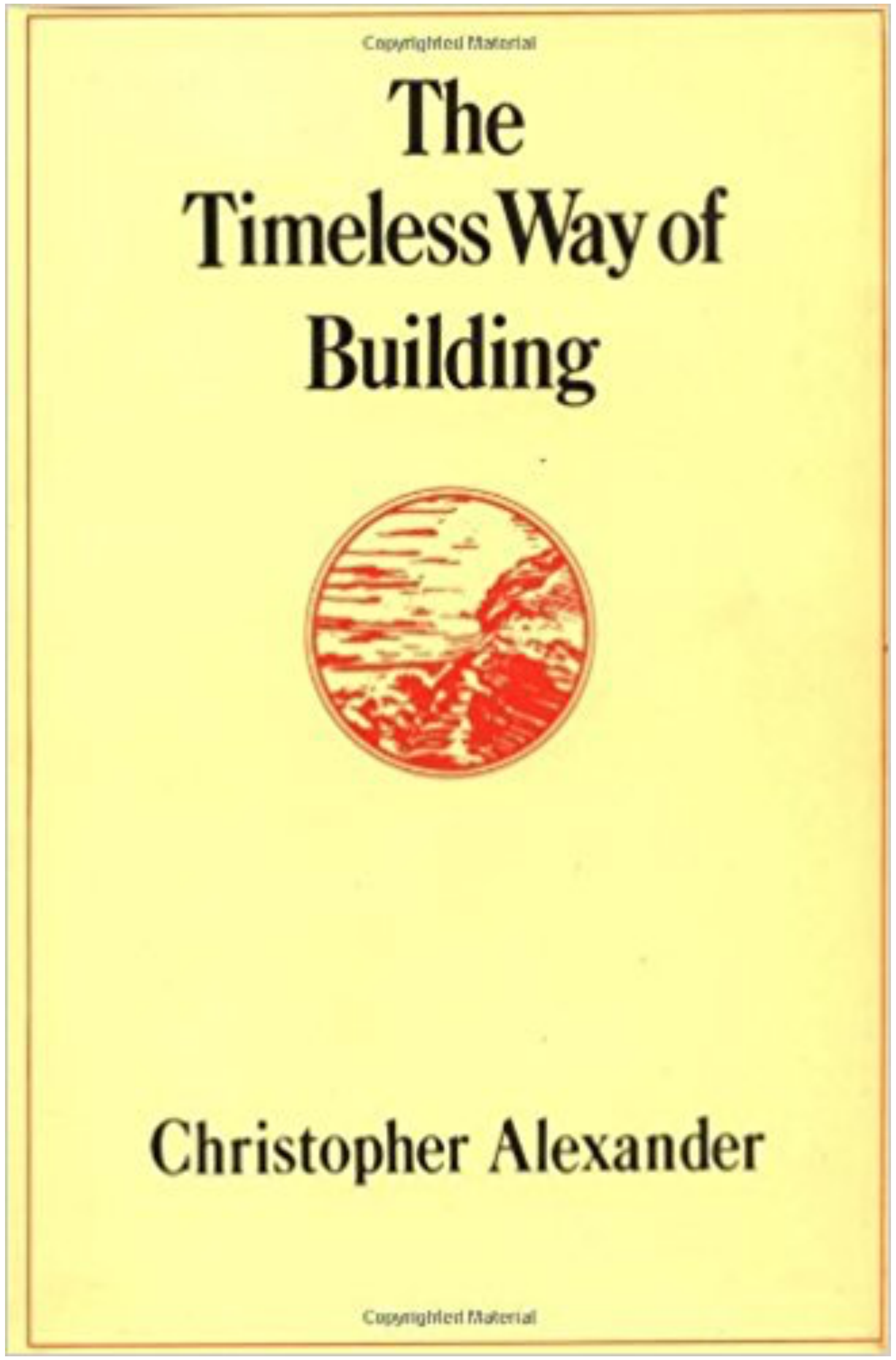
Design Patterns



Oregon State University

Announcements

Graded in-class assignment on Tuesday

Next Thursday, there will be in class Sprint 3 lab

Christopher Alexander

Architect, who asked the question ***"What makes good architectural design?"***

By studying high quality structures that solve similar patterns, he saw that ***patterns*** would appear the solutions to the problems.

He identified over 200 pattern for city planning, building design, gardens etc.

Patterns

"Each pattern is a three-part rule, which expresses a relation between a certain ***context****, a* ***problem****, and a* ***solution****."*

-Christopher Alexander

Patterns

Four elements describe the pattern:

