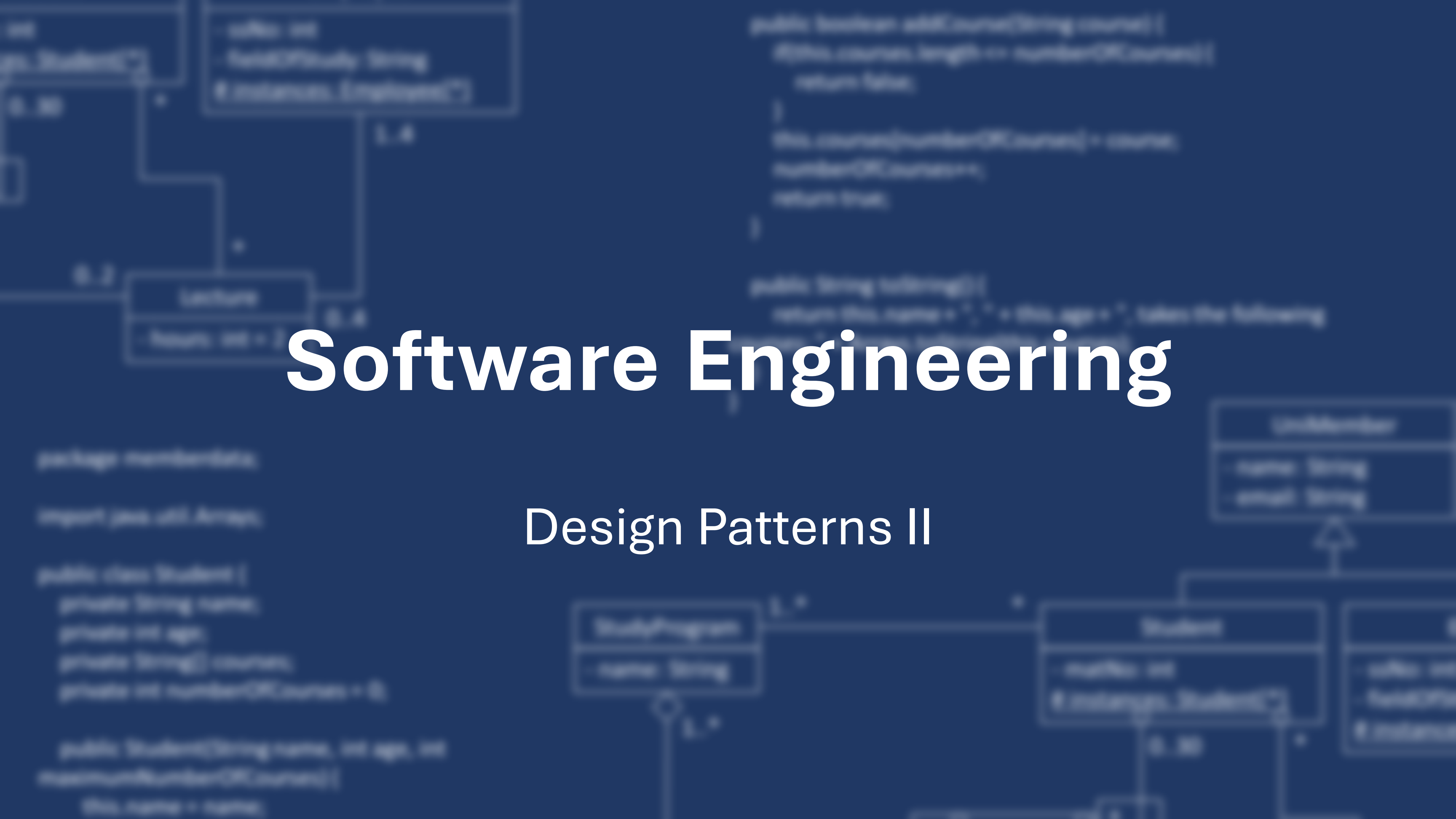


Software Engineering

Design Patterns II

Structure of the OOSE Lectures

Revisit and deepen basics of programming.

Revisit and deepen basics of object-oriented programming.

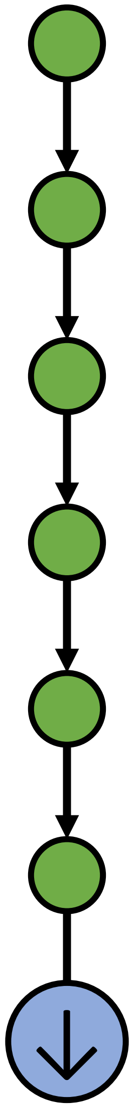
Cover advanced object-oriented principles.

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

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

Aims of this Lecture

In today's lecture:

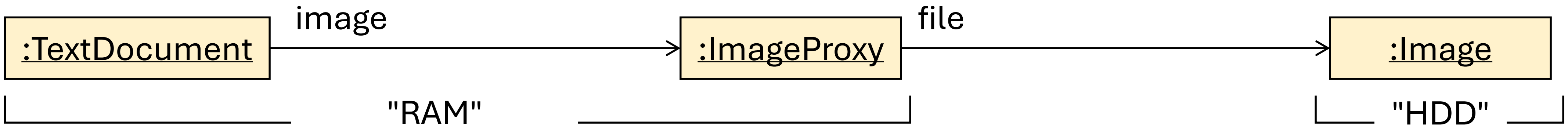
- Proxy
- Adapter
- Template Method
- State
- Strategy
- Decorator
- Visitor
- Twins

Proxy

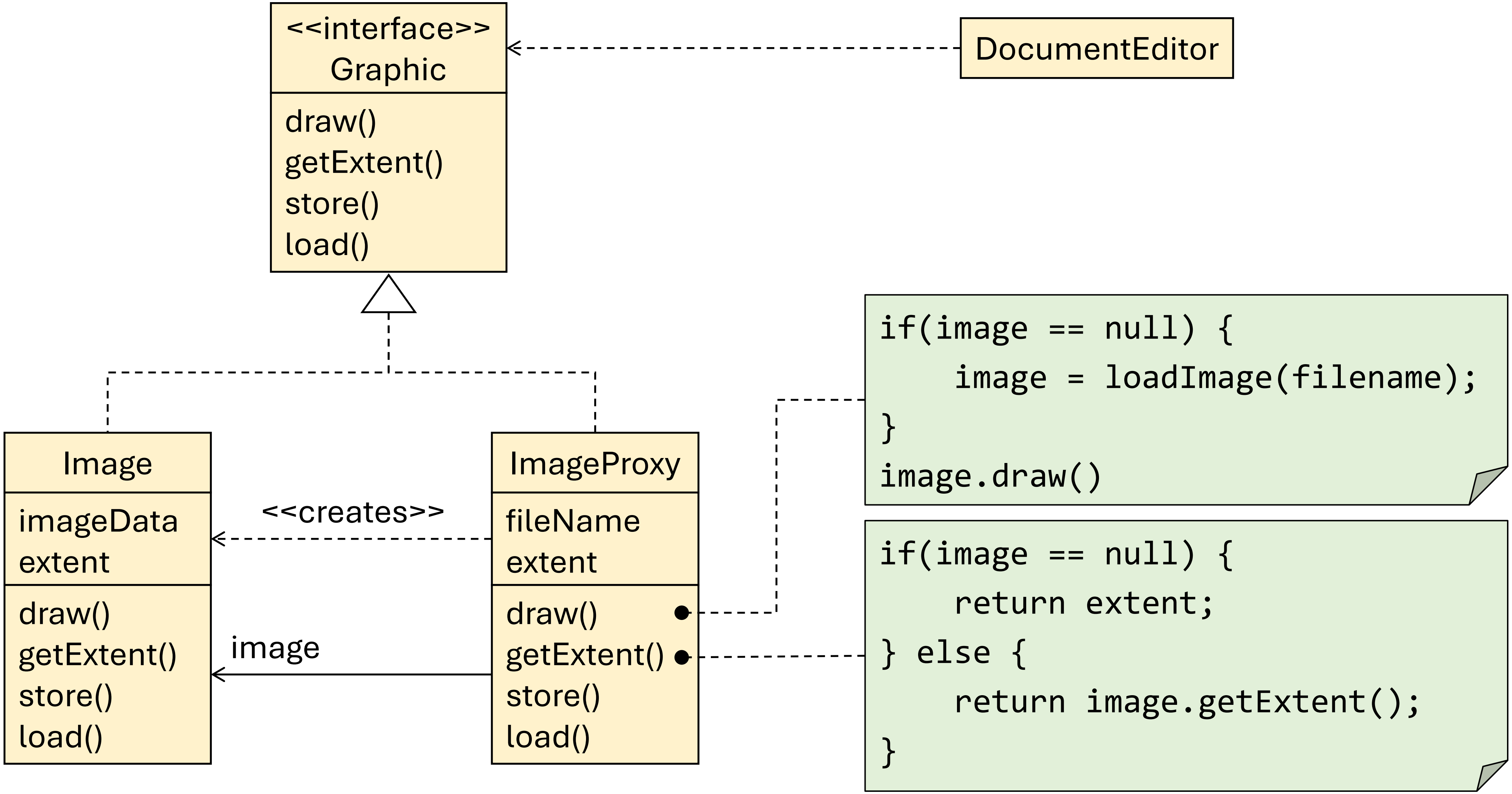
Proxy Pattern (also: Surrogate, Smart Reference)

The proxy pattern acts as a placeholder for a real subject and defers or blocks operations on the real subject, e.g., object creation or modification.

The proxy provides the same interface as the real subject and controls all messages send to it.



Example



<<interface>>
Graphic

draw()
getExtent()
store()
load()

DocumentEditor

Image

imageData
extent

draw()
getExtent()
store()
load()

<<creates>>

image

ImageProxy

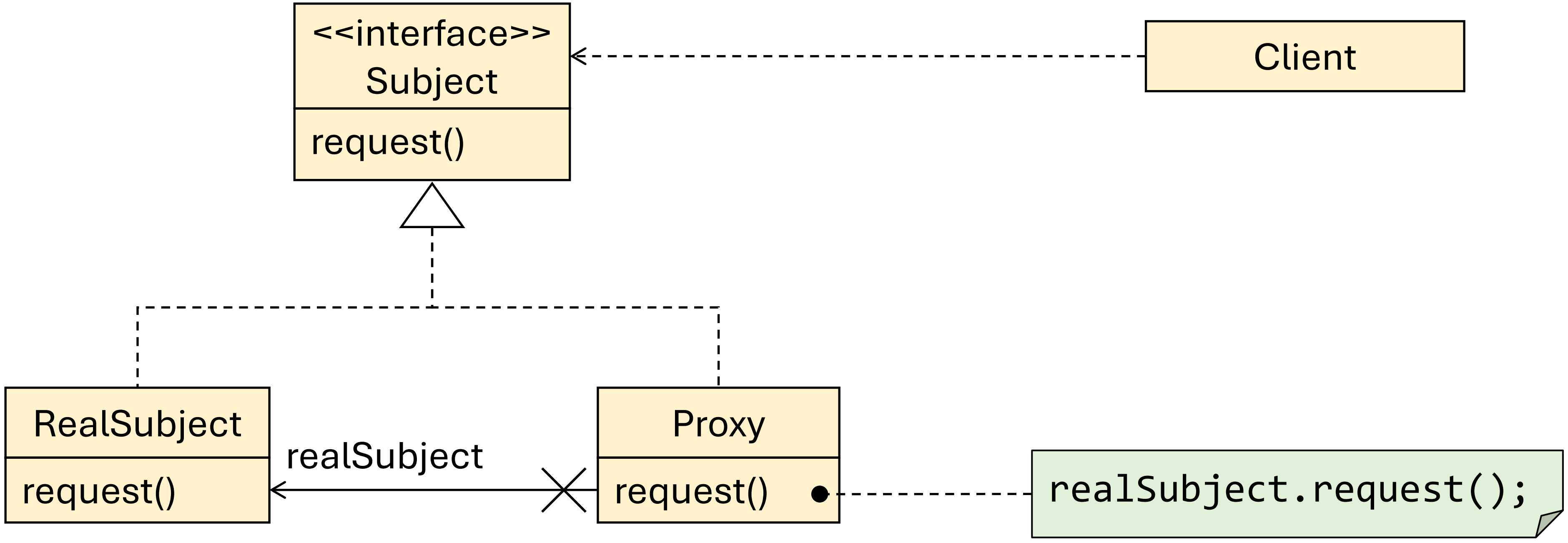
fileName
extent

draw()
getExtent()
store()
load()

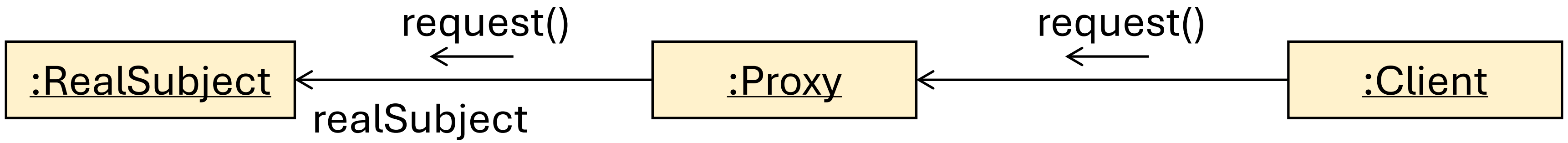
```
if(image == null) {
    image = loadImage(filename);
}
image.draw()
```

```
if(image == null) {
    return extent;
} else {
    return image.getExtent();
}
```

Schema



<<interface>>
Subject
request()
Client
RealSubject
request()
realSubject
Proxy
request()
realSubject.request();



request()
request()
:RealSubject
:Proxy
:Client
realSubject

Application

- Virtual Proxy: only create computational or memory intensive objects if really necessary.
- Remote Proxy: Replicate a remote object to reduce unnecessary network traffic or replace missing object.
    - See Remote Message Invocation.
- Concurrency Proxy: Prevent aliases on real subject from different threads by only having one reference on it from the proxy, which in turn is used by the threads. The proxy manages the locking.
- Copy-on-Write: Copying only increases a counter. The real copy is created when it is to be modified. → Prevent costly copying (e.g., deep clone).
- Protection Proxy: Manage access control to real subject.

Implementation of Consultation/Forwarding

- C++: Operator-Overloading
  - Proxy redefines dereferencing operator: `*anImage`
  - Proxy redefines member access operator:
    `anImage->extent()`
- Smalltalk: Reflection. Proxy redefines method "`doesNotUnderstand: aMessage`"
- Java: Reflection. Dynamic Proxy implements "InvocationHandler" Interface.
- JavaScript: Delegation. Proxy's `__proto__` points to real subject.
- Kotlin: keyword.

```
    class Proxy(ri: RealImage):
        RealImage by ri {... }
```

For all methods not defined in proxy, the compiler generates forwarding methods to the real subject.

Consequences, Advantages, Disadvantages

+ Open/close principle: You can introduce new proxies without changing the client.
+ Dependency inversion principle: The client code operates with the object through the common `Subject` interface. The dynamic type can be an arbitrary implementing class, thus the proxy or the real subject.

- More Code, as for all design patterns.
- More indirection (slightly slower).

Relation to other Patterns

- With adapter, you access an existing object via a different interface.
- With decorator, you access an existing object via an enhanced interface.
- With Proxy, the interface stays exactly the same.

- Facade is a proxy to an object structure but is does not necessarily have the same interfaces as the objects it substitutes.

- Decorator extends an existing object.

