# Subtyping and Inheritance

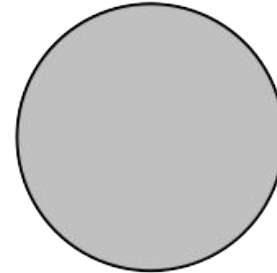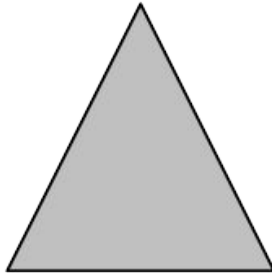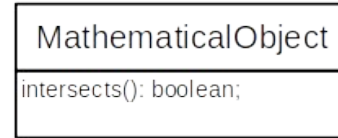Dirk Riehle, FAU Erlangen

## ADAP B03

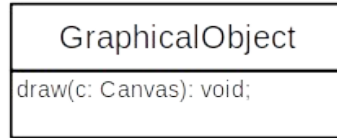# Agenda

1. What is subtyping?
2. Liskov substitutability principle
3. Applied to class hierarchies
4. Co- and contravariance
5. Multiple inheritance
6. Abstract superclass rule
7. Cascading class hierarchies

Homework

**2**

# 1. What is Subtyping?

# Subtyping Example 1 / 3

| GraphicalObject |
| --- |
| draw(c: Canvas): void; |

| MathematicalObject |
| --- |
| intersects(): boolean; |

**4**

# Subtyping Example 2 / 3

| GraphicalObject |
|---|
| draw(c: Canvas): void; |

| MathematicalObject |
|---|
| intersects(): boolean; |

DR

# Subtyping Example 3 / 3
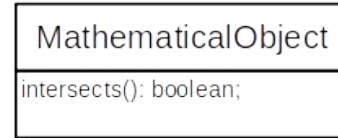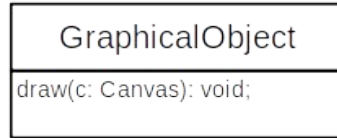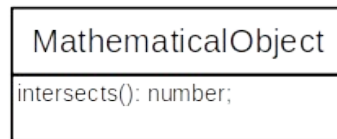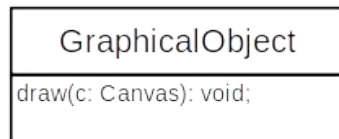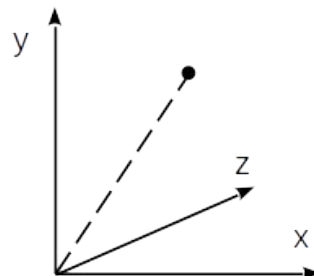
| GraphicalObject |
|---|
| draw(c: Canvas): void; |

| MathematicalObject |
|---|
| intersects(): number; |

# 2. Liskov Substitutability Principle

# The Subtype Requirement [1]

Let φ(x) be a property provable about objects x of type T. Then φ(y) should be provable for objects y of type S, where S is a subtype of T.

[1] A.k.a. the Liskov Substitutability Principle (LSP) [LW94]

# In Simpler Words

All properties that hold for instances of a supertype should also hold for instances of a subtype [DR]

# Even Simpler

Don't surprise use-clients

# Quiz: What's the Surprise?

If you make Rectangle a subtype of Square?

If you make Square a subtype of Rectangle?

If you make 2DLine a subtype of Point?

If you make Point a subtype of 2DLine?

