Some content from Elisa Baniassad

SOLID Design Principles

Reid Holmes

Source: [*Gamma et all, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995*]

- Program an Interface not an Implementation
- Favor Composition Versus Inheritance
- Find what varies and encapsulate it

Source: [*R. Martin, "Agile Software Development, Principles, Patterns, and Practices", Prentice-Hall, 2002*]

- Dependency-Inversion Principle
- Liskov Substitution Principle
- Open-Closed Principle
- Interface-Segregation Principle
- Reuse/Release Equivalency Principle
- Common Closure Principle
- Common Reuse Principle
- Acyclic Dependencies Principle
- Stable Dependencies Principle
- Stable Abstraction Principle

Source: [*Larman, "Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development", Prentice-Hall, 20*

- Design principles are codified in the GRASP Pattern
- GRASP (Pattern of General Principles in Assigning Responsabilities)
- Assign a responsibility to the information expert
- Assign a responsability so that coupling remains low
- Assign a responsability so that cohesion remains high
- Assign responsabilities using polymorphic operations
- Assign a highly cohesive set of responsabilities to an artificial class that does not represent anything in the problem domain (when you want to
- Don't talk to strangers (Law of Demeter)

Source: [*Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules", Commununication of ACM, 1972*]

- Information Hiding
- Modularity

Source: [*Hunt, Thomas, "The Pragmatic Programmer: From Journeyman to Master", Addison-Wesley, 1999*]

- DRY – Don't Repeat yourself
- Make it easy to reuse
- Design for Orthogonality
- Eliminate effects between unrelated things
- Program close to the problem domain
- Minimize Coupling between Modules
- Design Using Services
- Always Design for Concurrency
- Abstractions Live Longer than details

Source: [*Lieberherr,Holland, "Assuring Good Style for Object-Oriented Programs", IEEE Software, September 1989*]

- Law of Demeter

Source: [*Raymond, "Art of Unix Programming", Addison-Wesley, 2003*]

Pragmatic Programmer:

Eliminate Effects Between Unrelated Things –
design components that are:
self-contained,
independent,
and have a single, well-defined purpose

SOLID
review

SOLID (Single Responsibility)

Classes should do one thing and do it well.

SOLID (Single Responsibility)

Or check:

A description that describes a class in terms of alternatives is not one class, but a set of classes.

SOLID (Single Responsibility)

"A ClassRoom is la location where students attend tutorials **or** labs."

SOLID (Single Responsibility)

Things we can do to Yaks:

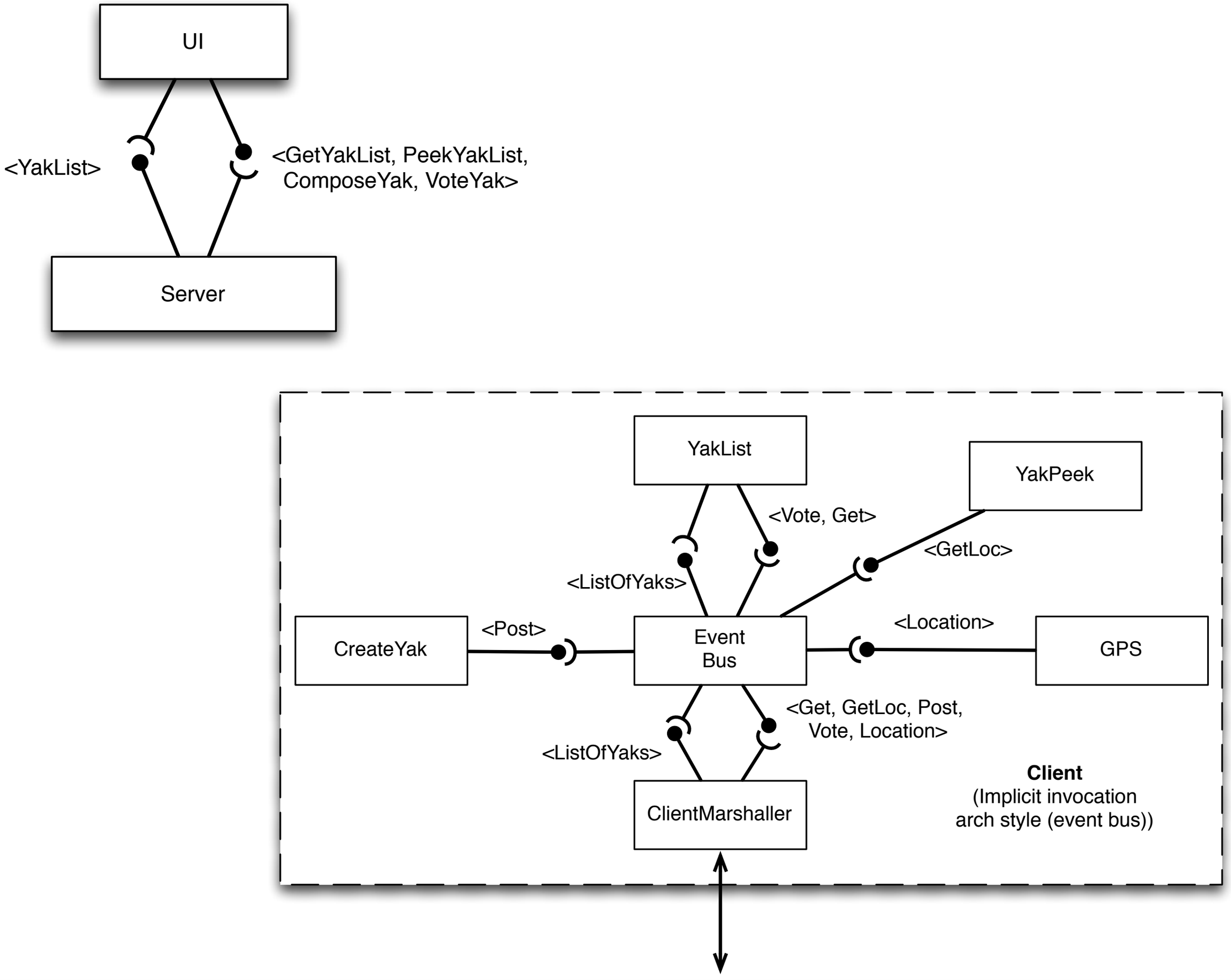
(from the midterm)

Compose

View

Peek

Vote


UI
<YakList>
<GetYakList, PeekYakList,
ComposeYak, VoteYak>
Server
YakList
YakPeek
<Vote, Get>
<GetLoc>
<ListOfYaks>
CreateYak
<Post>
Event
Bus
<Location>
GPS
<Get, GetLoc, Post,
Vote, Location>
<ListOfYaks>
ClientMarshaller
Client
(Implicit invocation
arch style (event bus))

SOLID (Single Responsibility)

- Strategy (small, targeted, algorithms)
- Command (invokers should be oblivious to actions)
- Visitor (accomplish specific tasks)
- State (centralize 3rd party complexity)

SOLID (Open/Close)

Classes should be open to **extension** and closed to **modification**.

SOLID (Open/Close)

Which design patterns support the open/close principle?

