Software Engineering

Design Patterns I

Structure of the OOSE Lectures

Revisit and deepen basics of programming.

Revisit and deepen basics of object-oriented programming.

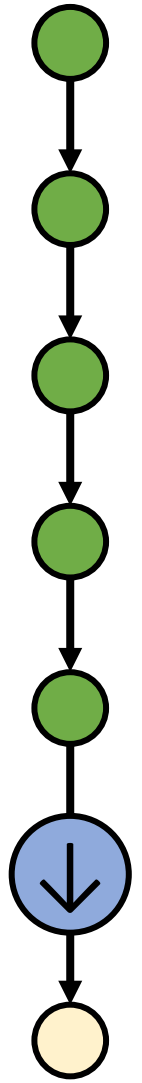
Cover advanced object-oriented principles.

How to model OO systems (UML) and map models to code.

Object-oriented modeling techniques.

Design patterns as means to realize OO concepts (I).

Design patterns as means to realize OO concepts (II).



Last Lecture

Got to know (OO) modeling principles.

About object behavior:

- Liskov
- Design by Contract.
- Behavior Protocols

About dependencies:

- SO(L)ID.
- Cohesion, Coupling, KISS, DRY and others.
- (How NOT to model software.)

Aims of this (and next) Lecture

- Understand what design patterns are.
- Get to know common design patterns:
  - Today: Observer, Facade, Singleton, Prototype, Factory Method, Composite.
  - Next time: Proxy, Adapter, Template Method, State, Strategy, Decorator, Visitor, Twins.
- See how patterns can be implemented.
- See how patterns help to solve common problems.

Design Patterns: Idea

Every discipline needs a common terminology. Software design patterns are documentations of solutions to reappearing problems that come up in a certain setting of opposing forces.

"Each pattern is a three-part rule that expresses a relation between

- a context,
- a system of forces that occur repeatedly in that context, and
- a software configuration that allows these forces to resolve themselves."

- Richard Gabriel, on the Patterns Home Page

Types of Design Patterns

Analysis Patterns: Structuring and refining analysis models.

- Martin Fowler: "Analysis Patterns", Addison-Wesley, 1997

Architectural Patterns: Define sets of subsystems, their roles and how they interact.

Software Design Pattern: Solutions to small-scale problems below component-level.

Idioms (or: *Coding Patterns*): Programming-language-specific patterns on code level.

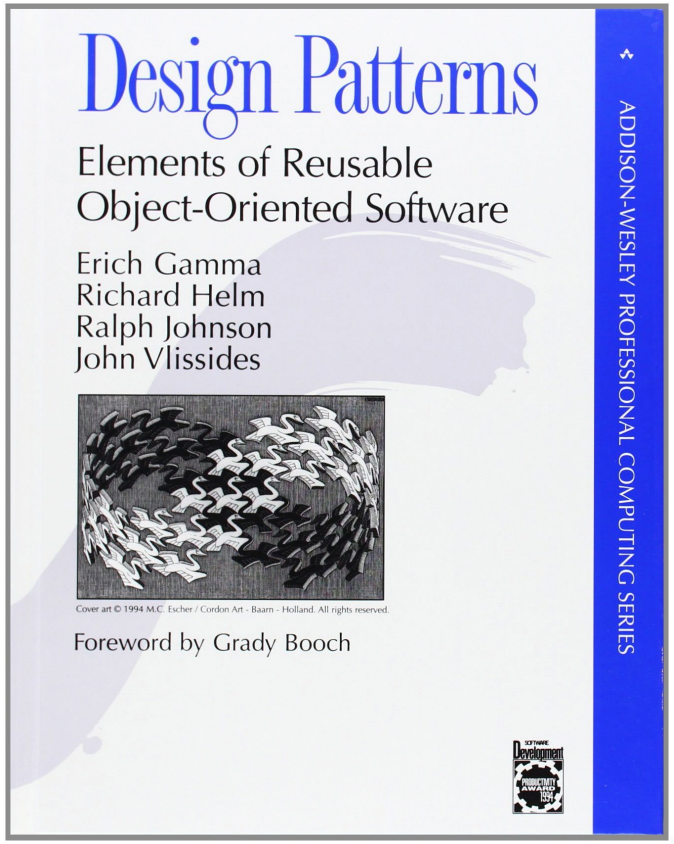
Further Types of Design Patterns

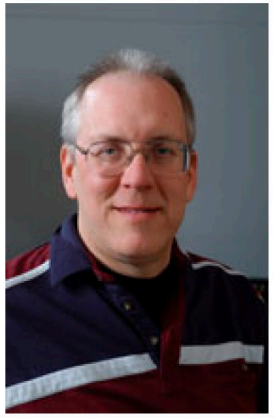
Organizational Patterns: Structuring and organizing groups of people that share a common goal.

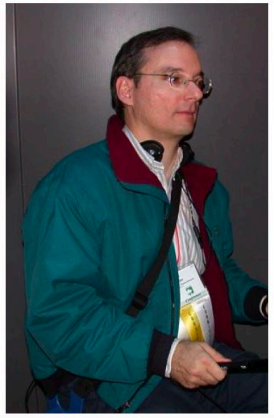
Anti-Patterns: How to not do something.

- It certainly is an anti-pattern to overdo anything you learn in the OOSE lectures ;)

"Gang of Four"







Object Creation
- Singleton
- Prototype
- Factory Method
- Abstract Factory
- Builder

Structure
- Facade
- Flyweight
- Composite
- Proxy
- Adapter
- Bridge

Behavior
- Observer
- Template Method
- Visitor
- Command
- Chain of Responsibility
- State
- Strategy
- Decorator
- Memento
- Mediator
- Interpreter
- Iterator

Guiding Questions

Throughout the following, we will answer these questions:

1. How to automatically update based on events? How to implement the MVC architecture? (Observer)
2. How to guarantee a fixed number of instances? (Singleton)
3. How to implement components? (Facade)
4. How to create instances of unknown classes? (Prototype and Factory Method)
5. How to create an object structure with propagating behavior? (Composite)
6. How to "replace" an object? (Proxy)
7. How to make two interfaces compatible? (Adapter)

Guiding Questions

Throughout the following, we will answer these questions:

8. How to enforce a certain order of instructions while making parts of them exchangeable? (Template Method)
9. How to make behavior exchangeable? (State/Strategy)
10. How to implement a state machine diagram? (State)
11. How to add/change/remove functionality to existing objects? (Decorator)
12. How to make operations work on heterogeneous object structures? How to implement double dispatch? (Visitor)
13. How to implement multiple inheritance? (Twins)

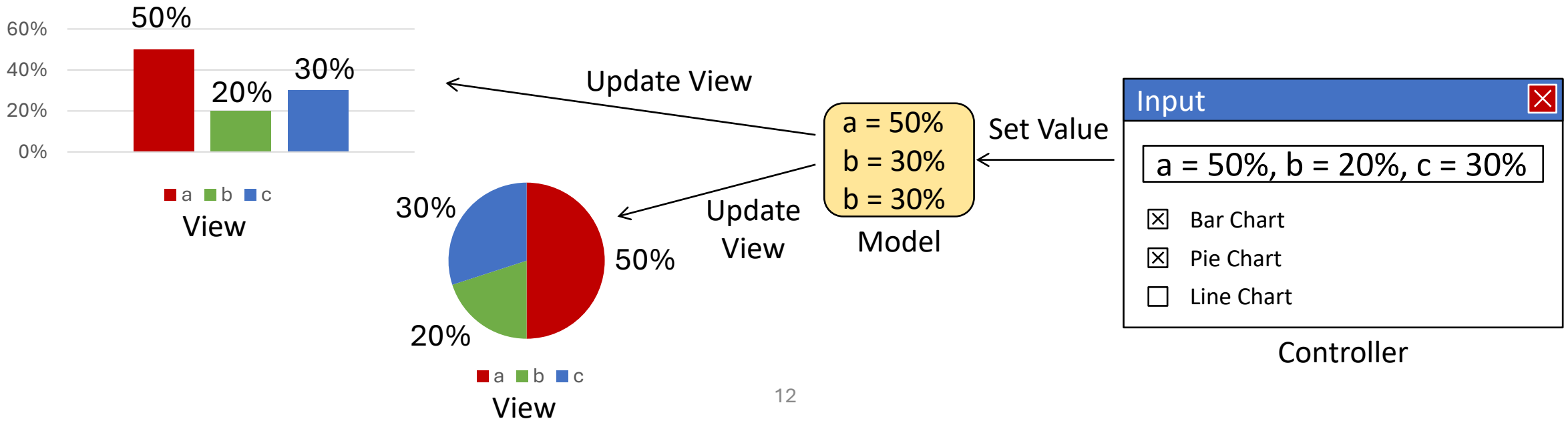
https://praxistipps.focus.de/nachbarn-beobachten-mich-das-koennen-sie-tun_104987

Observer

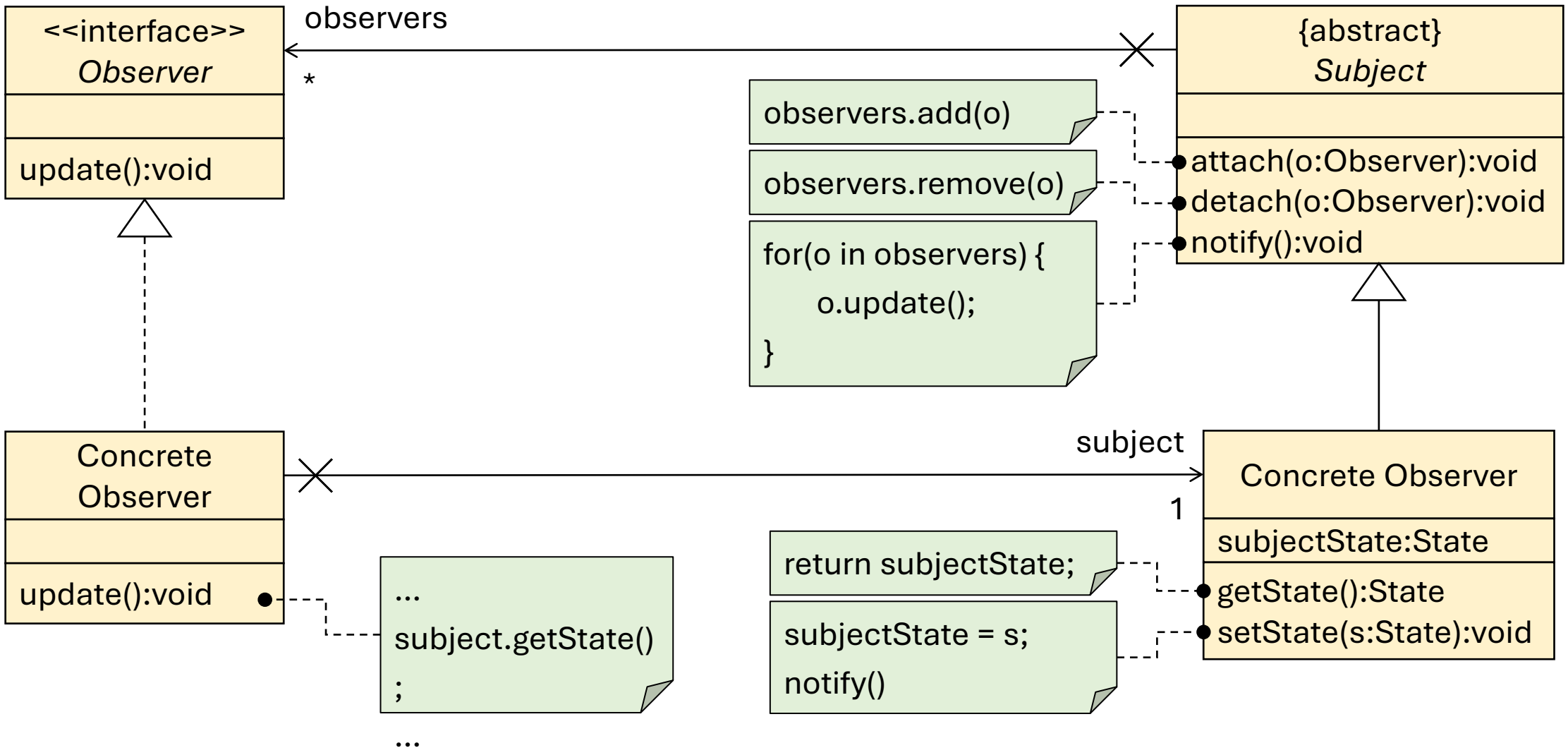
Observer Pattern

Using the observer pattern (other names: Dependents, Publish-Subscribe, Listener), we can automatically update objects if the state of a subject changes. However, these objects shall be loosely coupled.