Questions?

https://zenwire.de/de_DE/p/Adapter-Kartenleser-Lightning-Micro-SD-USB-iPhone/209

Adapter

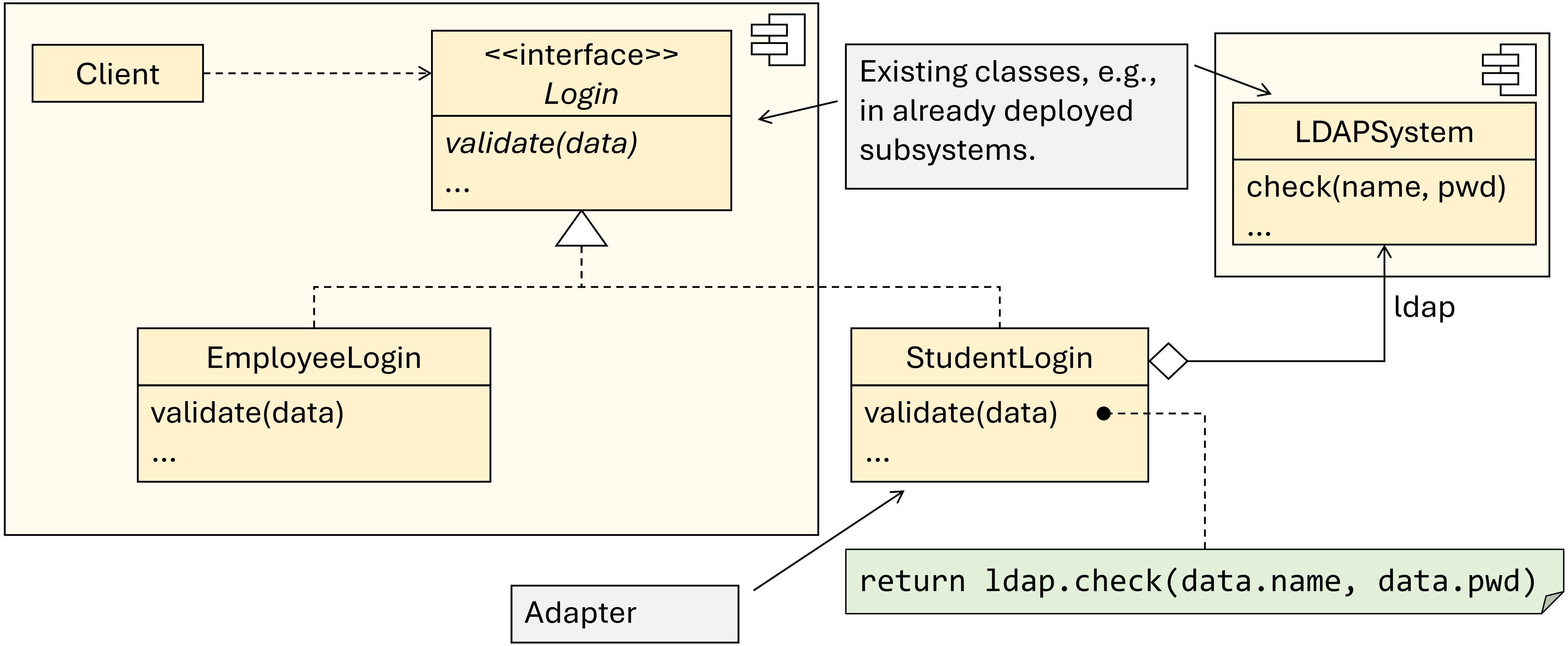
Adapter

The adapter pattern makes two different existing interfaces compatible to one another.

"Making compatible" can range from small modifications like changing a name to huge workarounds for completely different interfaces.

The object adapter adapts whole class hierarchies. Adapter subtypes and adaptee subtypes can be combined arbitrarily at runtime.

Example Configuration

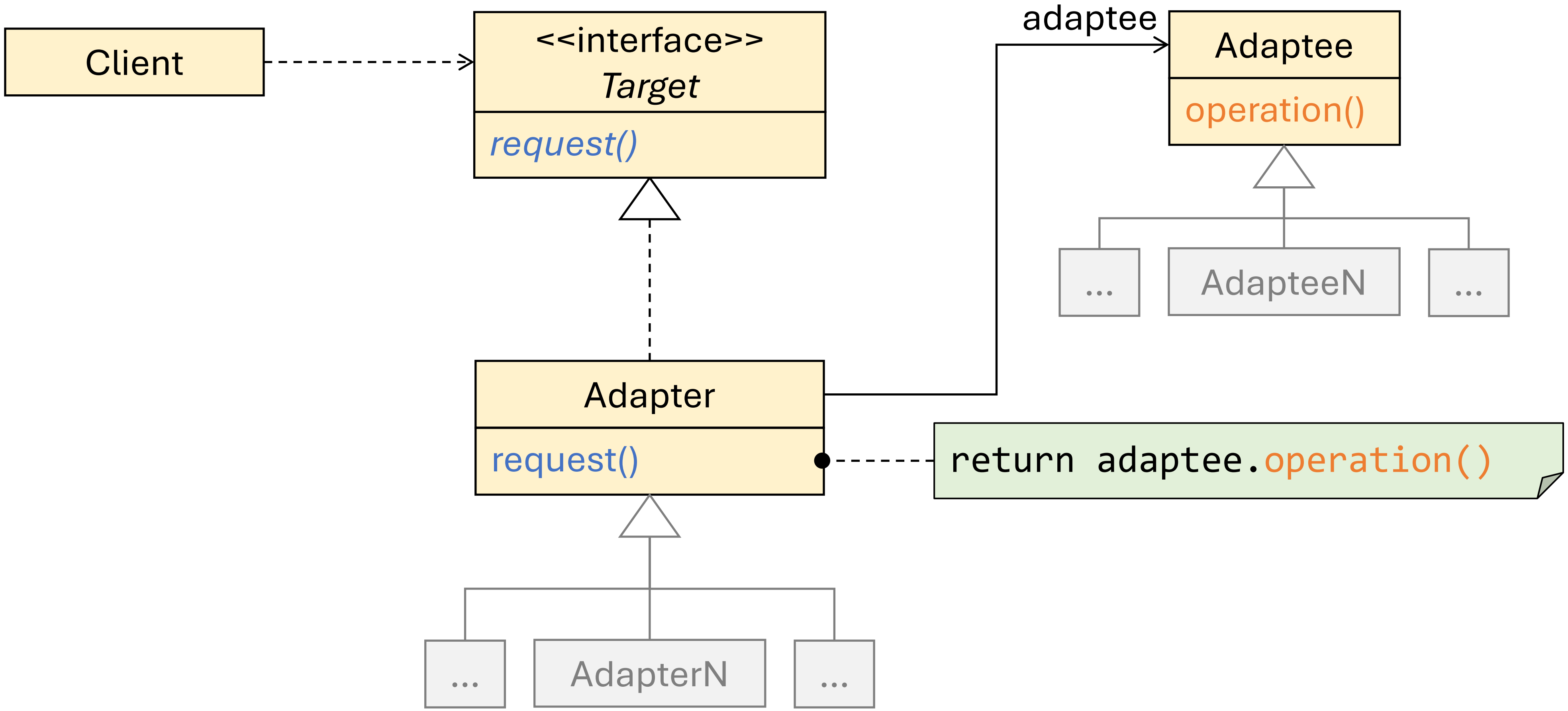


Existing classes, e.g., in already deployed subsystems.

```
return ldap.check(data.name, data.pwd)
```

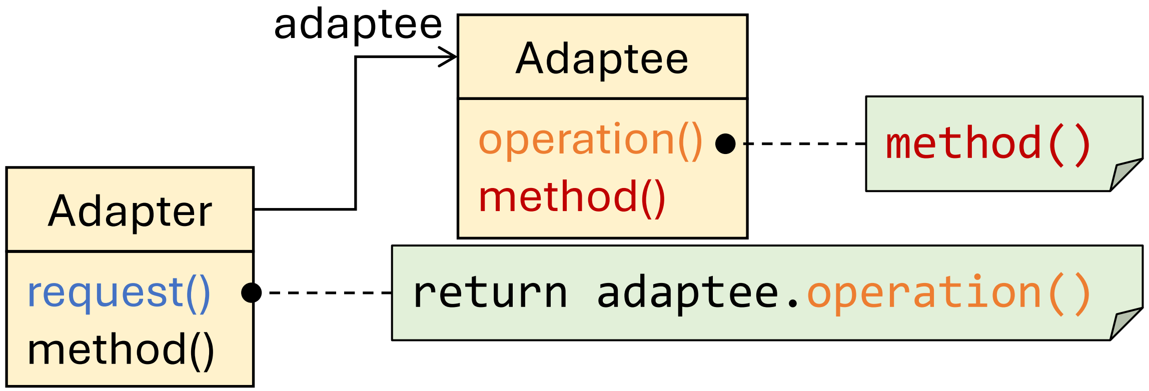
Adapter

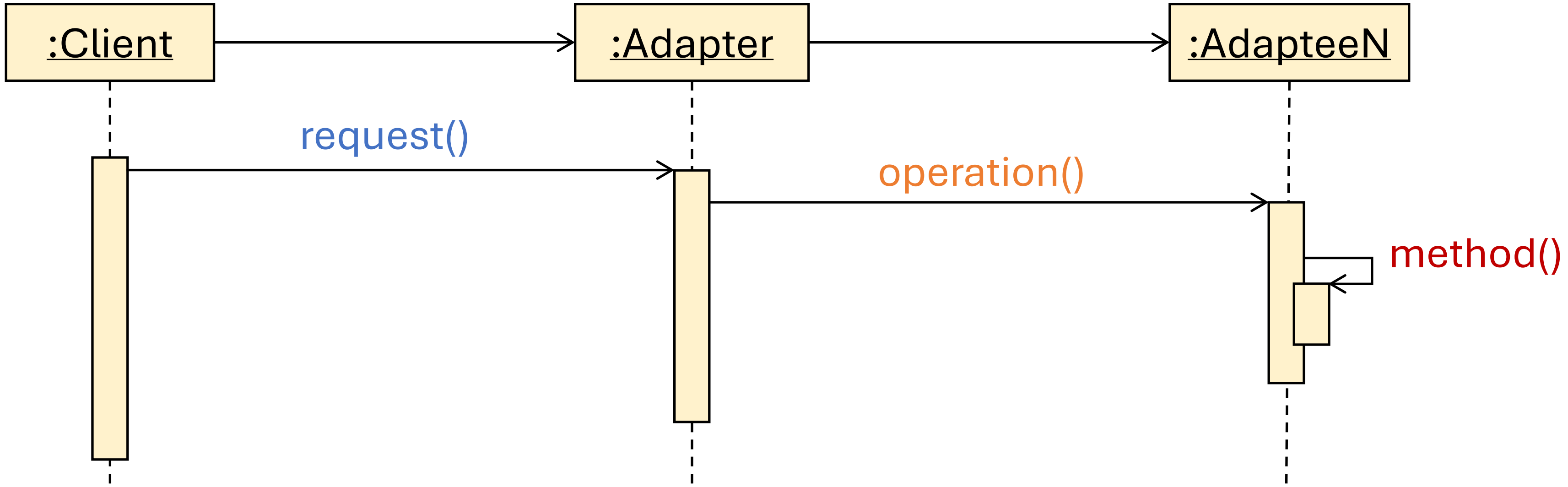
Object Adapter: Schema



Consequences

No overriding. In `Adaptee::operation`, we cannot call `Adapter::method` instead of `Adaptee::method`.





Consequences, Advantages, Disadvantages

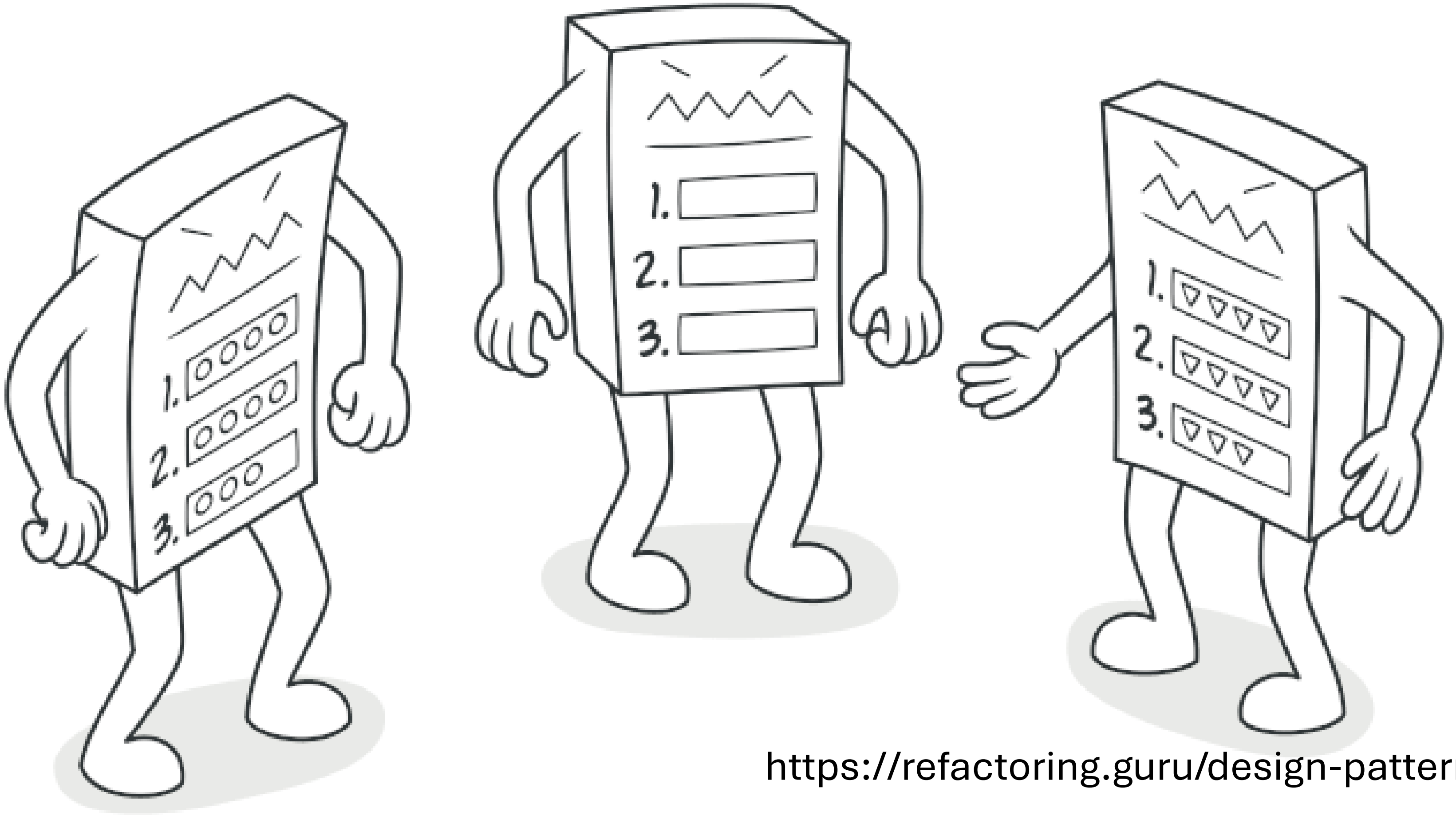
+ Open/close principle: You can introduce new adapters without changing the client.
+ Reusability: You can exchange whole subsystems by adding a matching adapter.
+ Single Responsibility Principle: Adapting code, like data conversion, is separated from the business logic.

- More Code, as for all design patterns.
- More indirection (slightly slower).

Relation to other Patterns

- With decorator, you access an existing object via an enhanced interface.
- With Proxy, the interface stays exactly the same.
- With adapter, you access an existing object via a different interface.

Questions?



https://refactoring.guru/design-patterns/template-method

Template Method

Template Method Pattern

With the template method pattern, we want to enforce a certain order of instruction while providing the possibility to redefine parts of it in specializations.

Idea: Extract redefinable parts as methods that are overridden in subclasses.

Implementation

Superclass

Template Method

```
public class Superclass {
   ...
   final public void operation(State newState) {
      this.doSomething(newState);
      this.notify();  //always notify at the end
   }
   protected void doSomething(State newState) {
   }
}
```

Subclass

Hook Method

```
public class Subclass extends Superclass {
   @Override
   protected void doSomething(State newState) {
      this.modifyMyState(newState);
   }
}
```

Implementation

General approach that follows the pattern by GoF:

- Hide hook methods (protected in Java and C++)
- Force overriding of methods (C++: pure virtual, Java: per default)
  - Keep abstract hooks minimal to allow easier extensibility.
- Forbid overriding of template method (Java: final, C++: do not declare as virtual)

Modern languages provide closures (in Java: lambda expressions) and "first-class operations" so that we can use them instead of the template method pattern.

Customizable Procedures with Lambdas

```
public class Superclass {
   ...
   final public void operation(State newState, Consumer<State> customOperation) {
      customOperation(newState);
      this.notify();  //always notify at the end
   }
}
```

```
Superclass sc = new Superclass();
sc.operation(Superclass::modifyMyState);
```

Consequences, Advantages, Disadvantages

+ Open/close principle: You can extend existing methods by changing parts of it.

+ Generalization/DRY principle: The common core of different methods is extracted in one central place.

○ Can be used as behavior protocol, but becomes increasingly complicated, the more complex method interplay is.

- More Code, as for all design patterns.

Relation to other Patterns

- Factory Method is a specialized template method.

- Strategy let's you change a method implementation at runtime via composition.

Questions?

Montana
North Dakota
Massachusetts
Maine
Oregon
Minnesota
Wisconsin
Idaho
South Dakota
Michigan
New York
Wyoming
Rhode Island
Pennsylvania
Connecticut
Nebraska
Iowa
Nevada
Ohio
New Jersey
Utah
Illinois
Indiana
Delaware
Colorado
Virginia
Maryland
Kansas
Missouri
California
Kentucky
Washington, D.C.
North Carolina
West Virginia
Tennessee
Oklahoma
Arizona
Arkansas
New Mexico
South Carolina
Alabama
Georgia
Mississippi
Texas

State

State

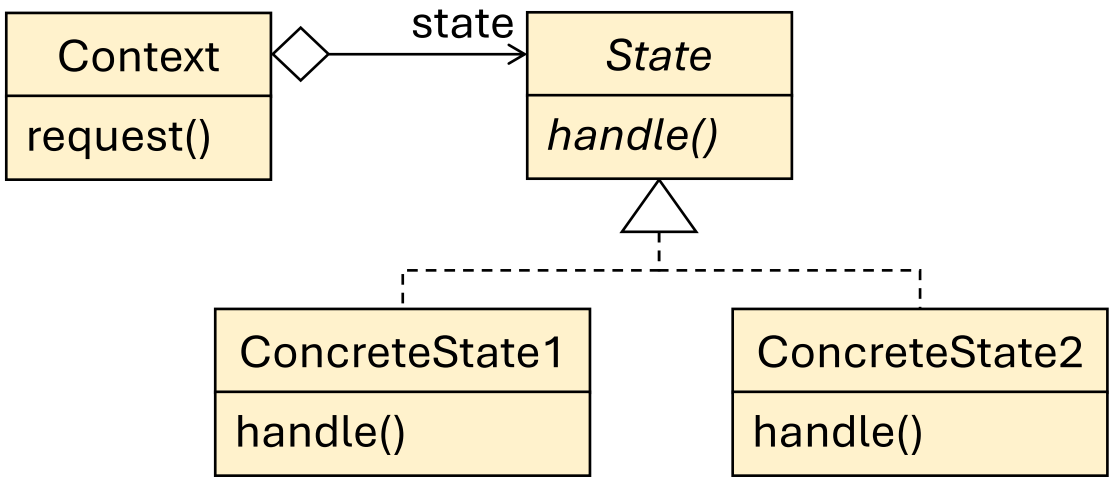
The state pattern is a way to model object-specific, state-specific, dynamically reconfigurable behavior.