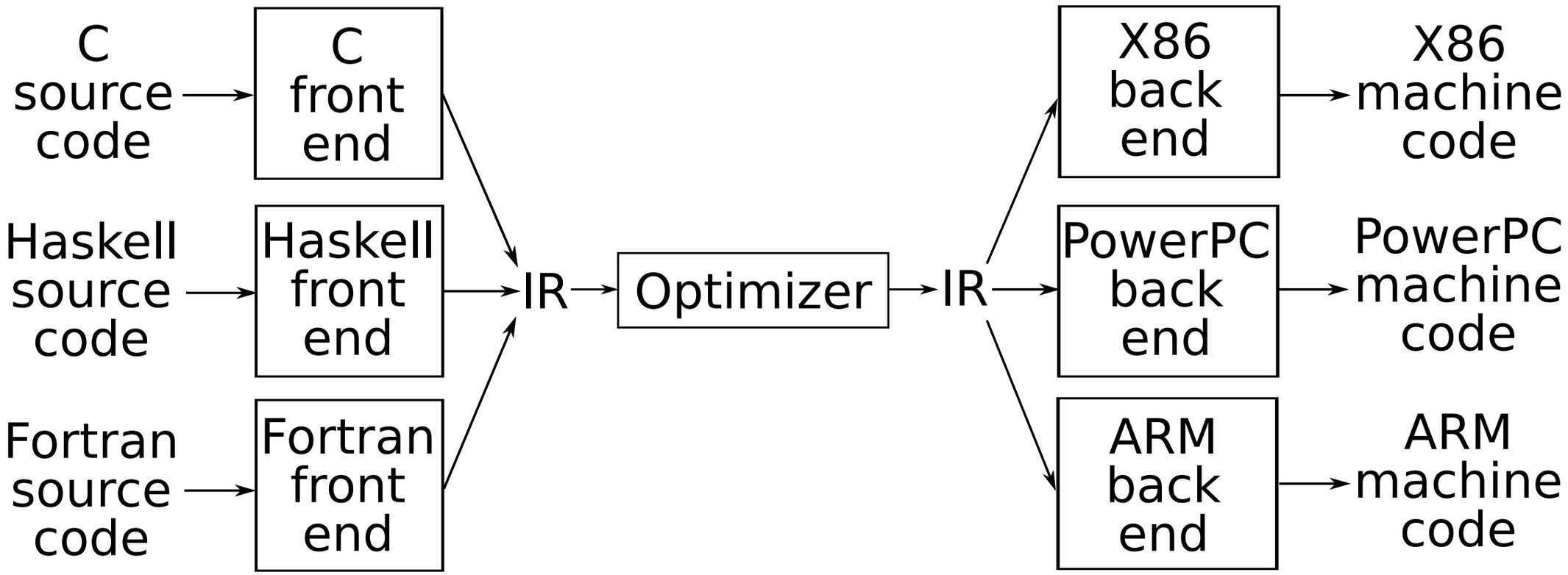
(These patterns are a subset of those patterns that help with *encapsulating what varies*. E.g., the 'extension' part is often expected to change.)

SOLID (Open/Close)

- Observer (extend set of observers)
  - w/o changing subject behaviour
- Strategy (extend algorithm suite)
  - w/o changing context or other algorithms
- State (specialize runtime behaviour)
  - w/o changing context or other behaviours
- Command (extend command suite)
  - w/o changing invoker
- Visitor (extend model analysis)
  - w/o changing data structure, traversal code, other visitors
- Decorator (extend object through composition)
  - w/o changing base classes
- Composite (extend component)
  - w/o changing clients / composites using any component

SOLID (Open/Close)

- How does the LLVM architecture in the midterm support Open/Close?



SOLID (Open/Close)

- Whenever your code is making behavioural changes based on internal flags or instanceof you are likely violating Open/Close. E.g.,

```
public interface IBillingService {

 Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {

    if (creditCard instanceof VisaCard) {
      …
    } else if (creditCard instanceof MasterCard) {
      …
    }
}
```

SOLID (Liskov substitution)

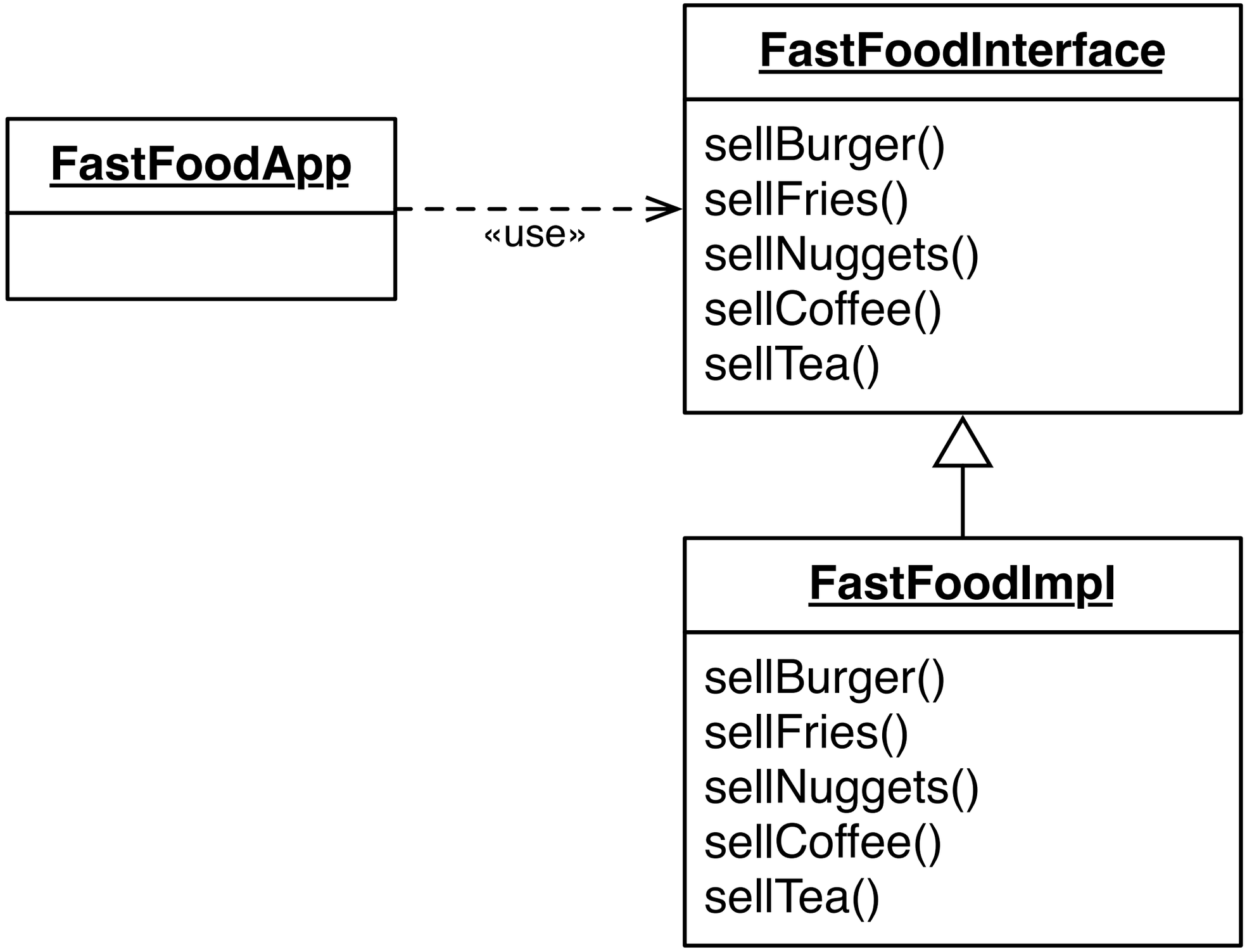
Most design patterns break down if LSP is violated.