A standard example is the consistent update of different graphical representations of a dataset, e.g. in Excel.



Structure (N:1, Pull-Model)



Roles and Responsibilities

Observer: `update()` as reaction to the state change of the subject.

Subject

- `attach(Observer o)`, `detach(Observer o)` to add or remove observers from the list.
- `notify()` to trigger the update of all observers.
- `setState(…)` as example of all state-changing operations.
- `getState()` as example of all operations revealing information about the state that may be interesting for observers.

Consequences

Independence:

- Observers can be added and removed without changing other observers or subjects.
- Observers can observe different (sub)types of subjects.

Unexpected updates: Small state changes already trigger updates of the observers, but often observers are only interested in a specific set of state changes.

- Possible solution: Push model (not in this lecture).

Implementation

The relation between subjects and observers may be implemented as a

- list (like in the example)
- change manager with a hash table. This allows to specifiy the update mechanism and defer updates.

To observe multiple subjects, the update method needs a parameter that differentiates which subject changed

- However, the question remains what type the parameter must have (not in this lecture).

Implementation

Who calls `notify()`?

- `setState()` method of the subject:
  - \+ Clients cannot forget to call `notify()`.
  - – Consecutive calls to state changing methods causes many updates.
- Clients:
  - \+ Accumulation of changes possible.
  - – Clients easily forget `notify()`.

- Use template method pattern!

Famous Applications

- Model-View-Controller-Framework in Smalltalk
- Java Foundation Classes / Swing (ActionListener)
- Client/Server-Models, e.g., Java RMI
- Push messages
- Weather Monitoring
- Publish-subscribe architectural model (but with a central broker; allows for asynchronous communication).

Consequences, Advantages, Disadvantages

+ Dependency inversion principle: Observers and subjects communicate through abstractions.
+ Open/close principle: You can add new subjects and new observers without affecting others.
+ Dynamic connecting: You can establish relations between an observer and a subject at runtime.
+ Loose coupling: Observers and subjects are not coupled tightly via direct references.

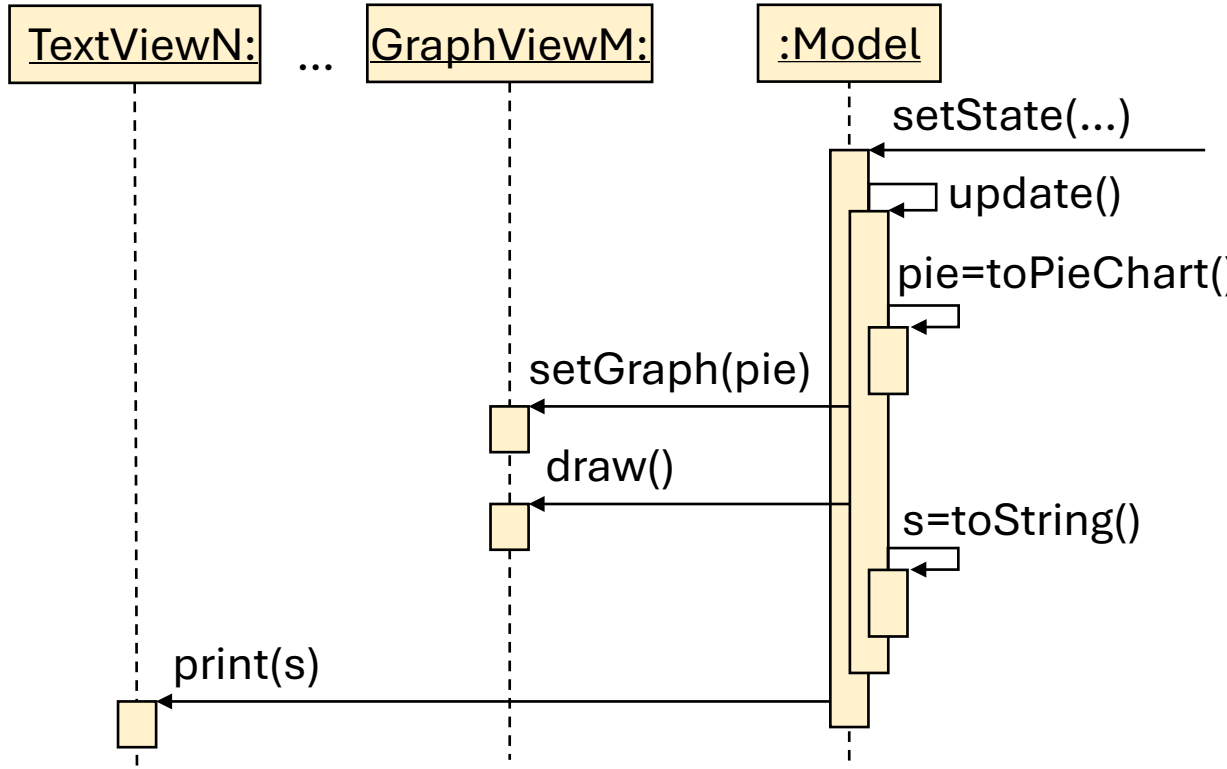
- More Code, as for all design patterns.
- More indirection (slightly slower).
- Observers are notified in unpredictable order (for this version).

Consequences of the **Observer Pattern**

Dependencies without Observer

Dependency View ← Model: A model knows the interfaces of all view types and must directly call their methods.

```
GraphView1.setGraph(this.toPieChart());
GraphView1.draw();
GraphView2.setGraph(this.toBarChart());
GraphView2.draw();
…
TextViewN.print(this.toString());
```

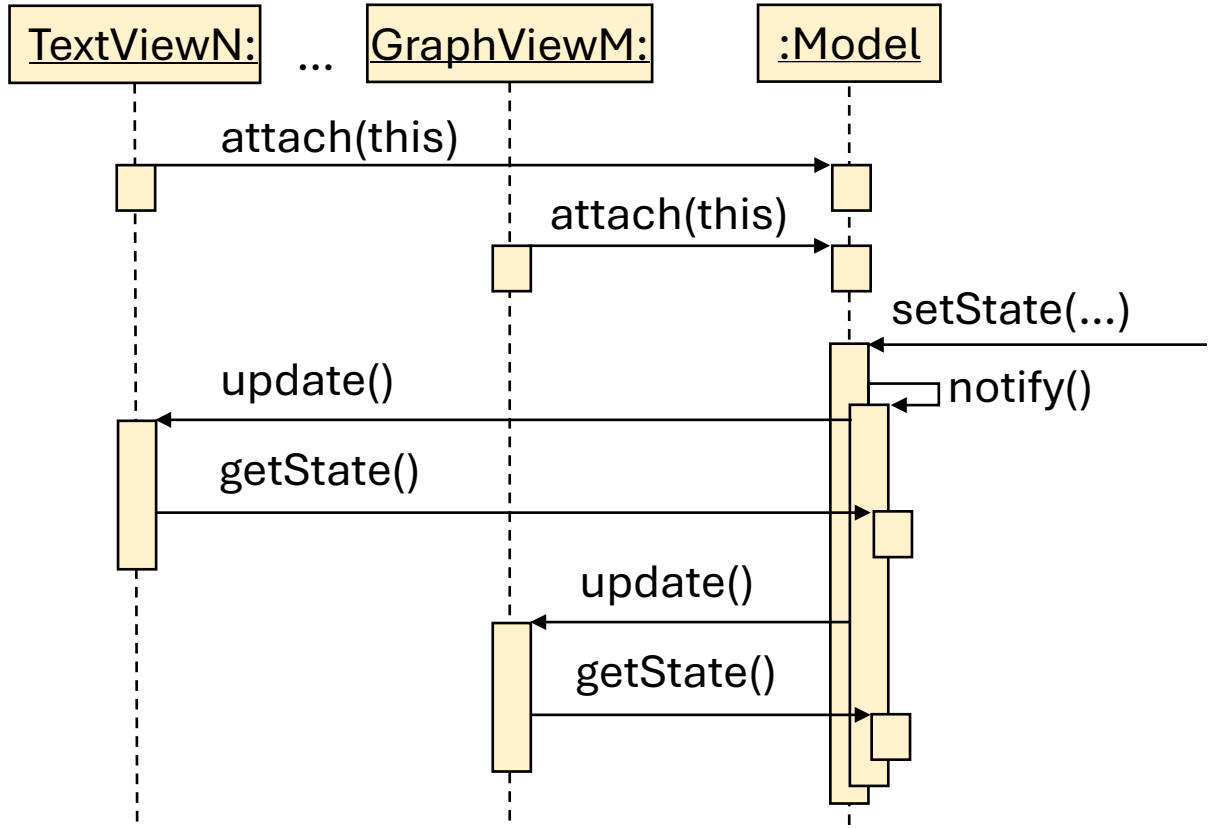


Dependencies with Observer

Dependency View → Model:

- A model only knows the abstract observer's interface.
- A concrete view knows the state-querying methods of a concrete model.

```
GraphView1.setGraph(this.toPieChart());
GraphView1.draw();
GraphView2.setGraph(this.toBarChart());
GraphView2.draw();
…
TextViewN.print(this.toString());
```



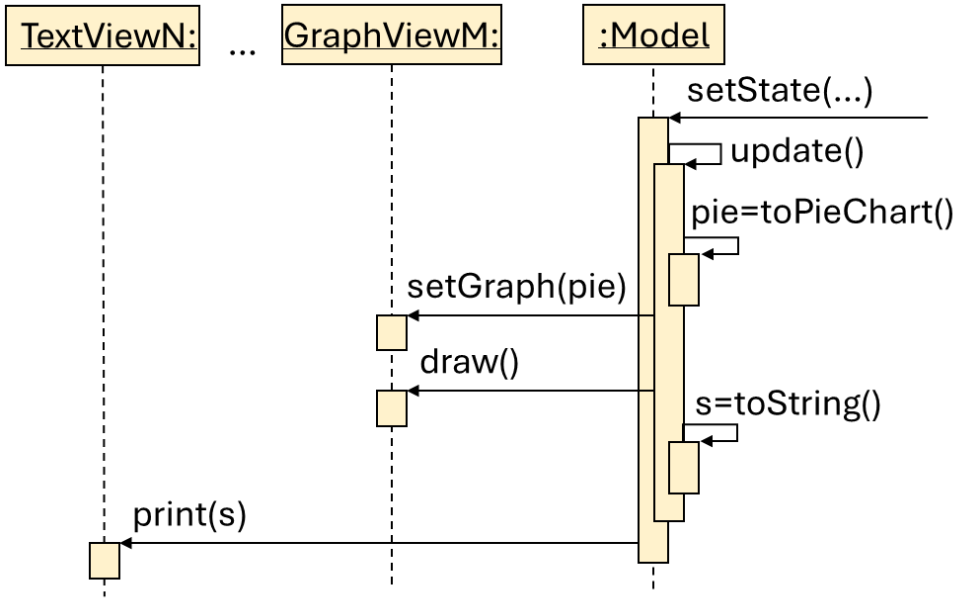
Dependency Inversion and Inversion of Control

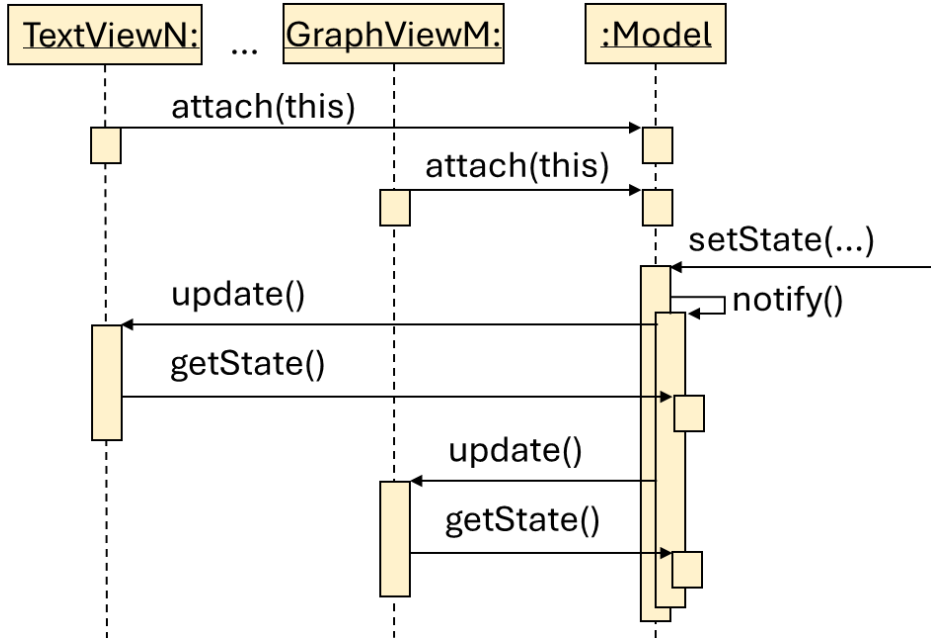
Dependency Inversion

- Without Observer: Model → Views.
- With Observer: Model ← Views.

