

# **Class and Interface Design**



Dirk Riehle, FAU Erlangen

**ADAP B02**

Licensed under [CC BY 4.0 International](https://creativecommons.org/licenses/by/4.0/)

# Agenda

- 
1. Classes vs. interfaces
  2. Abstract state model
  3. Program to an interface
  4. Design by primitives
  5. Simple class design
  6. Inheritance interface
  7. Class design evolution

# 1. Classes vs. Interfaces



# Objects and Classes



The modeling perspective (historically: Simula)

- An object is the representation of a phenomenon from a domain
- A class is a description of the commonalities of similar objects

The technology perspective (historically: Smalltalk)

- An object is an encapsulation of some program state
- A class is the implementation of how to change that state

# Classes and Interfaces



An interface is

- The abstract description of some object behavior

An abstract class is

- A partial implementation of an interface's behavior

An concrete class is

- A complete implementation of an interface's behavior

# (Use-Client) Interfaces



Have an abstract state model and state transitions

Are to be implemented by abstract and/or concrete classes

# Abstract Classes



Set up algorithmic scaffolding for concrete subclasses

Are used to implement an interface, to be extended by subclasses

# Concrete Classes



Include implementation state (fields)

Directly implement an interface or extend an abstract class

When paired with an interface, are also known as implementation classes

# In Typescript



A Typescript interface is a (use-client) interface

Any Typescript class has an (implicit) interface

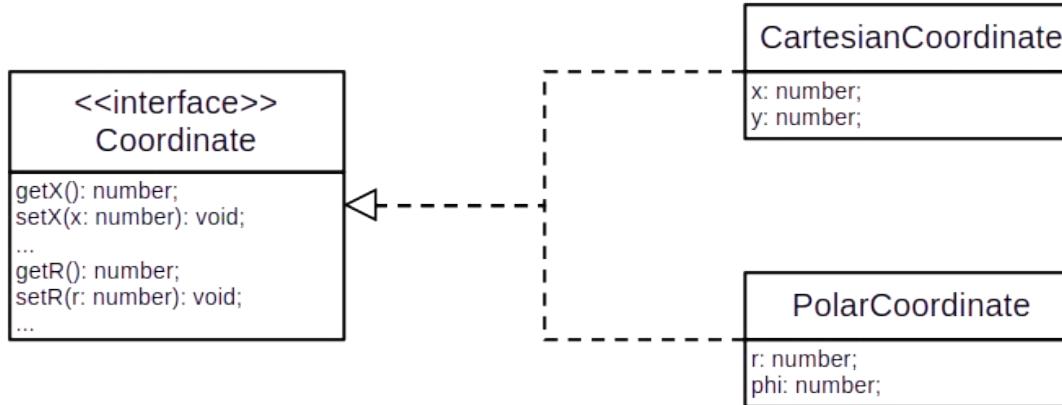
- The interface is conceptually separate from the implementation
- In other languages, e.g. C, interface and implementation are split in files