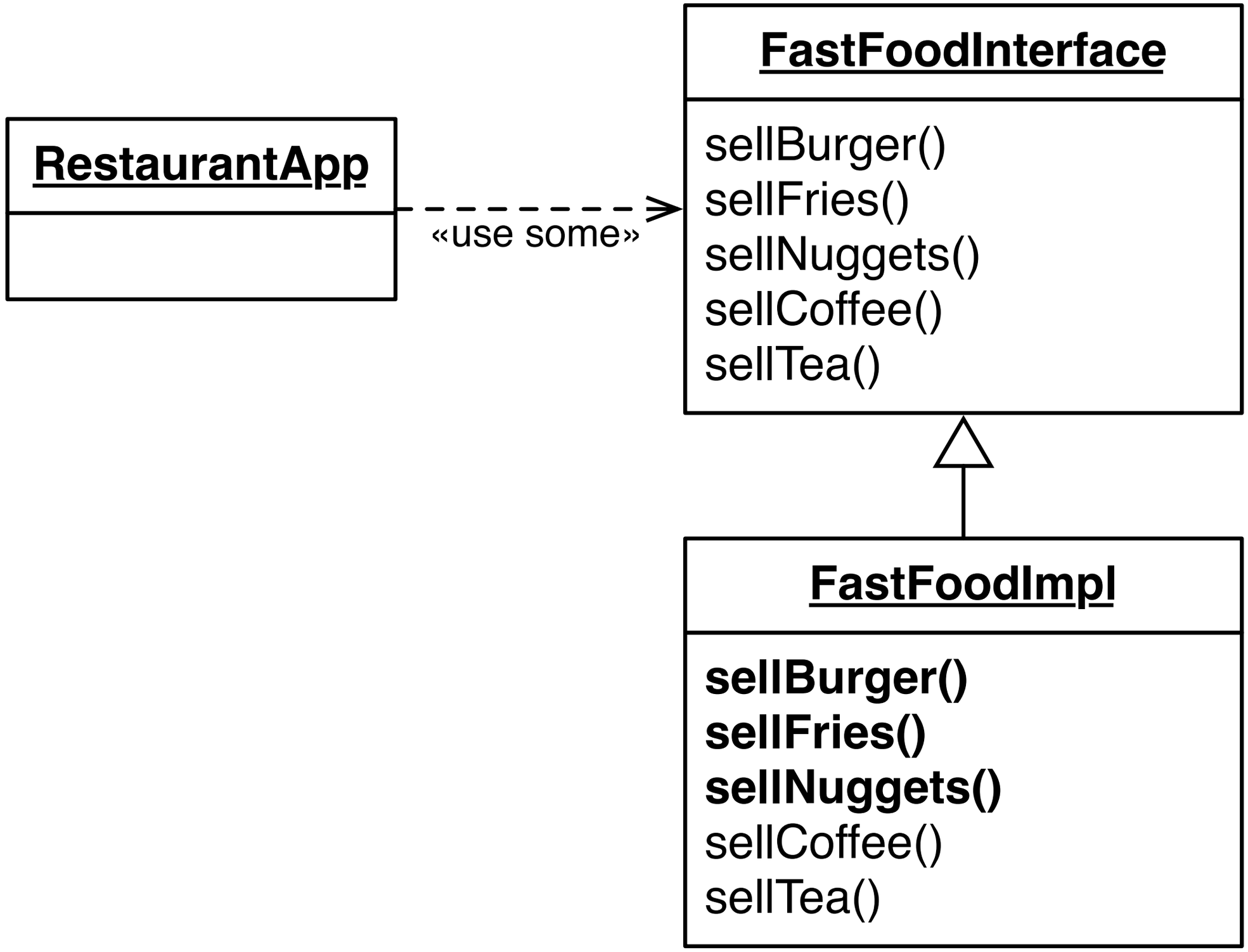
(Most design patterns are enabled through a layer of abstraction, typically provided through inheritance. When subtypes violate LSP inconsistencies can occur at runtime.)

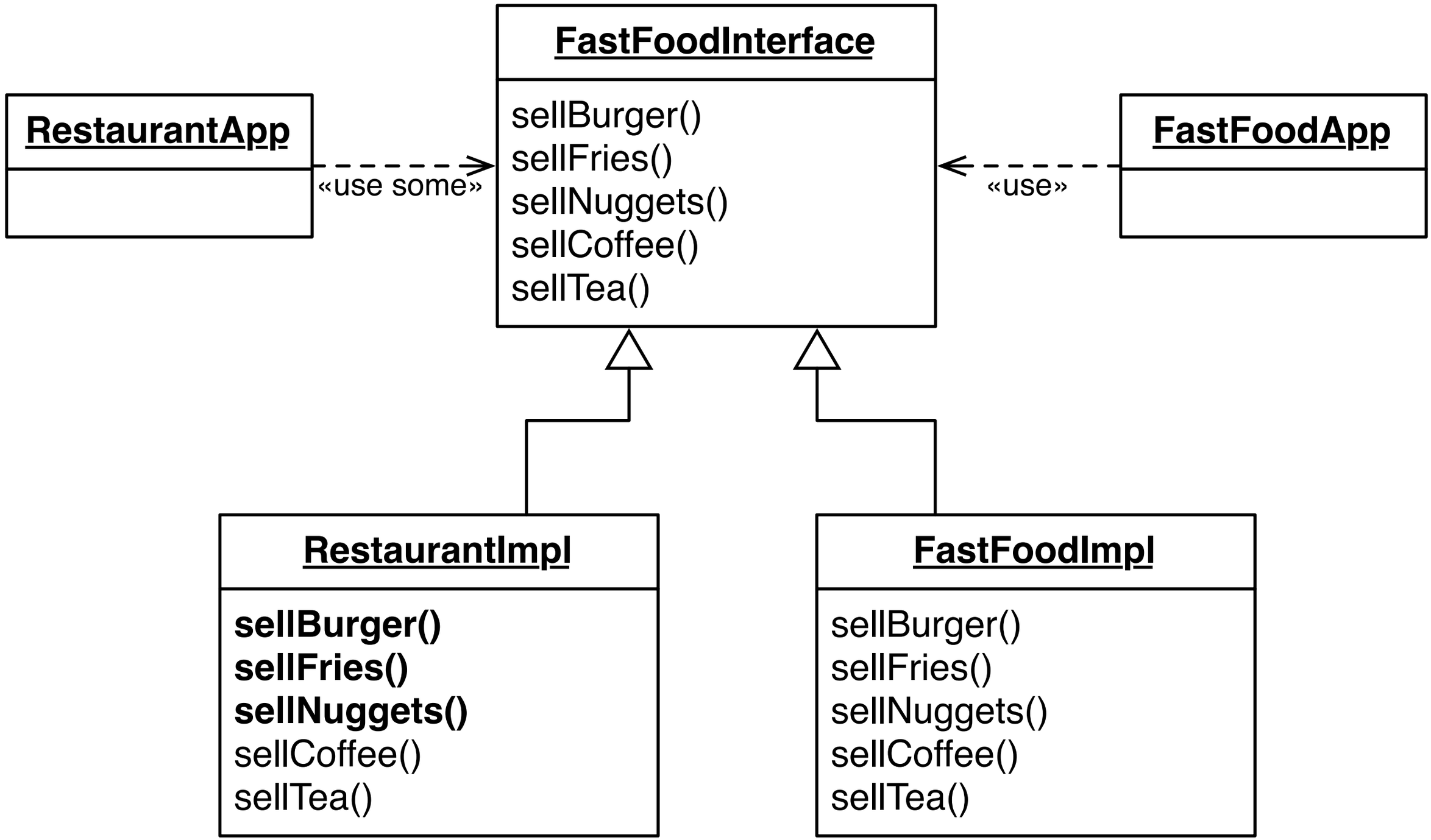
SOLID (Interface segregation)