Inversion of Control

- Without Observer: Model controls all updates.
- With Observer: Each view controls its own update.



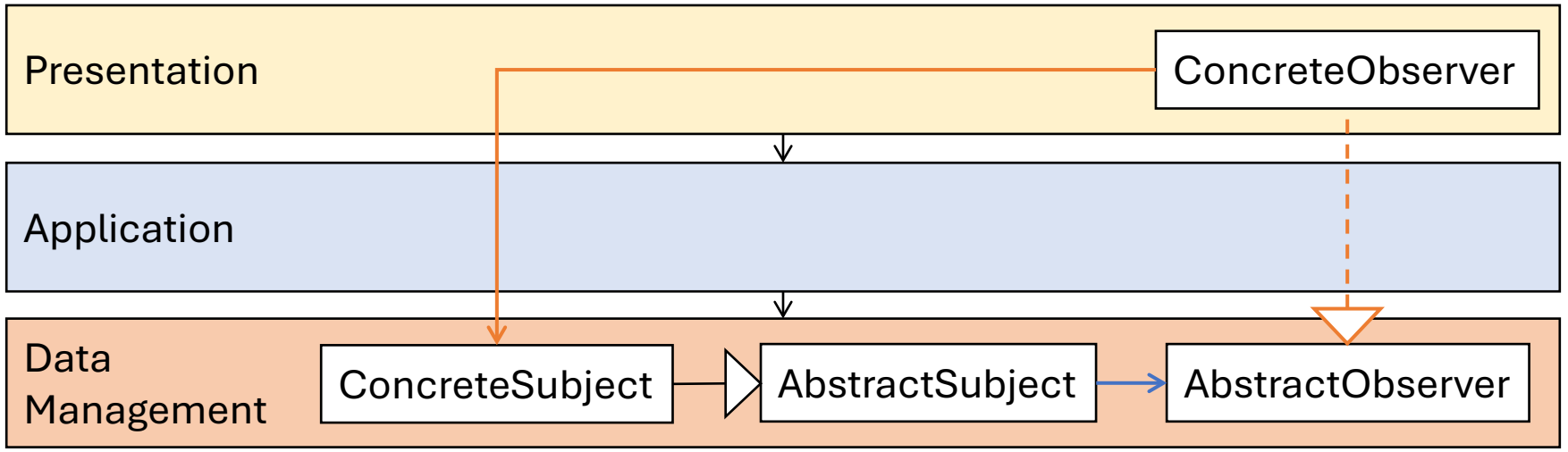


Observer and Layered Architecture

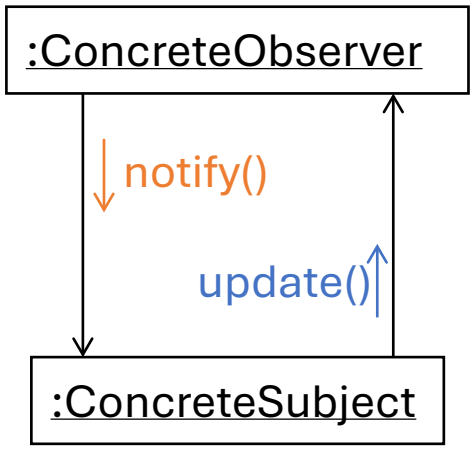
Subjects on lower layers may communicate with observers on higher layers without static dependencies violating the layer architecture.

(The same applies for objects in the application layer)

Static Dependency:
Hierarchical



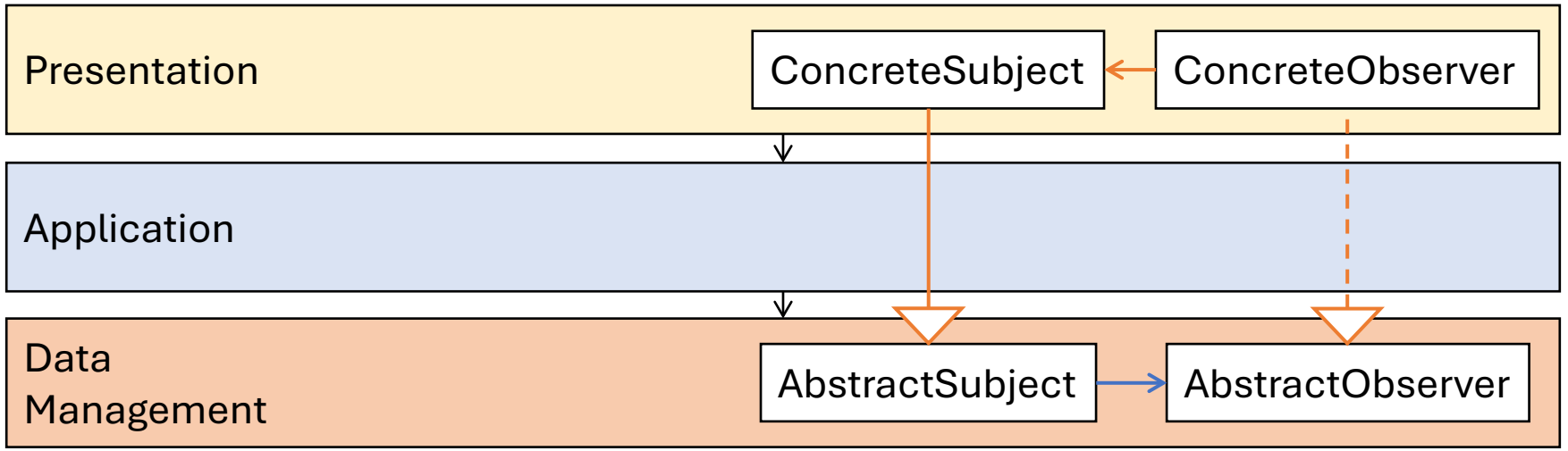
Dynamic Interaction:
Bidirectional



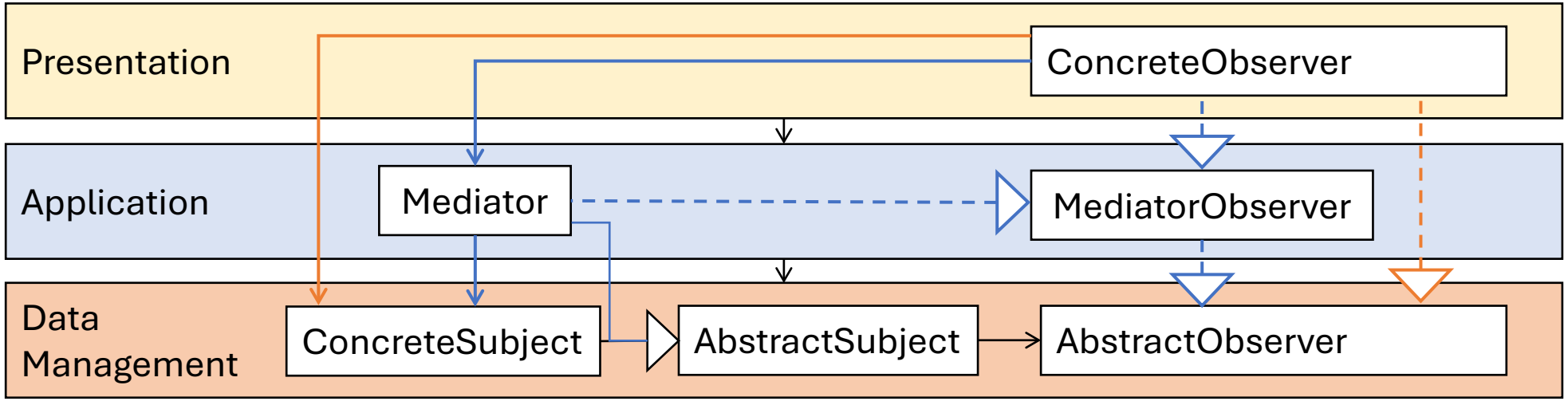
Observer and Layered Architecture

Concrete Subjects can also lie in the presentation layer -> views observe other views.

Concrete Observer must never lie in a layer below the concrete subject! Same for abstract observer and abstract subject.



Observer and Strict Layered Architecture

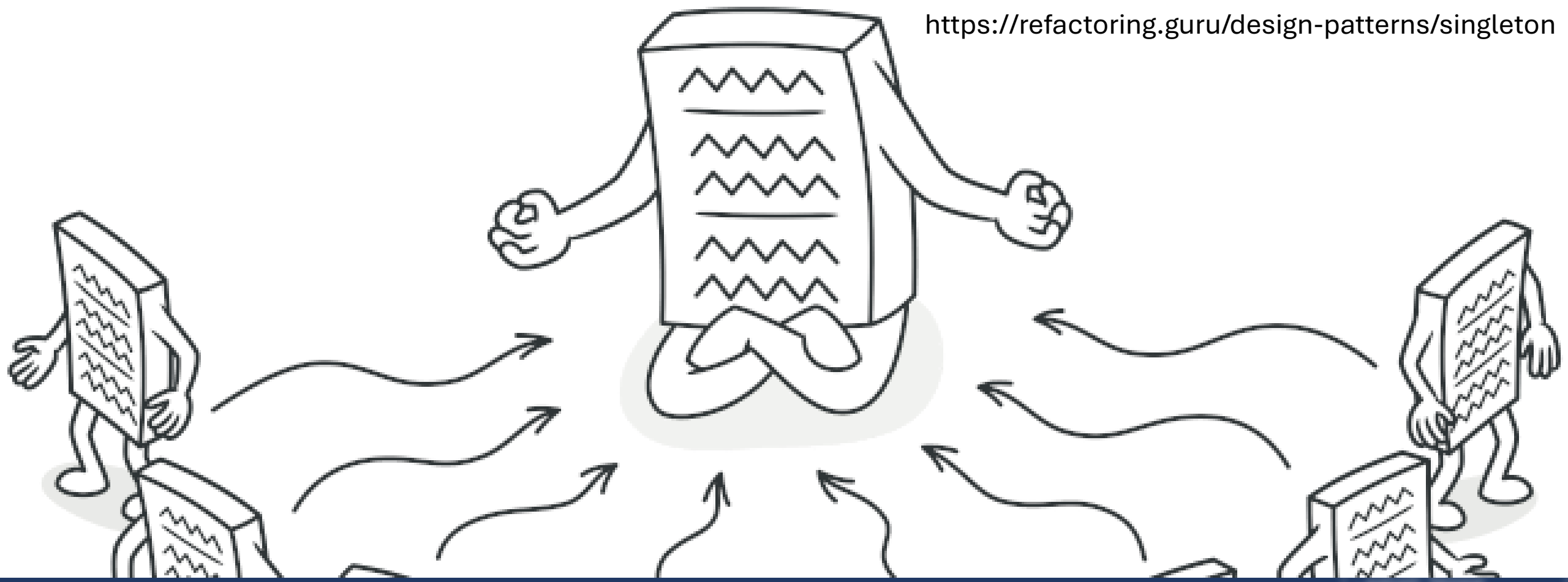


The orange dependencies violate a strict layered architecture.

The mediator play two roles: the concrete subject for the concrete observer and the concrete observer for the concrete subject.