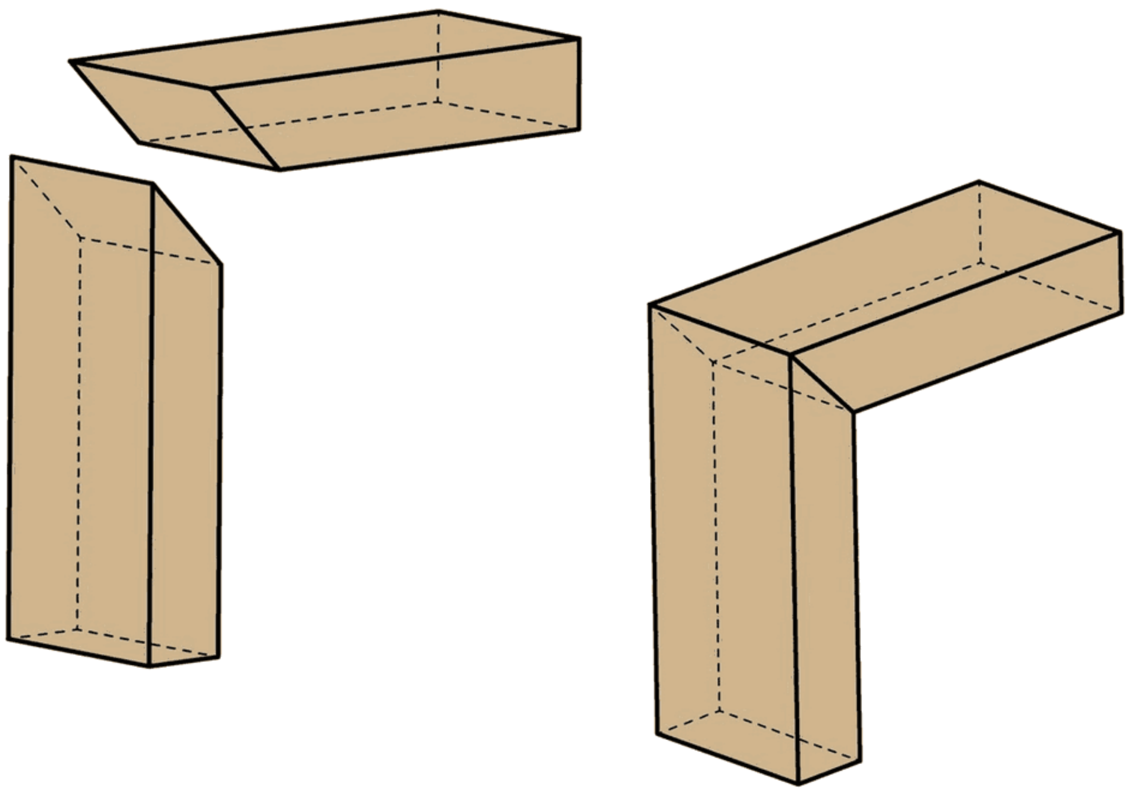
The ***name***

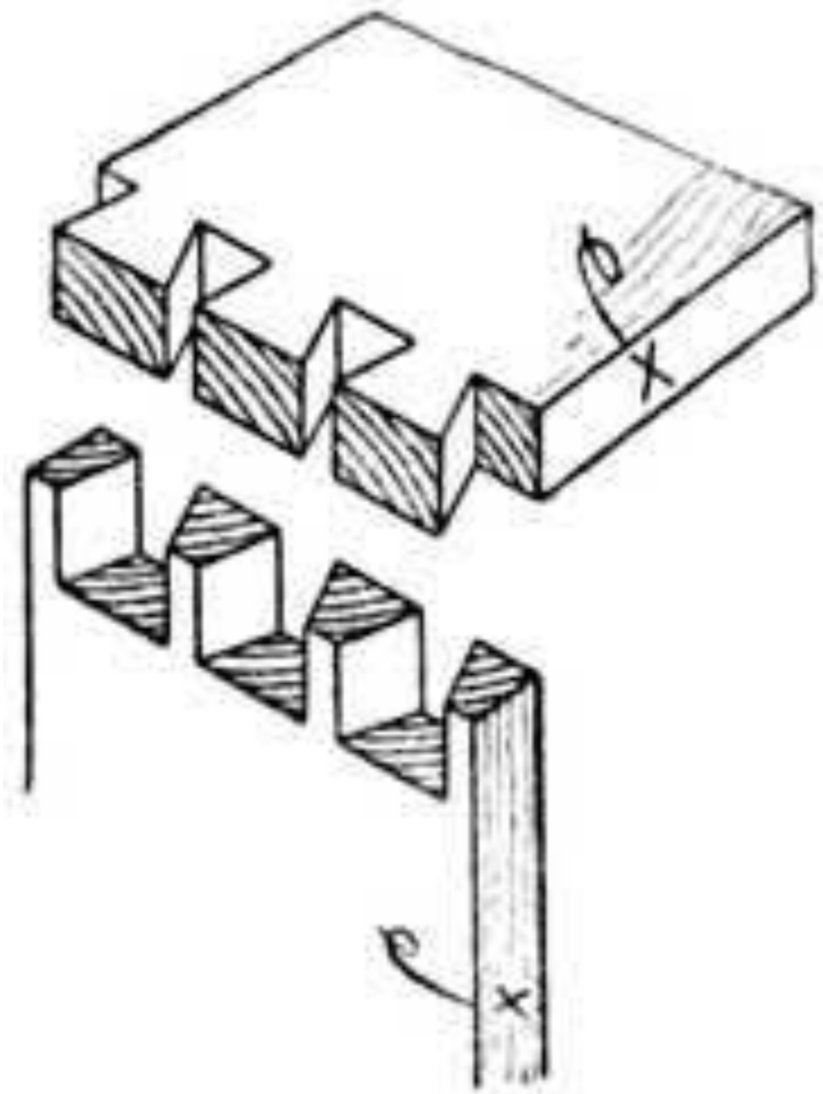
The ***purpose***; what problem is solves

How to solve the problem; the ***solution***

The ***constraints*** we have to consider in our solution

A higher level perspective





A higher level perspective

Patterns also describe a ***shared vocabulary.***

Which one is better?

“Should we use a dovetail or miter joint?”

or

“Should I make the joint by cutting down into the wood and then going back up 45 degrees and...”