A Typescript class can implement a Typescript interface

A Typescript class can be an abstract class

- The marked as abstract in its definition

# Class Model of Coordinates Example



DR

10

## **2. Abstract State Model**



# Abstract State Model

An abstract state model is

- A model of the valid state space of objects conforming to the model

An interface expresses an abstract state model

- An abstract state model does not map 1:1 on implementation fields

The abstract state model can be expressed using get methods

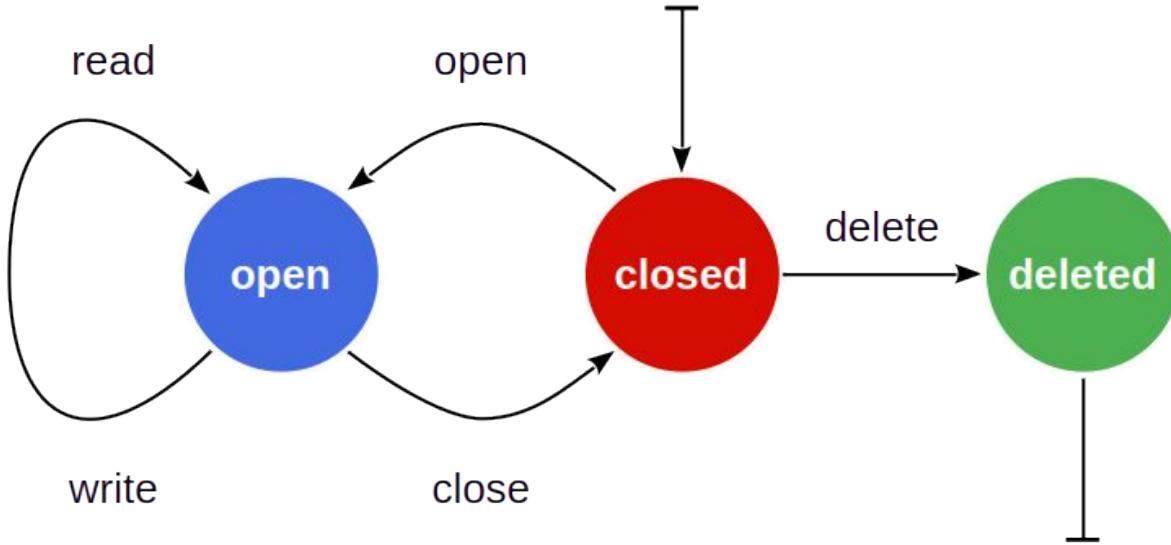
- Alternatively, it can be expressed as finite state machines

# State Model of Coordinate Example

```
import { Equality } from "./Equality";
import { Cloneable } from "./Cloneable";

export interface Coordinate extends Equality, Cloneable {
    ...
    getX(): number;
    getY(): number;
    ...
    getR(): number;
    getPhi(): number;
    ...
}
```

# State Model of File Example



DR

14

# File as Class (+ Interface)

```
export class File {  
    public isOpen(): boolean { ... }  
    public isClosed(): boolean { ... }  
  
    public read(): any[] {  
        this.assertIsOpenFile();  
        ...  
    }  
  
    public delete(): void {  
        this.assertIsClosedFile();  
        ...  
    }  
  
    protected assertIsOpenFile(): void { ... }  
    protected assertIsClosedFile(): void { ... }  
    ...  
}
```

# File Interface and ObjFile Implementation

```
export interface File {  
  
    isEmpty(): boolean;  
    isOpen(): boolean;  
    isClosed(): boolean;  
  
    read(): any[];  
    write(data: any[]): void;  
    delete(): void;  
  
}
```

```
export class ObjFile implements File {  
  
    protected data: Object[] = [];  
    protected length: number = 0;  
  
    public isEmpty(): boolean {  
        return this.length == 0;  
    }  
  
    public read(): Object[] { ... }  
    public write(data: Object[]): void { ... }  
    public delete(): void { ... }  
  
    ...  
}
```

### **3. Program to an Interface**



# The Program to an Interface Principle



Program to an interface, not an implementation

# How to Program to an Interface



Program to the abstract state model

- Do not program to the implementation state

Do not rely on what is not in the interface

- Do not expect implementation-generated side-effects
- Do not rely on specific performance unless guaranteed

# Class Implementation



A concrete class (a.k.a. implementation class)

1. Has an interface that combines all interfaces it implements
2. Is a complete implementation of that interface
3. Implements an abstract state model

The implementation is by way of implementation state (fields)

## 4. Design by Primitives



# Types of Interfaces



Use-client interfaces

Inheritance interfaces

# Design by Primitives



Design by primitives is a programming principle in which

- The implementation state of a class is encapsulated by primitive methods
- All other methods should utilize these primitive methods