Questions?

https://refactoring.guru/design-patterns/singleton



Singleton

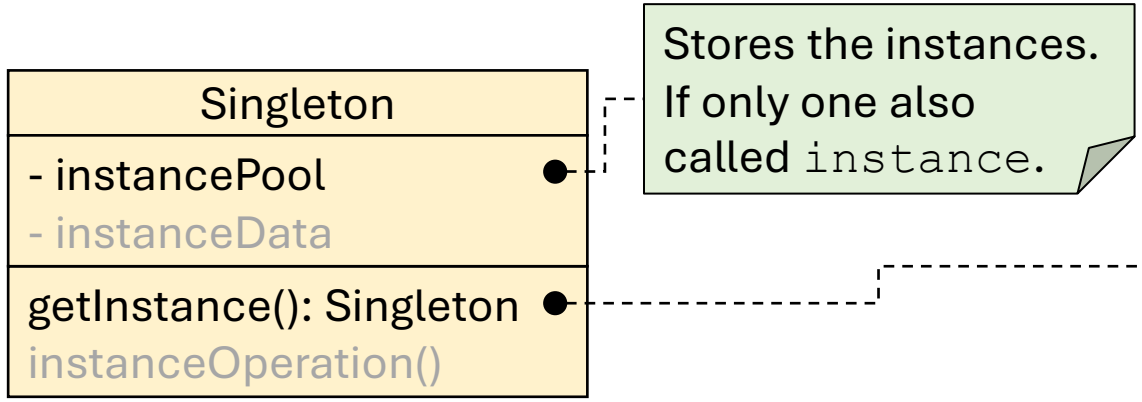
Singleton

Restrict the number of instances of a class, often only one.

Often singletons are facades, repositories, factories, ...

If we need a fixed number of instances (e.g., because of limited resources) we call that a multiton.

Schema and Implementation




```
private Singleton() {
   this.Singleton(arg1, ..., argN);
}
private Singleton(...) {  ... }
…
public static Singleton getInstance() {
   if (instancePool == null)
      instancePool = new Singleton();
   return instancePool;
}
```

The singleton only contains private constructors to prevent clients from creating arbitrarily many instances.

Attention! In Java, if no constructor is provided, the compiler generates a public standard constructor!

For more than one instance, a look up mechanism is necessary to retrieve specific instances.

Consequences, Advantages, Disadvantages

+ Guarantee that only a single instance (or a number < N) of a class exist.
+ Global access point to this/these instances.

- Global access points is basically the same as a global variable.
- Single responsibility principle: Class, additionally to its normal functionalities, also needs to care for its instantiation behavior.
- Needs special treatment in multithreaded settings.

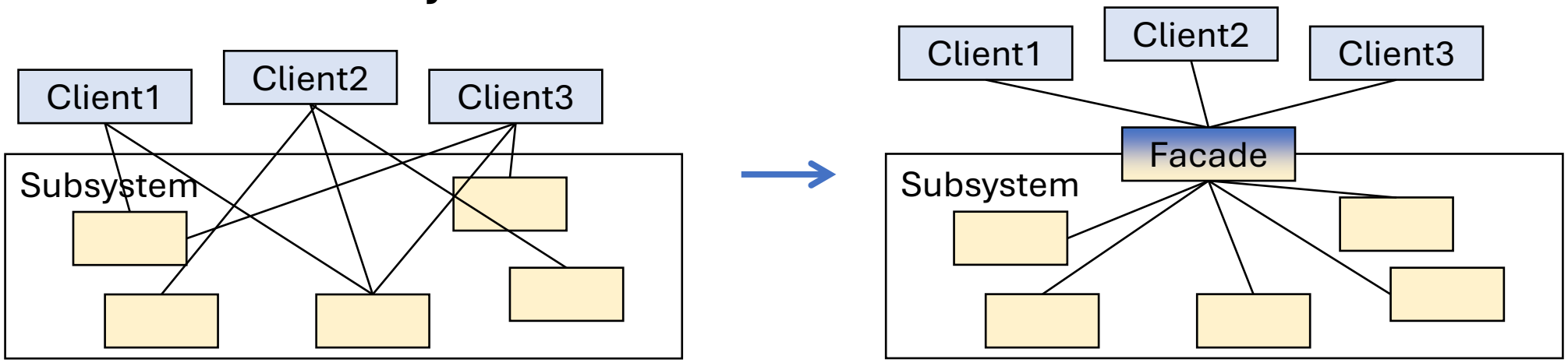
Questions?

https://de.m.wikipedia.org/wiki/Datei:Victorian_facades_on_16th_Street_in_San_Francisco.jpg

Facade

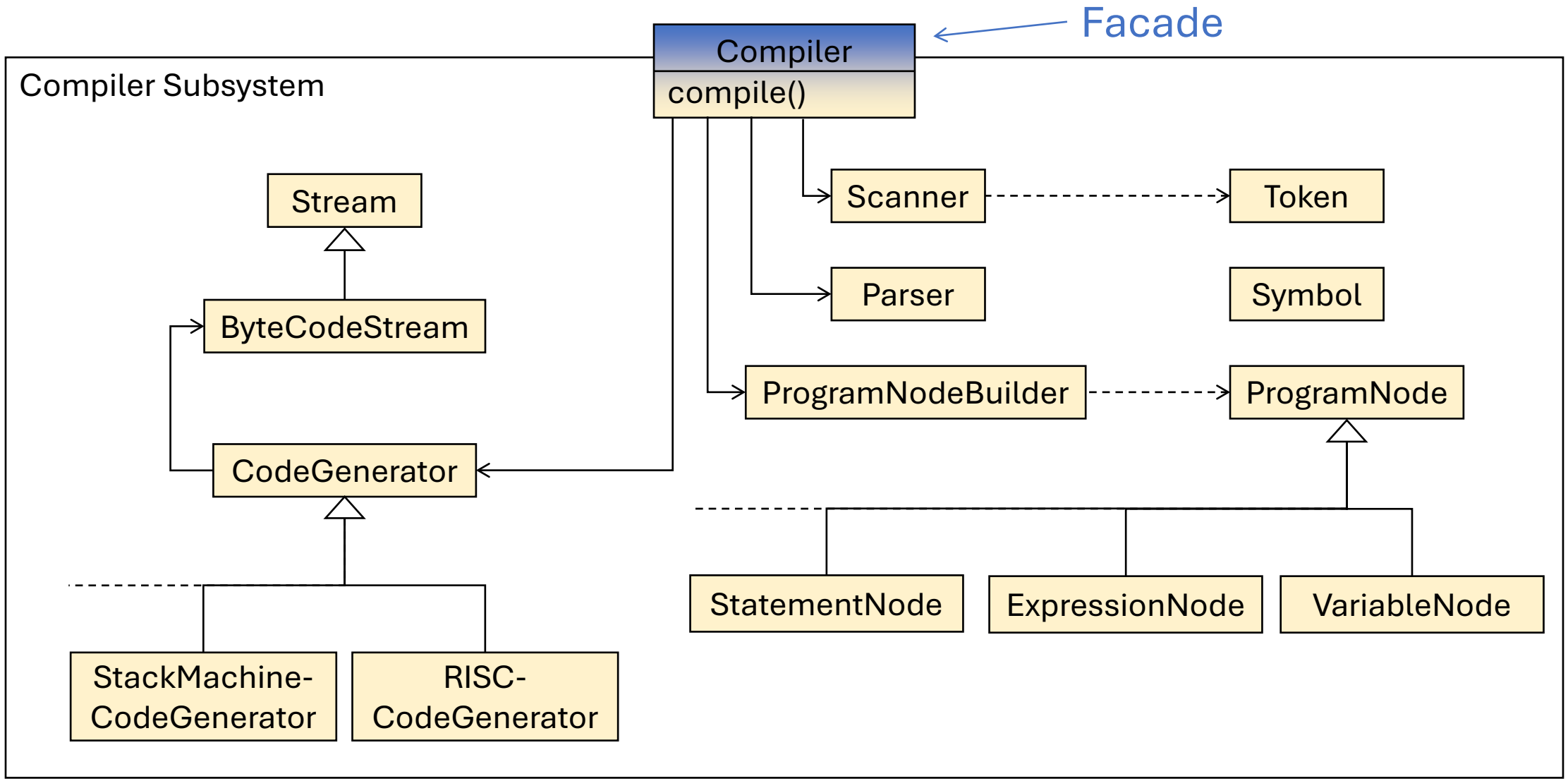
Implementing Subsystem Ports with Facades

With the facade pattern, we can reduce dependencies of clients to internals of a subsystem.



The facade object implements the service / interface of the subsystem. It is used by the clients of the subsystem and forwards all messages to the implementing classes. This way, clients only need to know the interface of the facade.

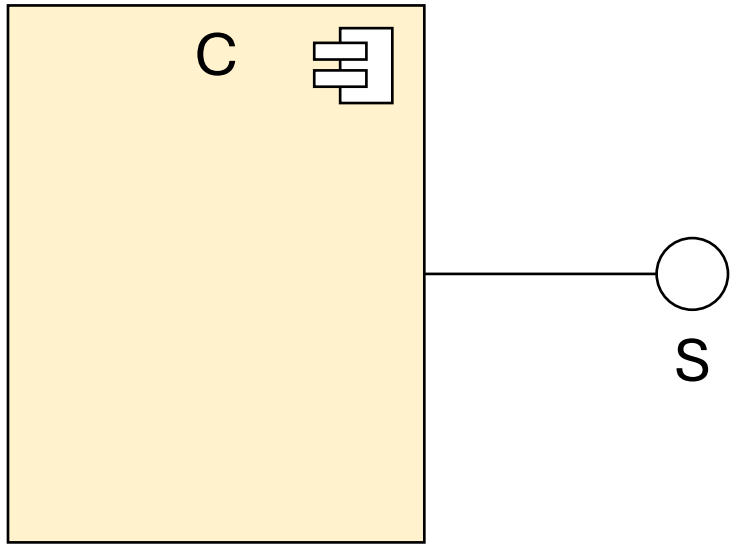
Facade Pattern: Example



Facade realizes Ports and Services

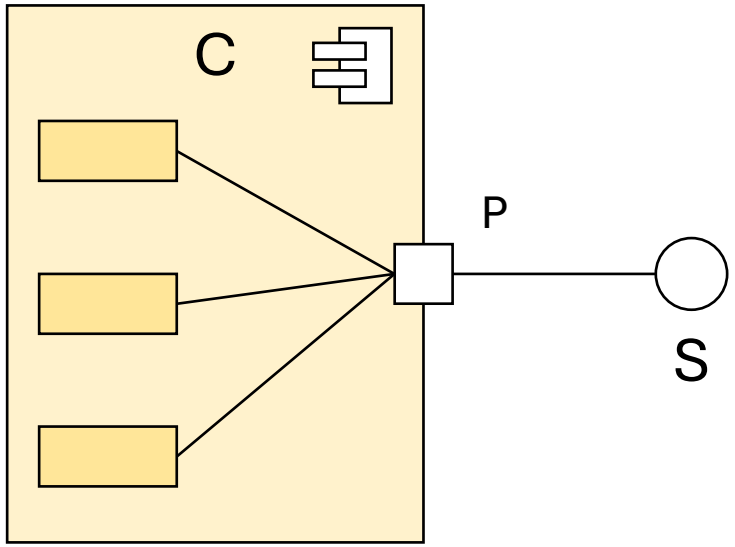
Black-Box View

Component C provides a service S.