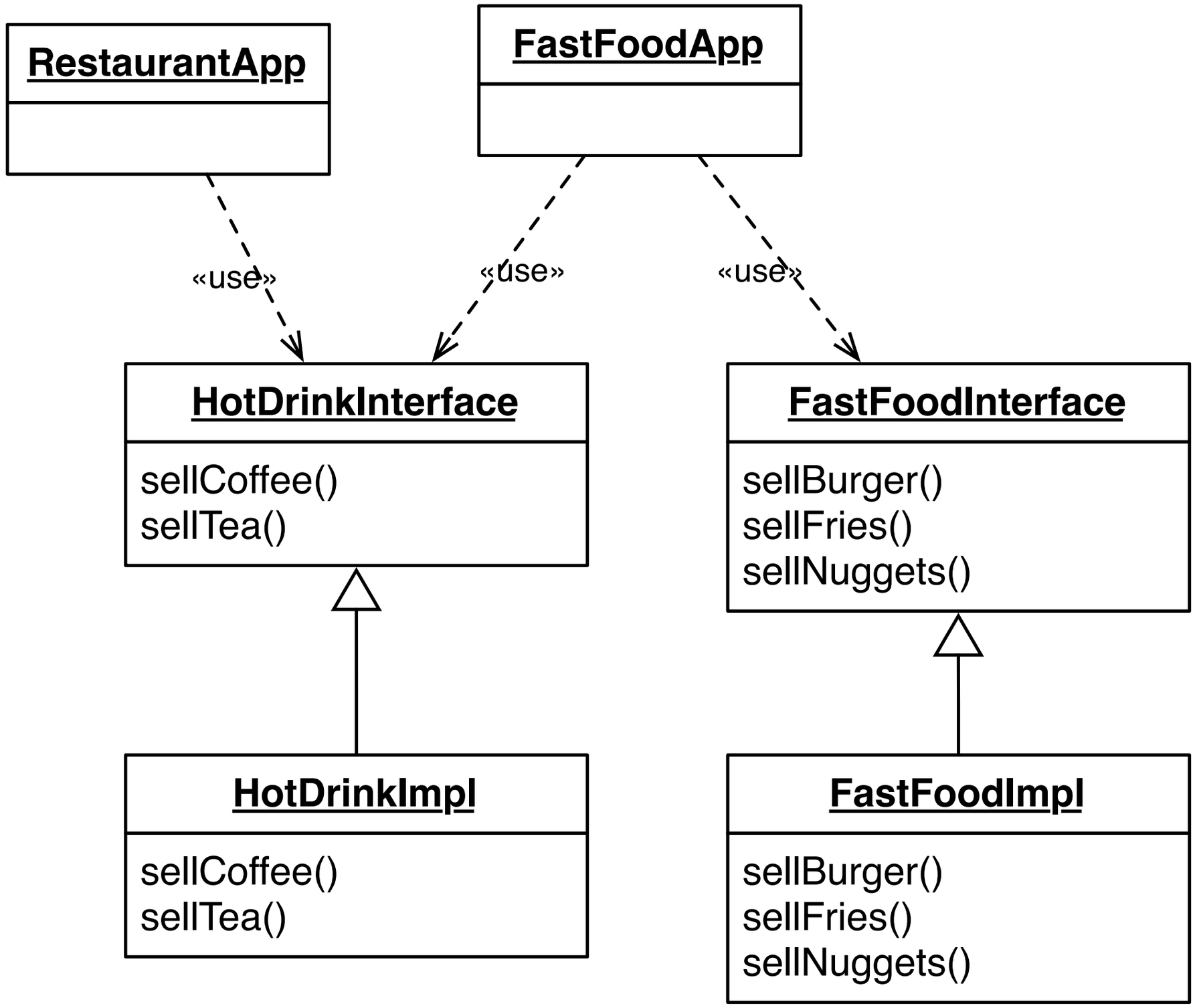
Clients should not be forced to **depend** on **interfaces** they do not use.

(Depending on irrelevant interfaces causes needless coupling. This causes classes to change even when interfaces they do not care about are modified.)

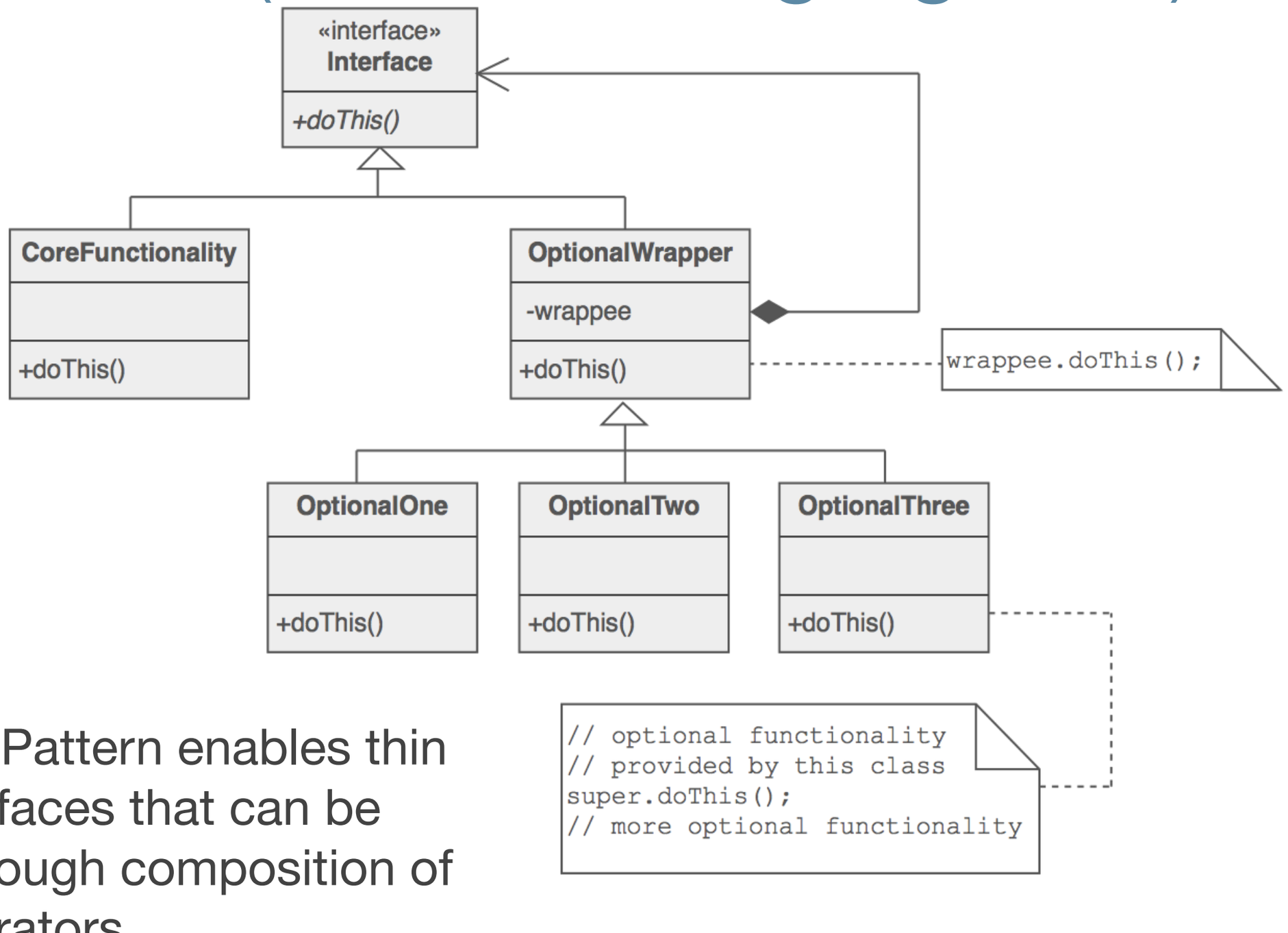

FastFoodInterface
sellBurger()
sellFries()
sellNuggets()
sellCoffee()
sellTea()
FastFoodApp
«use»
FastFoodImpl
sellBurger()
sellFries()
sellNuggets()
sellCoffee()
sellTea()