Applicability: The behavior of an object depends on its current (conceptual) state.

The state pattern applies best when state transitions are internally triggered, not by an outside user.

The naive implementation would cause many conditionals that select the behavior.

Schema and Implementation



- The definition of when transitions happen may be located centrally in the context or distributed in the state classes.
- State objects are either created on demand or up-front.
- Either use delegation or dynamic class modification.

Consequences, Advantages, Disadvantages

+ Open/Close principle: New states do not require any / much modification of the context.
+ Thread-Safety: State changes are atomic (one assignment)
+ State changes are made explicit.
+ Single responsibility principle: State-dependent behavior lies in its own class hierarchy. Related methods are encapsulated per state.

- Overkill for simple state-machines with few states that rarely change.

https://www.amazon.de/Hasbro-Risiko-Strategiespiel-deutsche-Sprachvariante/dp/B084MYXWPG

Strategy

Example

Matrix2D

```
Matrix2D matmatmul(Matrix2D a, Matrix2D b) {
   return new Matrix2D(
      a.m11 * b.m11 + a.m12 * b.m21,
      a.m11 * b.m12 + a.m12 * b.m22,

      a.m21 * b.m11 + a.m22 * b.m21,
      a.m21 * b.m12 + a.m22 * b.m22
   );
}
```

Matrix3D

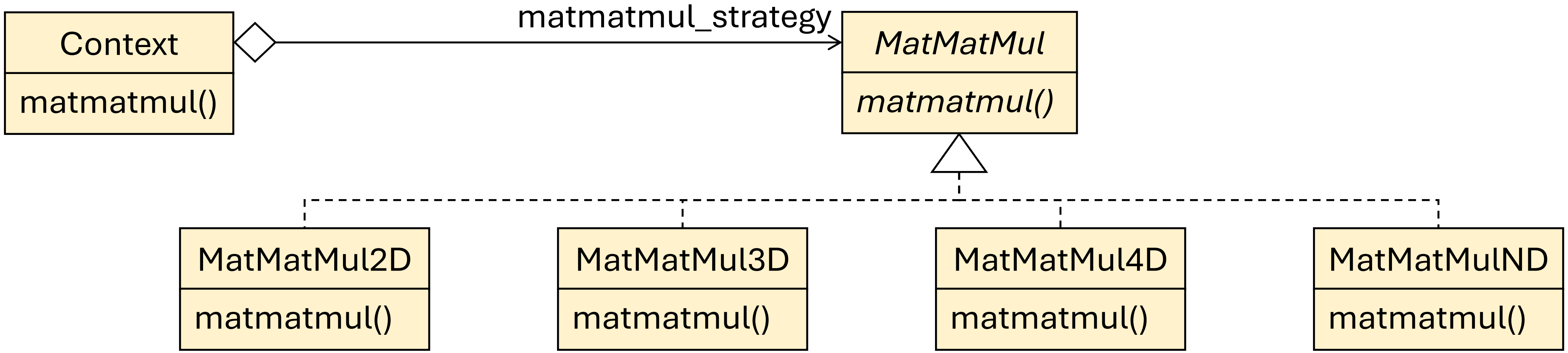
```
Matrix3D matmatmul(Matrix3D a, Matrix3D b) {
   return new Matrix3D(
      a.m11 * b.m11 + a.m12 * b.m21 + a.m13 * b.m31,
      a.m11 * b.m12 + a.m12 * b.m22 + a.m13 * b.m32,
      a.m11 * b.m13 + a.m12 * b.m23 + a.m13 * b.m33,

      a.m21 * b.m11 + a.m22 * b.m21 + a.m23 * b.m31,
      a.m21 * b.m12 + a.m22 * b.m22 + a.m23 * b.m32,
      a.m21 * b.m13 + a.m22 * b.m23 + a.m23 * b.m33,

      a.m31 * b.m11 + a.m32 * b.m21 + a.m33 * b.m31,
      a.m31 * b.m12 + a.m32 * b.m22 + a.m33 * b.m32,
      a.m31 * b.m13 + a.m32 * b.m23 + a.m33 * b.m33,
   );
}
```

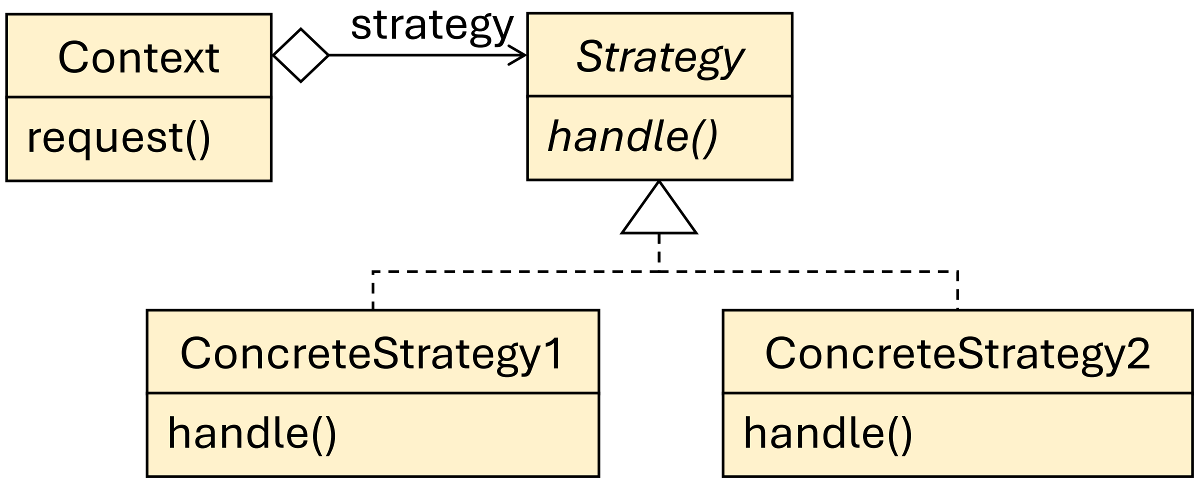
Example

Depending on the dimension of the matrices, we select an appropriate matrix multiplication algorithm.



Strategy and Schema

The strategy pattern encapsulates a family of behavior for dynamic exchangeability.



Applicability

The strategy is applicable everywhere where different variants of a behavior are required.

This way, we encapsulate data belonging to a behavior.

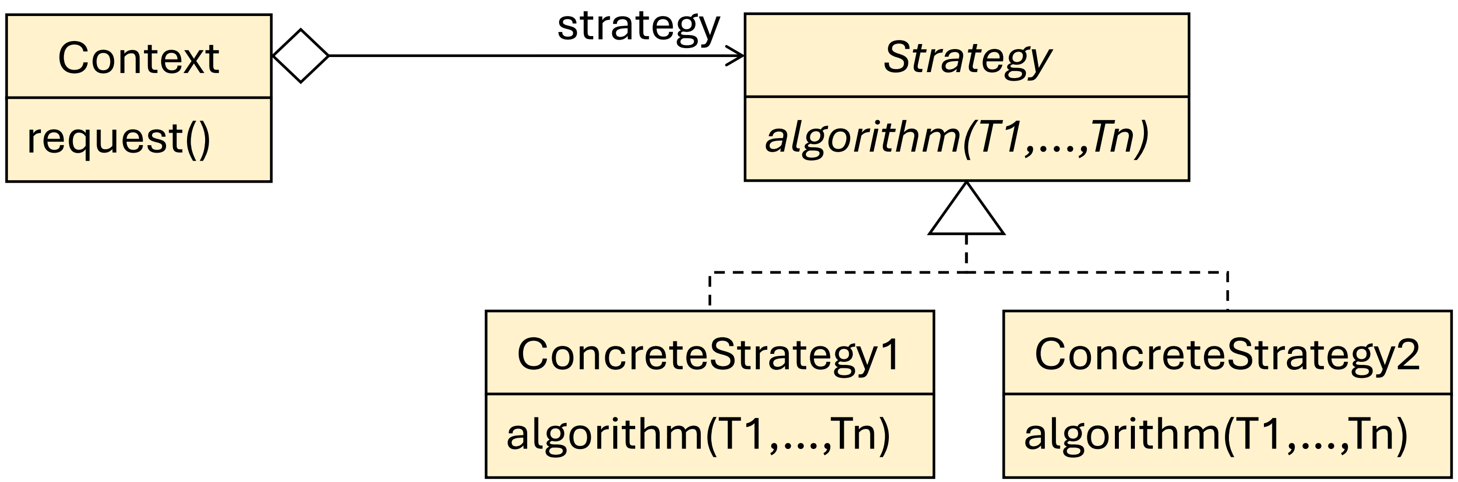
Behavior is dependent on outside constraints and thus changed by the outside (e.g., client calls `setStrategy`).

This differentiates the strategy from the state pattern: condition which triggers a change is neither sensible in context nor in any strategy class.

Implementation (for State and Strategy)

1. <u>Context supplies all data necessary to strategy</u>

- Easy to handle, but all parameters for all strategies need to be specified in the interface → bloated.
- Prone to change, as interface and all strategies have to be modified.



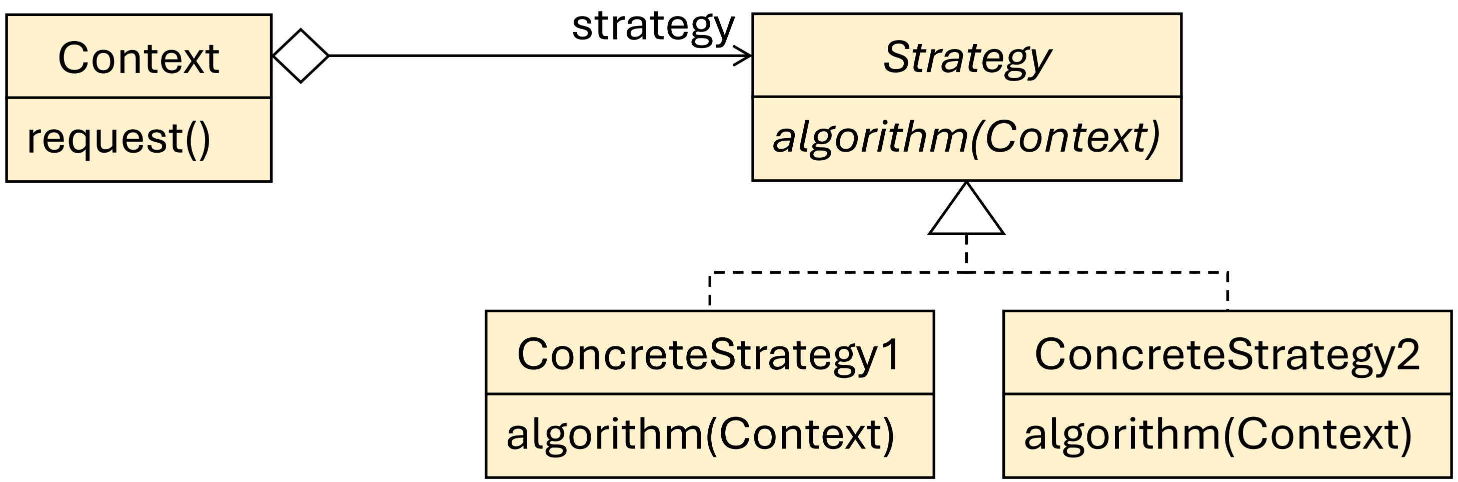
Implementation (for State and Strategy)

2. Context passes `this` to strategies

More flexible solution: Every strategy asks the context for what they need.

Multiple contexts can use the same strategy, if

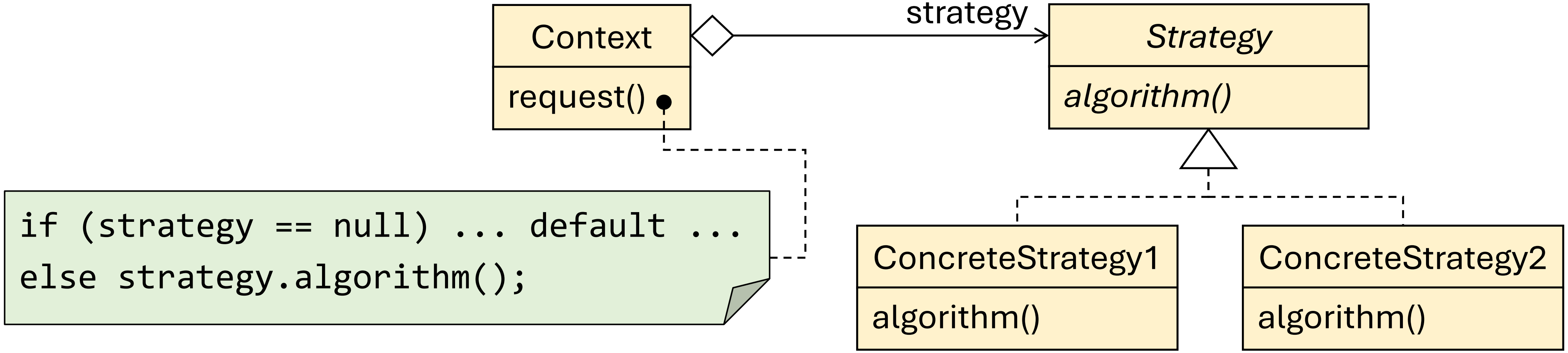
- strategies have no state,
- contexts all have the same interface.



Implementation: What if `null`?

Approach 1: Check for `null` in context.

This way, we save a subclass of Strategy. However, now the context needs to know what to do in this case, breaking the modularization effect.



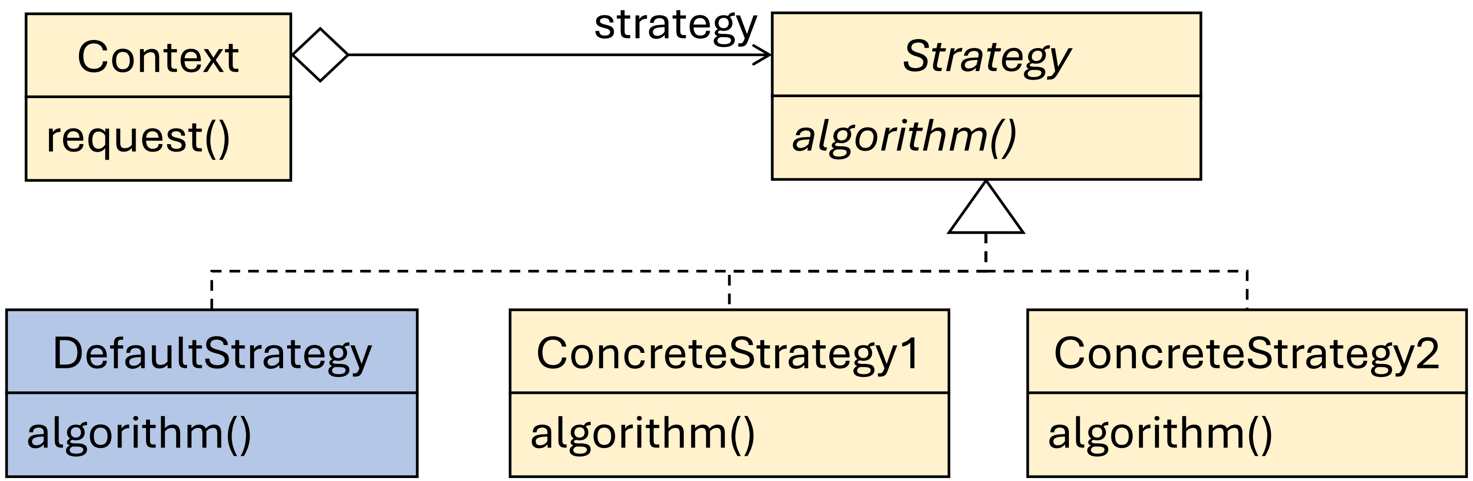
```
if (strategy == null) ... default ...
else strategy.algorithm();
```

Implementation: Null Object Pattern

Approach 2: Use the `null` object pattern.