Do not change any fields outside the primitive methods

# Use of Primitive Methods

```
public getX(): number {
    return this.doGetX();
}

protected doGetX(): number {
    return this.x;
}

public getR(): number {
    return Math.hypot(this.doGetX(), this.doGetY());
}

public setR(r: number): void {
    let phi: number = Math.atan2(this.doGetY(), this.doGetX());
    this.doSetX(r * Math.cos(phi));
    this.doSetY(r * Math.sin(phi));
}
```

# The Purpose of Design by Primitive



The encapsulation of implementation field access creates an indirection level that

- Creates flexibility (subclasses can override)
- Makes design by contract more effective

# 5. Simple Class Design



# How to Structure a Class Interface

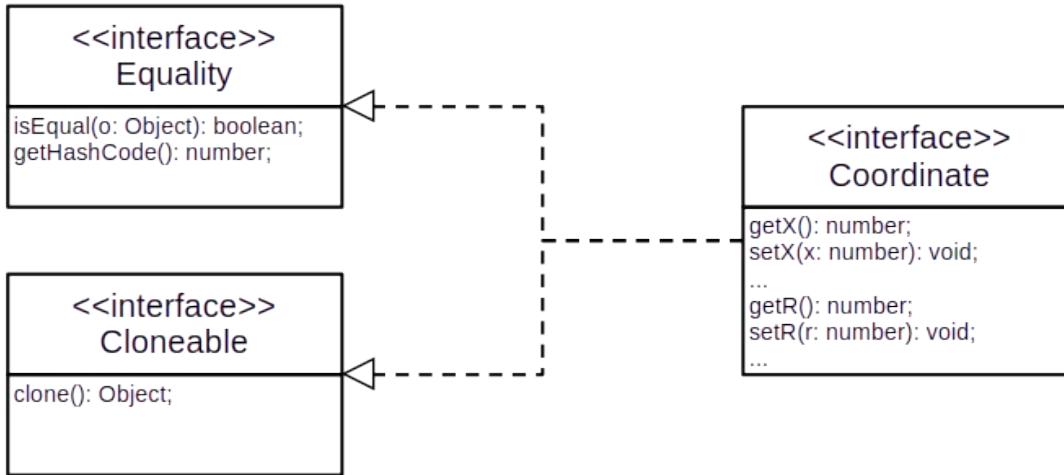


Group methods by trait or collaboration

- Get and set belong together, not getters and setters
- Methods of the same collaboration belong together
- Special purpose methods belong together

Break out reusable collaborations as interfaces

# Multiple Interfaces



DR

# Mixin Classes [1]



A mixin class is a (typically small) class that

- Provides reusable code focused on one aspect / trait / role of interest

A regular class then “mixes in” or inherits from one or more mixin classes to

- Compose the reusable code into the (larger) class

# How to Structure a Class Implementation



Group fields together, across traits or collaborations

- At top of file (most common) or bottom (less common)
- Because implementation state works across collaborations

# Tagging Interface



A tagging interface is

- An interface that serves as a marker to the compiler to add “some magic”
- Typically void of any methods
- A.k.a. marker interface