RestaurantApp
«use some»
FastFoodInterface
sellBurger()
sellFries()
sellNuggets()
sellCoffee()
sellTea()
FastFoodImpl
sellBurger()
sellFries()
sellNuggets()
sellCoffee()
sellTea()


FastFoodInterface
sellBurger()
sellFries()
sellNuggets()
sellCoffee()
sellTea()
RestaurantApp
«use some»
«use»
FastFoodApp
RestaurantImpl
sellBurger()
sellFries()
sellNuggets()
sellCoffee()
sellTea()
FastFoodImpl
sellBurger()
sellFries()
sellNuggets()
sellCoffee()
sellTea()


RestaurantApp
FastFoodApp
«use»
«use»
«use»
HotDrinkInterface
sellCoffee()
sellTea()
FastFoodInterface
sellBurger()
sellFries()
sellNuggets()
HotDrinkImpl
sellCoffee()
sellTea()
FastFoodImpl
sellBurger()
sellFries()
sellNuggets()

SOLID (Interface segregation)



The Decorator Pattern enables thin high-level interfaces that can be augmented through composition of concrete Decorators.

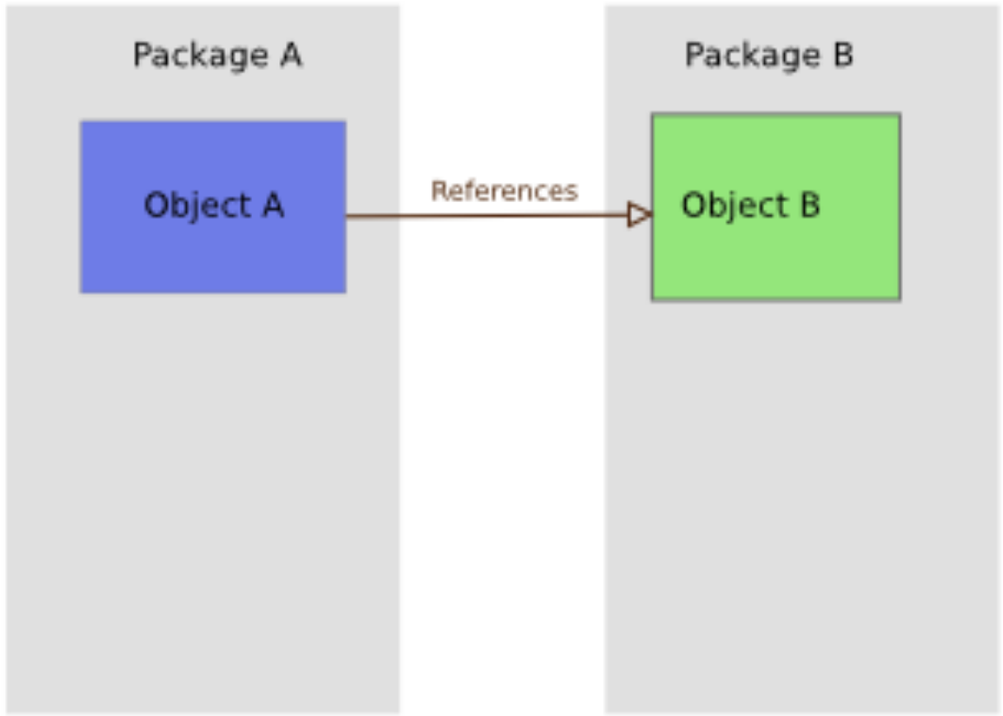
SOLID (Dependency inversion)

Depend on abstractions not implementations.

(High-level modules should not depend on low-level modules; instead, they should depend on abstractions.)

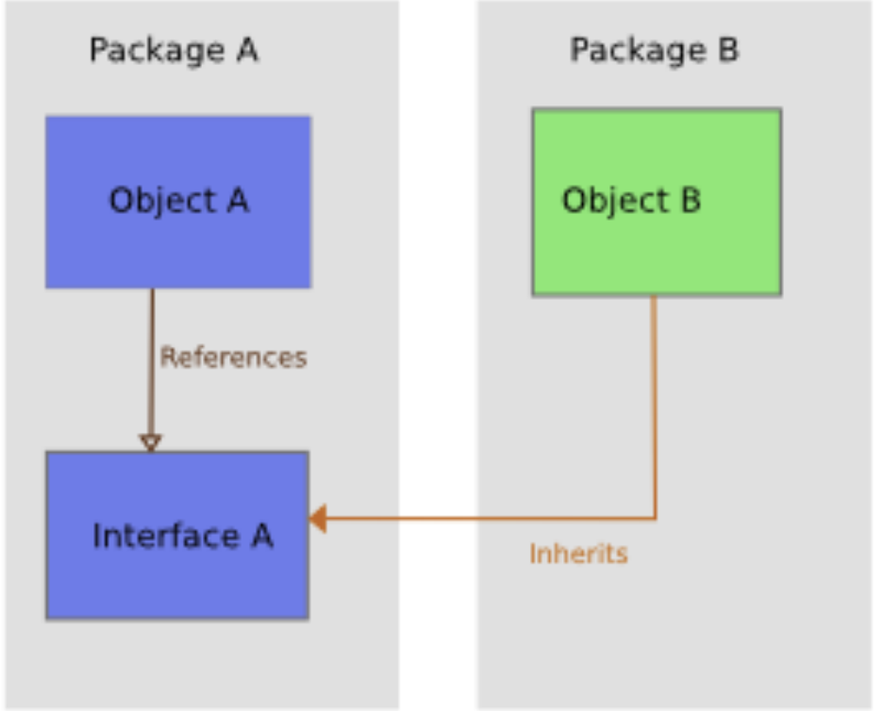
SOLID (Dependency inversion)

- From this:



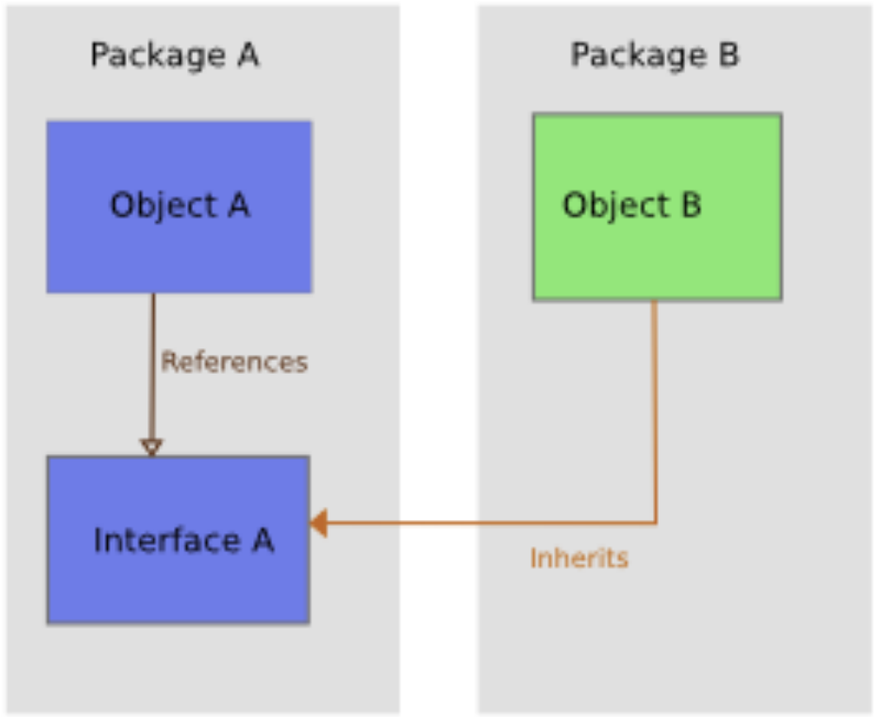
In the original version, reusing ObjectA requires reusing ObjectB. In the second, reusing A only requires an implementation of InterfaceA.