Replace every `Strategy s = null;` with Strategy `s = new DefaultStrategy();`.

This way, we stick to encapsulating strategies as their own objects and have a more readable code.



Consequences

- More indirection, more messages between context and strategy.
- More objects. This can be circumvented to some extend by reusing strategy objects, e.g., with the flyweight pattern (not in this lecture).

Consequences, Advantages, Disadvantages

+ Open/Close principle: You can add new strategies without affecting others and without changing the context.

+ You can swap algorithms at runtime.

+ Single responsibility principle: Each behavior is encapsulated in its own class.

- Overkill if you only have few variants.
- Most modern languages have functional aspects that allow to pass a strategy as a closure (lambda expression).

Relation to other Patterns

- State is a special strategy.

- "Decorator lets you change the skin of an object, while strategy lets you change the guts". [refactoring.guru]

- Template method lets you change parts of a method at compile time (static because of inheritance).

Questions?

https://www.architecturaldigest.com/story/dopamine-decor-is-the-feel-good-interior-trend-we-need

Decorator

Decorator

The decorator pattern allows to add new or changed functionality to an existing <u>conceptual</u> object dynamically at runtime.

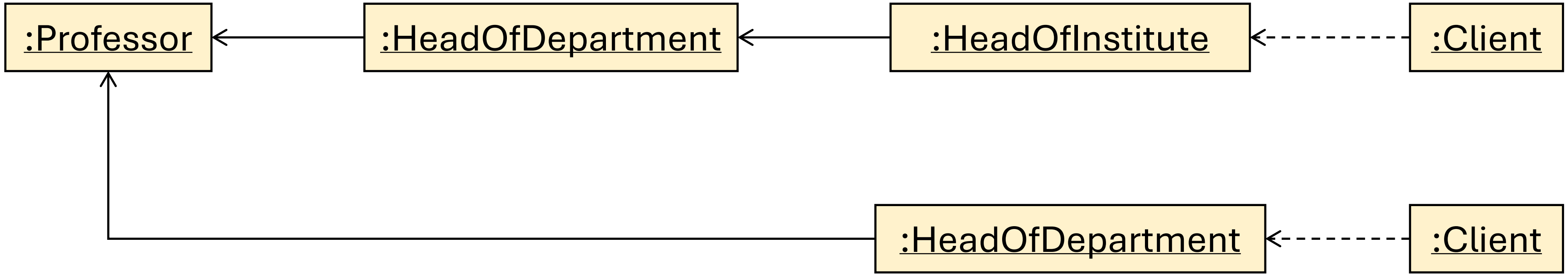
This is especially useful for object-specific, context-specific properties and for anticipation of change (open-closed principle).

Example: Assign roles to person objects, e.g., the professor who becomes head of the department… and later dean.

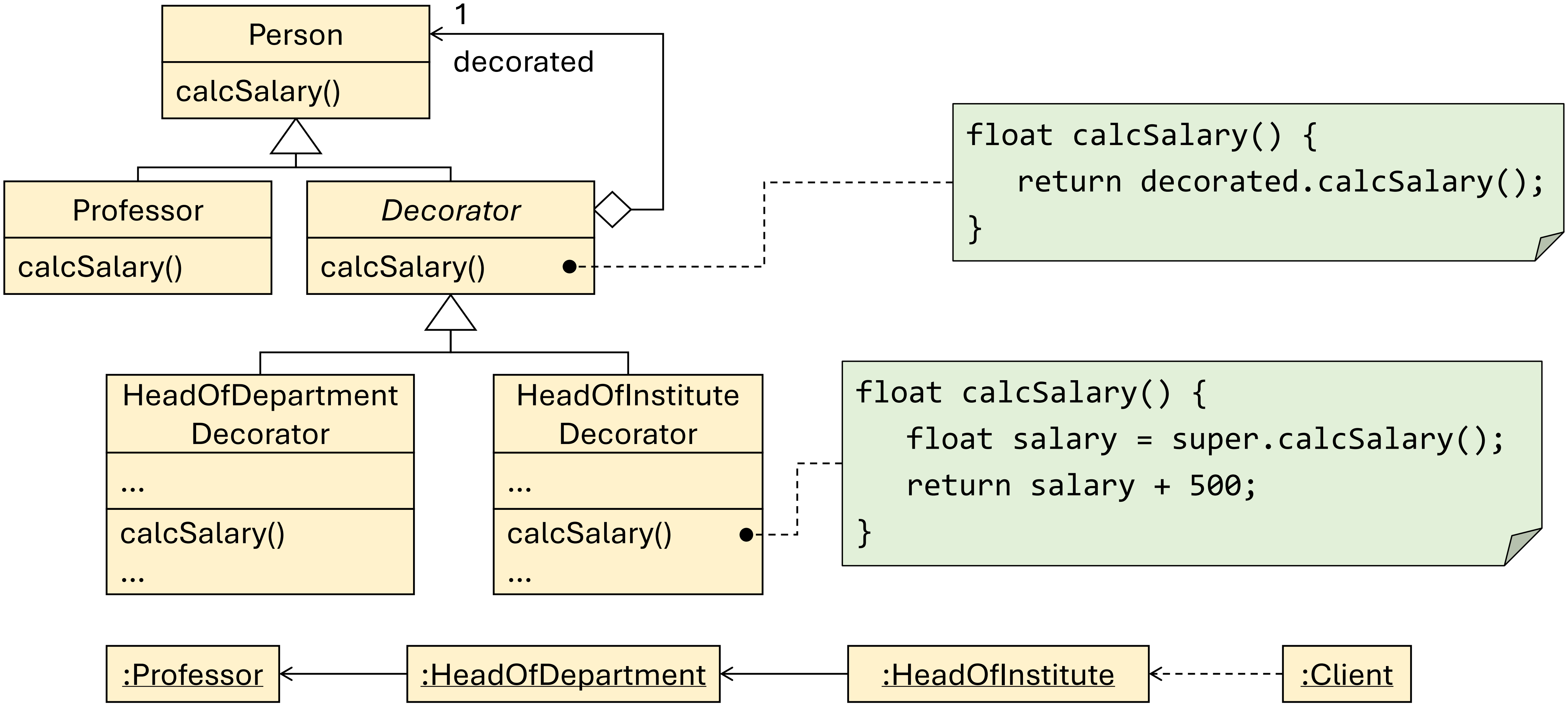
Approach

The new or changed functionality are distinct objects that lie between client and the real object.

Through accessing different decorators, clients see the same "core object" in different contexts.



Example



Person

calcSalary()

1

decorated

Professor

calcSalary()

Decorator

calcSalary()

```
float calcSalary() {
   return decorated.calcSalary();
}
```

HeadOfDepartment Decorator

…

calcSalary()

…

HeadOfInstitute Decorator

…

calcSalary()

…

```
float calcSalary() {
   float salary = super.calcSalary();
   return salary + 500;
}
```

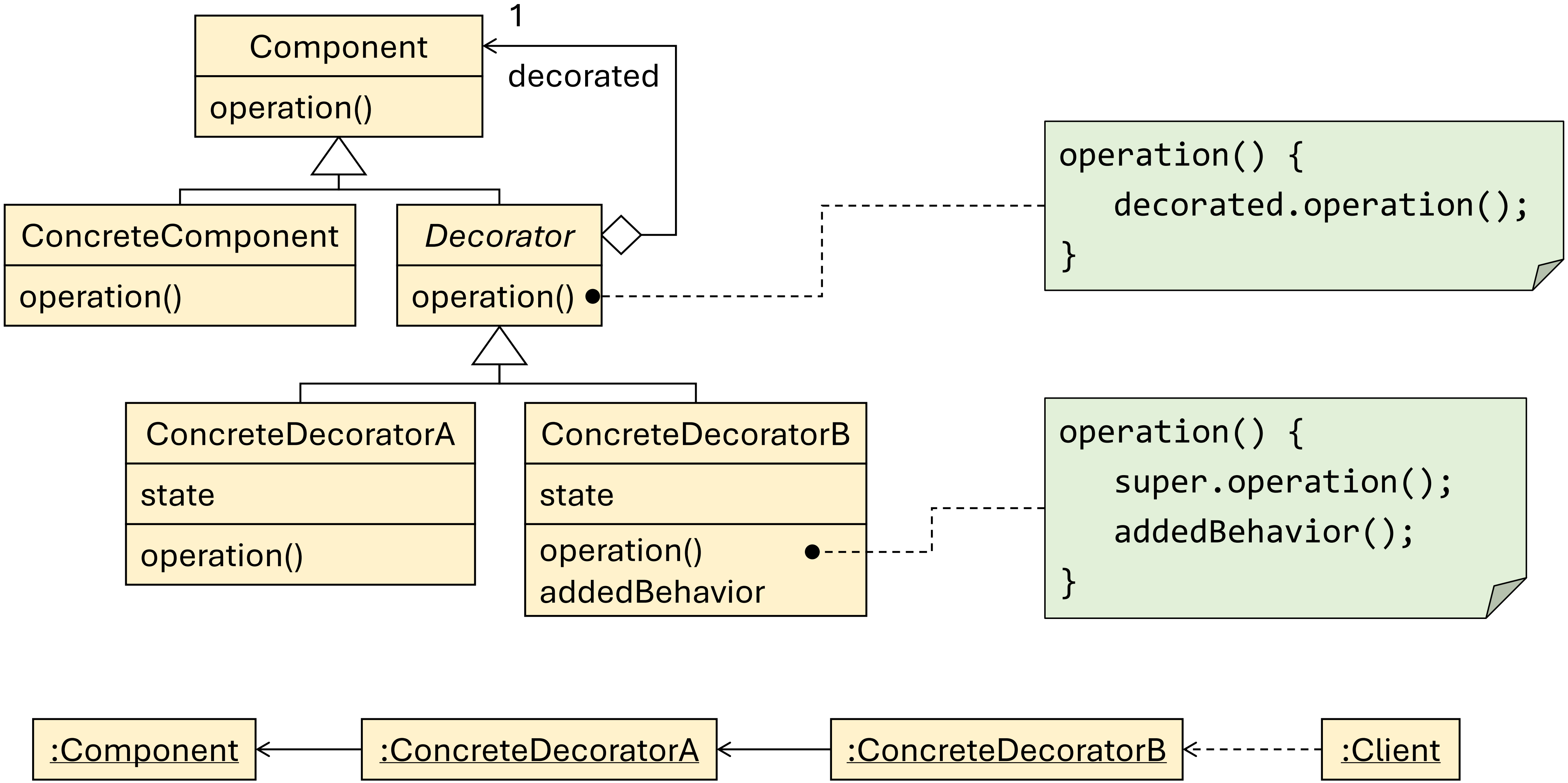
:Professor

:HeadOfDepartment

:HeadOfInstitute

:Client

Schema



```
operation() {
   decorated.operation();
}
```

```
operation() {
   super.operation();
   addedBehavior();
}
```

Applicability/Implementation

The decorator is applicable if the identity of an object is not important (see next slide for details).

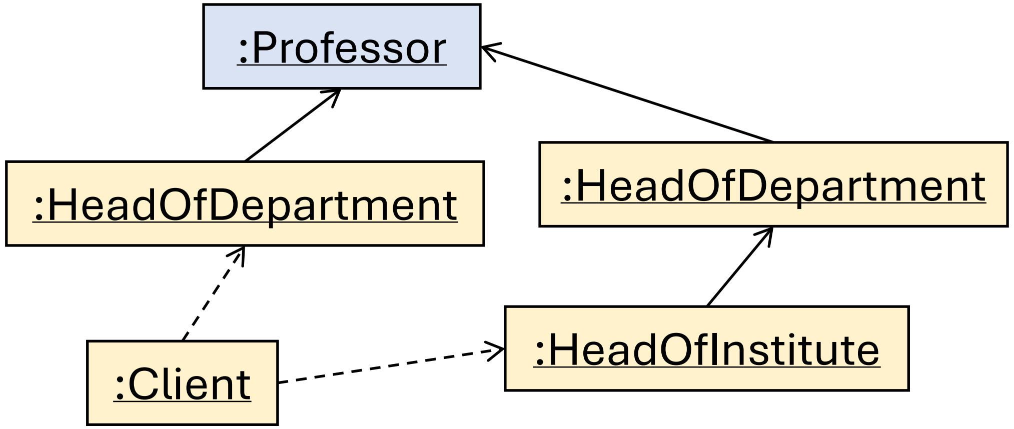
The decorator is not necessary or even bloat when the additional/changed functionality is static -> change the existing object directly.

Changing existing functionality if non-trivial, as we need to simulate overriding.

Implementation

Scenario: Different references to the same core object through different decorators.

```
//client code
Person p =  new Professor("Alice Bob");
Person hiHd = new HeadOfInsitute(
              new HeadOfDeparment(p));
Person hd = new HeadOfDeparment(p);
```



In this setting, clients testing for identity via == will not see that both paths point to the same core object, meaning that they refer the same conceptual object.

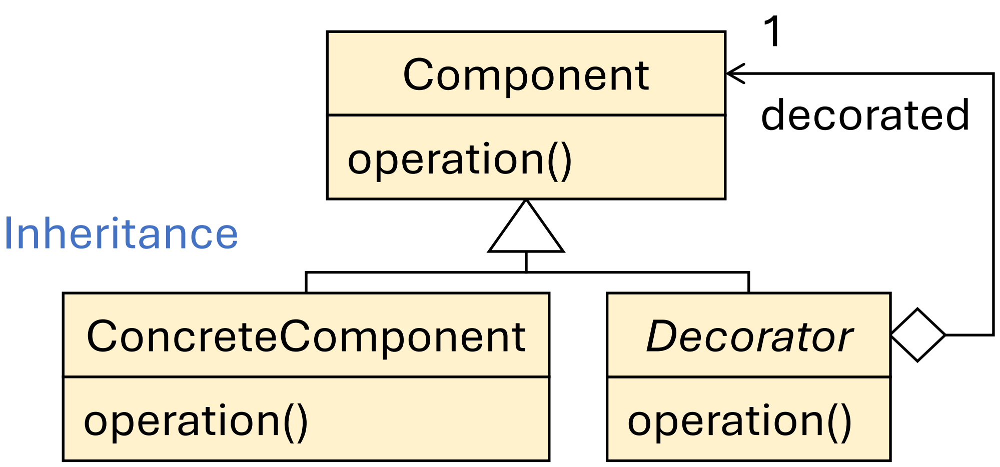
Solution: Use the `equals` method to implement this identity check.

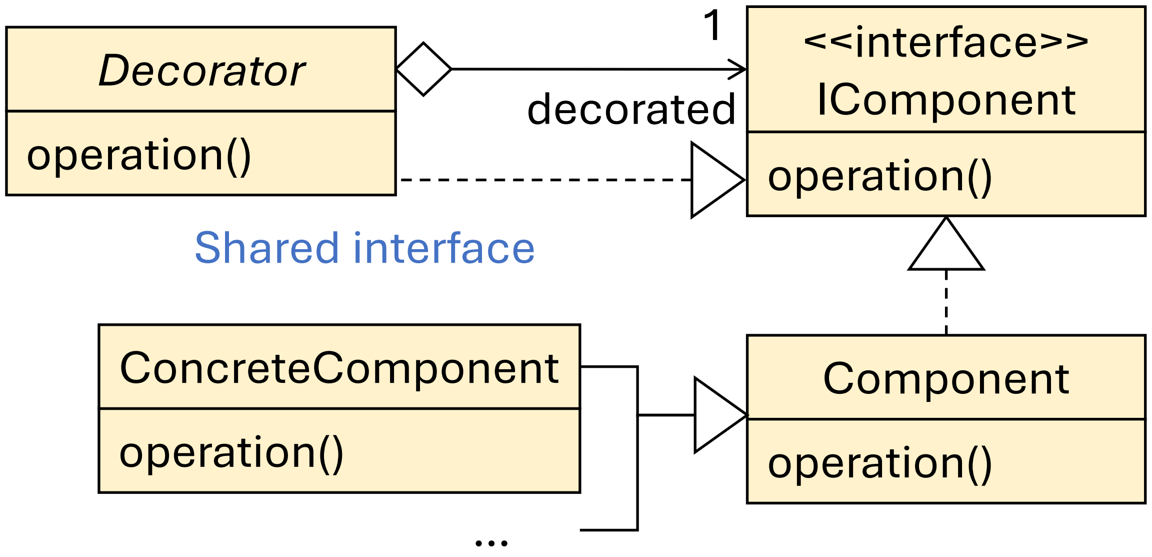
Consequences

The decorator allows dynamical, incremental addition of functionalities to an object as needed.

The functionalities are encapsulated in different objects. There may be many objects and thus the total set of functionalities can be hard to grasp.

Implementation





A more OO variant of the decorator is to let the component and the decorator implement a shared interface rather than declaring the decorator a subtype of the component.

- Reduced interfaces of the classes.
- Decorators are not flooded with state that is irrelevant to them.

Consequences, Advantages, Disadvantages

+ Open/Close principle: You can extend an object's behavior without adding a new subclass.

+ You can add different behavior dynamically to an object when necessary.