# 3. Applied to Class Hierarchies

# Subclasses as Extended Subtypes

Subclasses

● Add methods and state
● Do not constraint superclasses

https://profriehle.com
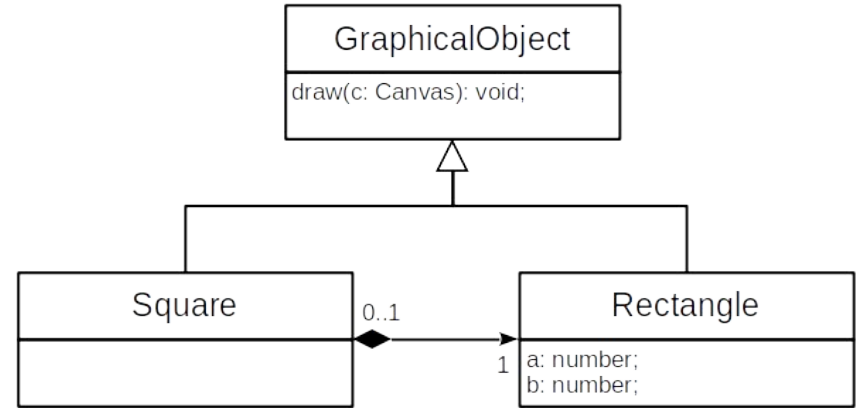
# Subclasses as Constrained Subtypes

Subclasses

● Constrain behavior in defined space

In method signatures

● Using covariant redefinition

Leads to dual hierarchies
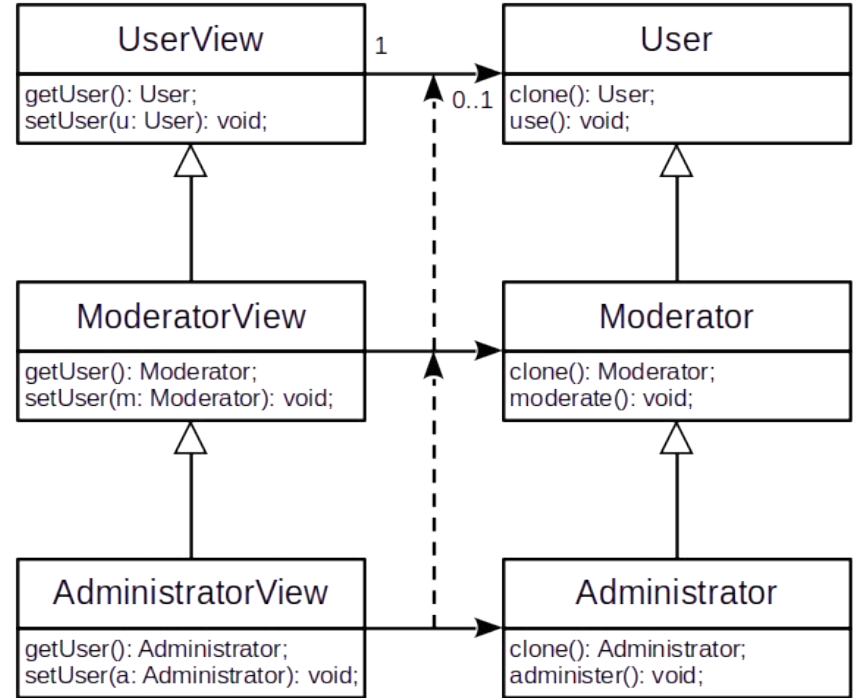
# Extract Superclass Refactoring

A refactoring is a

- Behavior-preserving transformation of existing code

The goal is to improve readability, remove redundancy, etc.

The extraction of an abstract superclass is a common refactoring

Fowler's catalog [1] lists Extract Superclass (without "Abstract" though)

[1] Martin Fowler. Refactoring (2nd Edition). Addison Wesley, 2018.

# 4. Co- and Contravariance

# Covariant Redefinition of Return Types

A return type has been covariantly redefined in a method definition, if

● The return type of the subtype's method is a subtype of the supertype's

Example of covariant redefinition of return type

● UserView.getUser(): **User** → ModeratorView.getUser(): **Moderator**

The subtype's method "returns less" than what the supertype's method promises

● Does not violate the LSP (is within expectations)

