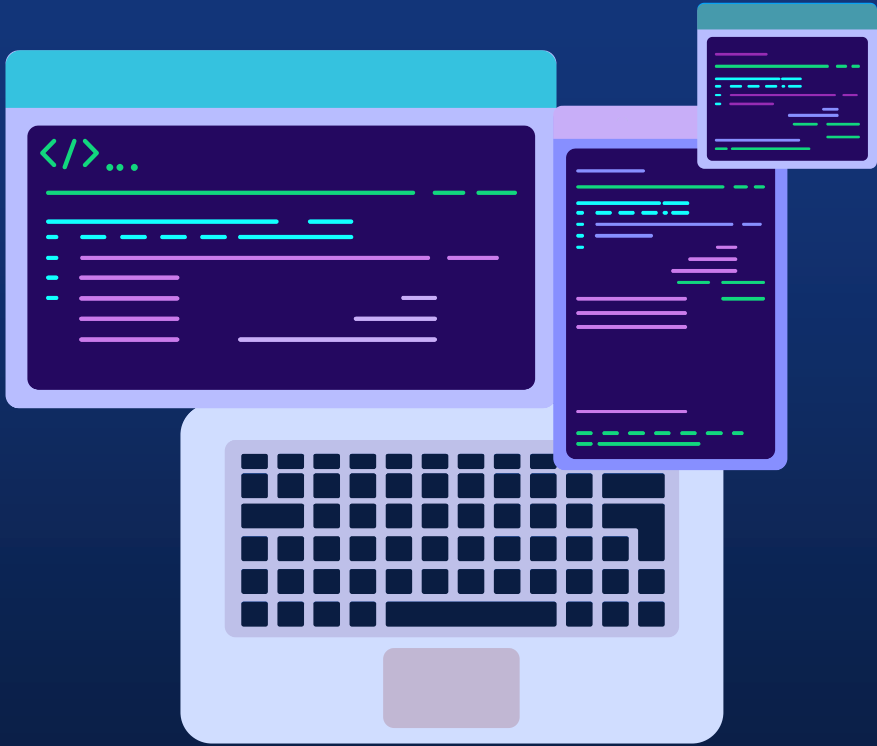
Software Design and Important concepts



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…

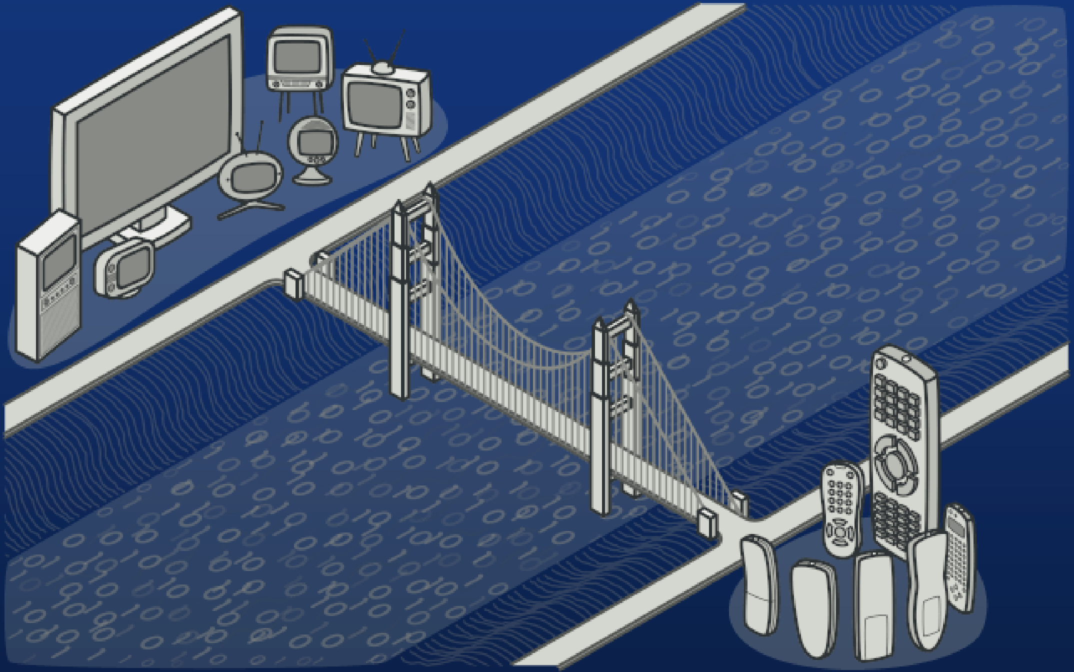
04

Design patterns

Agenda

Bridge	Example...
Composite	Example...