+ Single responsibility principle: Modified behavior is encapsulated in its own class.

- It is difficult to make behavior independent of the order of the decorators.
- Overkill if variable behavior is limited.

Relation to other Patterns

- With proxy, the interface stays exactly the same.
- With adapter, you access an existing object via a different interface.
- With decorator, you access an existing object via an enhanced interface.

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

- "Decorator lets you change the skin of an object, while strategy lets you change the guts". [refactoring.guru]

Questions?

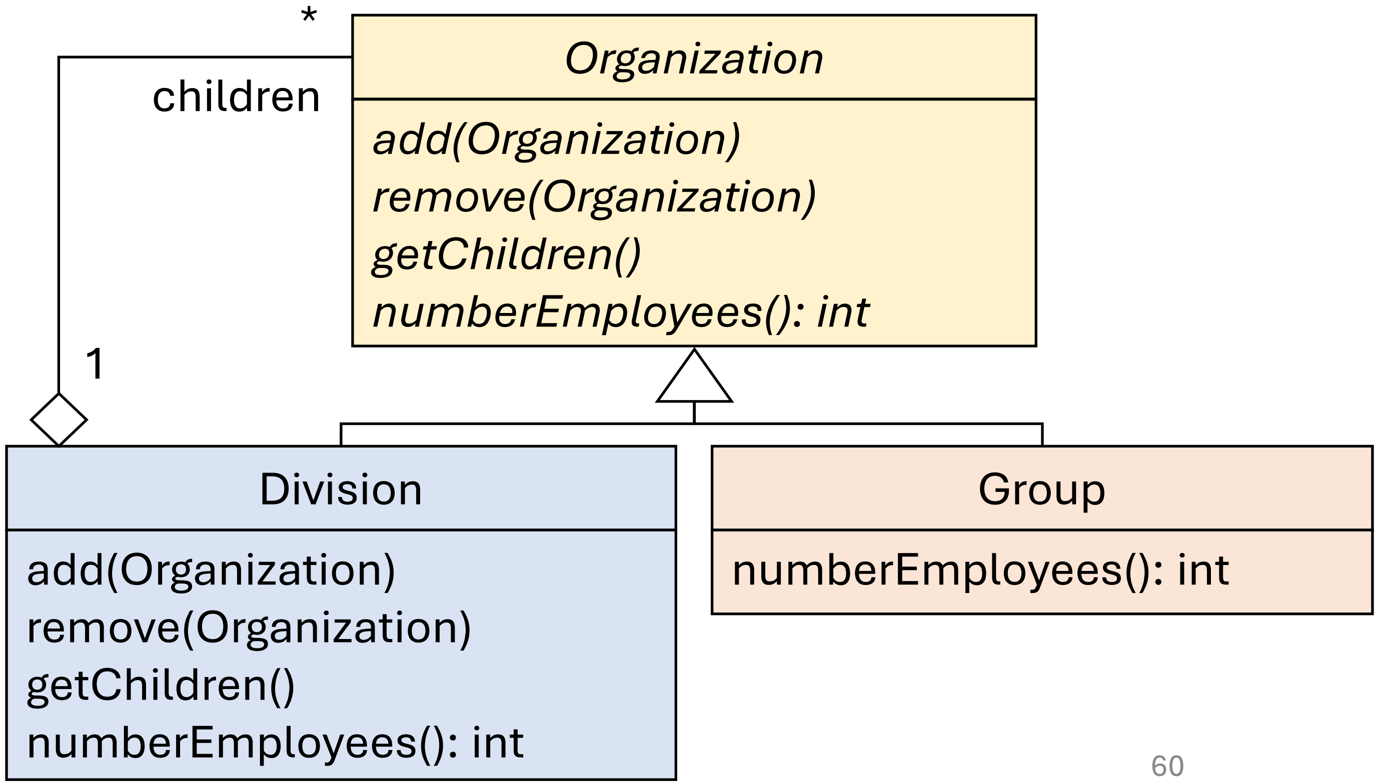
https://www.deutschland-monteurzimmer.de/ratgeber/besuch-in-monteurzimmer-ferienwohnung

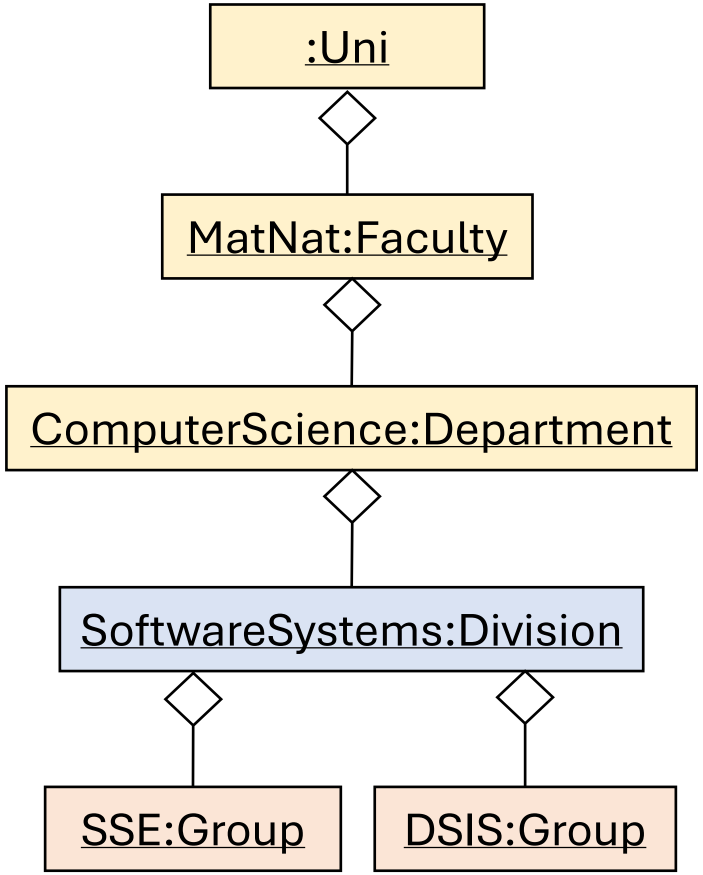
Visitor

Visitor

With the visitor pattern, we want to run operations on an arbitrary object structure without changing the classes of these objects.

Example: Structure of the university.





Visitor

Problem: certain operations are scattered through the classes of the object structure. Adding new functionality requires adding it to all these classes.

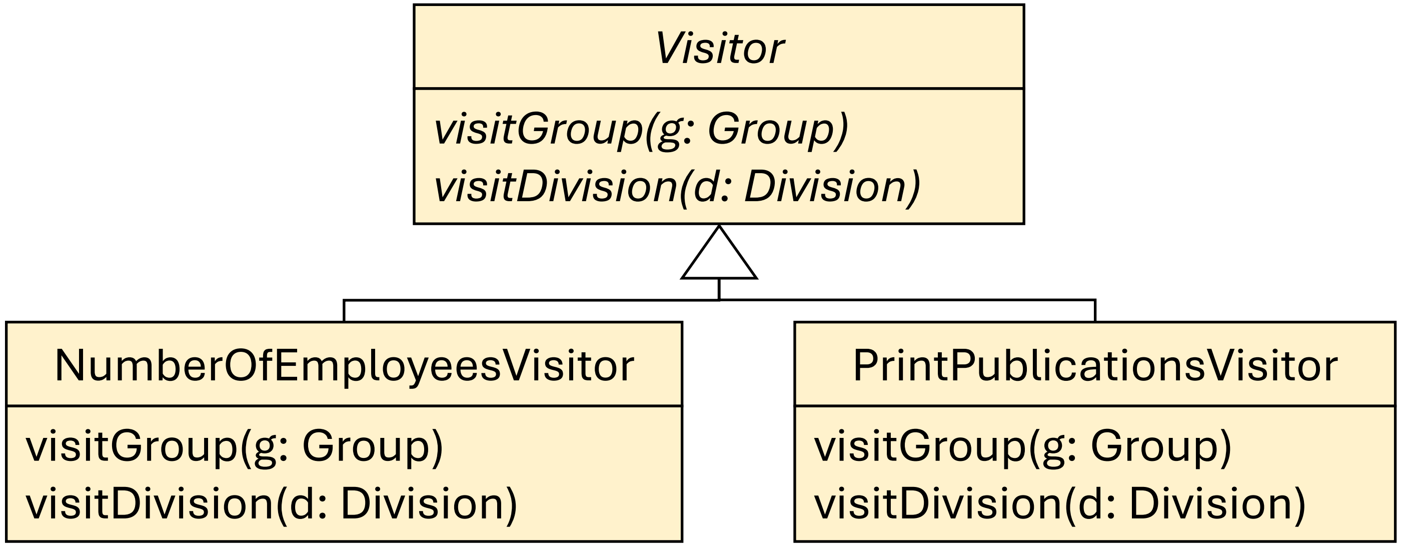
This is especially the case for the composite patter, which is why composite and visitor are often used together.

Approach (1)

The visitor is a class with a visit method per class of the object structure. Each visitor realizes one conceptual operation. That means a visitor is "a collection of how to run the same operation for different types".

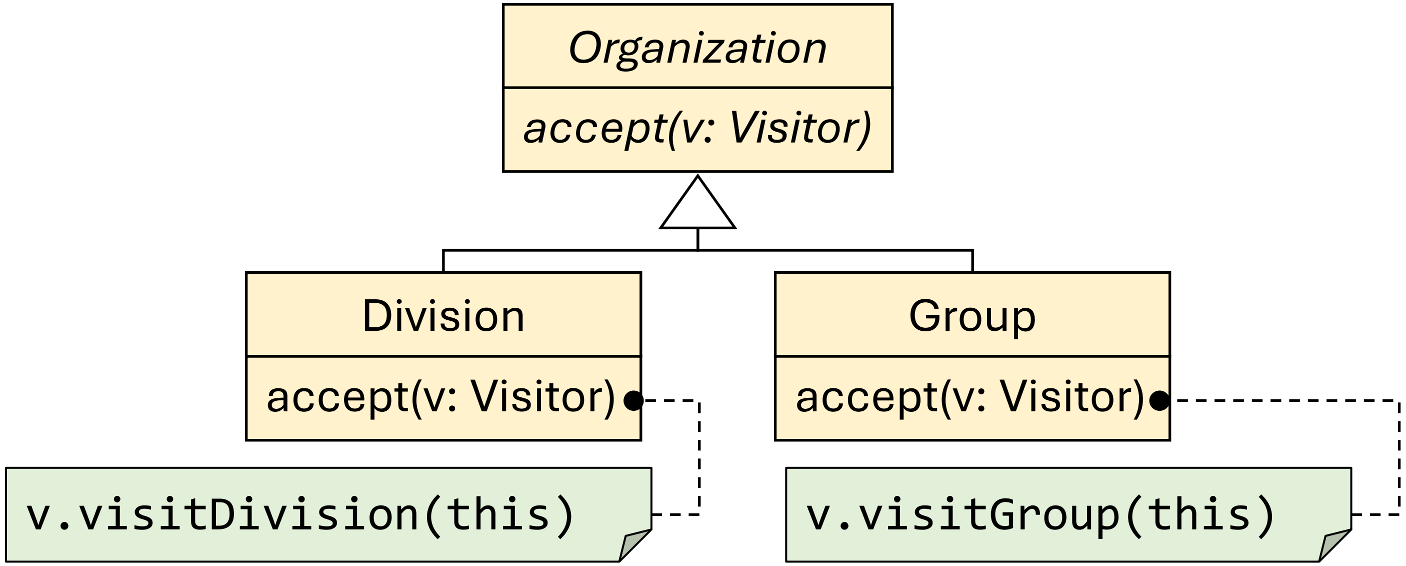
The object to be visited is passed as argument to visit.

In Java, the correct visit method can be found via overloading rules.

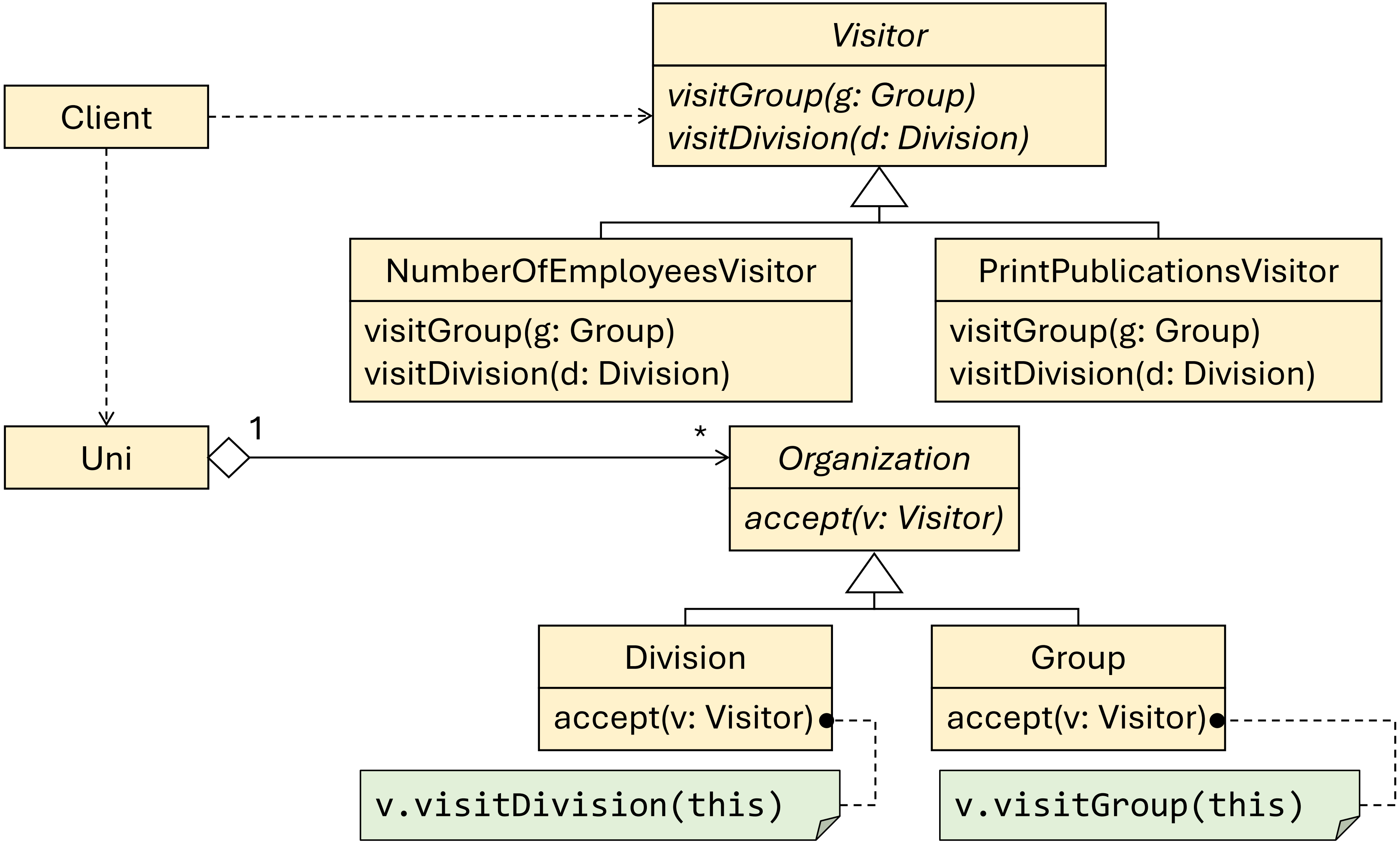


Approach (2)

In each class of the object structure, we simply add one new operation: accept(Visitor). These each just call the visit operation of the passed visitor and pass this as argument.



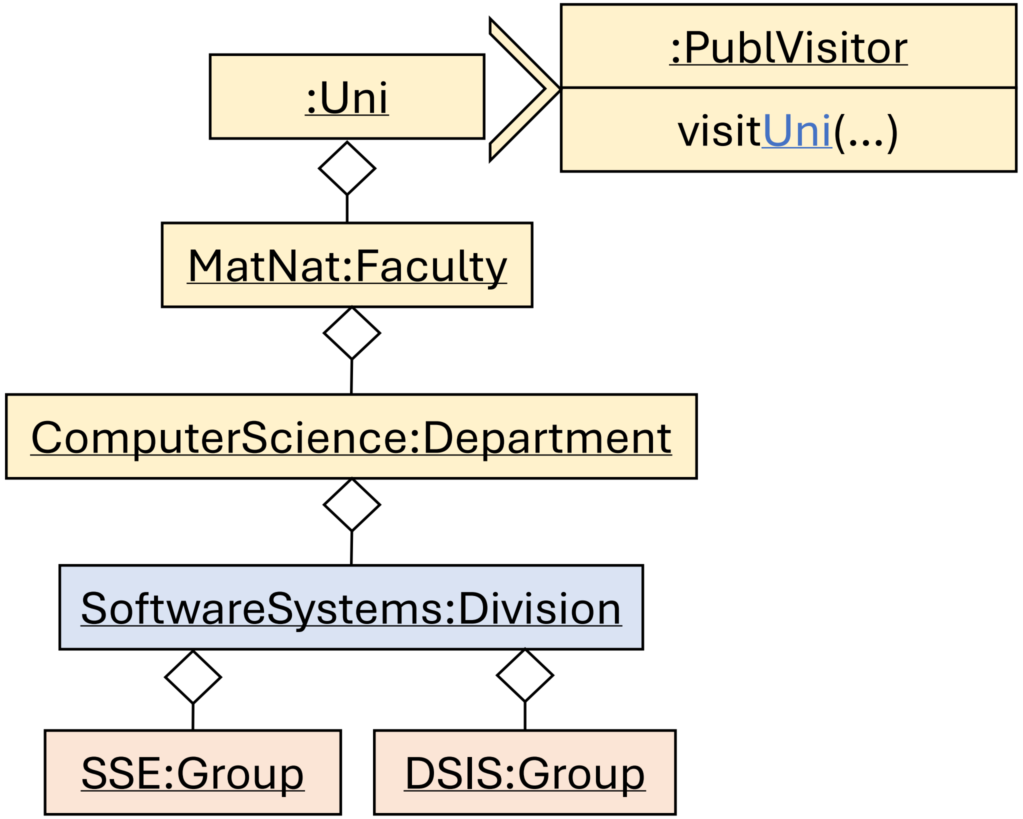
Complete Example



Runtime Example

The object structure consists of instances of different types. These are traversed by the visitor. On each instance, the visitor calls the accept method and passes itself as argument. In turn, in the accept method, the visitor's visit method is called and the this of the current object is passed back.