**17**

# Contravariant Redefinition of Return Types

A return type has been contravariantly redefined in a method definition, if

- The return type of the subtype's method is a supertype of the supertype's

Example of contrvariant redefinition of return type

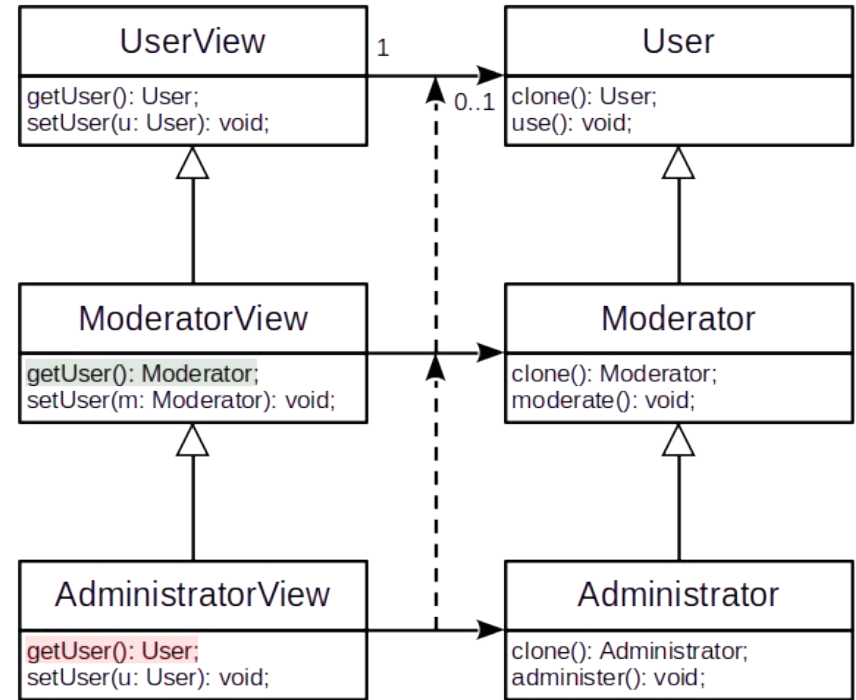- ModeratorView.getUser(): **Moderator** → AdministratorView.getUser(): **User**

The subtype's method "returns more" than what the supertype's method promises

- Violate the LSP, because clients of the supertype's methods might be surprised

**18**

# Users / Views Example 1 / 2

```
let modView1: ModeratorView =
      new ModeratorView(new Moderator());
let modAsUserView1: UserView =
      modView1 as UserView;
let mod1: Moderator =
      modAsUserView1.getUser() as Moderator;
mod1.moderate(); // should work, no problem


let adminView: AdministratorView =
      new AdministratorView();
let adminViewAsModView1: ModeratorView =
      adminView as ModeratorView;
let mod3: Moderator =
      adminViewAsModView1.getUser();
mod3.moderate() // will fail because mod3 is of dynamic type User
```

| UserView | 1 | User |
|---|---|---|
| getUser(): User; <br> setUser(u: User): void; | 0..1 | clone(): User; <br> use(): void; |

| ModeratorView | | Moderator |
|---|---|---|
| getUser(): Moderator; <br> setUser(m: Moderator): void; | | clone(): Moderator; <br> moderate(): void; |

| AdministratorView | | Administrator |
|---|---|---|
| getUser(): User; <br> setUser(u: User): void; | | clone(): Administrator; <br> administer(): void; |