To this:

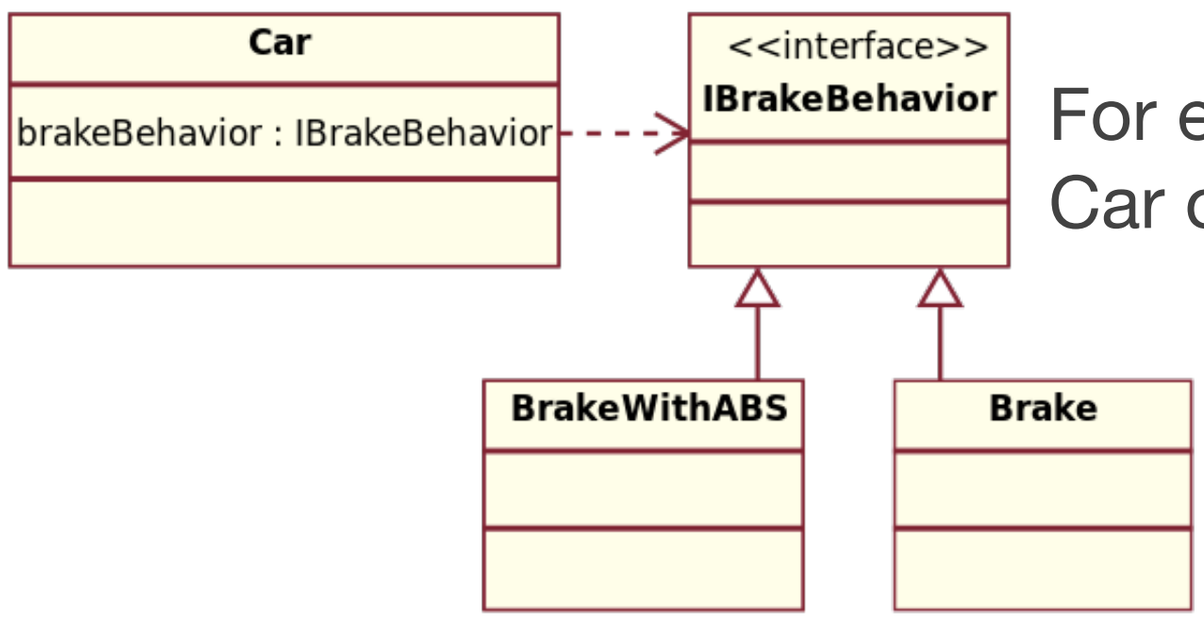


Instantiating instances of InterfaceA still ‘leaks’ details about concrete implementations; this is what Dependency Injection aims to solve.

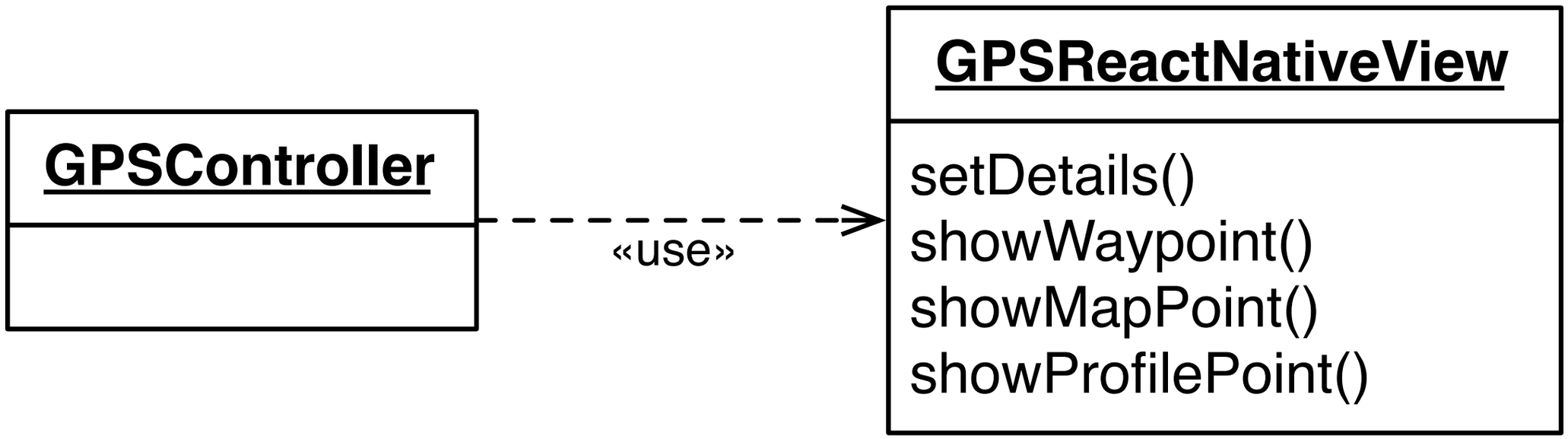
SOLID (Dependency inversion)