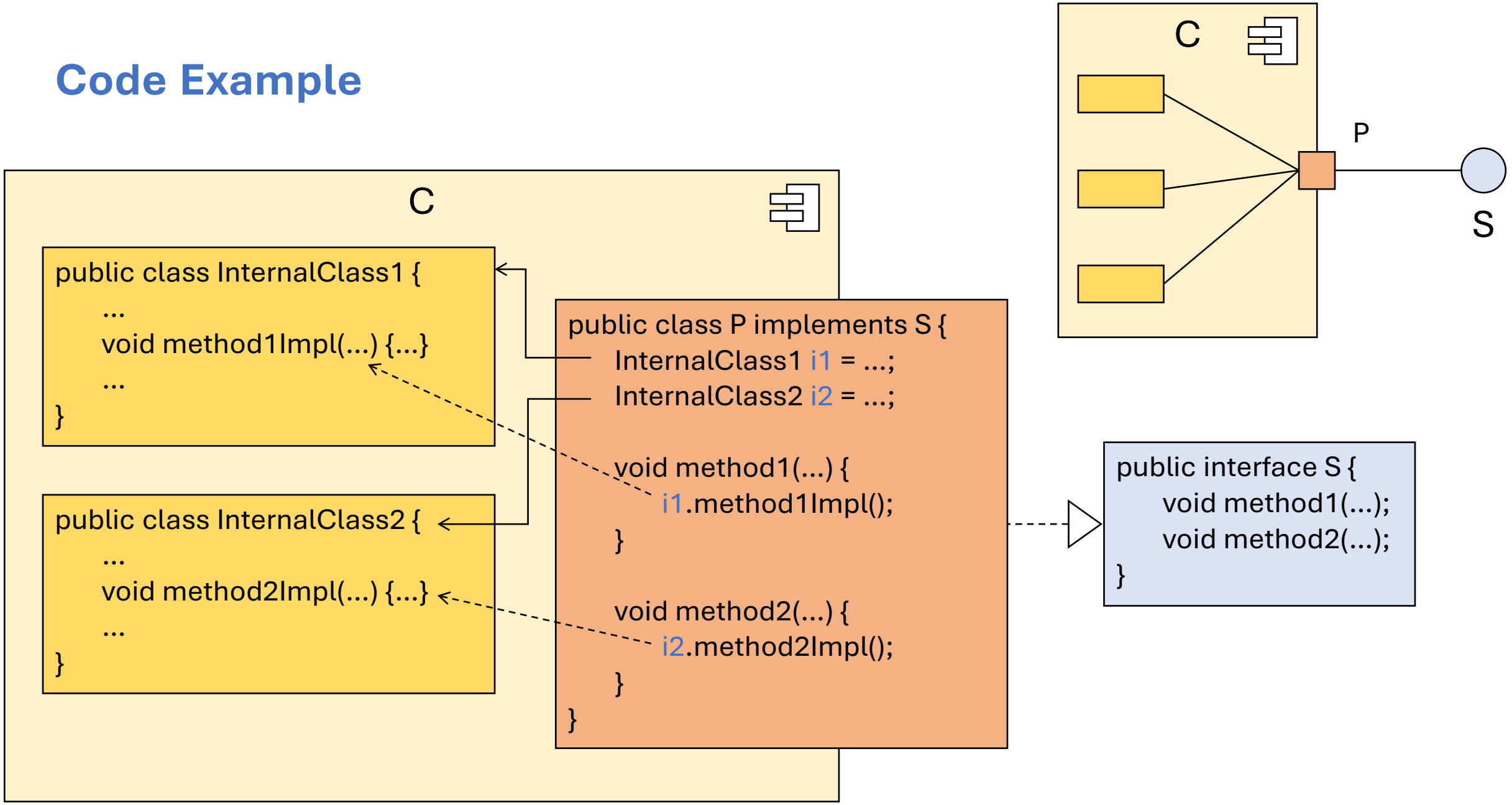


Internal View

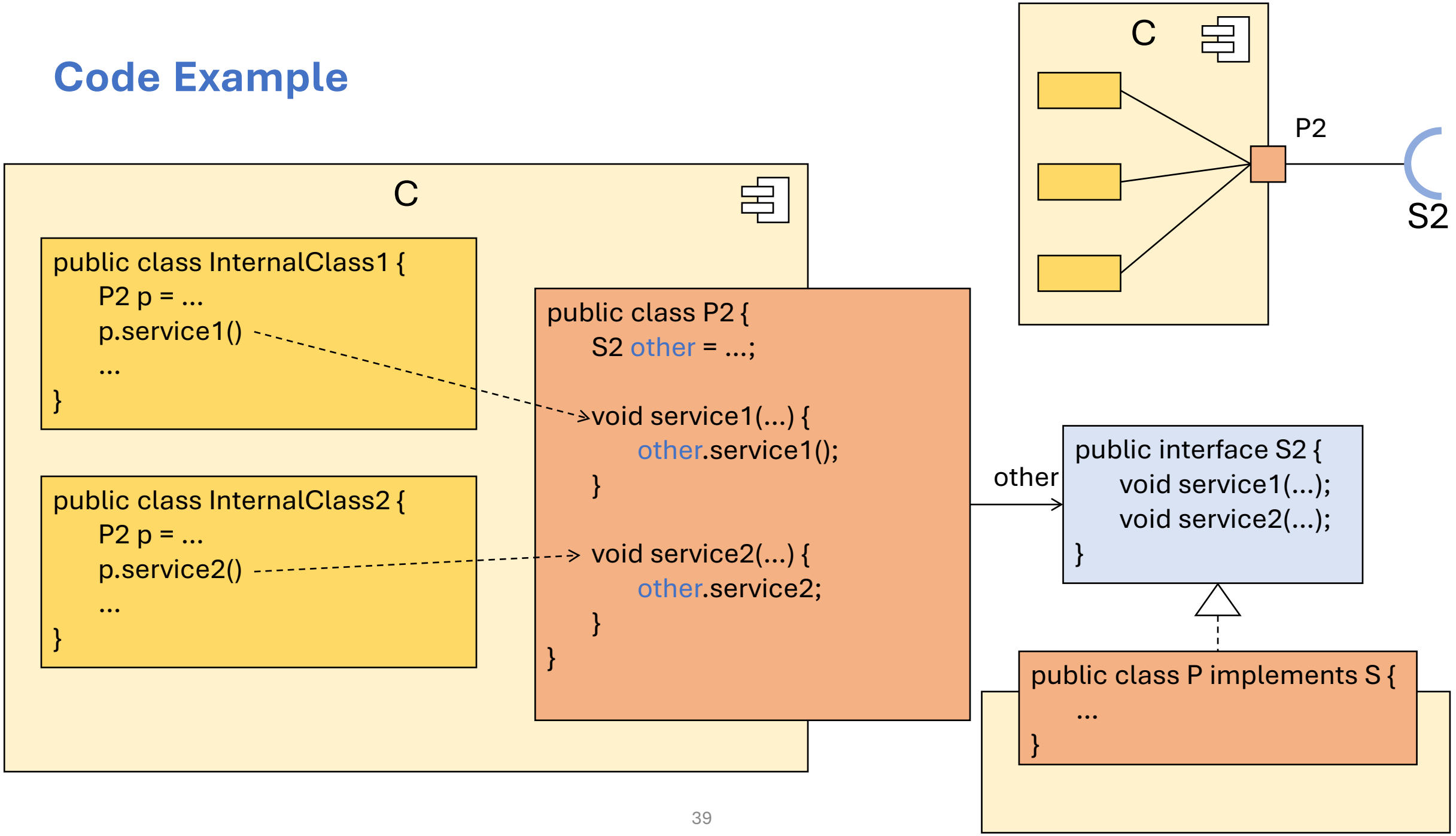
S is realized by a port P which is a facade.



Code Example



Code Example



Full Implementation of Components

Facades realize the ports of a component. What is still missing is the "black-box" property. For this, define all classes within the component that are not ports as package-private.

Usually, facades not only forward messages to its subsystem's internal objects but are also in charge of instantiating the subsystem.

Singleton: Facades (ports) are often singletons, as we only need one, or multitons, if we need multiple components of the same port.

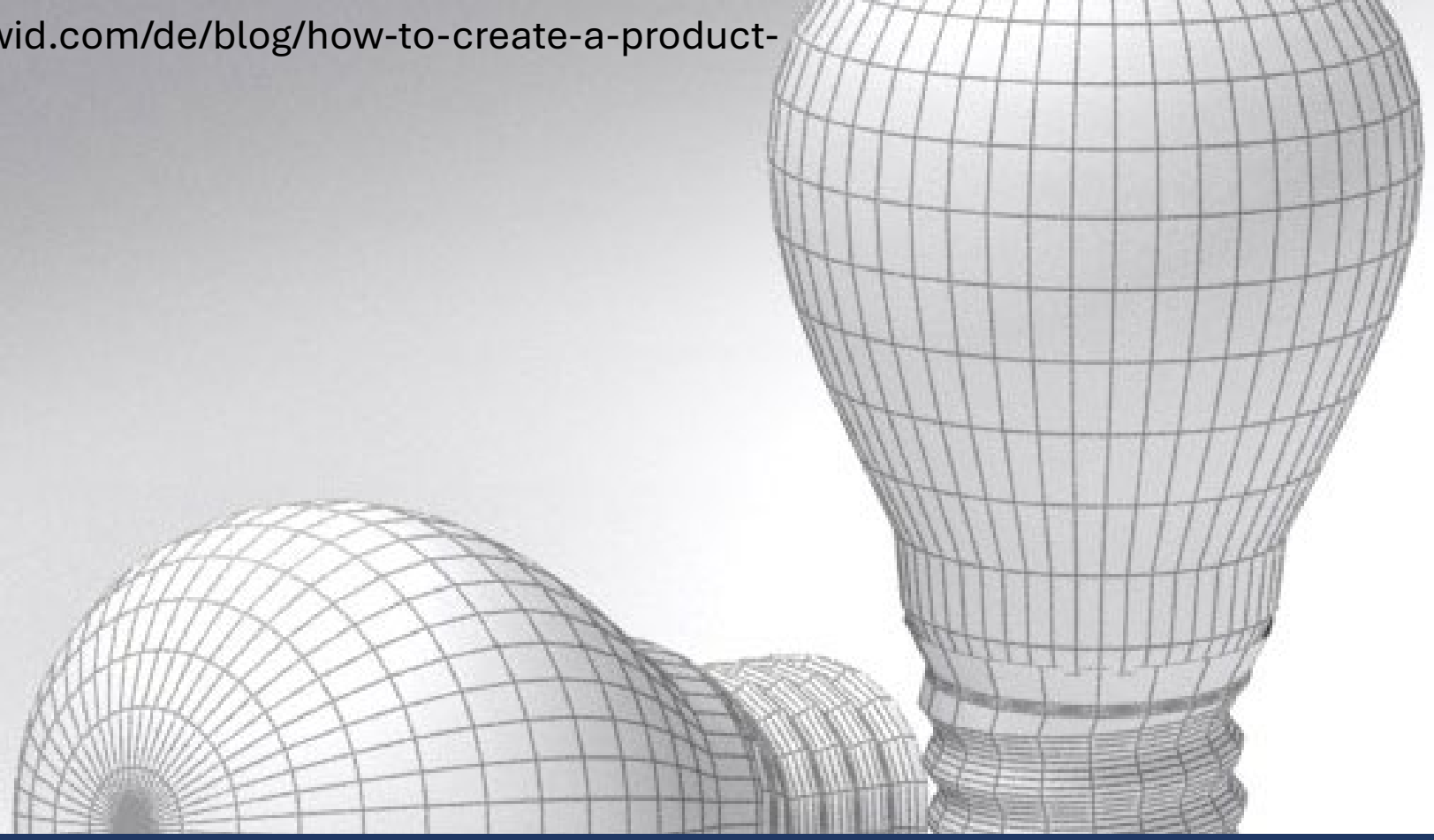
Consequences, Advantages, Disadvantages

+ Reduction of dependencies between classes of a subsystem and its users.
+ Can act as a central communication hub.

- Danger to become a god class to its subsystem.

Questions?

https://www.ecwid.com/de/blog/how-to-create-a-product-prototype.html



Prototype

Prototype Pattern

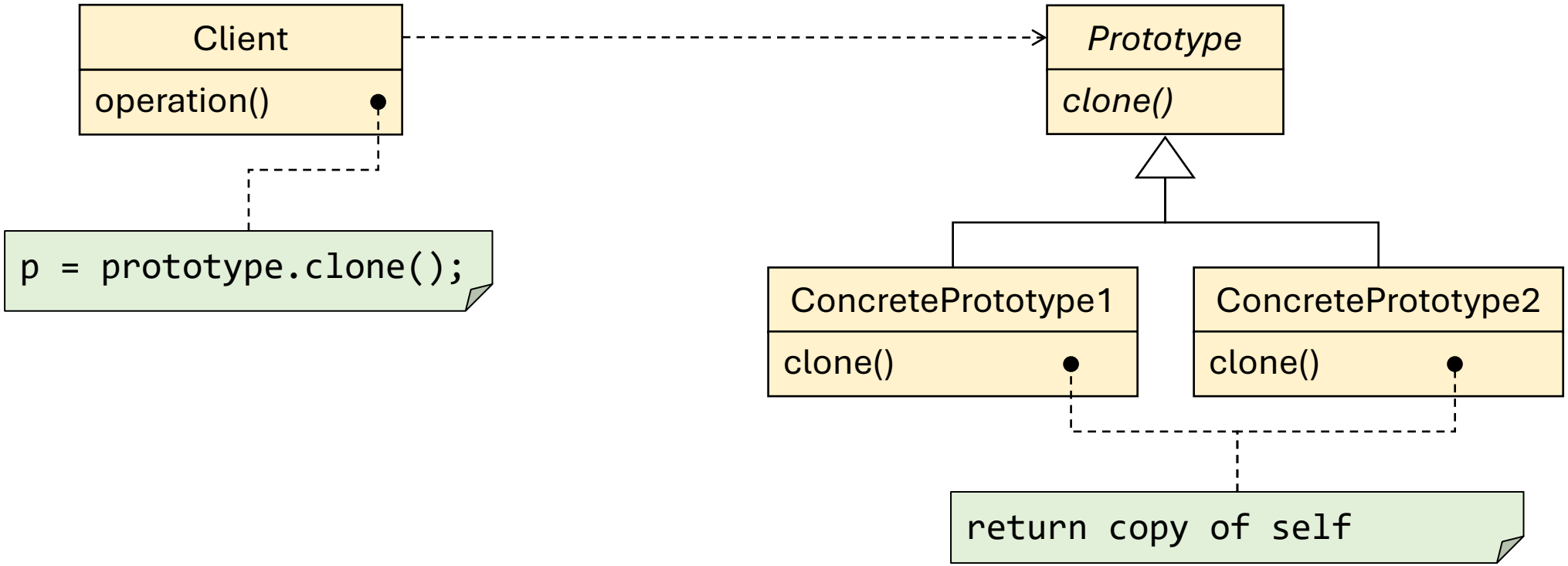
The prototype pattern enables prototype-based programming in a class-based environment.