**19**

# **Covariant Redefinition of Argument Types**

An argument type has been covariantly redefined in a method definition, if

- The argument type of subtype's method is a subtype of the supertype's

Example of covariant redefinition of argument type

- User.setUser(u: **User**): void → Moderator.setUser(m: **Moderator**): void

The subtype's method "accepts less" than what the supertype's method promises

- This violates the LSP and only makes sense if you think in relationships

# Contravariant Redefinition of Argument Types

An argument type has been contravariantly redefined in a method definition, if

- The argument type of the subtype's method is a supertype of the supertype's

Example of contravariant redefinition of argument type

- ModeratorView.getUser(): **Moderator** → AdministratorView.getUser(): **User**

The subtype's method "accepts more" than what the supertype's method promises

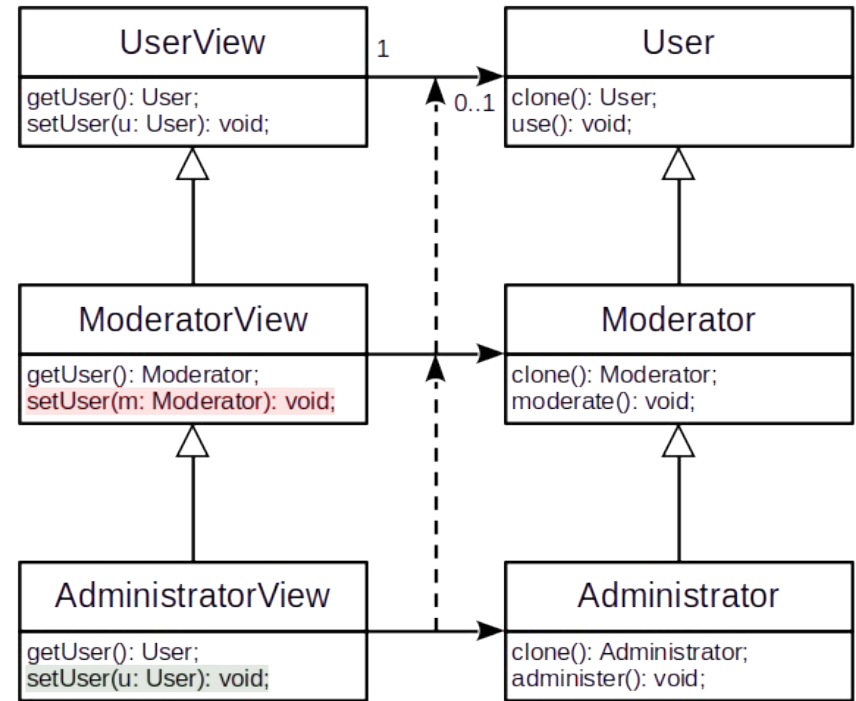- Does not violate the LSP but also makes little sense in practice

# Users / Views Argument Type Example

```
let modView2: ModeratorView =
    new ModeratorView(new Moderator());
let modAsUserView2 =
    modView2 as UserView;
modAsUserView2.setUser(new User()); // setup
let mod2: Moderator =
    modView2.getUser(); // creates failure p
mod2.moderate(); // should fail
```

```
adminView.setUser(new User());
let user1: User = adminView.getUser();
user1.use(); // no problem
let admin1: Administrator = user1 as Administrator;
admin1.administer(); // will fail but also was not promised
```



**UserView**
getUser(): User;
setUser(u: User): void;

**User**
clone(): User;
use(): void;

1    0..1

**ModeratorView**
getUser(): Moderator;
setUser(m: Moderator): void;

**Moderator**
clone(): Moderator;
moderate(): void;

**AdministratorView**
getUser(): User;
setUser(u: User): void;

**Administrator**
clone(): Administrator;
administer(): void;

DR

**22**

# Co- and Contravariance in Typescript

| | Covariant Redefinition | Contravariant Redefinition |
|---|---|---|
| **Return type** | ● is allowed | ● is not allowed |
| **Argument type** | ● is allowed [2]<br>● should not be allowed [1] | ● is allowed |

