Design by Contract

Dirk Riehle, FAU Erlangen

ADAP B04

Agenda

1. Design by contract
2. Expressing contracts
3. Implementing contracts
4. Contract violations
5. Contract pragmatics

Homework

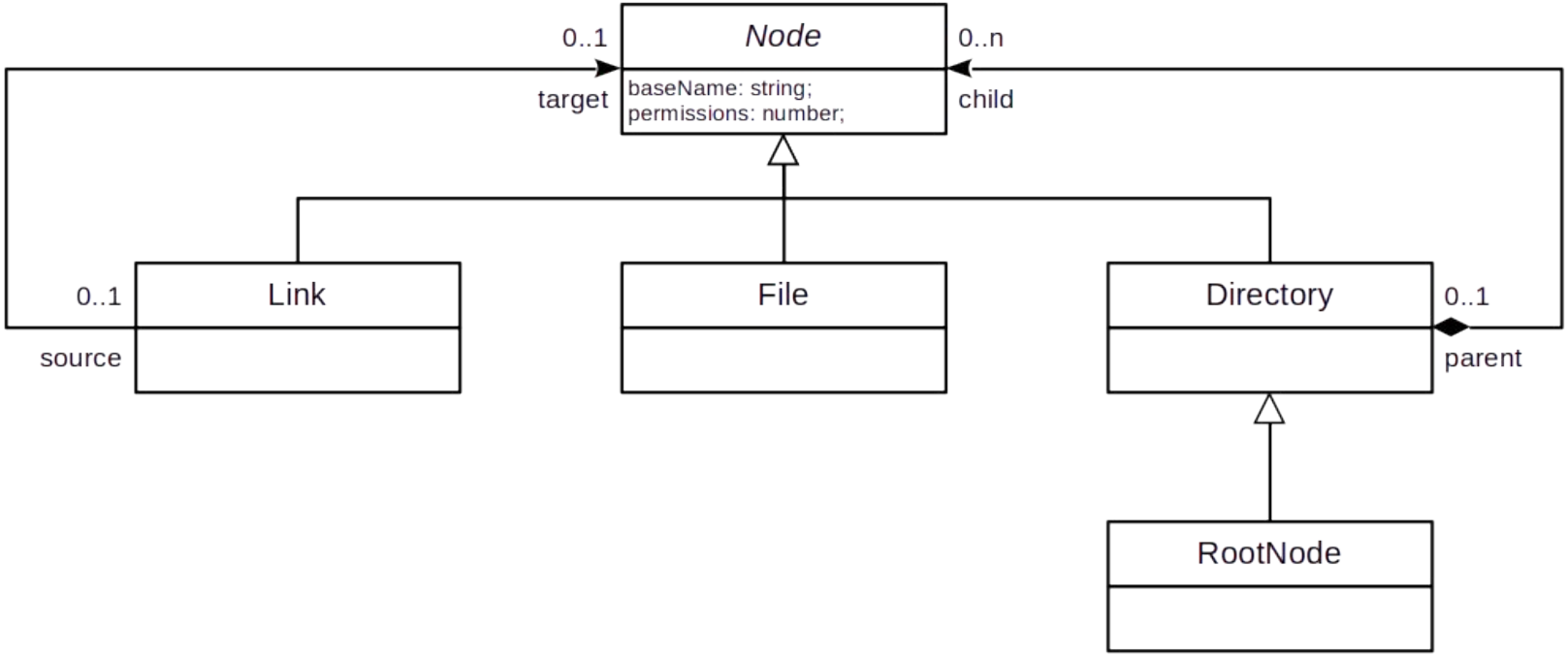
Did You Previously Learn about Design by Contract?