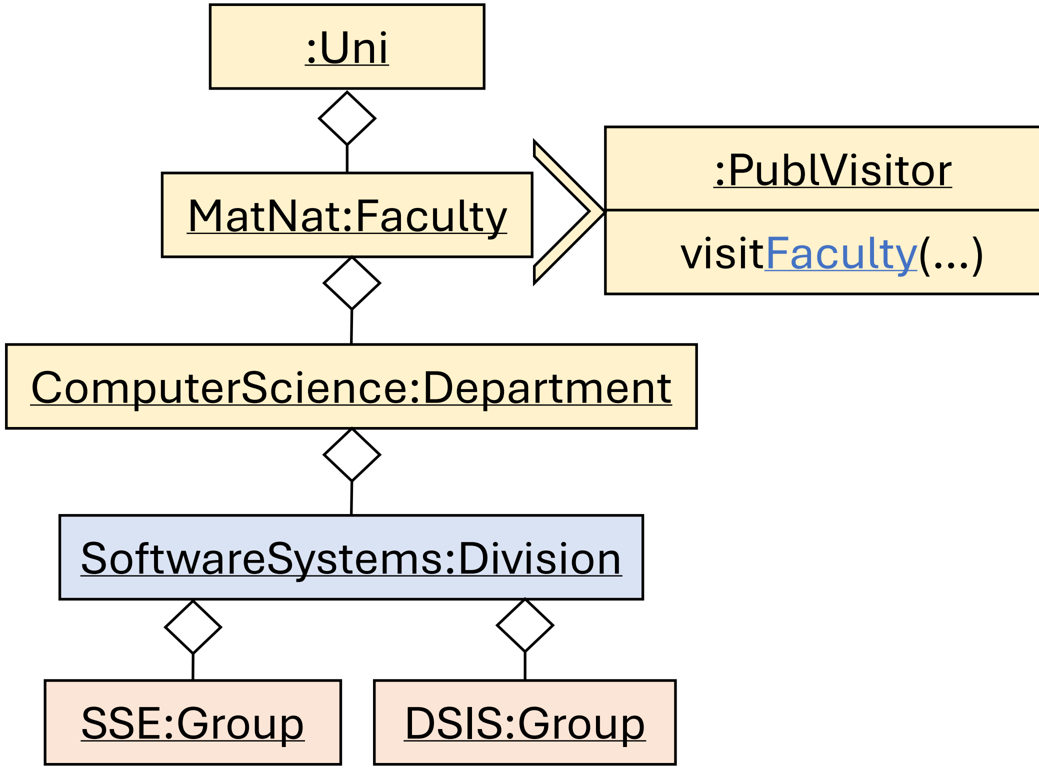
Then, the correct implementation of the operation the visitor represents is executed.



Runtime Example

The object structure consists of instances of different types. These are traversed by the visitor. On each instance, the visitor calls the accept method and passes itself as argument. In turn, in the accept method, the visitor's visit method is called and the this of the current object is passed back.

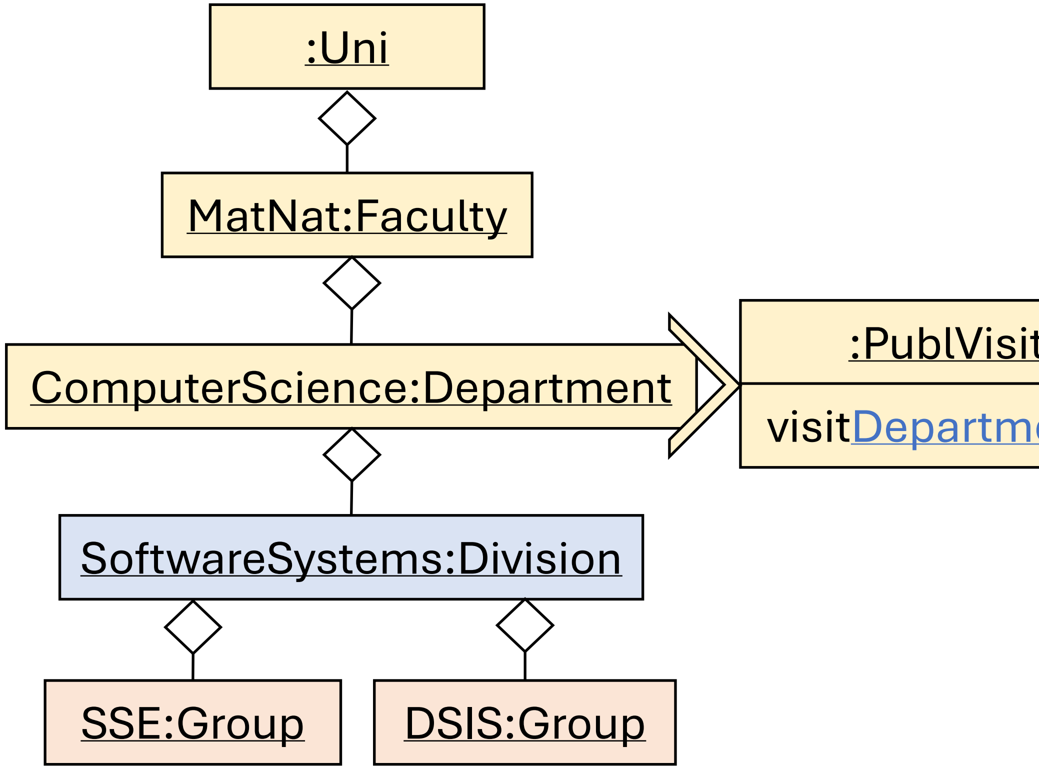
Then, the correct implementation of the operation the visitor represents is executed.



Runtime Example

The object structure consists of instances of different types. These are traversed by the visitor. On each instance, the visitor calls the accept method and passes itself as argument. In turn, in the accept method, the visitor's visit method is called and the this of the current object is passed back.

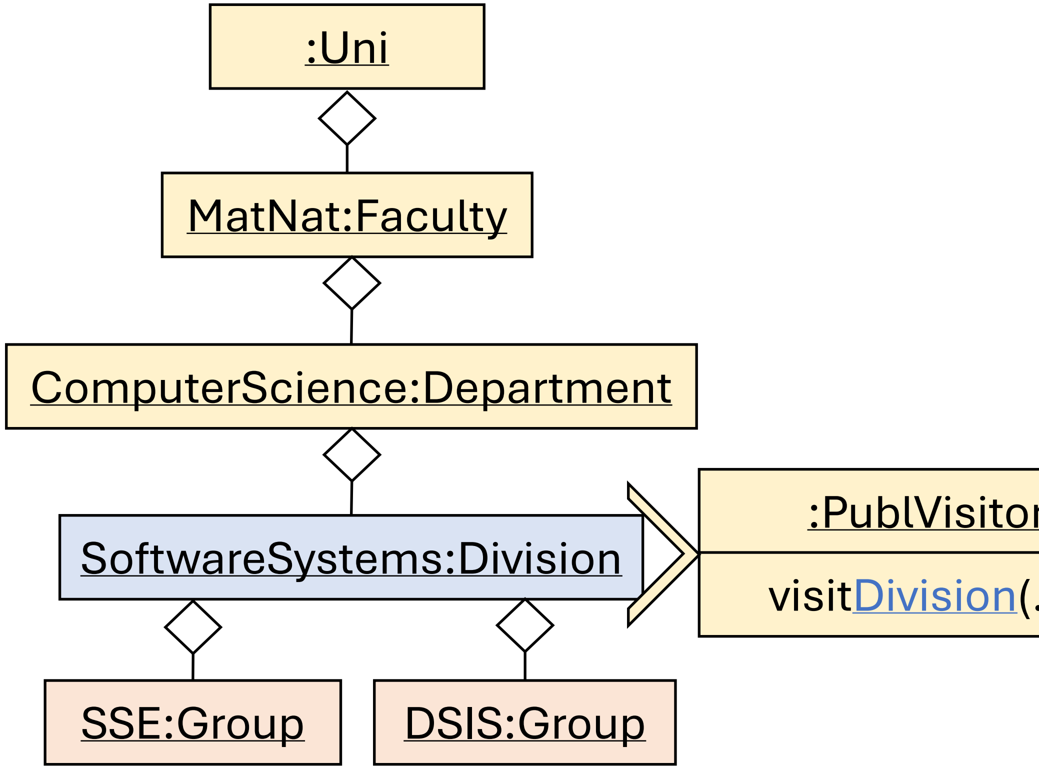
Then, the correct implementation of the operation the visitor represents is executed.



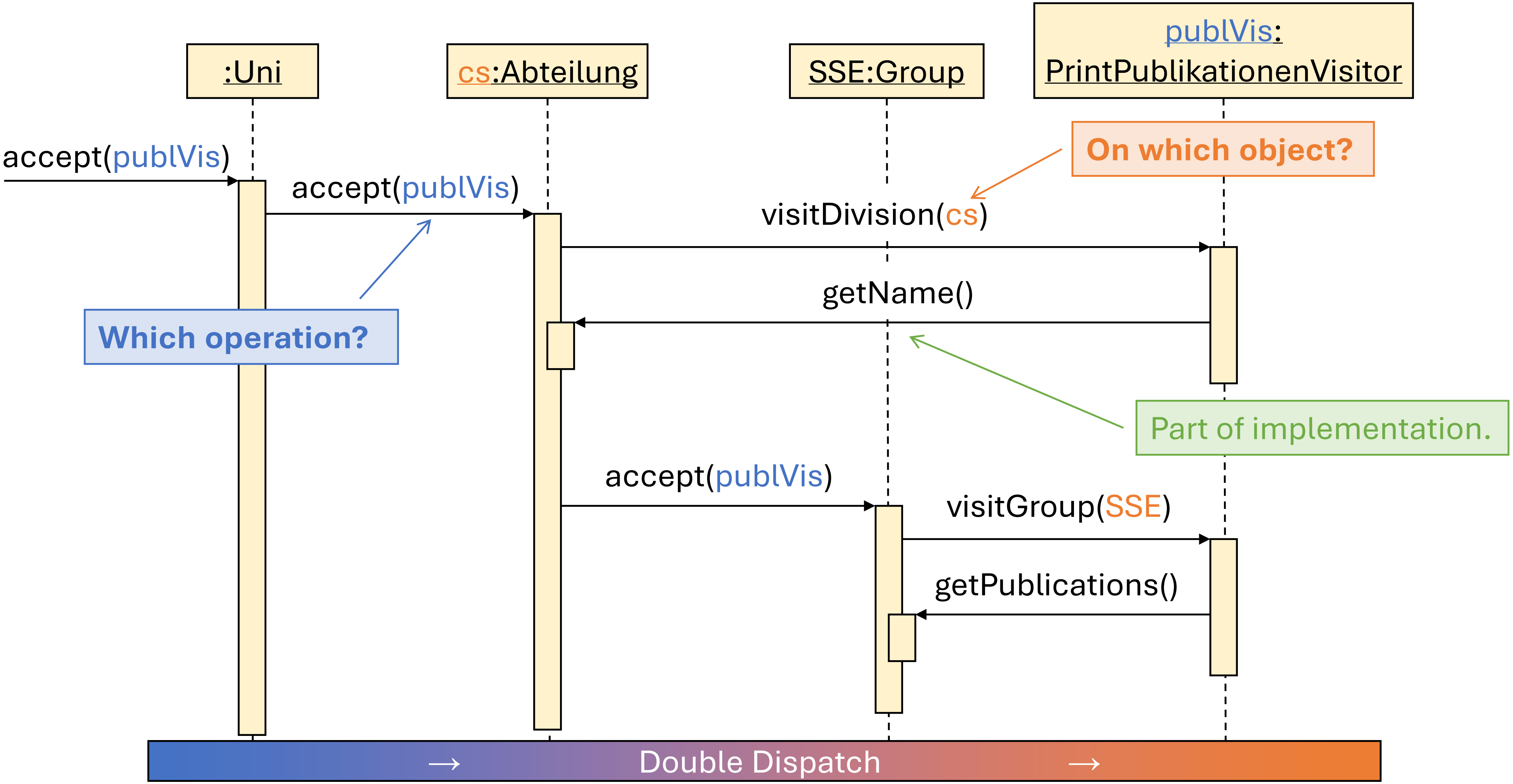
Runtime Example

The object structure consists of instances of different types. These are traversed by the visitor. On each instance, the visitor calls the accept method and passes itself as argument. In turn, in the accept method, the visitor's visit method is called and the this of the current object is passed back.

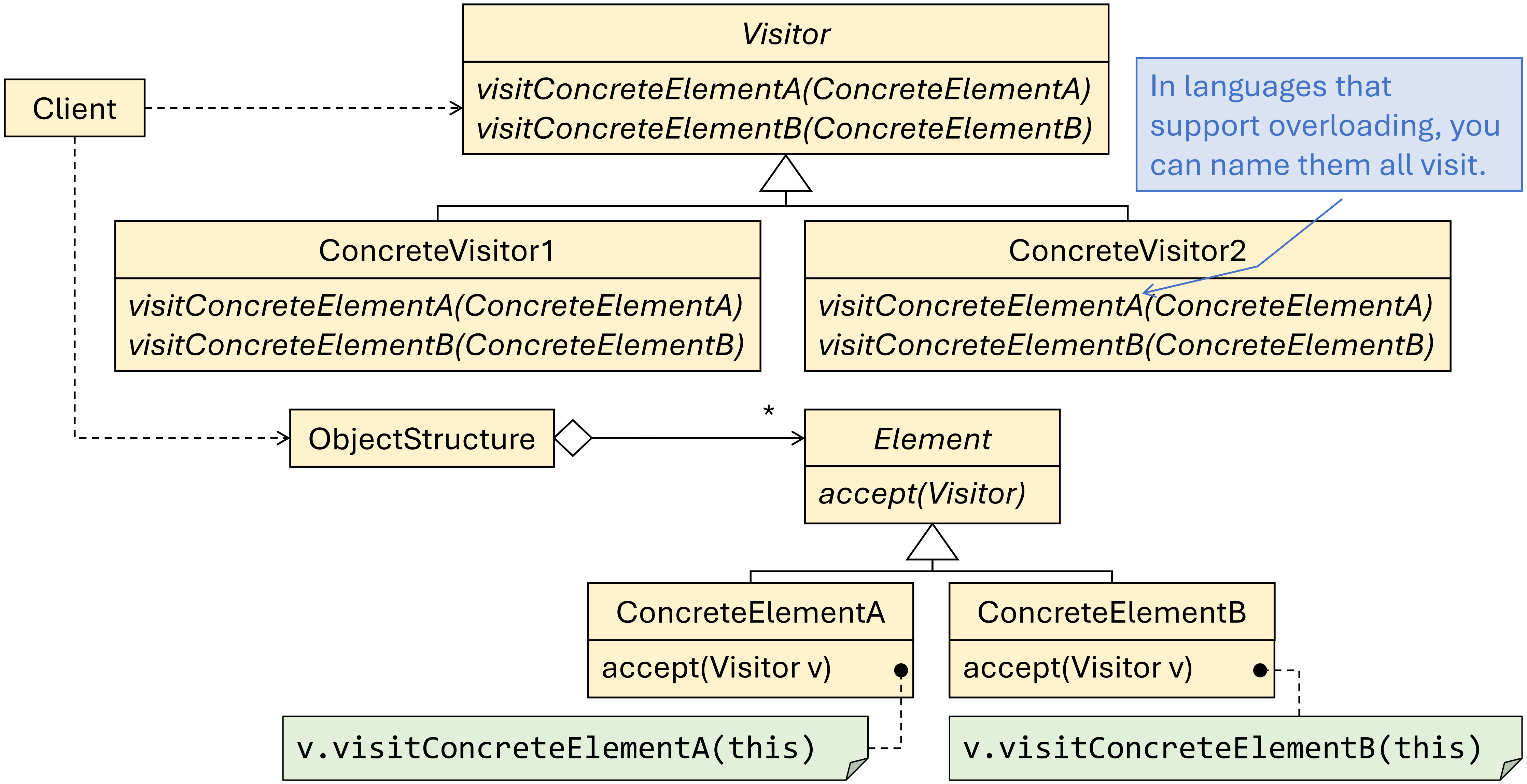
Then, the correct implementation of the operation the visitor represents is executed.