A higher level perspective

The former avoids getting bogged down in details

The former relies on the carpenter's ***shared knowledge***

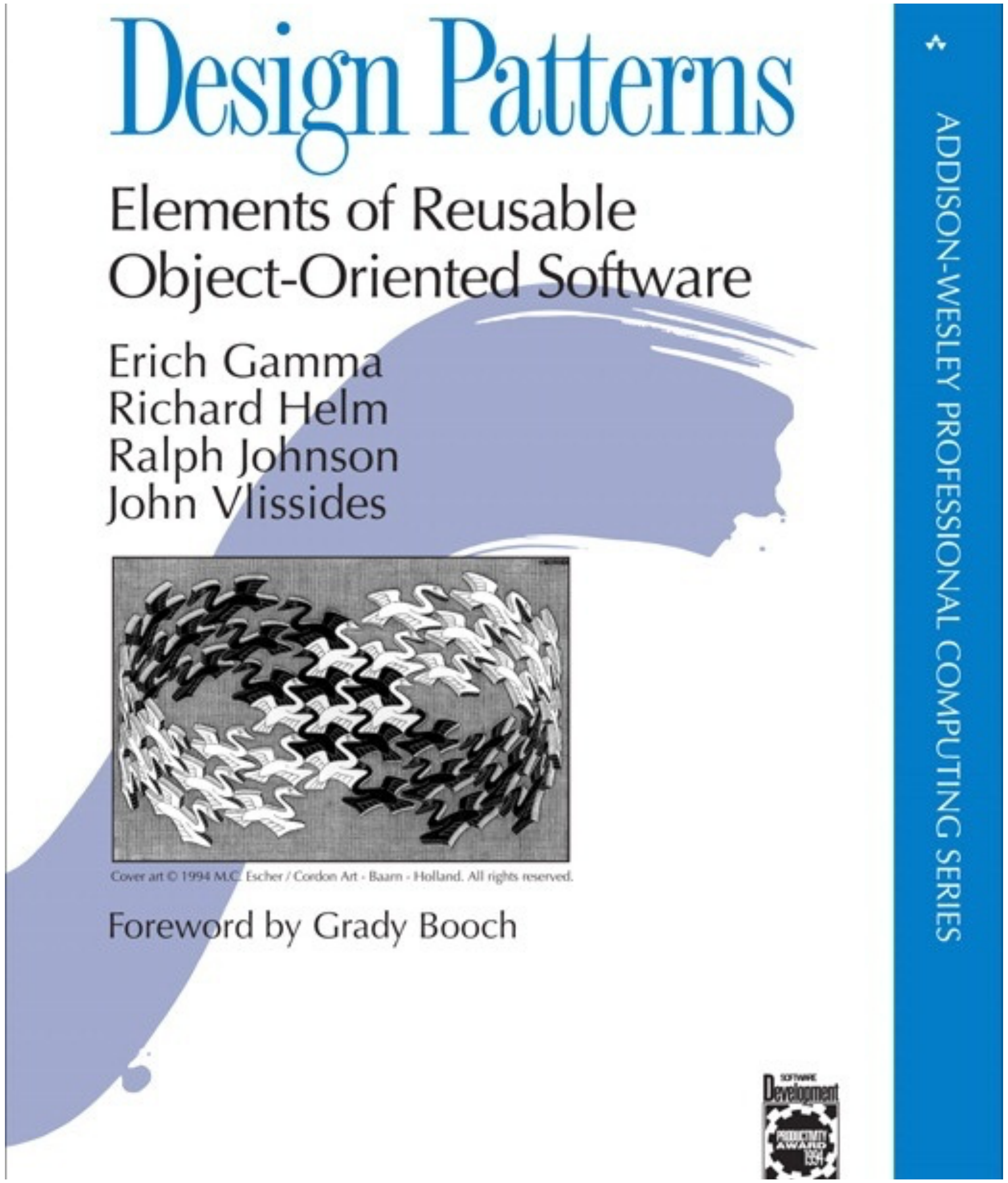
> *[Design patterns] distill and provide a means to reuse the design knowledge gained by experienced practitioners.*

-G.O.F.

Software Design Patterns

The seminal book published by the "Gang of Four."

They propose 23 patterns, organized in 3 categories.



Key Features of a Pattern

Name

Intent: The purpose of a pattern.

Problem: What problem does it solve?

Solution: The approach taken.

Participants: The entities involved in the pattern.

Consequences: The effect the pattern has on your code.

Implementation: Example ways to implement it

Structure: a class diagram

Software Design Patterns

3 Categories:

Creational: they abstract away the object instantiation (creation)

Structural: are concerned with how classes and objects are composed to form larger structures

Behavioral: are concerned with algorithms and the assignment of responsibilities between objects.

Creational patterns

Why not use `new`?

`new` binds you to a specific class

What if you want to instantiate different classes (in a class hierarchy) depending on different conditions?

What if you want to restrict how many objects are created (e.g. access to limited resources)

Singleton

Intent: ensure a class has only one instance, and provide a global point of access to it.

Motivation: having a single instance of a class is sometimes important; e.g. There can be only one file system or event thread.

Singleton

We want to restrict access such that this is no longer possible:

```
Singleton s = new Singleton();
```

Instead, we want to do this:

```
Singleton s = Singleton.getInstance();
```

We want to ensure that only a ***unique instance*** exists.

```java
public class Singleton {
    private static Singleton s = null;

    private Singleton() {}

    public static Singleton getInstance() {
        if (s == null)
            s = new Singleton();
        return s;
    }
}
```

```java
public class Singleton {
    private static Singleton s = null;

    private Singleton() {}

    ...tInstance() {
    ...);

}
```

Declaring the constructor *private* means we cannot create instances outside of the class.

Therefore, we control where an object can be instantiated.

```java
public class Singleton {
    private static Singleton s = null;

    private Singleton() {}

    public static Singleton getInstance() {
        if (s == nu
            s = new
        return s;
    }
}
```

The ***static*** keyword allows us to access fields and methods without an instance:

```java
Singleton.getInstance();
```

```
public cla
    privat

    privat

    public static Singleton getInstance() {
        if (s == null)
            s = new Singleton();
        return s;
    }
}
```