Idea: Objects are created via cloning existing objects. Each prototype provides a clone() method.

```
void myMethod(Prototype proto) {
   Prototype clone = proto.clone(); //Instead of "new SpecificType();"
   ...
}
```

```
myMethod(new SpecificType());
```

Prototype: Schema



Prototype Pattern: Applicability

Prototypes allow to decouple from the specific structure, type, combination, ... of objects.

This way, we do not need to rely on any type to be known up-front. This comes in handy especially when we want to "plug-in" new classes, either in code or at runtime by loading classes.

It is also handy when there are only a few feasible combinations of state. Then, it is easier to create prototypes of these combinations up-front and clone them rather than instantiate new instances each time with an allowed set of attributes.

Prototype Pattern: Consequences

Object creation becomes dynamically adjustable, as the specific type of the object to be created is deferred.

`prototype.clone()` instead of `new ClassName()`

Factory methods achieve a similar goal but introduce an additional hierarchy of factory classes. The prototype pattern does not need additional class hierarchies.

Prototype Pattern: Implementation

Initializing clones: Since a uniform interface of the `clone()` method is key to the prototype pattern, we cannot pass arguments to it to modify the clone. Instead, introduce an initialize method or use existing getter and setter.

Alternatives: Class-objects in reflective languages.

- E.g., via `newInstance()` method in Java.
- This is effectively prototype-based programming.

Consequences, Advantages, Disadvantages

+ Open/close principle: You can extend existing client code using new subclasses, even dynamically at runtime.
+ Dependency inversion principle: The client code operates with the object through the common `Prototype` interface. The dynamic type can be an arbitrary implementing class.

- More Code, as for all design patterns.
- Instantiation is no longer Initialization (we need an additional init method).

Relation to other Patterns

- Abstract factories may use factory methods or prototypes.
- Prototypes can be used to save the same Command multiple times into a history.
- Factory method and prototype both achieve type-independent instantiation. Factory method is based on class-inheritance (static), prototype does not rely on inheritance and is thus more dynamic.
- Prototypes can be implemented as singletons.

Questions?

Factory Method

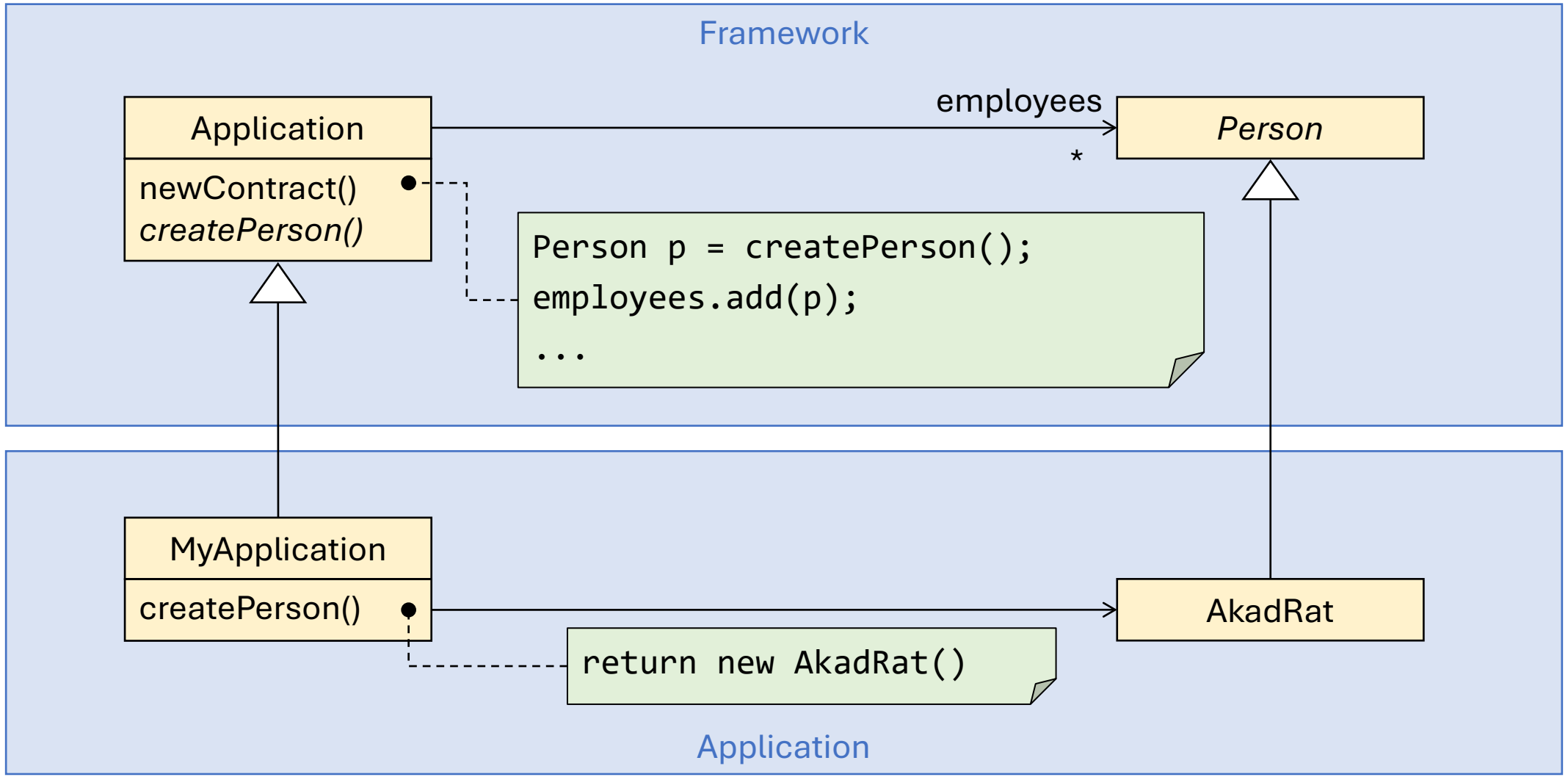
Factory Method

The factory method pattern allows us to defer the specific type of an object to-be-created to the runtime.

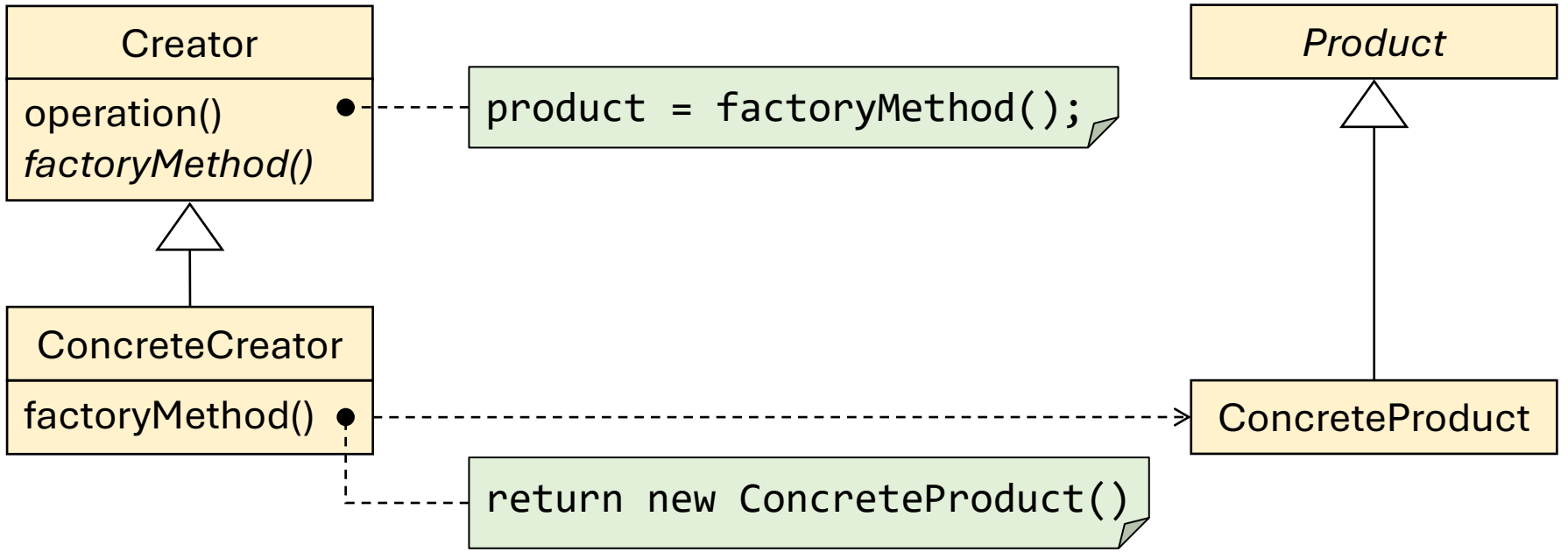
This is especially handy in frameworks. The framework defines the interfaces of the concrete products, and the main code uses the factory methods instead of constructors. We then "plug-in" the factory methods that perform the instantiation.

This way, the framework decides when an object is created, and our plugin decides what class the object is instantiated from.

Example


Framework
Application
newContract()
createPerson()
employees
*
Person
Person p = createPerson();
employees.add(p);
...
MyApplication
createPerson()
AkadRat
return new AkadRat()
Application

Schema



- Factory method may be abstract or have a default implementation.
- Creator defines the interface of the factory method.
- Concrete creator redefines factory method as needed.

Implementation Variants

Fixed product type: each subclass creates one product type.

+ Advantage: Easy extensible.
- Disadvantage: New subclass per product type.

```
class Creator {
   Product create() {
      return new MyProduct();
   }
}
```

Parameterized factory method, where the parameter value encodes the product type.

+ Advantage: Only one or few subclasses needed.
- Disadvantage: Caller needs to pass deciding argument → Coupling to given set.

```
class Creator {
   Product create(int id) {
      if (id==1)
         return MyProduct1();
      if (id==2)
         return MyProduct2();
      ...
    }
}
```

Application and Consequences

- Support creating instances of a class that are not known in advance.
- Code becomes more abstract and thus more reusable, as it is less dependent on specific types.
- Artificial classes just for subclassing.
- Real world example: Smalltalk's Model-View-Controller Framework
  - FactoryMethod-Template: defaultController
  - Hook-Method: defaultControllerClass

Distinction and Relation to other Patterns

- Template method: often calls factory methods.
- Abstract factory: uses factory methods.
- Prototype: alternative to avoid introducing new classes but needs `initialize` method.

Consequences, Advantages, Disadvantages

+ Open/close principle: You can extend existing client code using new subclasses, even dynamically at runtime.
+ Single Responsibility Principle: The code for object creation is extracted to its own class.
+ Dependency inversion principle: The client code operates with the object through the common `Creator` superclass. The dynamic type can be an arbitrary derived class.
+ Does solve the limitation of the Prototype pattern.

- More Code, as for all design patterns. Here especially, as we need one new subclass per new type.