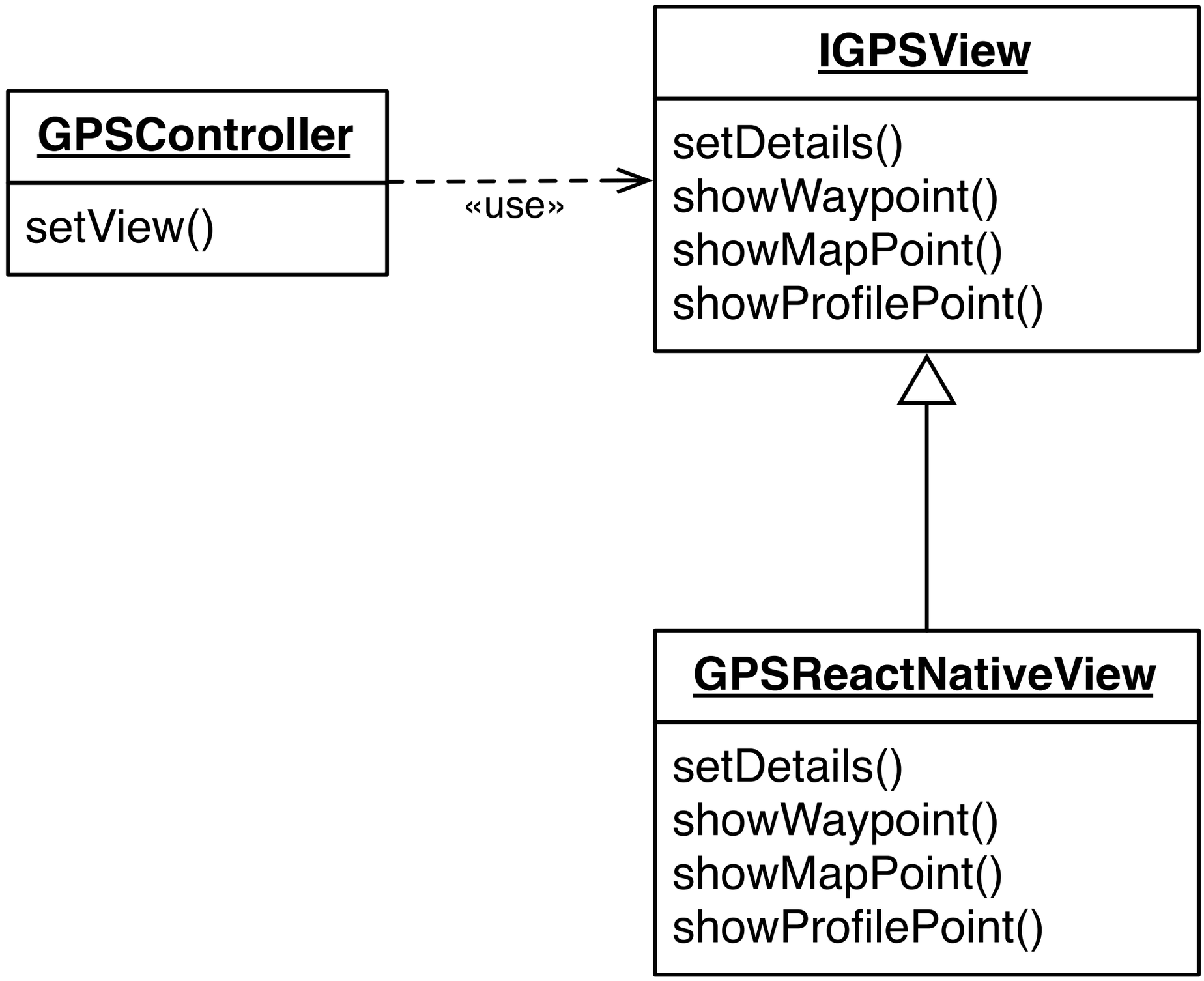


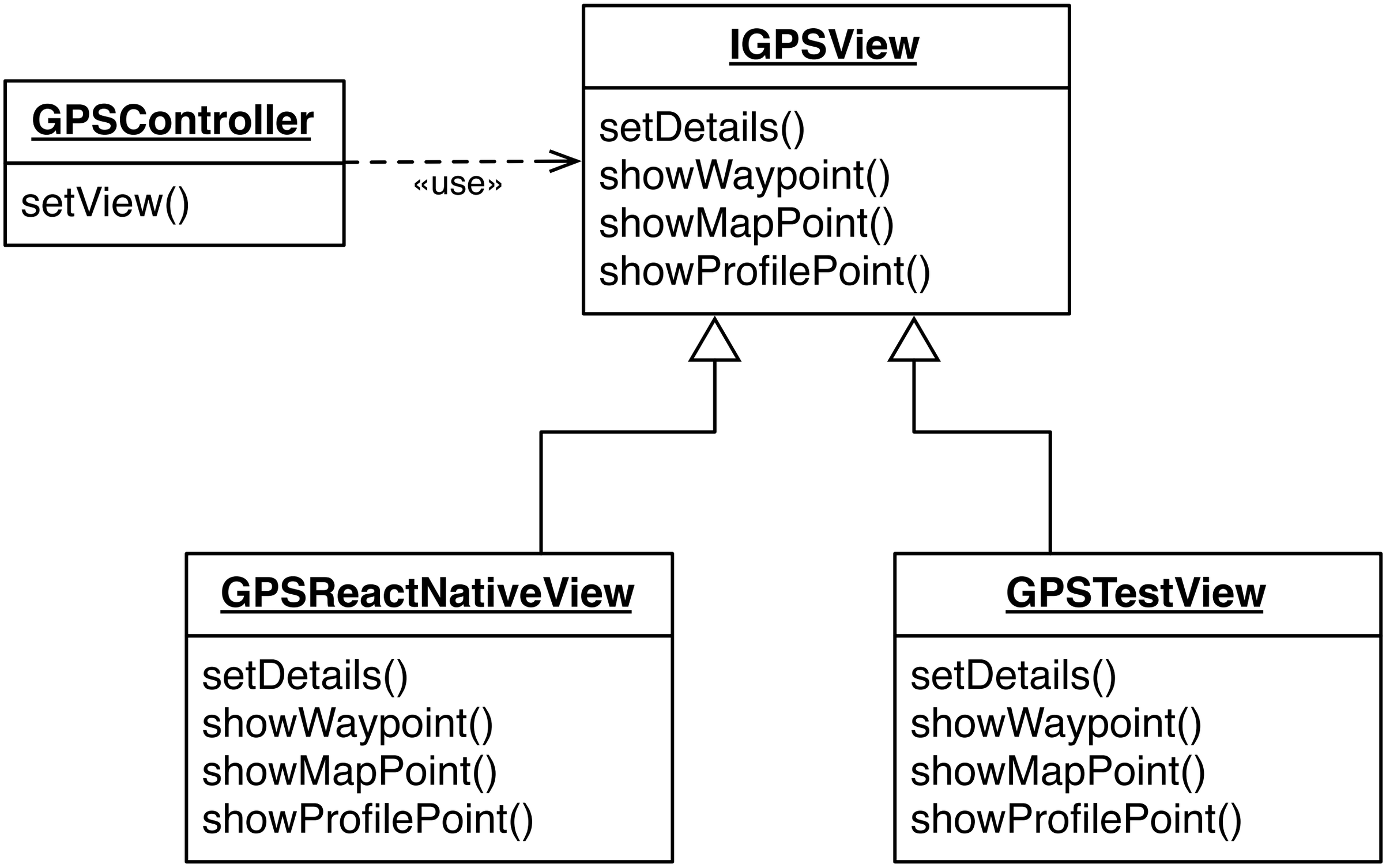
Many design patterns look just like this (from the client's perspective).