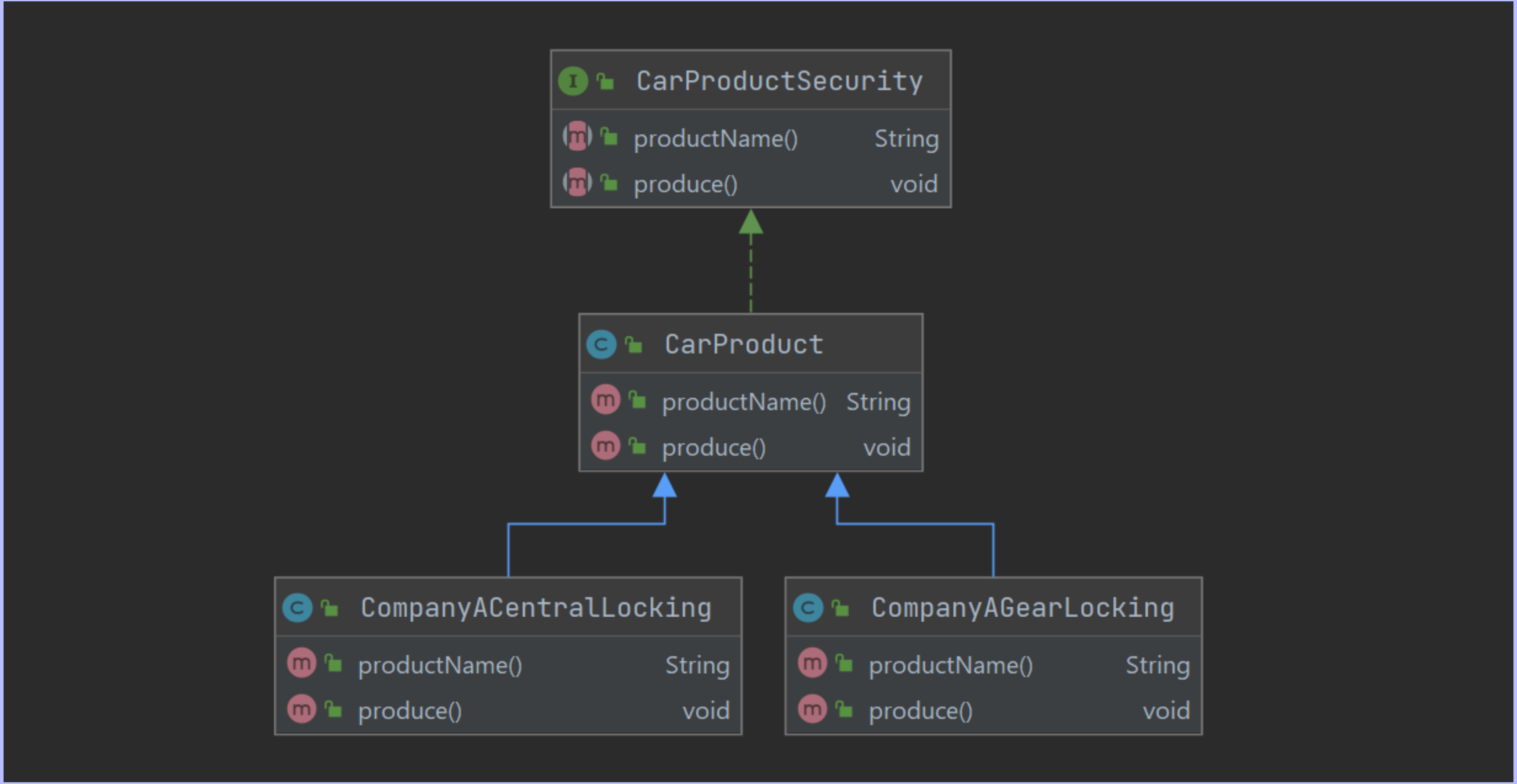
Bridge



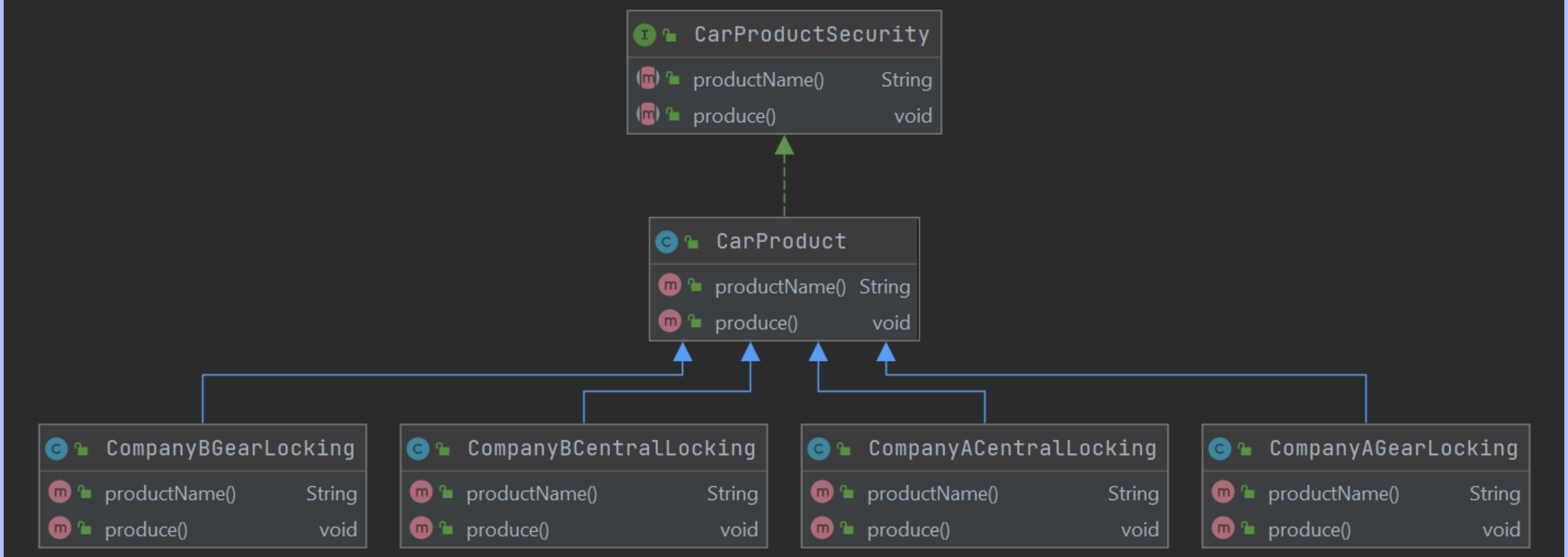
When to use?

- To avoid that there is too strong a link between the representation of the objects and their implementation.
- So that changes in the implementation of the objects have no impact on the interactions between the objects and their clients.
- To avoid getting too complex class hierarchies