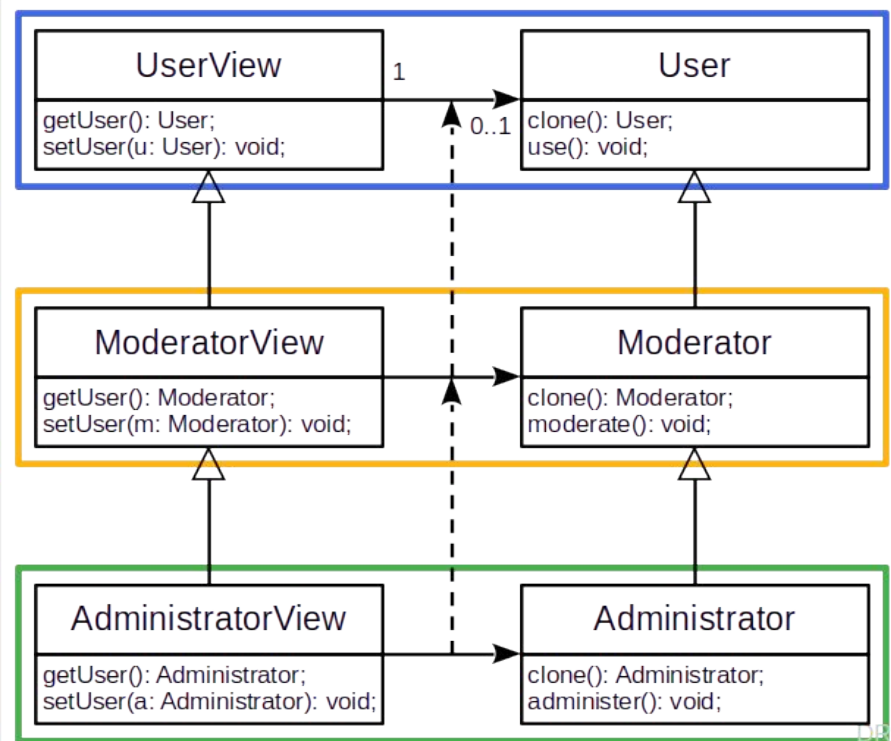[1] Should not be allowed because it violates the LSP
[2] Only makes sense if class (role type) is part of a collaboration

**23**

# Parallel Class Hierarchies

Parallel class hierarchies are

- Two related class hierarchies, subclassed in parallel
- Often using covariant redefinition of both return and argument types

The design focus is on the collaboration
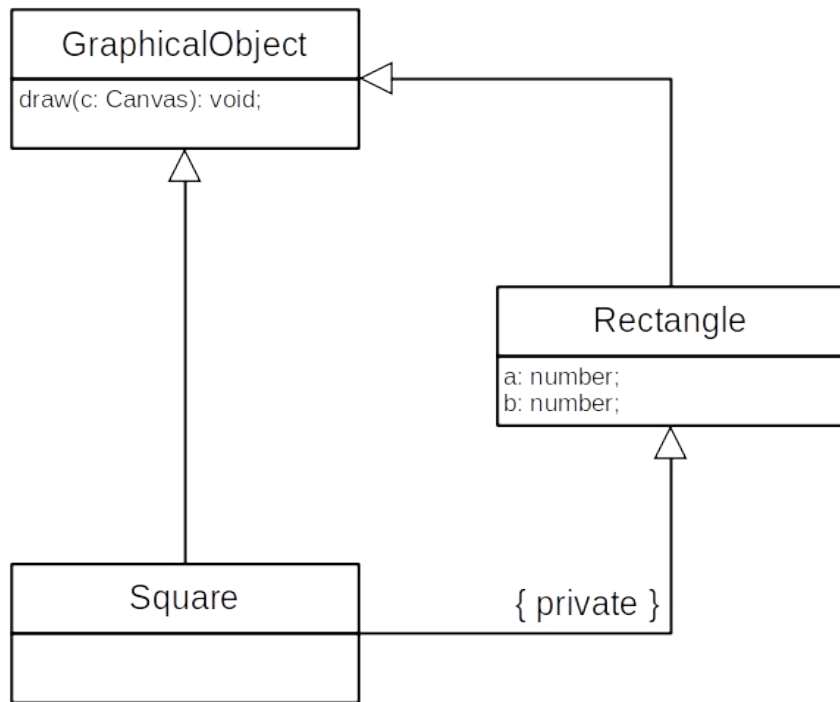
# 5. Multiple Inheritance