File Terminology

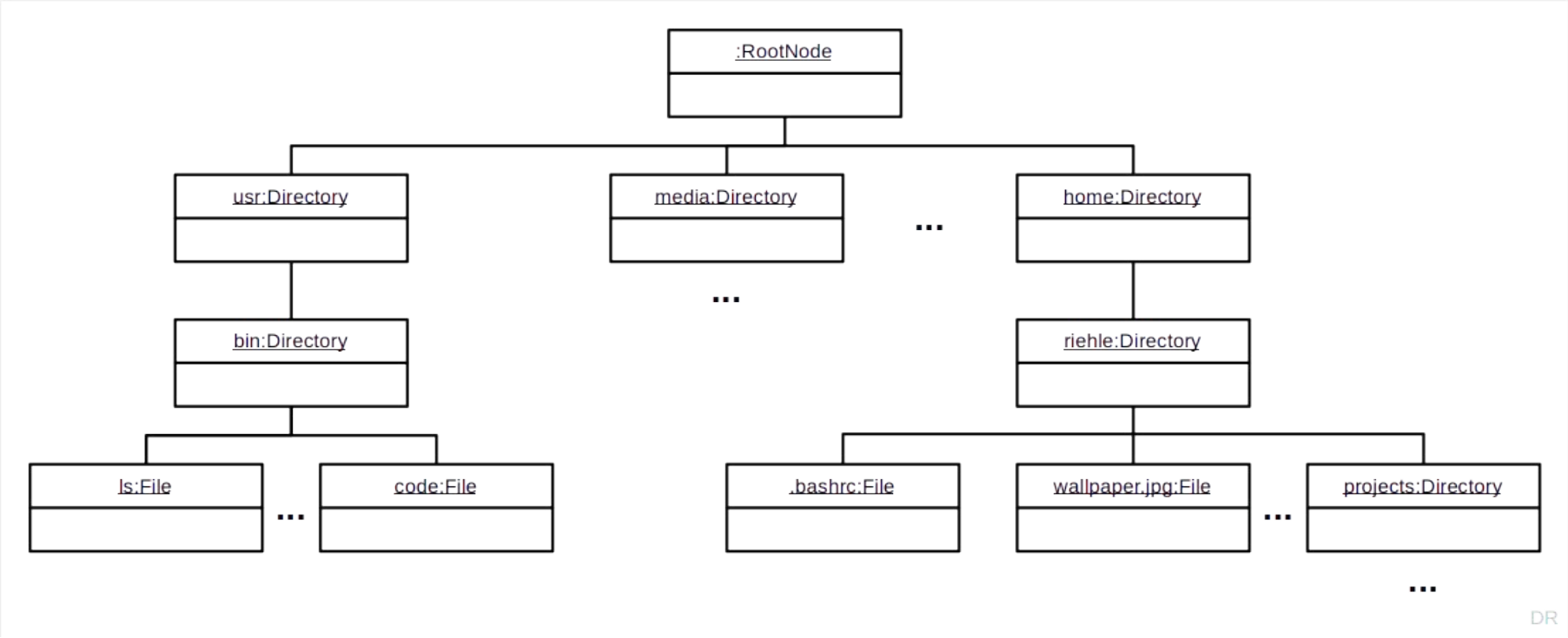
Given the file “/usr/bin/ls”

- “ls” is called the **base name**
- “/usr/bin” is called the **dir(ectory) name**
- “/usr/bin/ls” is called the **full name**

Extended Example



Quiz: Find “/usr/bin/ls”



1. Design by Contract

Design by Contract [M91]

Design by contract views

- Software design as a succession of contracting decisions

[M91] Meyer, B. (1991). Design by Contract. Chapter 1 in Mandrioli, D., Meyer, B. (1991). Advances in Object-oriented Software Engineering. Prentice-Hall.