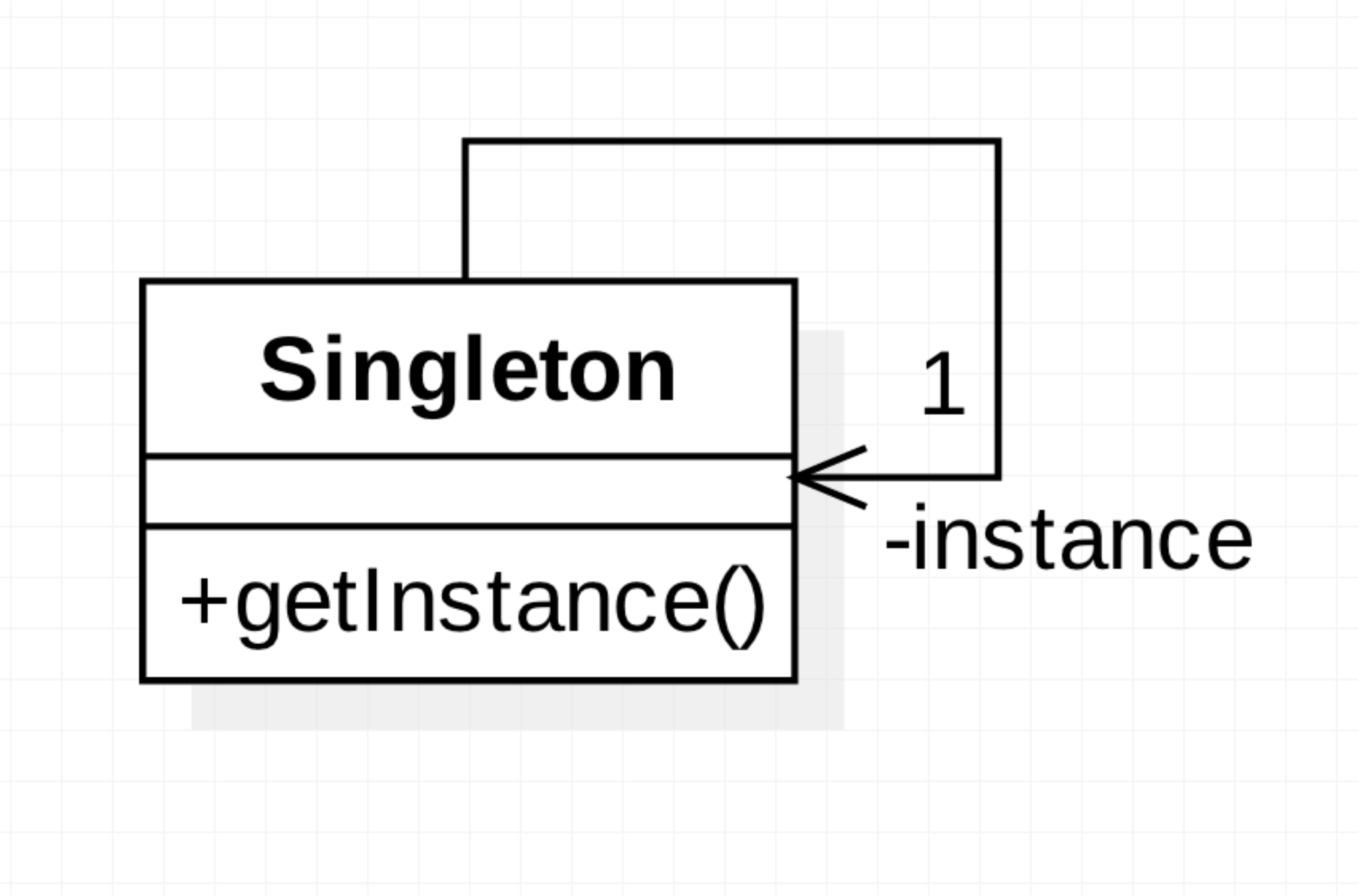
This is called ***lazy initialization.*** We only create the object when we need them.

```java
public class Singleton {
    private static Singleton s = null;

    private Singleton() {}

    public static Singleton getInstance() {
        if (s == null)
            s = new Singleton();
        return s;
    }
}
```

Structure



Pros & Cons

Pros:

Easy instance management

Cons:

It acts like a global variable and shares all the problems associated with them

Breaks SRP, as the objects now has to control it's own lifetime cycle

Factory Method

Intent: Define an interface for creating an object. Let subclasses decide which class to instantiate.

Motivation: Frameworks use abstract classes and maintains a relationship between objects. The framework also instantiates the objects.

The instantiated objects are sometimes knows as ***products.***

Factory Method

Like with Singletons, we want to restrict how objects are created. This should be impossible:

```
Transport Truck = new Truck();
```

And it is replaced with:

```
Transport object =
 logistics.makeInstance();
```

```java
public class RoadLogistics extends
 Logistics {

    private RoadLogistics() {}

    public Transport makeInstance() {
        return new Truck();
    }
}
```

We declare the constructor as ***private,*** to control how objects are created.

```
public class Ro
 Logistics {

    private RoadLogistics() {}

    public Transport makeInstance() {
        return new Truck();
    }
}
```

```java
public class RoadLogistics extends
 Logistics {

    private RoadLogistics() {}

    public Transport makeInstance() {
        return new Truck();
    }
}
```

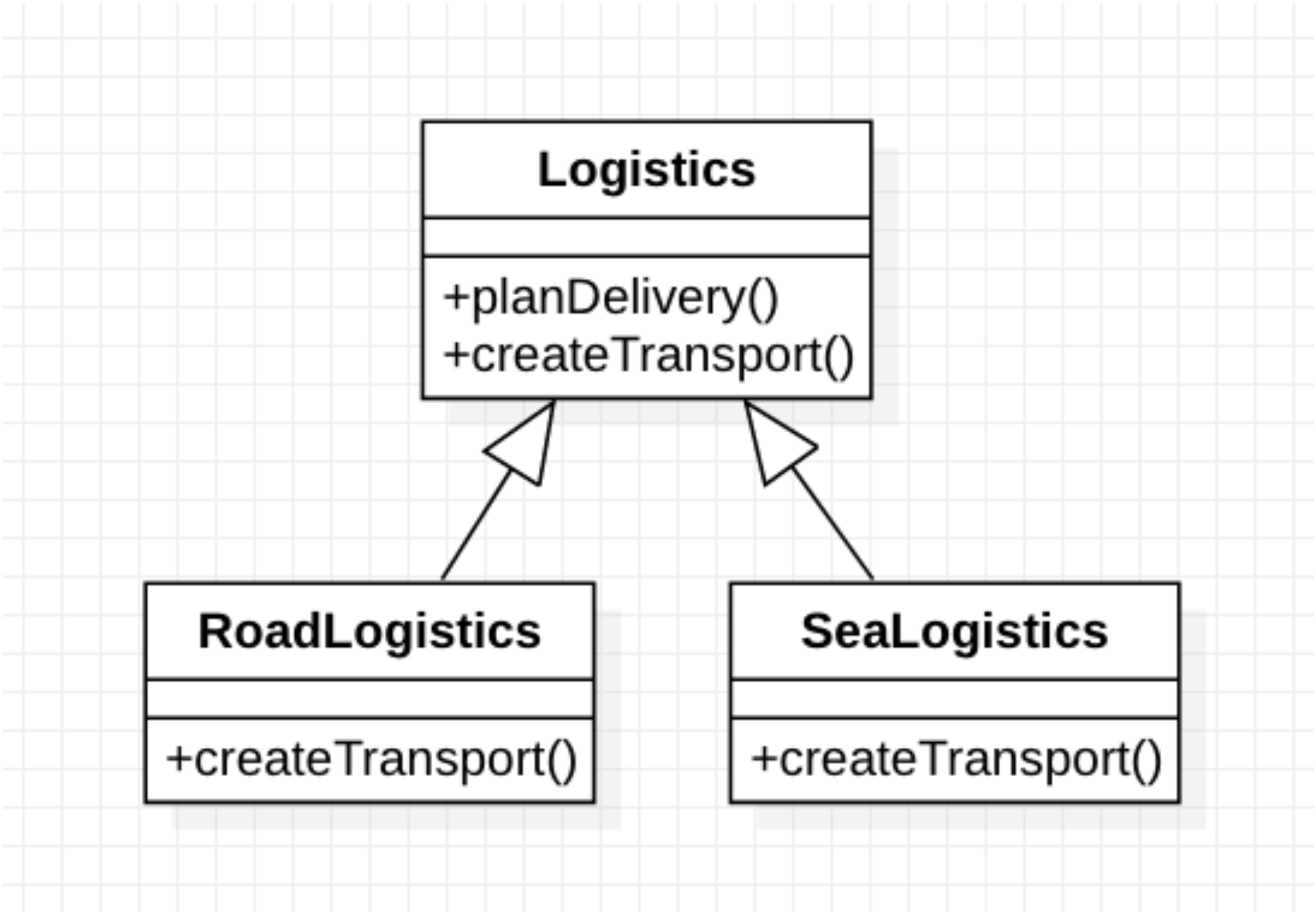
```java
public class RoadLogistics extends
 Logistics {

    private RoadLogistics() {}

    public Transport makeInstance() {
        return new Truck();
    }
}
```