Relation to other Patterns

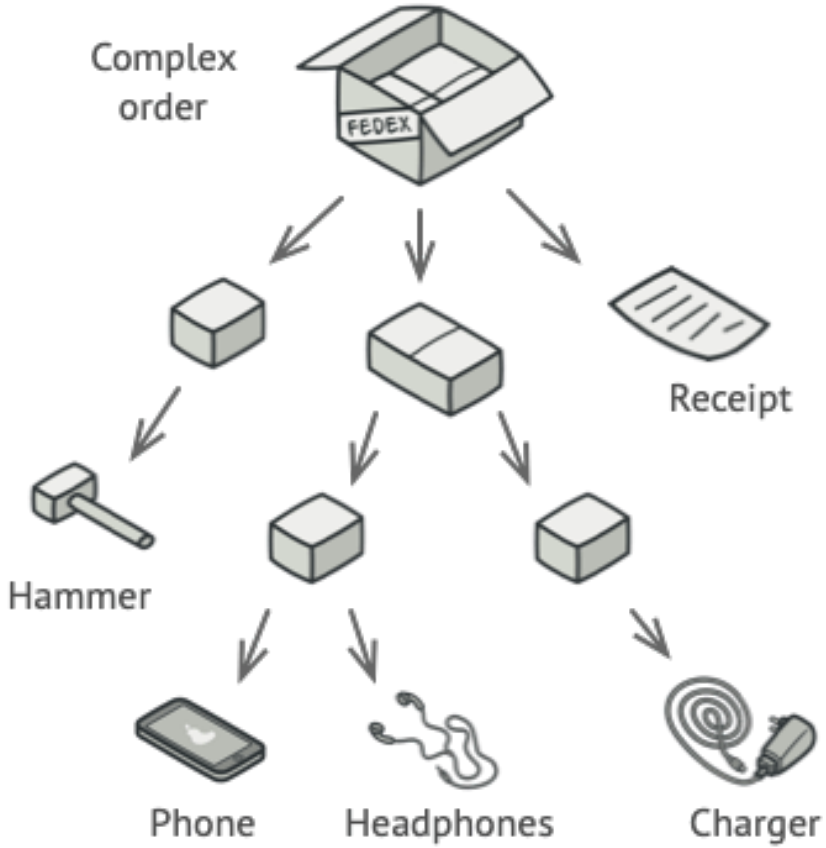
- Abstract factories may use factory methods or prototypes.

- Factory method and prototype both achieve type-independent instantiation. Factory method is based on class-inheritance (static), prototype does not rely on inheritance and is thus more dynamic.

- Factory method is a special template method.

Questions?

https://refactoring.guru/design-patterns/composite

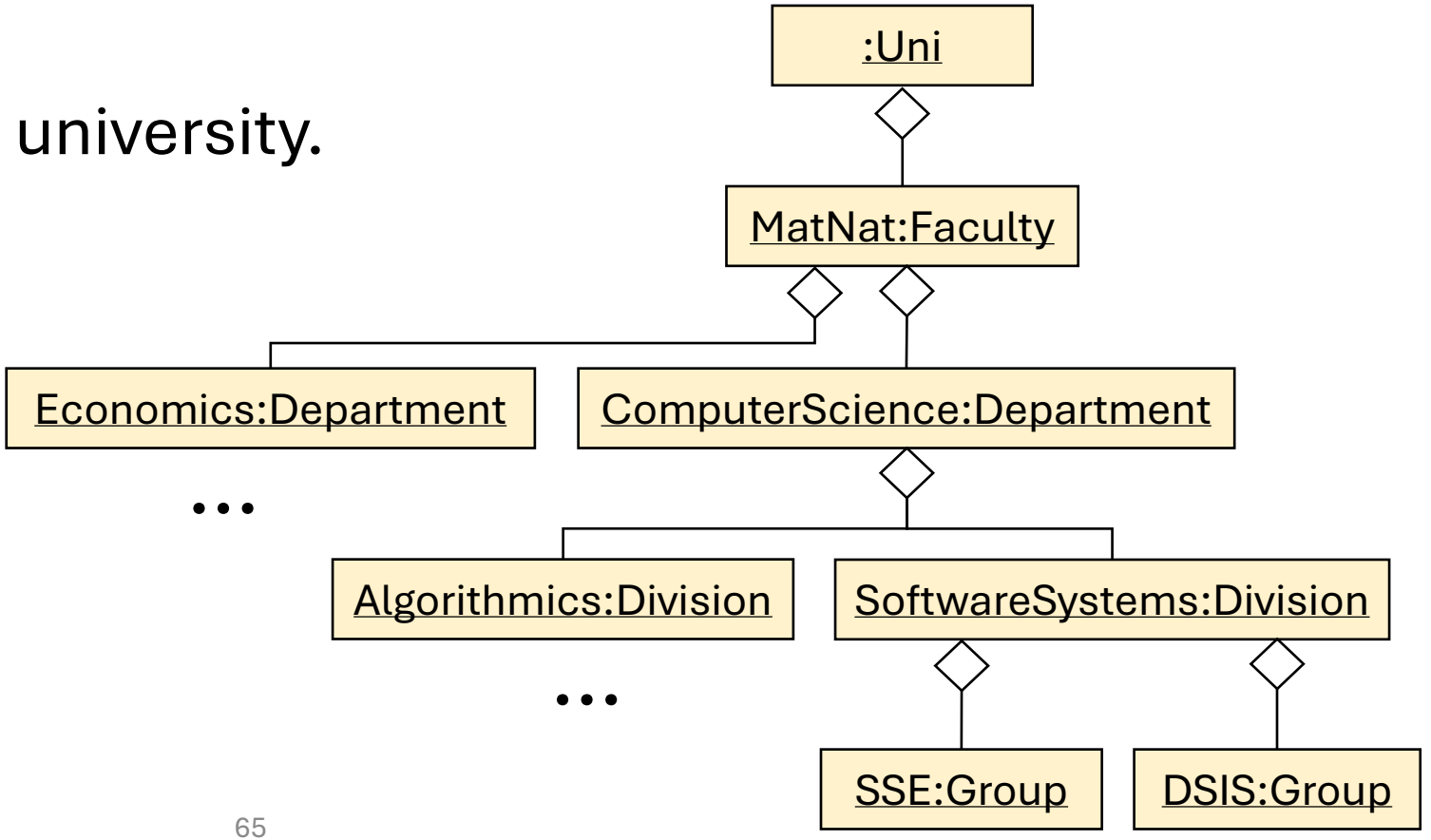


Composite

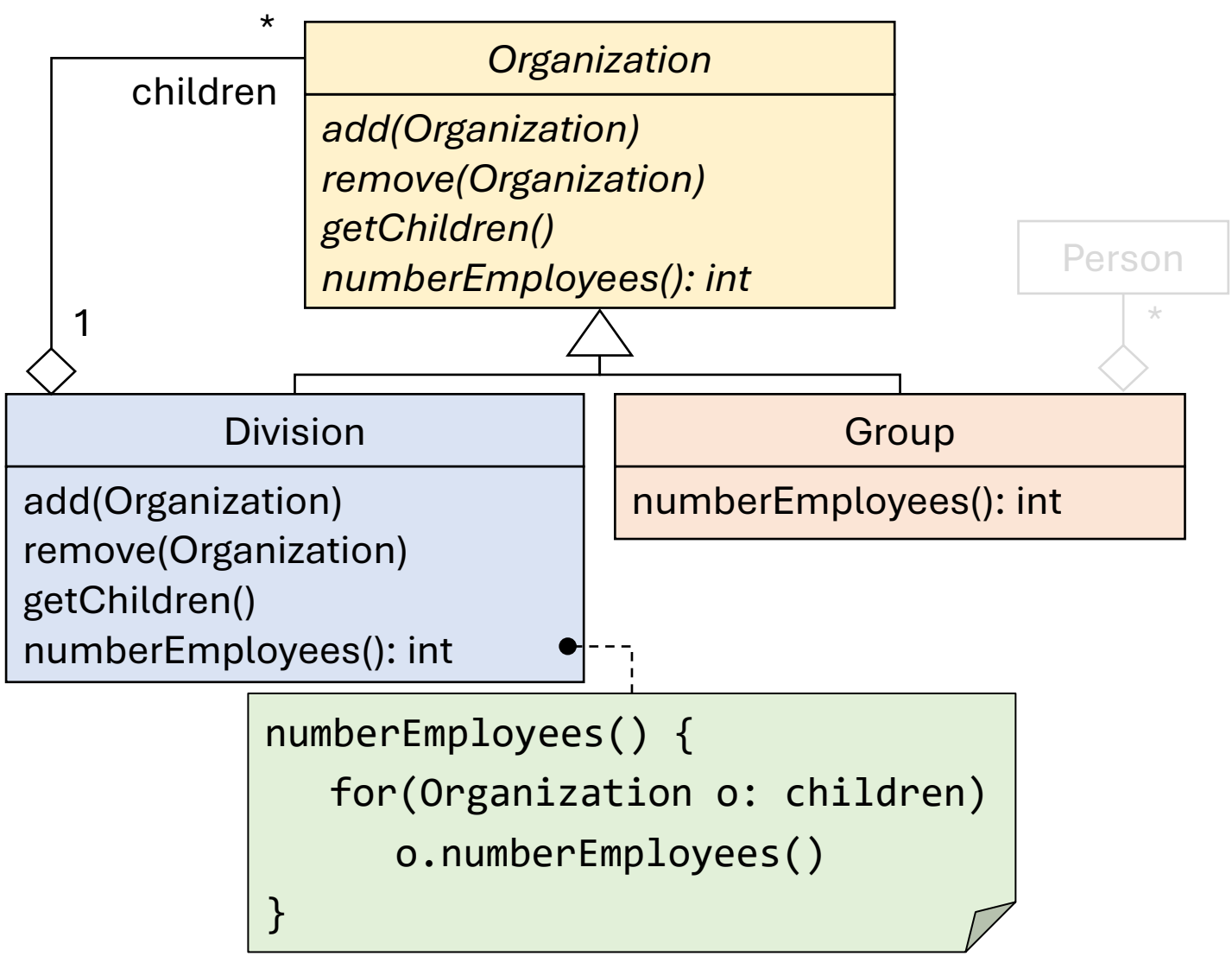
Composite

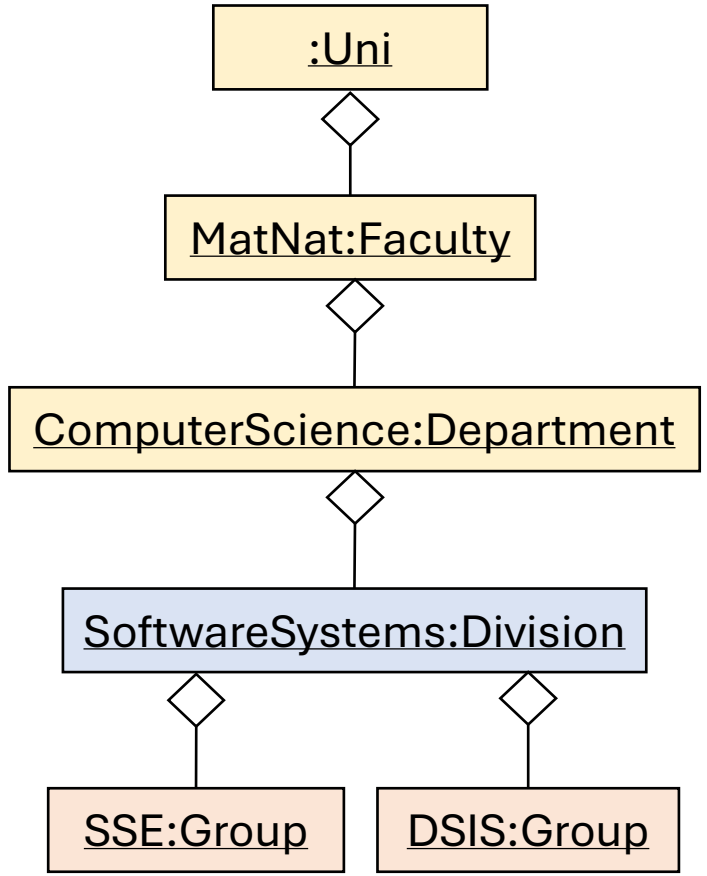
The composite pattern lets us realize a recursive aggregation structure. The composition acts as one whole conceptual object, i.e., operations on the root are propagated to the composites.

Example: Structure of the university.