In Java, example tagging interfaces are Cloneable and Serializable

## 6. Inheritance Interface

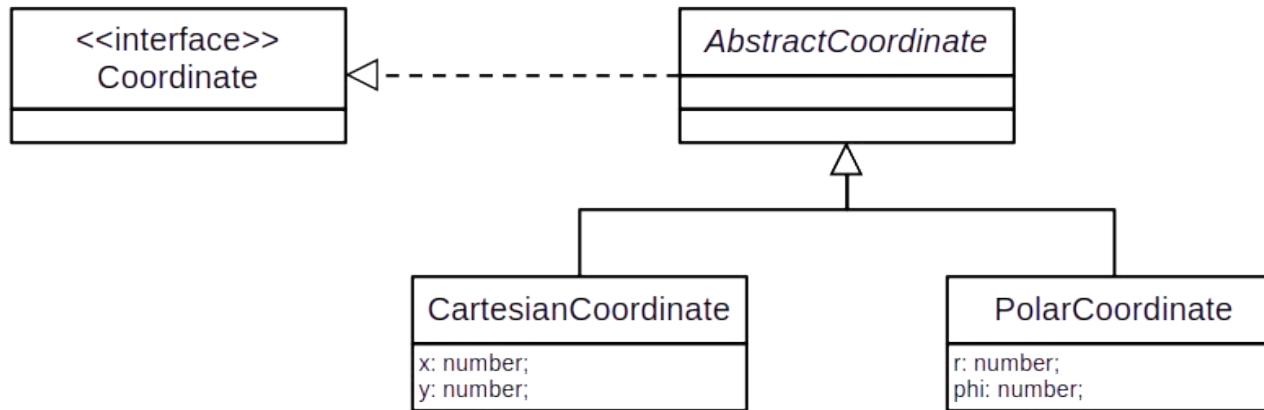


# Inheritance Interface

An inheritance interface is

- An internal abstract interface defined by a superclass for its own use
- To be implemented by subclasses to complete the implementation

# Inheritance Interface in the Coordinate Example



DR

# Design of Inheritance Interfaces



## The inheritance interface

- Typically consists of primitive methods
- Is protected (only for class internal use)

# Inheritance Interface of AbstractCoordinate

```
export abstract class AbstractCoordinate implements Coordinate {  
    ...  
  
    protected abstract doGetX(): number;  
    protected abstract doSetX(x: number): void;  
    protected abstract doGetY(): number;  
    protected abstract doSetY(y: number): void;  
    protected abstract doGetR(): number;  
    protected abstract doSetR(r: number): void;  
    protected abstract doGetPhi(): number;  
    protected abstract doSetPhi(phi: number): void;  
}
```

# Narrow Inheritance Interface Principle



Inheritance interfaces should be as small (narrow) as possible

Small = minimal number of methods needed to complete the implementation

This allows for expedient implementation of subclasses

Subclasses can still override more methods for more efficient implementations

# Thinking the Class Hierarchy Bottom-up



Subclasses call methods of superclasses to use its functionality

This may be any kind of method including helper methods

# Thinking the Class Hierarchy Top-down



Superclasses implement algorithms (for example, as template methods)

Superclasses call methods of subclasses through the inheritance interface

This reuses algorithms while allowing for behavior variation of the algorithms

# The Open / Closed Principle



A class should be open for extension and closed for modification

Open for extension through a well-defined inheritance interface

Closed for modification by not violating the use-client interface

## 7. Class Design Evolution

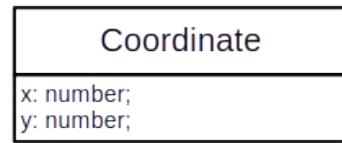


# Typical Class and Interface Evolution



1. Simple class
2. Interface extraction
3. Addition of implementation classes
4. Introduction of reusable abstract superclass