We create the object inside the method. We can now ***control*** which object we create, and subclasses can change this

Structure



Logistics
+planDelivery()
+createTransport()
RoadLogistics
+createTransport()
SeaLogistics
+createTransport()

Pros & Cons

Pros:

Follows OCP

Makes adding new products easy

Moves creating objects to one place

Cons:

Extra complexity due to the class hierarchy

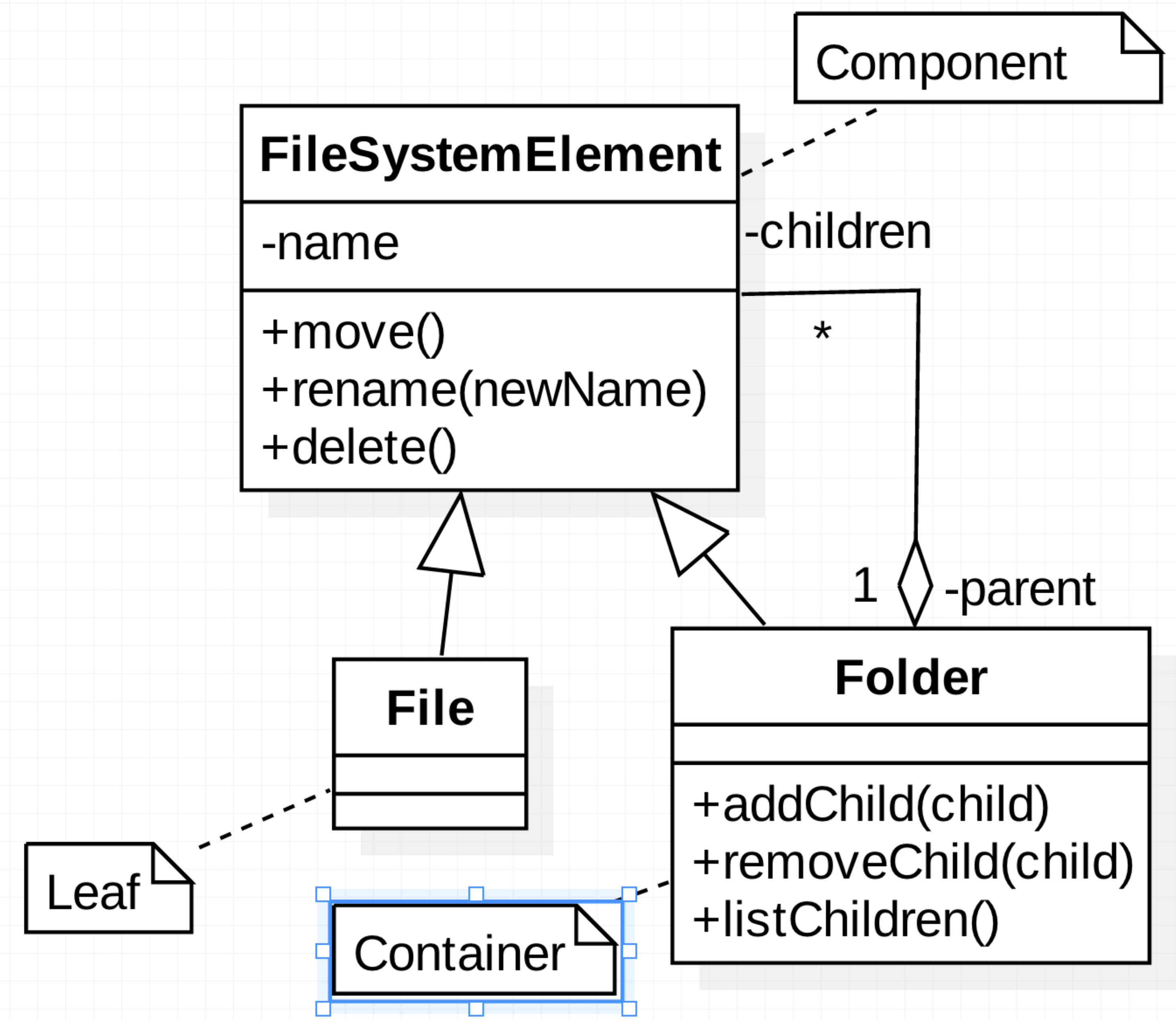
Structural Patterns

Composite

Intent: compose objects into tree structures to represent part-whole hierarchies. Clients can treat individual objects and compositions **uniformly**.

Motivation: A file system has files and folders. Users want to manipulate files and folders the same way (e.g. move, rename, delete etc).