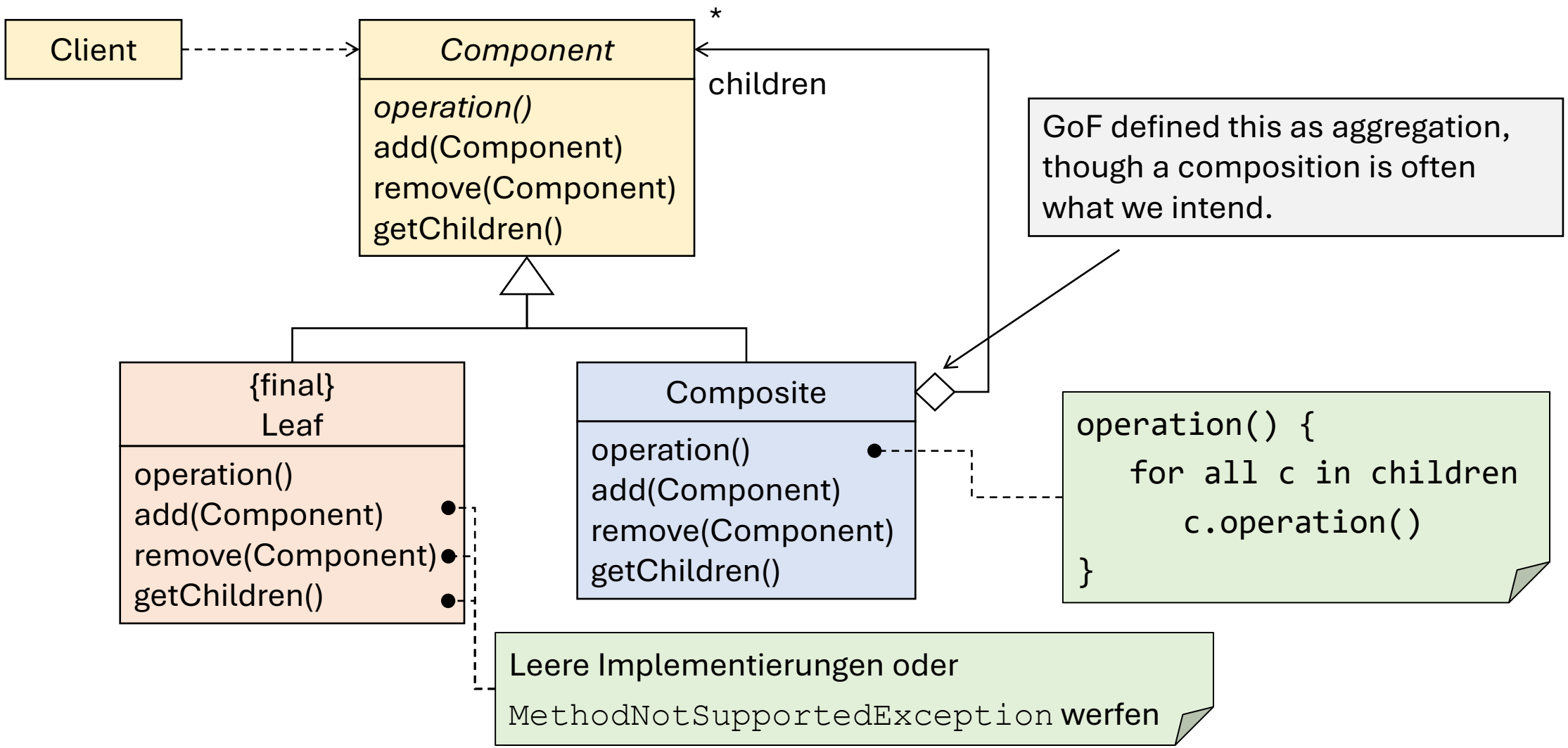


Example


*
children
Organization
add(Organization)
remove(Organization)
getChildren()
numberEmployees(): int
Person
*
1
Division
add(Organization)
remove(Organization)
getChildren()
numberEmployees(): int
Group
numberEmployees(): int
numberEmployees() {
for(Organization o: children)
o.numberEmployees()
}


:Uni
MatNat:Faculty
ComputerScience:Department
SoftwareSystems:Division
SSE:Group
DSIS:Group

Schema

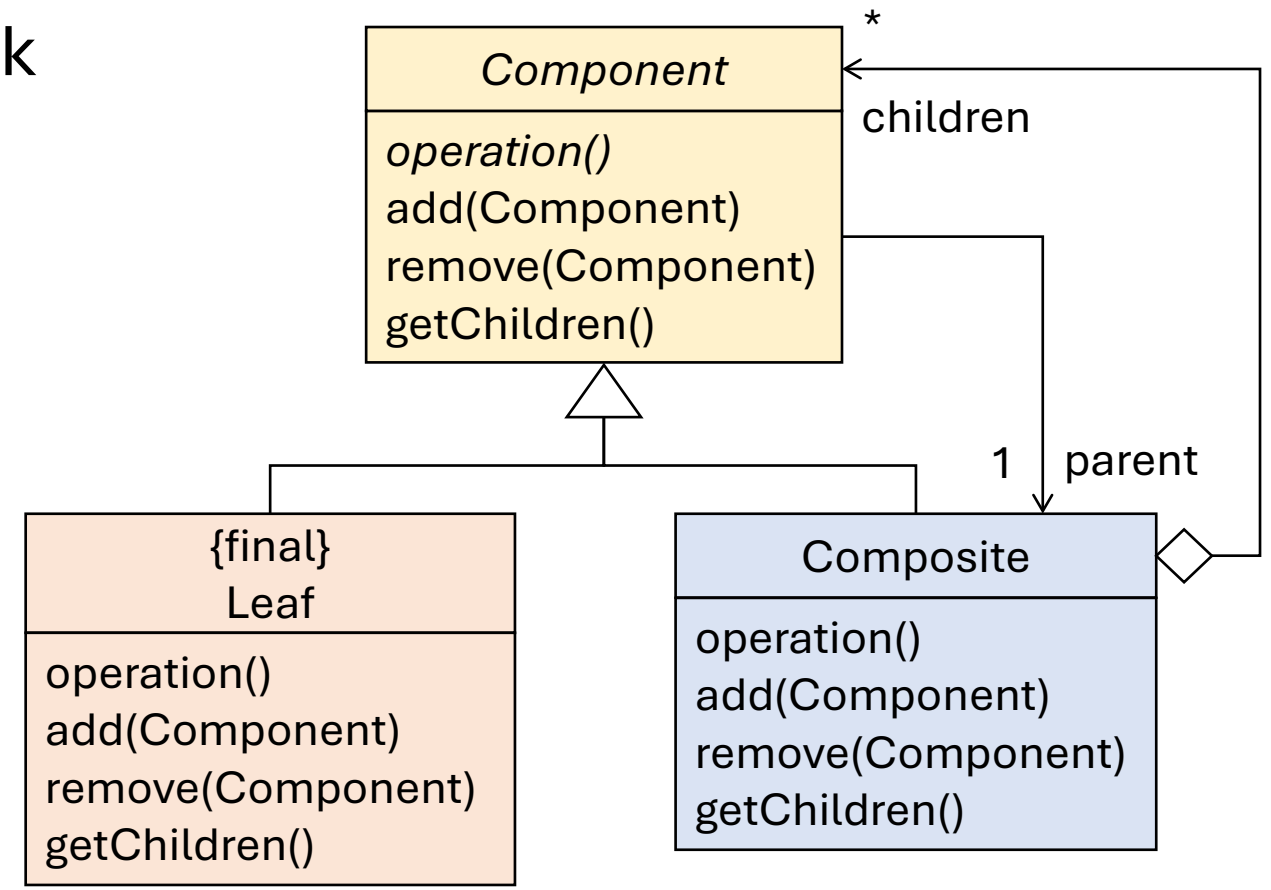

Client
Component
operation()
add(Component)
remove(Component)
getChildren()
*
children
GoF defined this as aggregation, though a composition is often what we intend.
{final}
Leaf
operation()
add(Component)
remove(Component)
getChildren()
Composite
operation()
add(Component)
remove(Component)
getChildren()
operation() {
for all c in children
c.operation()
}
Leere Implementierungen oder
MethodNotSupportedException werfen

Responsibilities

- Component
  - Common interface of all parts.
  - Default implementations.
  - Provides interface for access to parts.
  - May provide interface to access parent object.
- Leaf
  - Often the "real" data.
  - Override the access-to-parts interface with empty methods or exceptions.
- Composite: Forward operations to parts.

Implementation

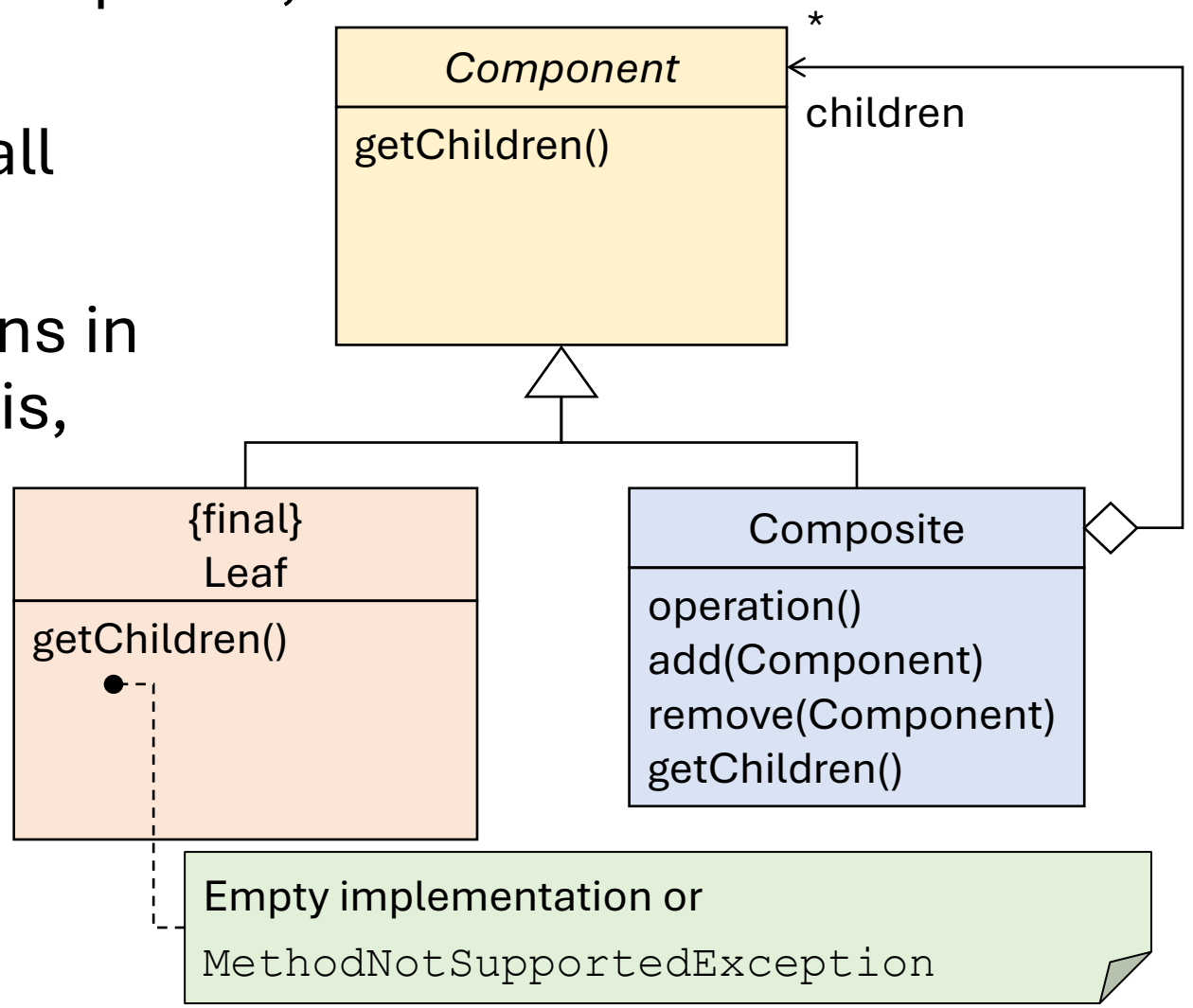
- If components should know to what composites they belong: back reference.
- Components that are part of multiple composites.



Implementation

Management operations into composite, as they are irrelevant for leafs.

- In contrast to idea to handle all components equally.
- Empty management operations in component. Leafs leave it as is, composites override it.



Consequences, Advantages, Disadvantages

+ Open/Close principle: You can add new object types to the tree structure without affecting client code or other object types.
+ Working on an object structure is more convenient due to method propagation.

- Overkill if size of object structure is small.
- Clients might need to perform runtime type checks (instanceof) to differentiate between composites and leafs.

Relation to other Patterns

- Composite and decorator have a similar structure since both rely on recursive compositions.

- You can use visitors to execute an operation on a composite object structure.

Questions?

Summary

In today's lecture:

- Observer
- Singleton
- Facade
- Prototype
- Factory Method
- Composite