Bridge - One

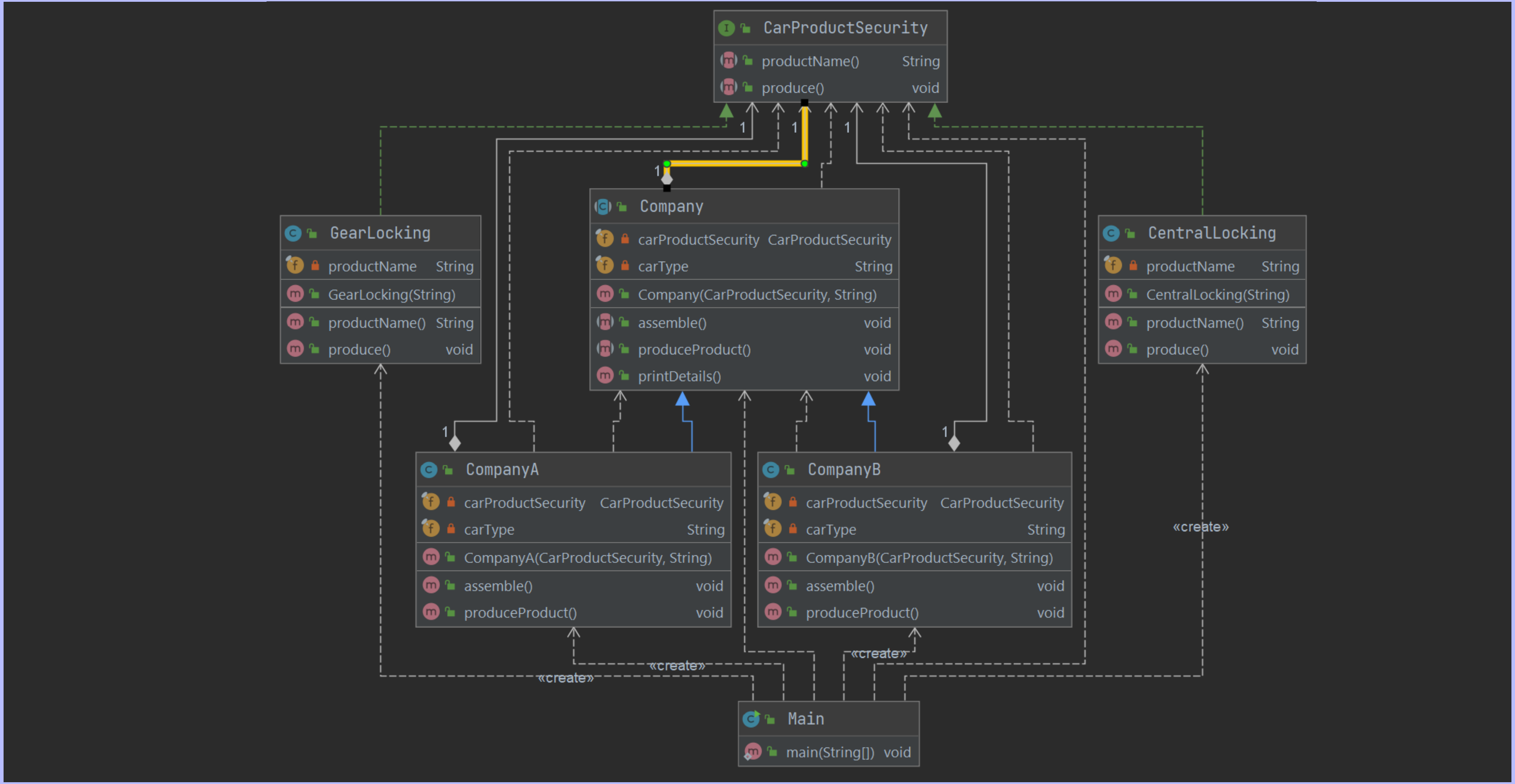

CarProductSecurity
productName() String
produce() void
CarProduct
productName() String
produce() void
CompanyACentralLocking
productName() String
produce() void
CompanyAGearLocking
productName() String
produce() void

Bridge - Two



CarProductSecurity
productName() String
produce() void
CarProduct
productName() String
produce() void
CompanyBGearLocking
productName() String
produce() void
CompanyBCentralLocking
productName() String
produce() void
CompanyACentralLocking
productName() String
produce() void
CompanyAGearLocking
productName() String
produce() void

Bridge


CarProductSecurity
productName() String
produce() void
Company
carProductSecurity CarProductSecurity
carType String
Company(CarProductSecurity, String)
assemble() void
produceProduct() void
printDetails() void
GearLocking
productName String
GearLocking(String)
productName() String
produce() void
CentralLocking
productName String
CentralLocking(String)
productName() String
produce() void
CompanyA
carProductSecurity CarProductSecurity
carType String
CompanyA(CarProductSecurity, String)
assemble() void
produceProduct() void
CompanyB
carProductSecurity CarProductSecurity
carType String
CompanyB(CarProductSecurity, String)
assemble() void
produceProduct() void
Main
main(String[]) void
1
1
1
1
1
1
«create»
«create»
«create»
«create»

CONS

You might make the code more complicated by applying the pattern to a highly cohesive class.