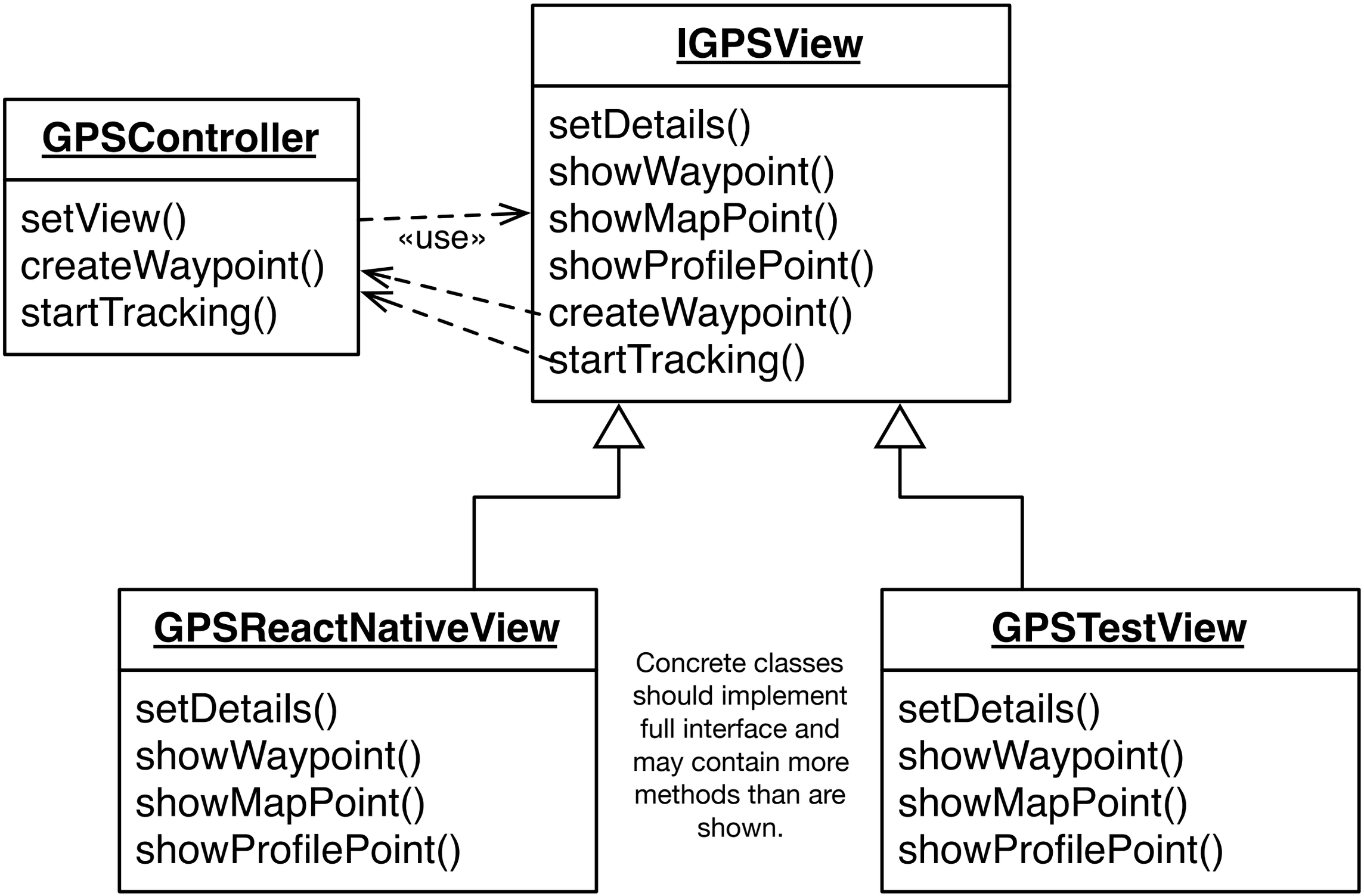


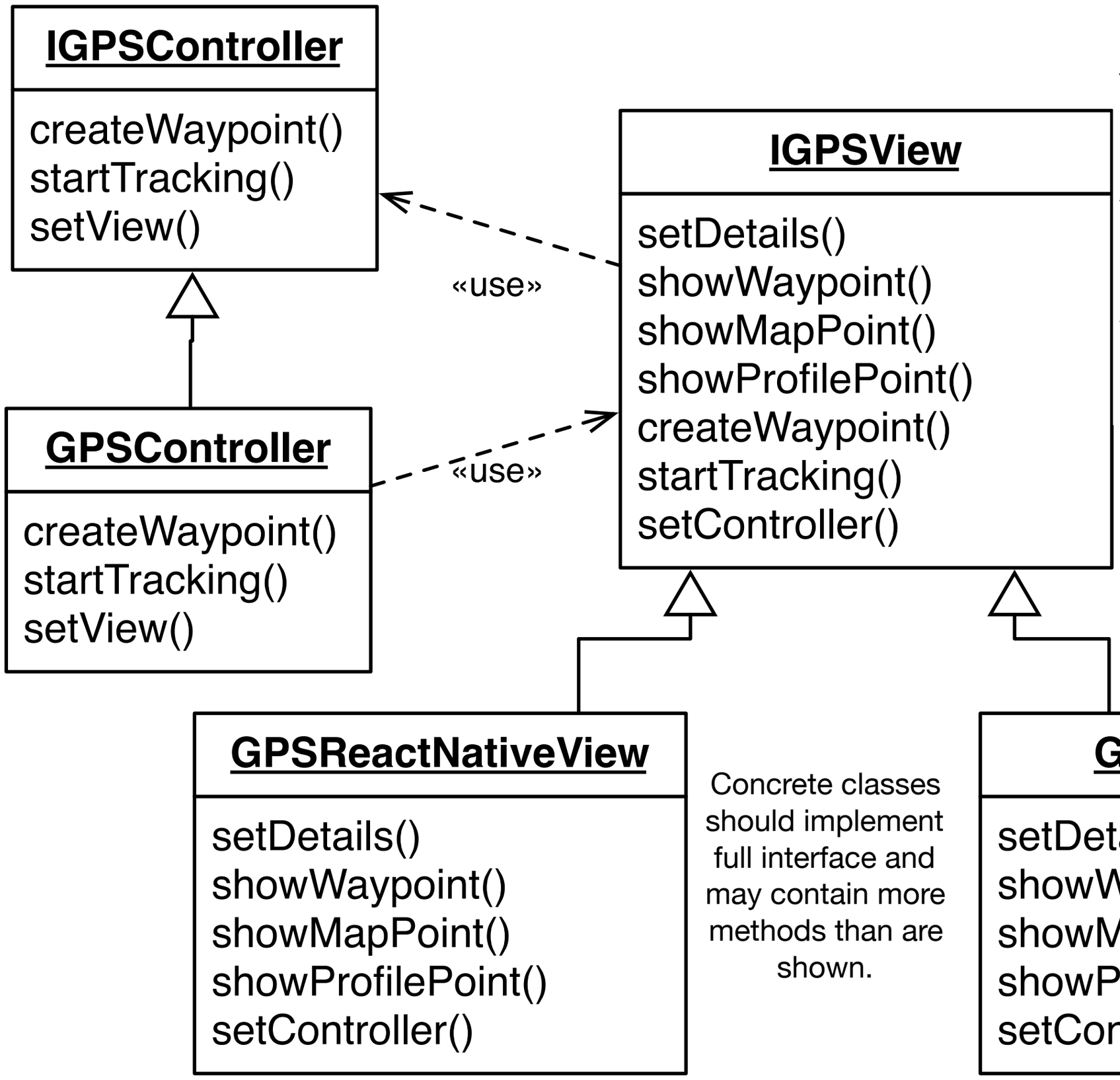
For example, in this strategy example, Car only depends on IBrakeBehavior.


GPSController
«use»
GPSReactNativeView
setDetails()
showWaypoint()
showMapPoint()
showProfilePoint()


GPSController
setView()
«use»
IGPSView
setDetails()
showWaypoint()
showMapPoint()
showProfilePoint()
GPSReactNativeView
setDetails()
showWaypoint()
showMapPoint()
showProfilePoint()


GPSController
setView()
«use»
IGPSView
setDetails()
showWaypoint()
showMapPoint()
showProfilePoint()
GPSReactNativeView
setDetails()
showWaypoint()
showMapPoint()
showProfilePoint()
GPSTestView
setDetails()
showWaypoint()
showMapPoint()
showProfilePoint()


IGPSView
setDetails()
showWaypoint()
showMapPoint()
showProfilePoint()
createWaypoint()
startTracking()
GPSController
setView()
createWaypoint()
startTracking()
«use»
GPSReactNativeView
setDetails()
showWaypoint()
showMapPoint()
showProfilePoint()
Concrete classes should implement full interface and may contain more methods than are shown.
GPSTestView
setDetails()
showWaypoint()
showMapPoint()
showProfilePoint()



This now gives you more flexibility for testing:

```
IGPSController c = new GPSController();
IGPSView v = new GPSTestView();
c.setView(v);
v.setController(c);
```

Since you have full control of which view or controller is used.

You could also reuse the view w/o the controller, just reusing its interface definition and providing your own controller implementation.