Interaction



Schema



Visitor
visitConcreteElementA(ConcreteElementA)
visitConcreteElementB(ConcreteElementB)
Client
ConcreteVisitor1
visitConcreteElementA(ConcreteElementA)
visitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
visitConcreteElementA(ConcreteElementA)
visitConcreteElementB(ConcreteElementB)
ObjectStructure
*
Element
accept(Visitor)
ConcreteElementA
accept(Visitor v)
ConcreteElementB
accept(Visitor v)

```
v.visitConcreteElementA(this)
```

```
v.visitConcreteElementB(this)
```

In languages that support overloading, you can name them all visit.

Implementation

Who defines how the object structure is traversed?

It can either be the visitor itself (at the end of each visit, call accept of the next object), or the object (at the end of each accept method, call accept of the next object).

Consequences

- Functional decomposition: visitor encapsulates methods that realize the same operation.

- Adding new operations is easy: Just implement a new visitor.

- Adding new classes of the object structure is difficult: New visit method in all visitors.

Applicability

- Stable set of classes of the object structure (rarely new classes) but often new operations needed.
- Heterogenic structure, classes have different interfaces.

Consequences, Advantages, Disadvantages

+ Open/Close principle: You can add new behavior that works with objects of different types without changing these classes.

+ Single responsibility principle: Multiple (type specific) variants of the same method are encapsulated in the same class.

+ Visitors work especially good with composites.

- Overkill if size of object structure is small.

- Adding new classes that visitors should work on requires to change all visitors.

- Visitors might need access to private variables of an object, which requires breaking its encapsulation.

Relation to other Patterns

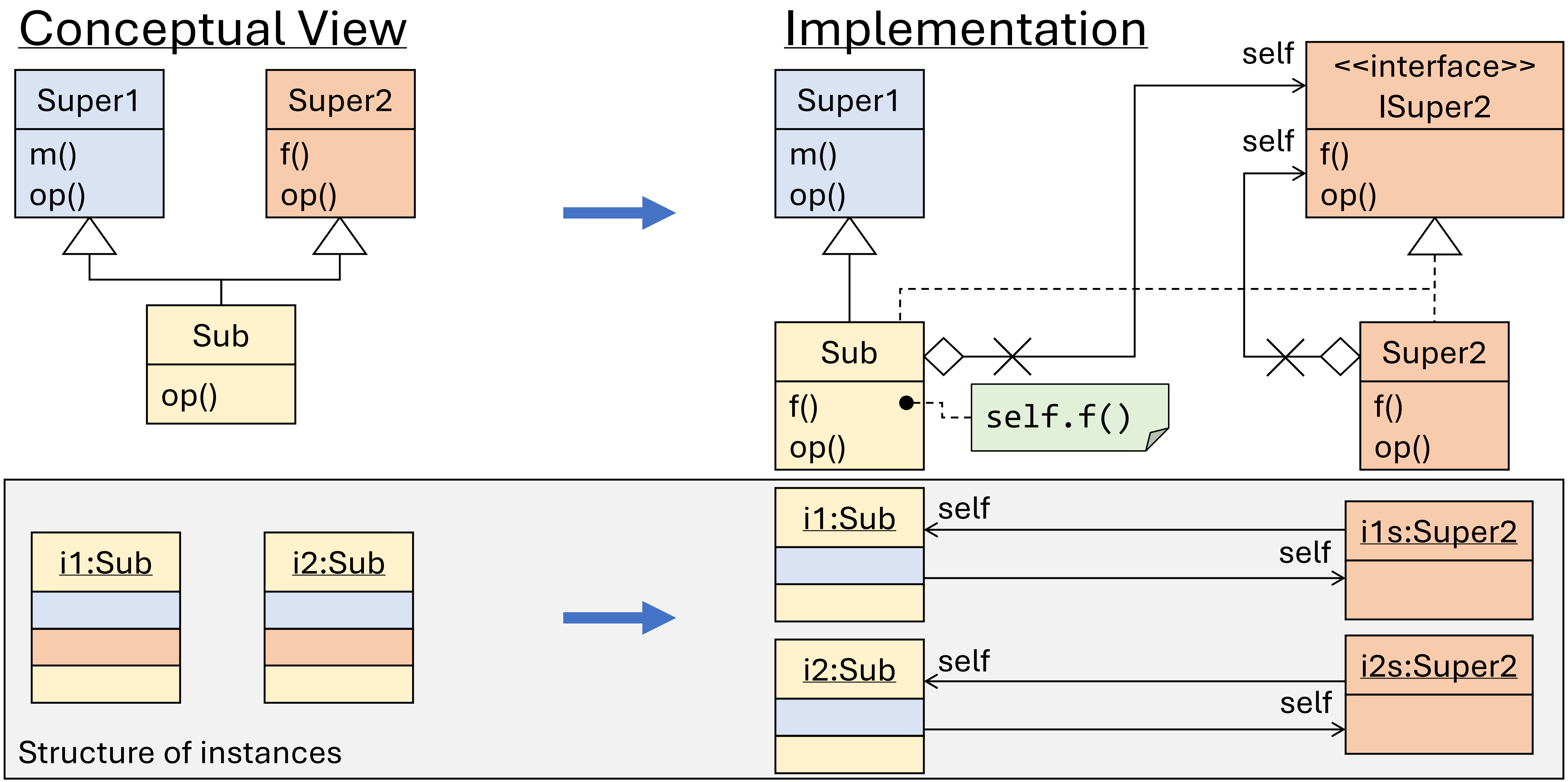
- You can use visitors to execute an operation on a composite object structure.

- Visitor can be used to extend an existing object (structure)'s interface, as with a decorator. However, decorators allow recursive stacking of decorator object, while visitors must always operate on the real subject.

Questions?

The Twins or Multiple Inheritance

Multiple Inheritance



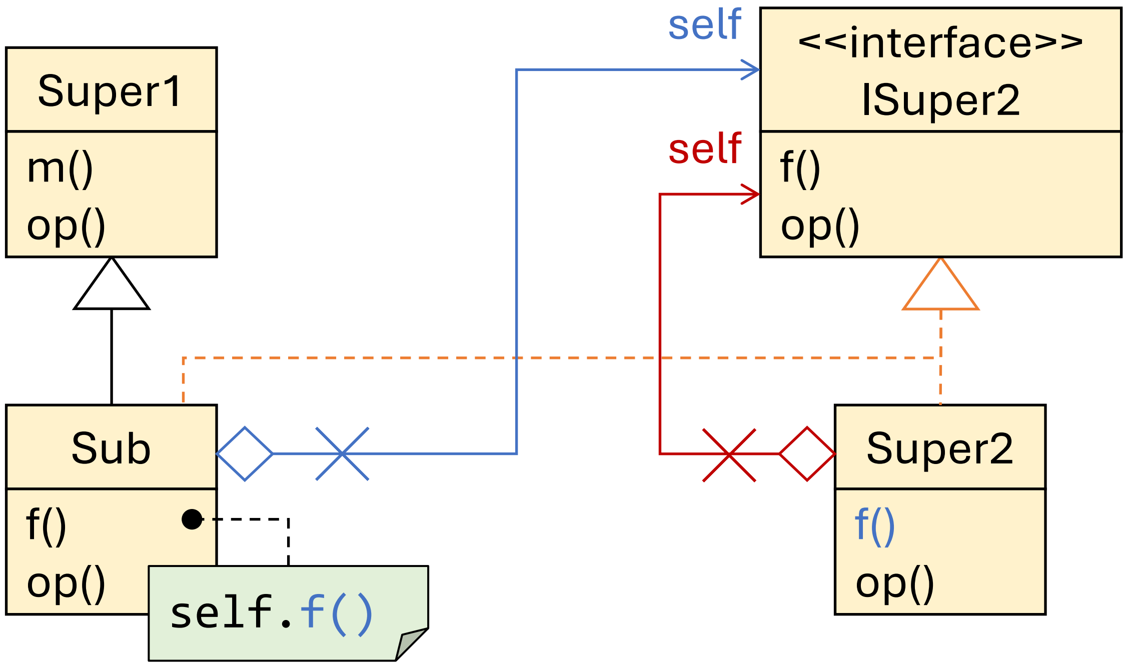
Multiple Inheritance in Java

The twin pattern is a way to realize multiple inheritance in languages that do not offer it as a language feature.

Applicable only when there are no semantic conflicts between superclass members, e.g., a clone method in both superclasses where one implements shallow cloning, and one implements deep cloning.

Approach:

- Code reuse via aggregation and forwarding.
- Subtyping via common interface.
- Overriding via "manual this".



Implementation Constraints and Problems

- If `Sub` and `Superclasses` lie in different packages,
  - protected methods of the "superclass" must be made public,
  - protected variables of the "superclass" must be made public or must be encapsulated via public getter/setter to be visible in the subclass.
- `self` has the type of the common interface. It is mutually passed in the constructors of the twins.
- Manual propagation of messages tedious and error-prone.
- Only easy to realize if there are no existing clients to the subclass and superclass already.

When do we not need Multiple Inheritance?

- If we only need multiple subtyping and no code reuse → implement multiple interfaces.

- If we do not need subtyping. E.g., a stack that uses a list to implement data storage.

Consequences, Advantages, Disadvantages

+ Resembles multiple inheritance as closely as possible.
+ Single responsibility principle: Functionalities of `ISuper2` are implemented in separate class `Super2`, `Sub` only delegates.

- More Code, as for all design patterns. Extensive manual work.
- More indirection (slightly slower).
- Open/close principle: Adding multiple inheritance to existing classes is tedious, requires extensive refactoring or behavior change and might even not be possible.

Questions?

Split Objects

Split Objects

Split objects are objects that interact as one conceptual unit with

- seemingly one identity
- seemingly one state
- seemingly one behavior

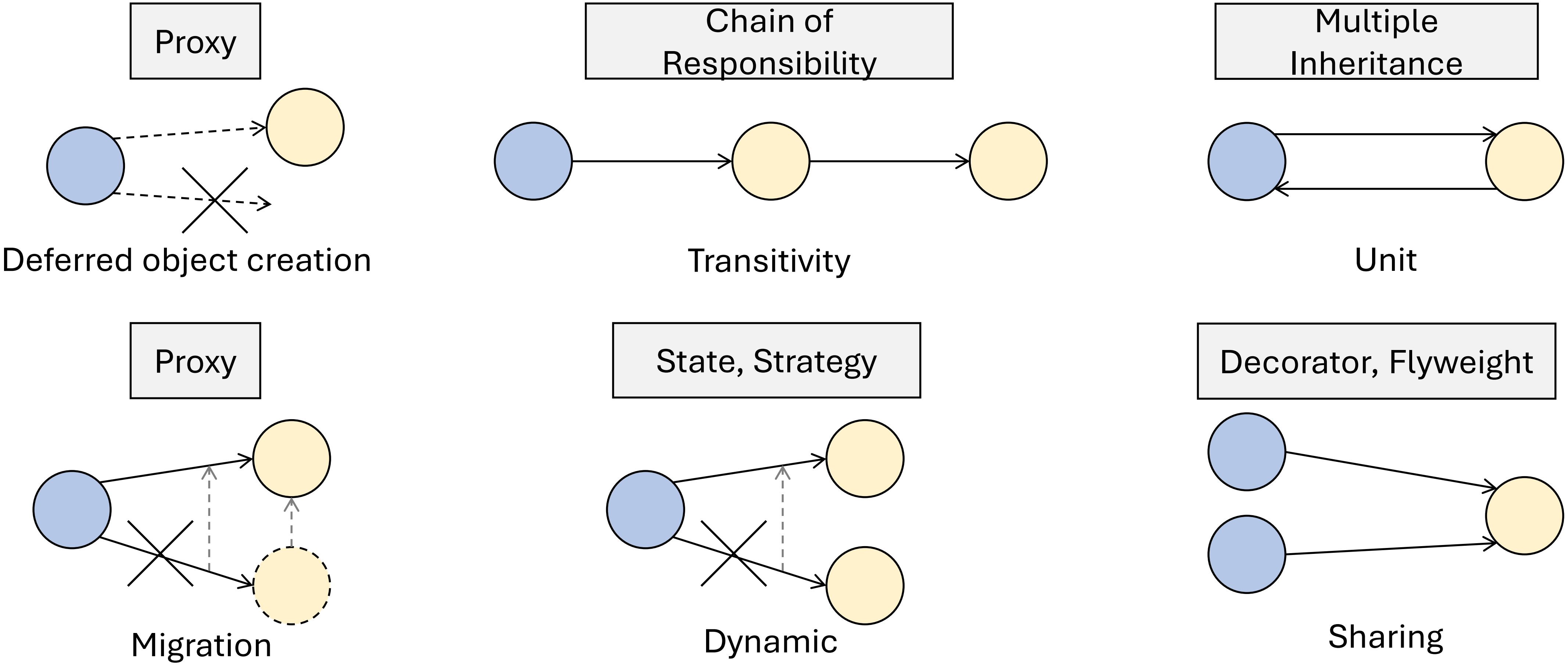
Clients of split objects have the impression to interact with one object only.

Motivation:

- Foreseen decomposition: Parts of an object (state/behavior) should be exchangeable.
- Unforeseen composition: Adding state/behavior to an object.

Split Objects

Multiple objects, but only one ● is visible to clients, providing the interface to the conceptual whole.



Proxy

Deferred object creation

Chain of Responsibility

Transitivity

Multiple Inheritance

Unit

Proxy

Migration

State, Strategy

Dynamic

Decorator, Flyweight

Sharing

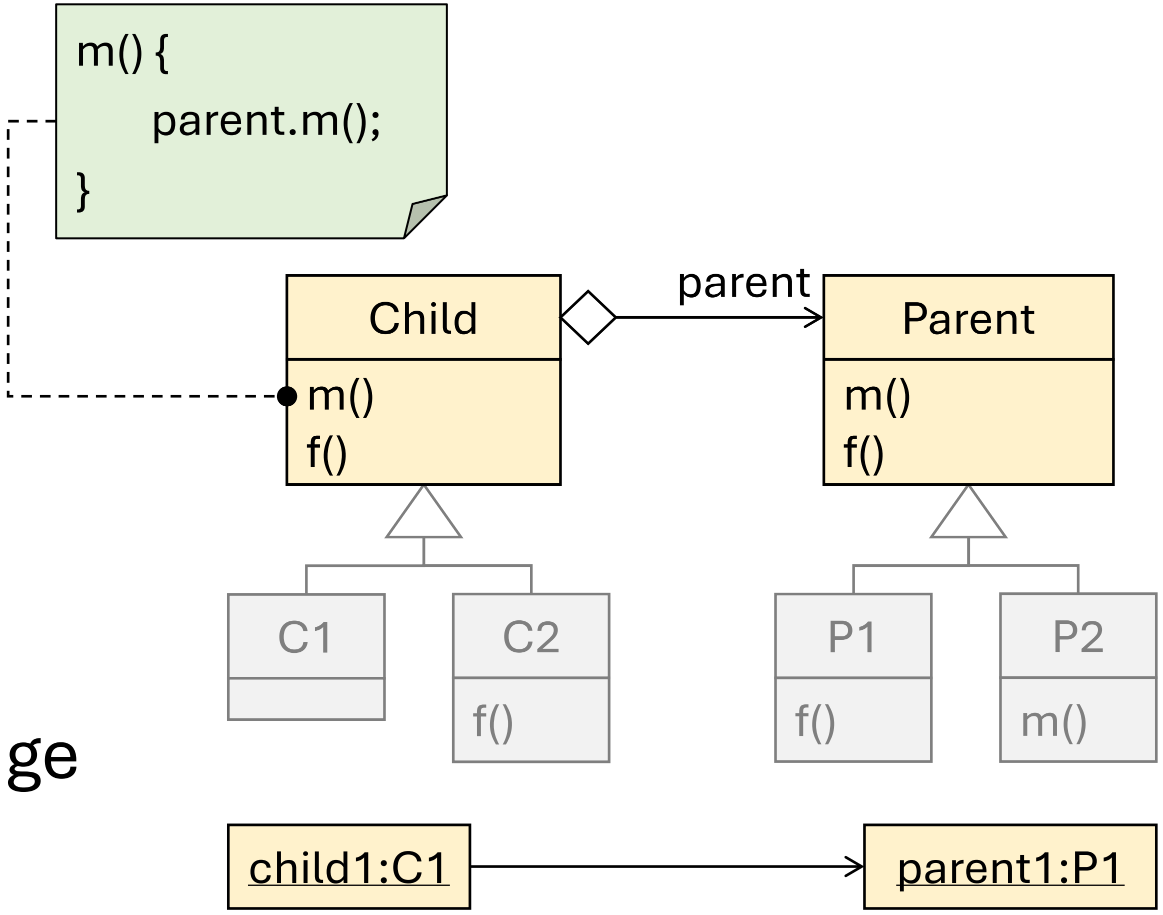
Common Structure

Aggregation,

- Child as "whole",
- Parent as "part".