Composite



Component
FileSystemElement
-name
+move()
+rename(newName)
+delete()
-children
*
1
-parent
File
Folder
+addChild(child)
+removeChild(child)
+listChildren()
Leaf
Container

```java
public abstract class FileSystemElement {
    public void move(){}
    public void rename(String newName){}
    public void delete(){}
}

public class Folder extends FileSystemElement {
    private List<FSE> children = new ArrayList<>();

    public void addChild(FileSystemElement child) {
        children.add(child);
    }

    public void removeChild(FileSystemElement child) {
        children.remove(child);
    }
}

public class File extends FileSystemElement {
    // do file stuff
}
```

```
public abstract class FileSystemElement {
    public void move(){}
    public void rename(String newName){}
    public void delete(){}
}
```

```
public clas
    private

    public
        ch
    }

    public
        ch
    }
}

public clas
    // do 
}
```

The base class contains the ***common operations*** for a "File System Element"

`abstract` means that it cannot be instantiated.

Oregon State University

```
public abstract class FileSystemElement {
    public void move(){}
    public void rename(String newName){}
    public void delete(){}
}

public class Folder extends FileSystemElement {
    private List<FSE> children = new ArrayList<>();

    public void addChild(FileSystemElement child) {
    }

    pu
    }
}

publi
    //
}
```

The `Folder` class is the ***container***.

It's responsibility is managing and accessing the children.

It may override some common operations (e.g. `delete`)

Oregon State University

```java
public abstract class FileSystemElement {
    public void move(){}
    public void rename(String newName){}
    public void delete(){}
}

public class Folder extends FileSystemElement {
    private List<FSE> children = new ArrayList<>();

    public void addChild(FileSystemElement child) {

    }

    pu                                          d) {

    }
}
```

The **File** class is the ***leaf,*** as it had no children.

```java
public class File extends FileSystemElement {
    // do file stuff
}
```


Oregon State University

Pros & Cons

Pros:

It's easy to add new types of components

Clients can manipulate both types homogeneously.

Cons:

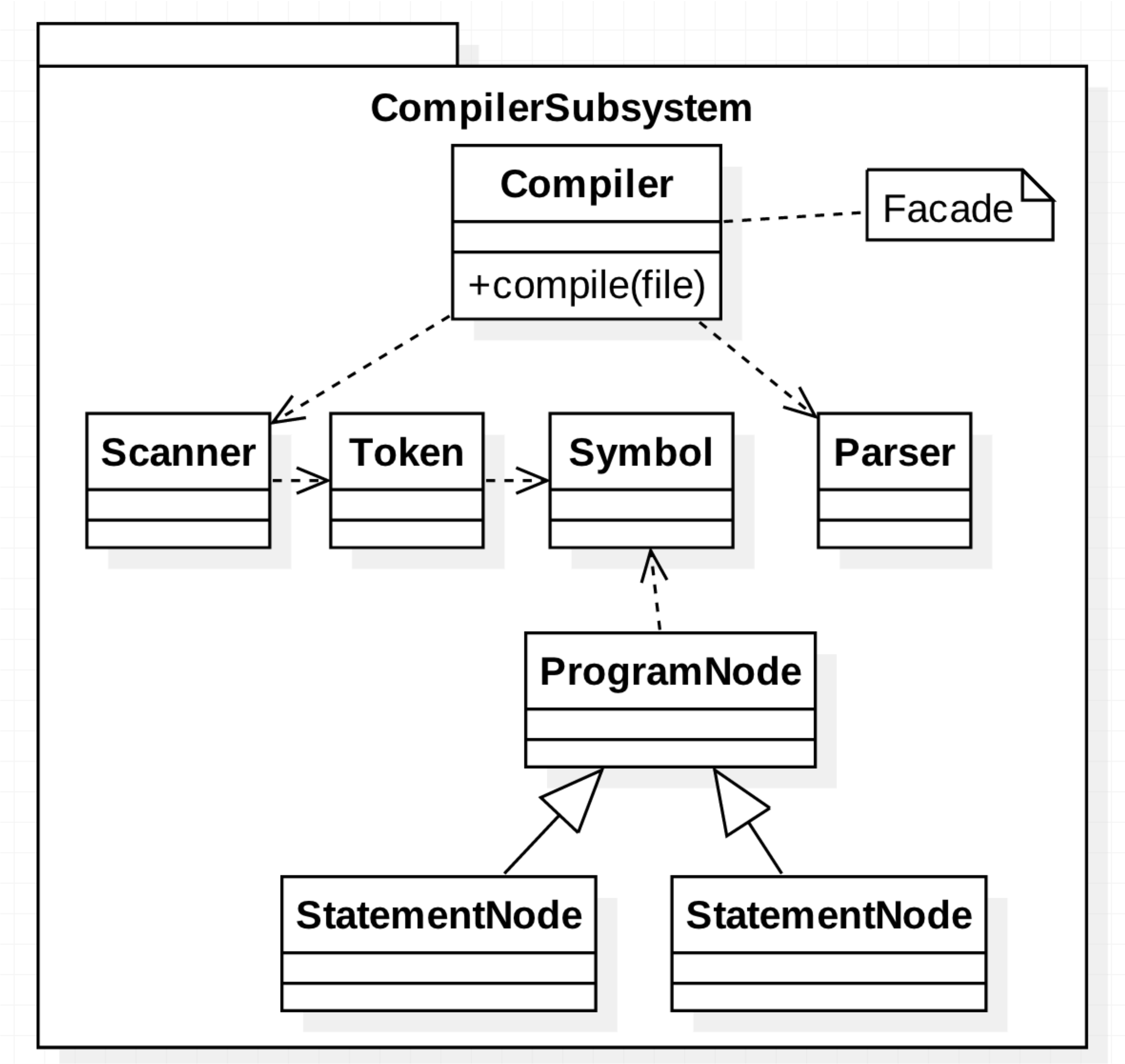
It's hard to restrict the types of a component (design is too general)

Facade

Intent: provide a unified interface to a set of interfaces in a subsystem. It defines a higher-level interface that makes the subsystem easier to use.

Motivation: Structuring a system into subsystems reduces complexity. Facade provides a single, simplified interface to the subsystem.