# Multiple Inheritance

Multiple inheritance is when

- A class has 2+ superclasses

Does not necessarily imply substitutability

- Cf. C++'s private inheritance

Not a Typescript feature
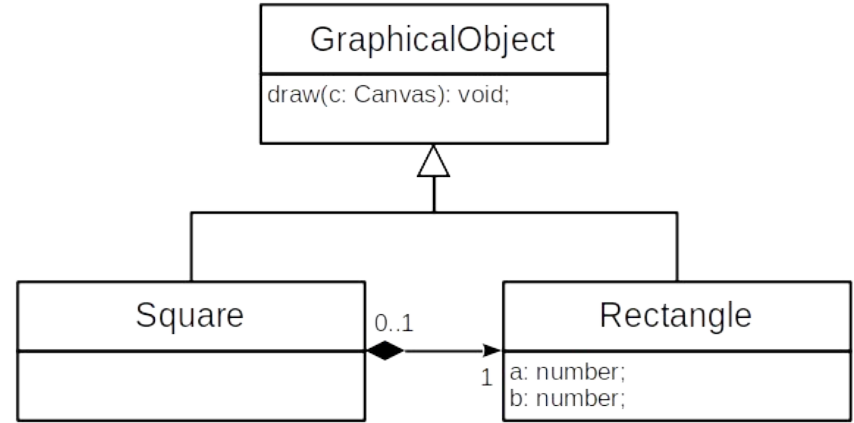


DR

https://profriehle.com

# Implementation Delegation

Implementation delegation is when

- A class delegates its implementation

Generally better than multiple inheritance

- Choose delegation over inheritance

# Composition over Inheritance

The composition over inheritance principle states that

- You should favor object composition over class inheritance

A.k.a. delegation over inheritance (principle)

# 6. Abstract Superclass Rule

# Inheritance vs. Abstractness

Inheritance is

- A relationship between two classes

Abstractness / concreteness

- A relationship between a class and its instances

# Abstract Superclass Rule (ASR)

All superclasses must be abstract

Corollary: Never subclass a concrete class

# ASR in Framework vs. Application

In a framework

- Leaf classes may be abstract (awaiting subclassing)
- Leaf classes may be concrete (if ready to use)

In an application (based on a framework)

- Framework leaf classes may be abstract if unused
- Application leaf classes must be concrete