Eventually subclasses.

Forwarding: Child forwards message to parent.



```
m() {
      parent.m();
}
```

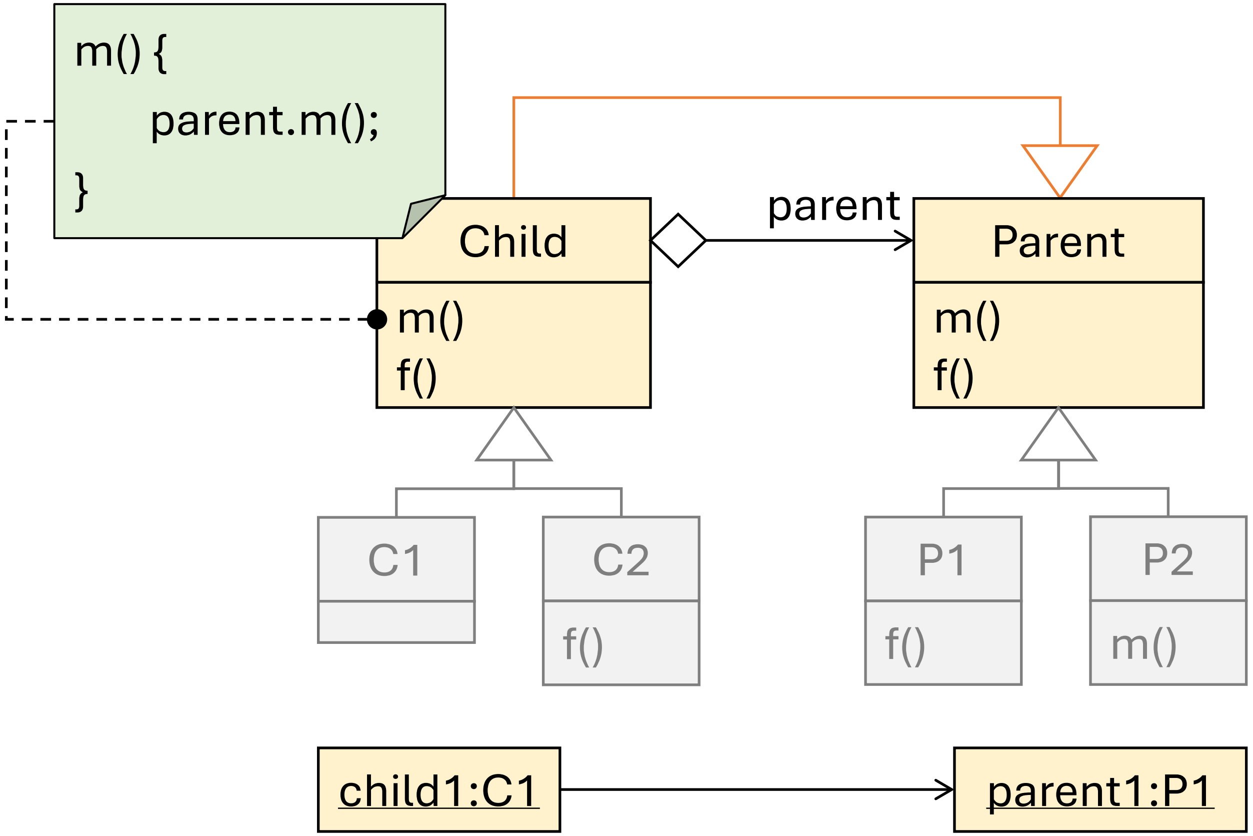
Relations between Child and Parent

		Model relations		
		Resending	Consultation	Delegation
Object-oriented relations	Forwarding: Child forwards message to parent.	✓	✓	✓
	Subtyping: Child provides parent interface.	✗	✓	✓
	Overriding: "this" is bound to child.	✗	✗	✓

Subtyping Implementation

Variant 1: Child is subclass of Parent.

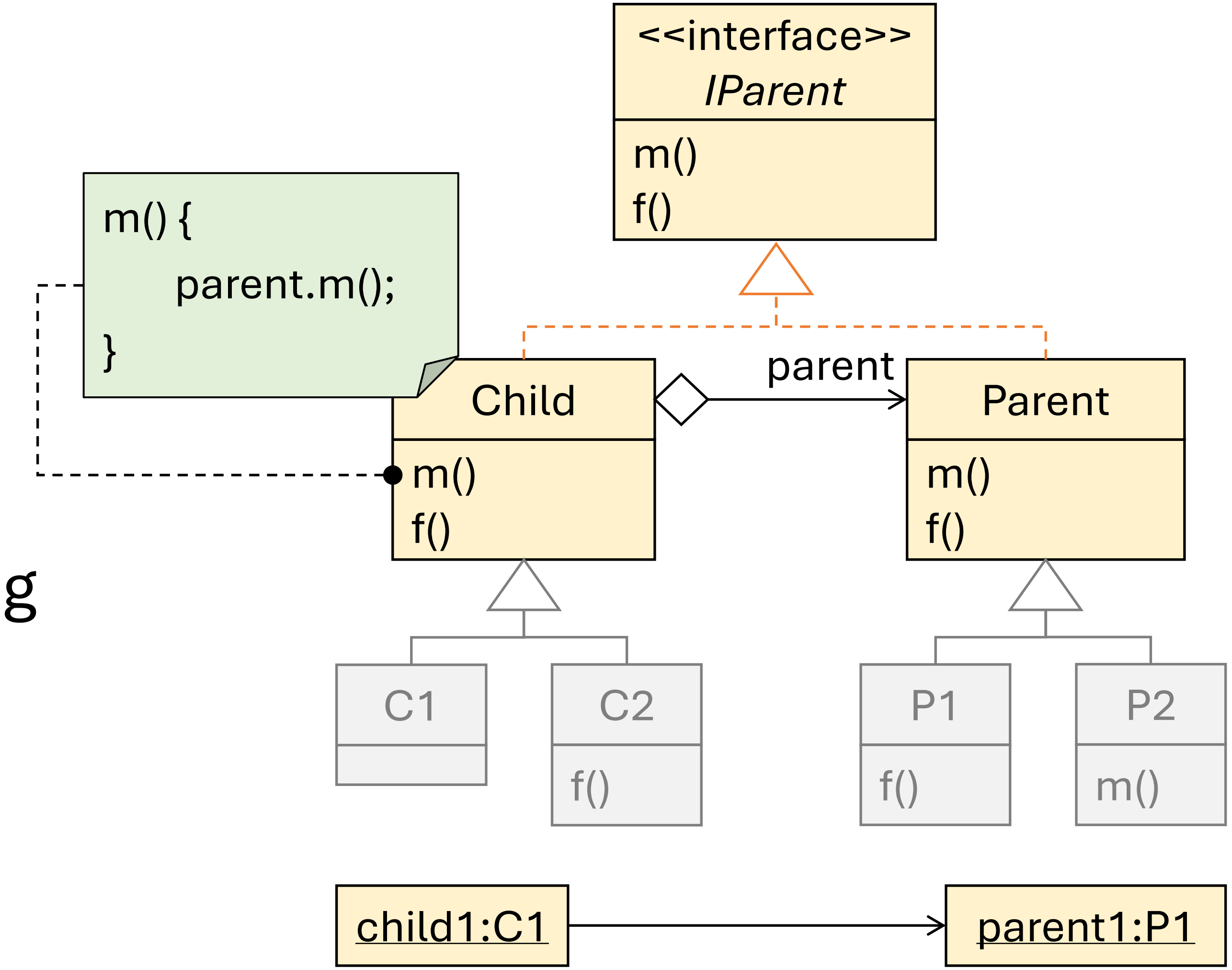
+ Easy.
- Child inherits data of parent.
- No multiple delegatees in single inheritance setting.



Subtyping Implementation

Variant 2: Child implements same interface as Parent.

+ No data duplication.
+ Clean separation of subtyping and code reuse.

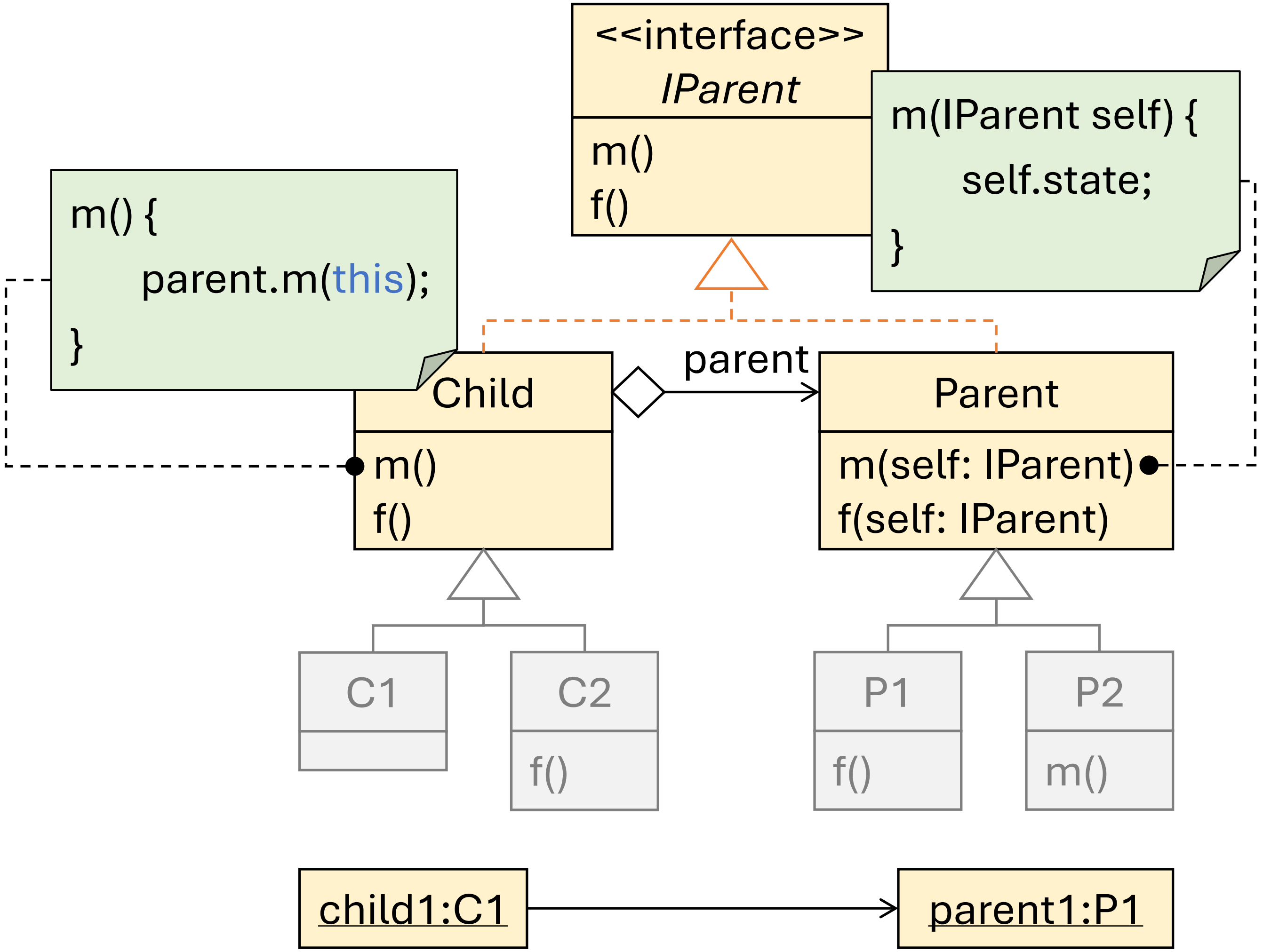


Overriding Implementation

All methods of the Parent class accept a parameter that simulates "`this`".

In the child classes, for methods that should be delegated, we forward the message to the parent object and pass `this` as argument to `self`.

In the parent's methods, we always refer to `self` in all places where we would otherwise refer to `this`.



```
m() {
    parent.m(this);
}
```

```
m(IParent self) {
    self.state;
}
```

Questions?

When to use what?

Creating objects of unknown type: Factory method or prototype.

Restricting the number of instantiations of a class: Singleton.

Creating an object structure with propagation of messages: Composite.

Substituting an object: Proxy.

Making two different interfaces compatible: Adapter.

Automatically react to changes/events: Observer

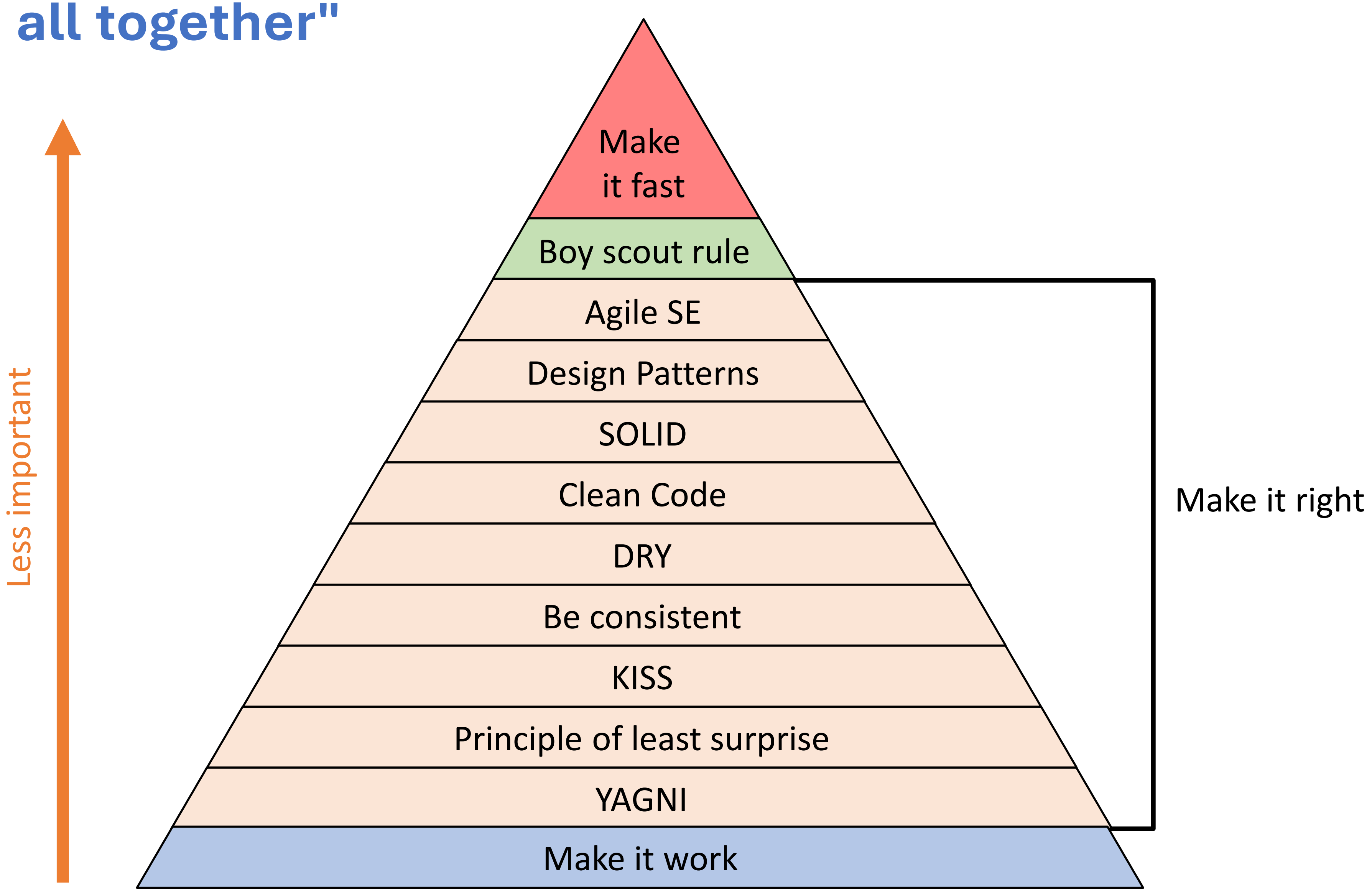
Reducing the dependencies of clients to a set of classes: Facade.

Adding/changing behavior…

- State dependent: State pattern.
- Of parts of a method: Template method.
- Of an entire method: Strategy.
- Of an existing object: Decorator.
- To an entire set of classes or object structure: Visitor.

Multiple Inheritance: Twins.

"Putting it all together"



Less important

Make it fast

Boy scout rule

Agile SE

Design Patterns

SOLID

Clean Code

DRY

Be consistent

KISS

Principle of least surprise

YAGNI

Make it work

Make it right

Summary

In today's lecture:

- Proxy
- Adapter
- Template Method
- State
- Strategy
- Decorator
- Visitor
- Twins