# 1. Simple Class



DR

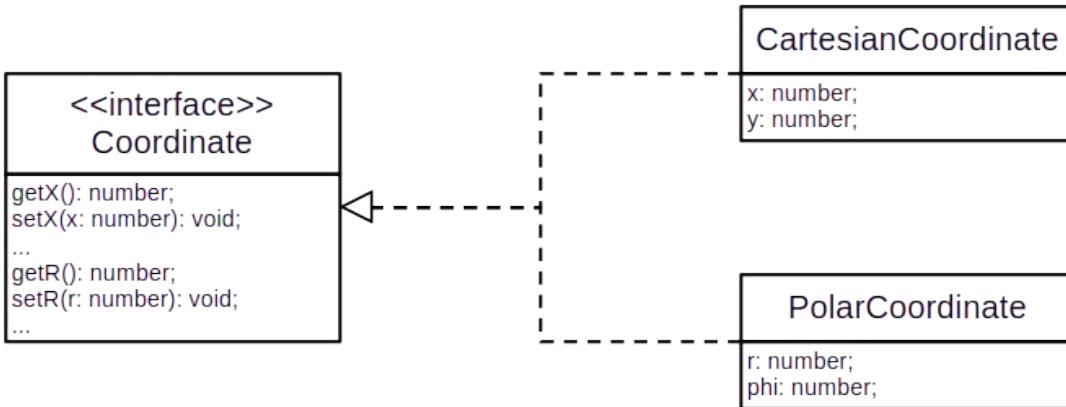
43

## 2. Interface Extraction



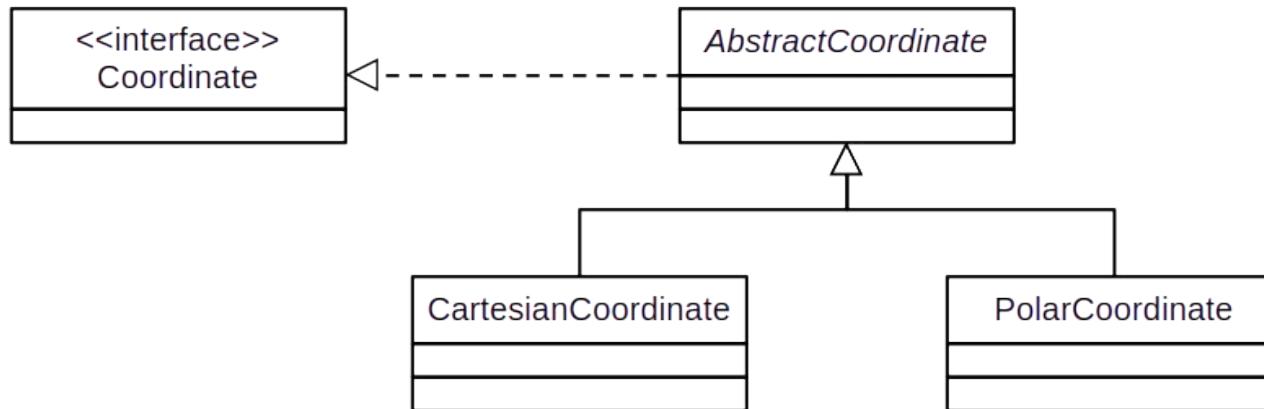
DR

### 3. Addition of Implementation Classes



DR

# 4. Introduction of Abstract Superclass



DR

46

# Naming Conventions of Classes [1]

## Recommended

- Interfaces
  - Coordinate
- Abstract classes
  - AbstractCoordinate
- Concrete classes
  - DefaultCoordinate
  - GenericCoordinate
  - CartesianCoordinate
  - PolarCoordinate

## Discouraged

- Interfaces
  - ICoordinate
- Abstract classes
  - CoordinateImpl
- Concrete classes
  - CartesianCoordinateImpl
  - ...

# How to Name Classes (English Grammar)

It is best to go with the flow of the (English) language chosen by your project

- General rule: Adjective sequence + class name
- Ordering of adjective: By how strongly they bind
  - Opinion, size, age, shape, color, origin, material, purpose

## Examples

- FunnyLargeRipeStraightYellowJamaicanSoftEdibleBanana
- LargeEuropeanWoodenRockingChair

# Summary



1. Classes vs. interfaces
2. Abstract state model
3. Program to an interface
4. Design by primitives
5. Simple class design
6. Inheritance interface
7. Class design evolution

# Thank you! Any questions?



[dirk.riehle@fau.de](mailto:dirk.riehle@fau.de) – <https://oss.cs.fau.de>

[dirk@riehle.org](mailto:dirk@riehle.org) – <https://dirkriehle.com> – [@dirkriehle](#)

# Legal Notices



## License

- Licensed under the [CC BY 4.0 International](https://creativecommons.org/licenses/by/4.0/) license

## Copyright

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