Facade


CompilerSubsystem
Compiler
+compile(file)
Facade
Scanner
Token
Symbol
Parser
ProgramNode
StatementNode
StatementNode

Pros & Cons

Pros:

Isolates clients from subsystem components

Minimizes the dependency of the client code on the subsystems

Cons:

The Facade class risks accumulating a lot of responsibility because it is linked to all the classes in the application

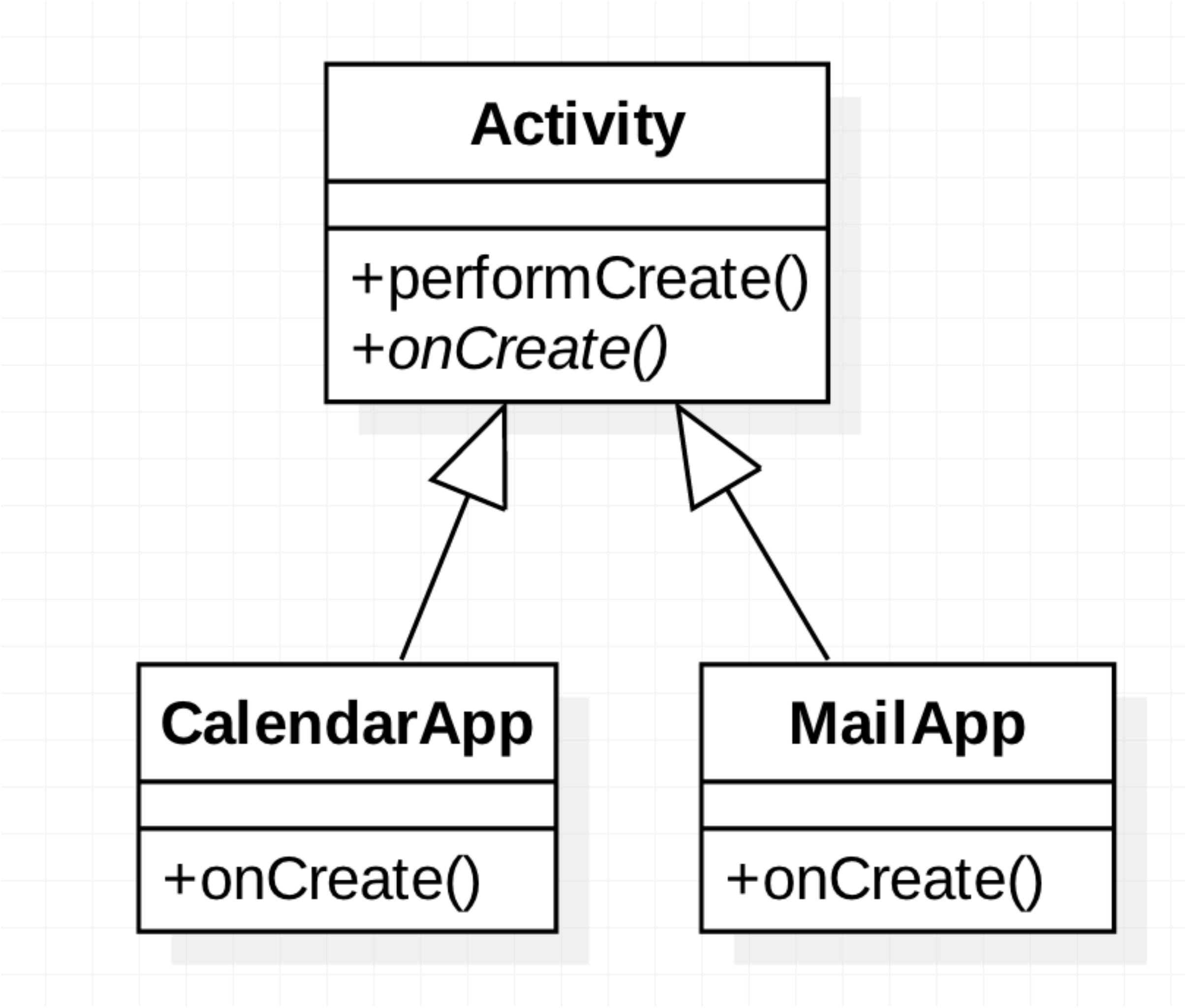
Behavioral Patterns

Template Method

Intent: define a skeleton of an algorithm and defer some steps to subclasses.

Motivation: The Android OS must support multiple types of app. These apps all have a common lifecycle and need to handled uniformly by the OS.

Template Method



Template Method

The base class provides the ***basic steps*** of the algorithm.

The subclasses provide the ***details.***

```java
public abstract class Activity {
    final void performCreate(Bundle icicle) {
        restoreHasCurrentPermissionRequest(icicle);
        onCreate(icicle);
        mActivityTransitionState.readState(icicle);
        performCreateCommon();
    }

    public abstract void onCreate(Bundle bundle);
}

public class MyApp extends Activity {
    @Override
    public void onCreate(Bundle bundle) {
        // app specific stuff goes here
    }
}
```

```java
public abstract class Activity {
    final void performCreate(Bundle icicle) {
        restoreHasCurrentPermissionRequest(icicle);
        onCreate(icicle);
        mActivityTransitionState.readState(icicle);
        performCreateCommon();
    }

    public abstract void onCreate(Bundle bundle);
}

public class MyApp e
    @Override
    public void onCreate(Bundle bundle) {
        // app specific stuff goes here
    }
}
```

(Part of) the algorithm for starting an `Activity`.

```java
public abstract class Activity {
    final void performCreate(B
        restoreHasCurrentPermi
        onCreate(icicle);
        mActivityTransitionSta
        performCreateCommon();
    }

    public abstract void onCreate(Bundle bundle);
}

public class MyApp extends Activity {
    @Override
    public void onCreate(Bundle bundle) {
        // app specific stuff goes here
    }
}
```

The specific details are handled by the subclasses.