**32**

# ASR and LSP

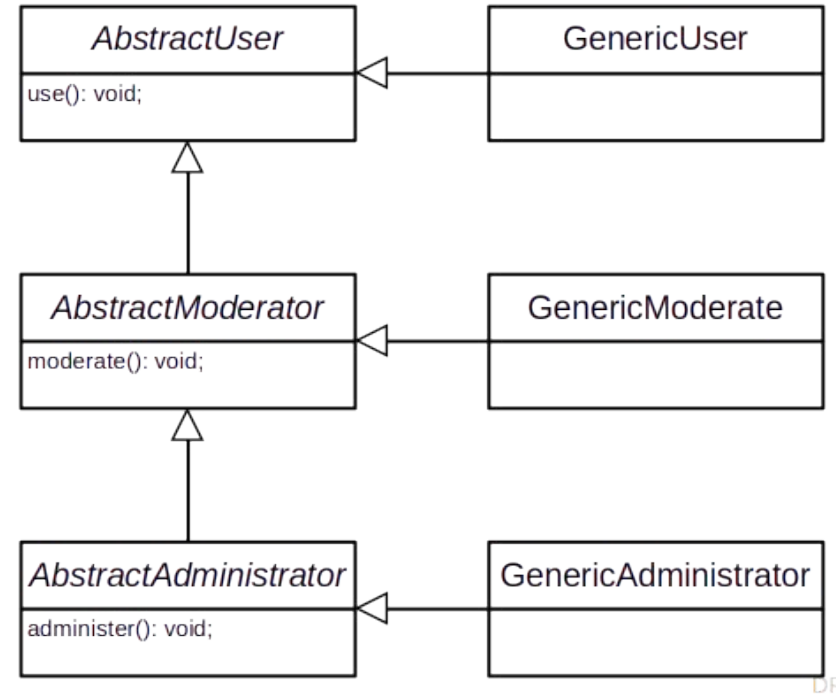The ASR helps to comply with the LSP

- The ASR automatically casts subclasses as constrained subtypes
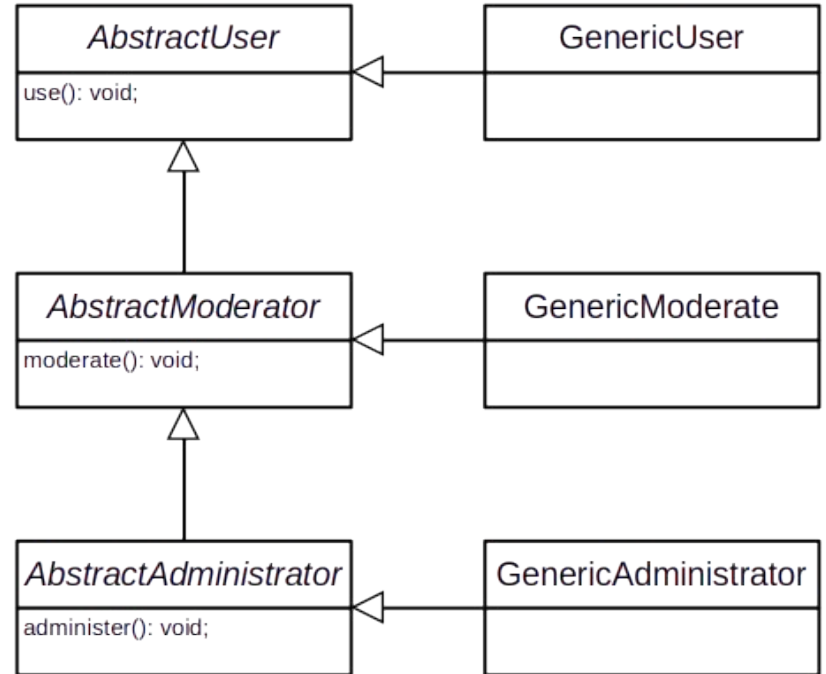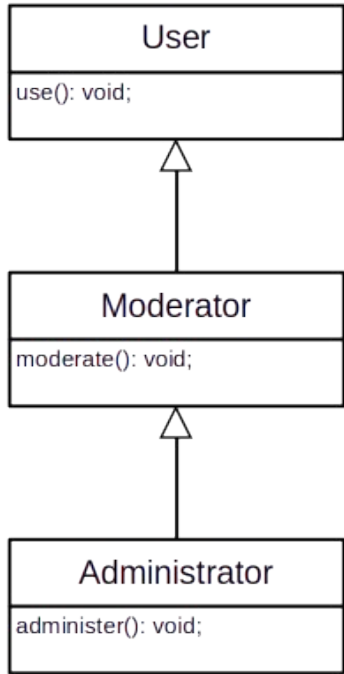
# Pragmatics of ASR