PROS

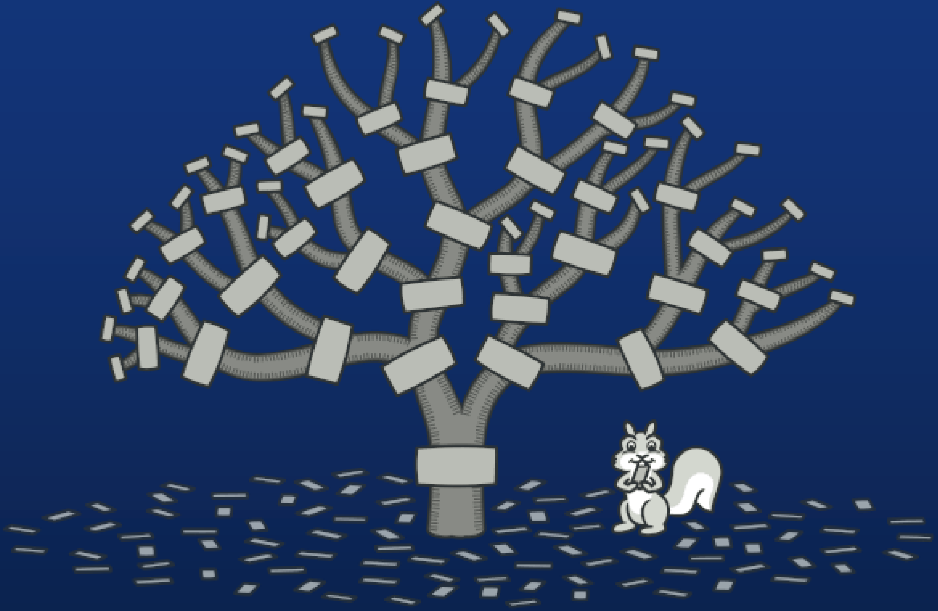
You can create platform-independent classes and apps.

The client code works with high-level abstractions.

Open/Closed Principle.

Single Responsibility Principle.

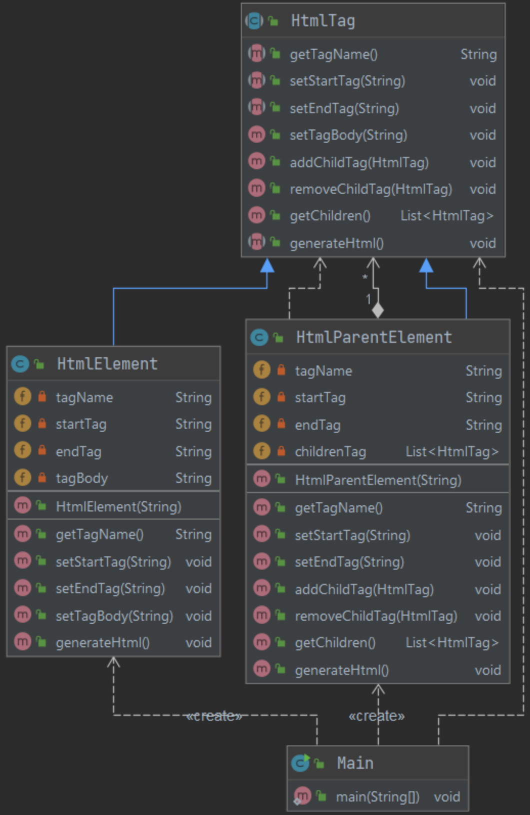
Composite



When to use?

- When you want to represent part-whole hierarchies of objects.
- When you want clients to be able to ignore the difference between compositions of objects and individual objects.

Composite


HtmlTag
getTagName() String
setStartTag(String) void
setEndTag(String) void
setTagBody(String) void
addChildTag(HtmlTag) void
removeChildTag(HtmlTag) void
getChildren() List<HtmlTag>
generateHtml() void
*
1
HtmlElement
tagName String
startTag String
endTag String
tagBody String
HtmlElement(String)
getTagName() String
setStartTag(String) void
setEndTag(String) void
setTagBody(String) void
generateHtml() void
HtmlParentElement
tagName String
startTag String
endTag String
childrenTag List<HtmlTag>
HtmlParentElement(String)
getTagName() String
setStartTag(String) void
setEndTag(String) void
addChildTag(HtmlTag) void
removeChildTag(HtmlTag) void
getChildren() List<HtmlTag>
generateHtml() void
«create»
«create»
Main
main(String[]) void

CONS

It might be difficult to provide a common interface for classes whose functionality differs too much.

PROS

You can work with complex tree structures more conveniently.

Open/Closed Principle. You can introduce new element types into the app without breaking the existing code.