Pros & Cons

Pros:

Helps eliminate code duplication

Easy to customize the algorithm

Cons:

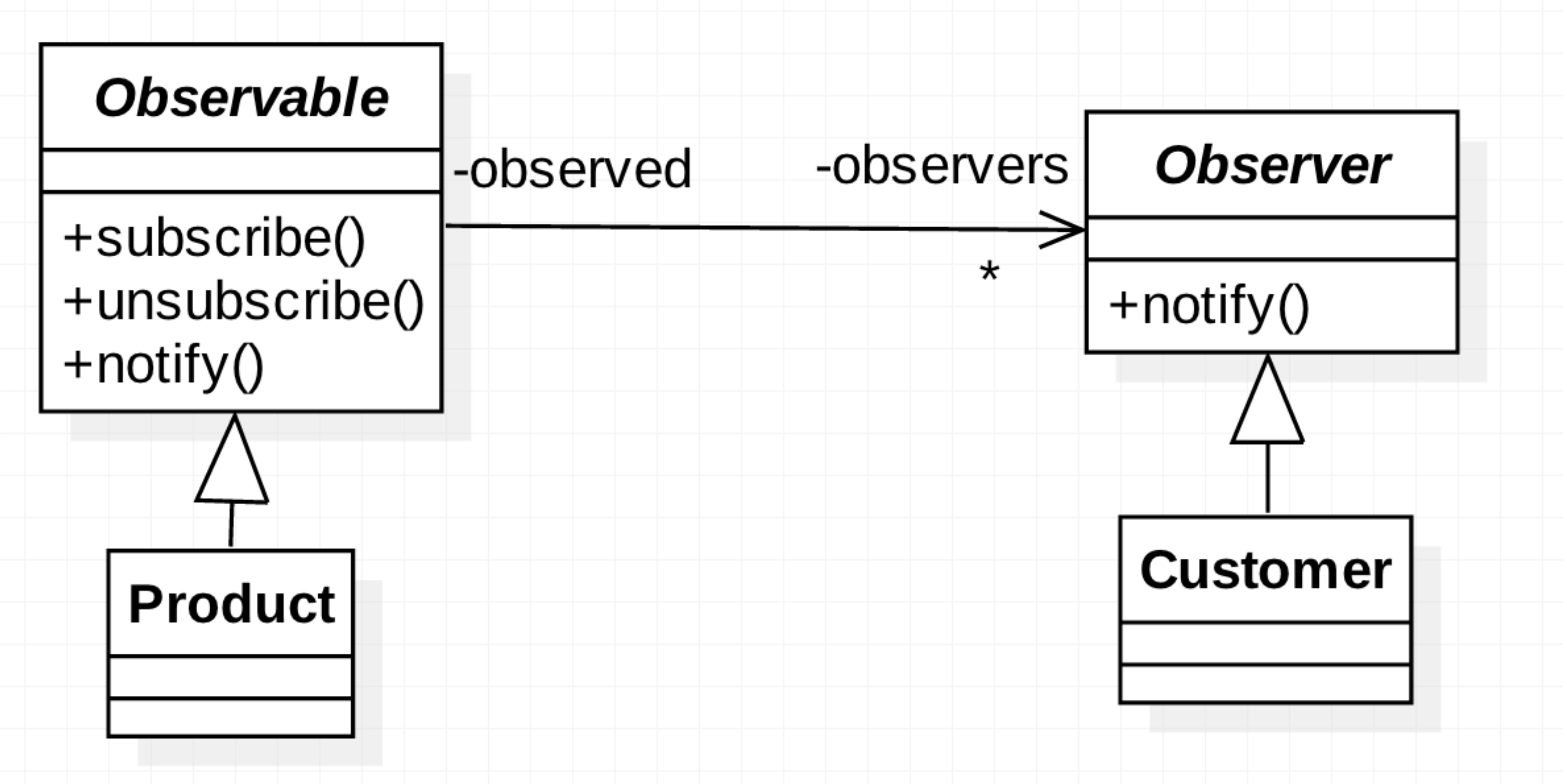
Your options are limited by the existing skeleton

Observer

Intent: Define a one-to-many relationship between objects, so that when one object changes its state, the dependents are notified and updated automatically.

Motivation: An online store is about to receive a large shipment of a high demand product. The store wants to notify the customers when the product is in stock.

Observer



Observable
+subscribe()
+unsubscribe()
+notify()
-observed
-observers
*
Observer
+notify()
Product
Customer

```java
public abstract class Observable {

    private List<Observer> observers = new ArrayList<>();

    public void subscribe(Observer o) {
        observers.add(o);
    }

    public void unsubscribe(Observer o) {
        observers.remove(o);
    }

    public void notify(Object data) {
        for(Observer o : observers) {
            o.notify(data);
        }
    }

}

public abstract class Observer {
    public abstract void notify(Object data);
}
```


Oregon State University

```java
public abstract class Observable {

    private List<Observer> observers = new ArrayList<>();

    public void subscribe(Observer o) {
        observers.add(o);
    }

    public void unsubscribe(Observer o) {
        observers.remove(o);
    }

    publi
        fo

        }
    }

}

public abs
    public abstract void notify(Object data);
}
```

The `Observable` keeps track of *observers* and provides methods to subscribe and unsubscribe

```
public abstract class Observable {

    private List<Observer> observers = new ArrayList<>();

    public void subscribe(Observer o) {
        observers.ad
    }

    public void unsu
        observers.re
    }

    public void notify(Object data) {
        for(Observer o : observers) {
            o.notify(data);
        }
    }

}

public abstract class Observer {
    public abstract void notify(Object data);
}
```

It also handles notifying the observers

```java
public class Product extends Observable{
    public void updateStock(int units) {
        // .....
        this.notify(units);
    }
}

public class Customer extends Observer {
    public void notify(Object data) {
        // react to the notification
    }
}
```

```java
public class Product extends Observable{
    public void updateStock(int units) {
        // .....
        this.notify(units);
    }
}

public class Customer ex
    public void notify((
        // react to the
    }
}
```

The client can notify all the observers of the change.

```
public class Product extends Observable{
    publi
        /
        t
    }
}
```

The observer can deal with the notification in any way it sees fit.

```
public class Customer extends Observer {
    public void notify(Object data) {
        // react to the notification
    }
}
```

Pros & Cons

Pros:

Observers are isolated from Observables

You can dynamically subscribe and unsubscribe

Cons:

The order in which Observers are called might not be deterministic.