Logically separate abstract class from generic implementation subclass

Pragmatically, merge implementation class into abstract class

Make abstract class concrete but maintain inheritance interface



**34**

# Class Hierarchy Evolution
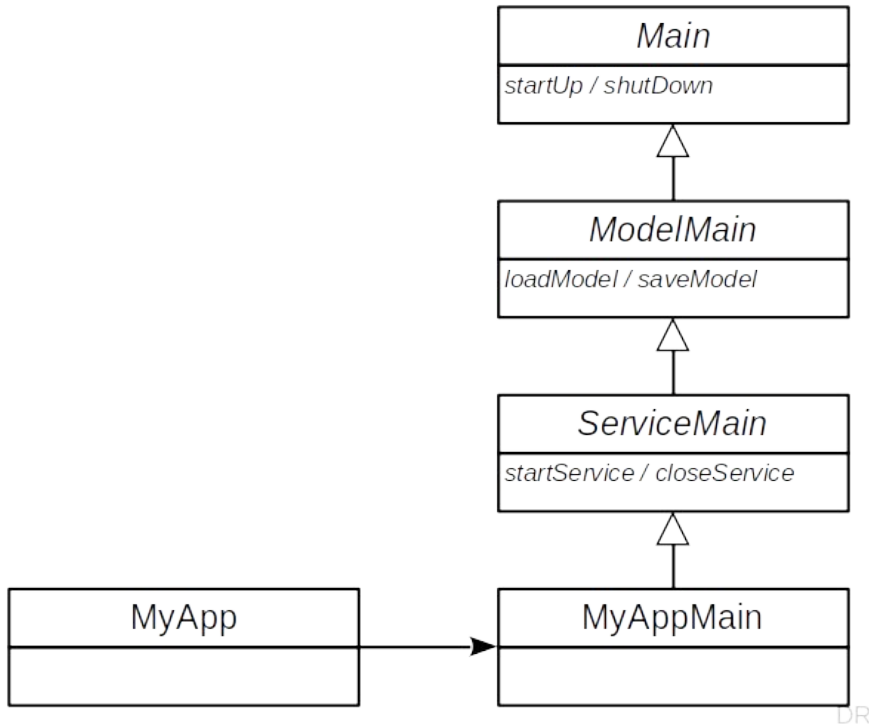
# 7. Cascading Class Hierarchies

# Before and After Methods

Before and after methods wrap a method's main body

They typically come in pairs and are about a meta issue

- The before method sets something up
- The after method tears it down

# App with Service Example



```
import { MyAppMain } from "./MyAppMain";

function main(args: string[]) {
    let appMain: MyAppMain = new MyAppMain();
    appMain.run(args);
}

let args: string[] = process.argv;
args = args.slice(2);
main(args);
```

**38**

# Cascading Inheritance Interfaces 1 / 2

```typescript
export abstract class Main {

    public run(args: string[]): void {
        this.parseArgs(args);
        this.startUp();
        this.execute();
        this.shutDown();
    };

    protected parseArgs(args: string[]): void {
        // do nothing (expect subclass to override)
    }

    protected startUp(): void {
        // do nothing (expect subclass to override)
    }

    protected abstract execute(): void;

    protected shutDown(): void {
        // do nothing (expect subclass to override)
    }

}
```

```typescript
import { Main } from "./Main";

export abstract class ModelMain extends Main {

    protected startUp(): void {
        super.startUp();
        this.loadModel();
    }

    protected loadModel(): void {
        // do nothing (expect subclass to override)
    }

    protected shutDown(): void {
        this.saveModel();
        super.shutDown();
    }

    protected saveModel(): void {
        // do nothing (expect subclass to override)
    }

}
```

**39**

# Cascading Inheritance Interfaces 2 / 2

```typescript
import { ModelMain } from "./ModelMain";

export abstract class ServiceMain extends ModelMain {

    protected startUp(): void {
        super.startUp();
        this.startService();
    }

    protected startService(): void { /* ... */ }

    protected execute(): void {
        // start main event loop
    }

    protected shutDown(): void {
        this.closeService();
        super.shutDown();
    }

    protected closeService(): void { /* ... */ }

}
```

```typescript
import { ServiceMain } from "./ServiceMain";

export class MyAppMain extends ServiceMain {

    protected loadModel(): void {
        // do something
    }

    protected startService(): void {
        // do something
    }

    protected saveModel(): void {
        // do something
    }

    protected closeService(): void {
        // do something
    }

}
```

**40**

# Homework

# Homework Instructions

- Extract AbstractName superclass from StringName and StringArrayName
  - Identify and implement the narrow (minimal) inheritance interface
  - Move as much as you sensibly can into the AbstractName class

- Adapt your previous work to this homework as you see fit
- Commit homework by deadline to homework folder

# Summary

1. What is subtyping?
2. Liskov substitutability principle
3. Applied to class hierarchies
4. Co- and contravariance
5. Multiple inheritance
6. Abstract superclass rule
7. Cascading class hierarchies

**43**

# Thank you! Any questions?

dirk.riehle@fau.de – https://oss.cs.fau.de

dirk@riehle.org – https://dirkriehle.com – @dirkriehle

# Legal Notices

License

- Licensed under the [CC BY 4.0 International](#) license

Copyright

- © 2012, 2018, 2024 Dirk Riehle, some rights reserved