Contracts

A contract specifies rights (benefits) and obligations

- Between a client (consumer) and contractor (supplier)
- Contracts are (ideally) exhaustive; there are no hidden clauses

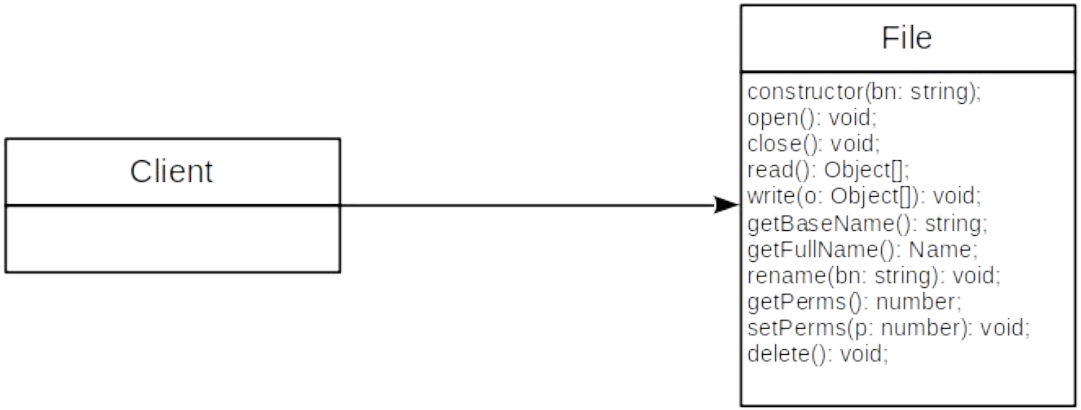
Rights and obligations are mutual

- A client obligation (precondition) is contractor's right
- A contractor obligation (postcondition) is a client's right

A contract protects both sides of the deal

- The client is guaranteed a result
- The contractor is guaranteed a specified operating environment

Client-File Contract 1 / 2



Client-File Contract 2 / 2

	Rights	Obligations
Client	See contractor obligations	• Provide valid base name • Don’t open an open file • Don’t open a deleted file • Don’t close a closed file • Don’t close a deleted file • Don’t read from a closed file • Don’t read from a deleted file • Don’t write to a closed file • …
Contractor	See client obligations	Perform functions properly

Where do you check that the obligations are met?

Defensive Programming

Defensive programming

- Wikipedia: “[...] the programmer never assumes a particular function call or library will work as advertised”
- Meyer: “[...] protect every software module by as many checks as possible, even those which are redundant with checks made by the clients.”

Problems with defensive programming

- Multiplies the amount of checking code
- Leads to bloated, hard-to-read, slow code

Redundant code is (mostly) a bad idea

Benefits of Design by Contract

Leads to well-specified interfaces

Leads to clean separation of work

Makes software more reliable

2. Expressing Contracts

Expressing Contracts

1. Preconditions
2. Postconditions
3. Class invariants

1. Preconditions

A precondition is

- A boolean condition to be met for successful method entry

The purpose is to guarantee a safe operating environment

- If violated, the method should not be executed

The client must make sure preconditions are met

- A violation in the preconditions indicates a bug in the client

Preconditions are method-level components of a contract

Precondition Example

```
import { InvalidStateException } from "./InvalidStateException";
import { Node } from "./Node";

enum FileState { OPEN, CLOSED, DELETED };

export class File extends Node {
   protected state: FileState = FileState.CLOSED;

   public open(): void {
       this.assertIsInState(FileState.CLOSED);
       // ...
   }

   protected doGetState(): FileState {
       return this.state;
   }

   protected assertIsInState(state: FileState) {
       if (state != this.doGetState()) {
           throw new InvalidStateException("invalid file state");
       }
   }
}
```

2. Postconditions

A postcondition is

- A boolean condition guaranteed after successful method exit

The method must make sure postconditions are met

- A violation of a postcondition indicates a bug in the method

Postconditions are method-level components of a contract

Postcondition Example

```
public open(): void {
    this.assertIsInState(FileState.CLOSED);
    // do something
    this.assertIsInState(FileState.OPEN);
}

protected assertIsInState(state: FileState) {
    if (state != this.doGetState()) {
        throw new InvalidStateException("invalid file state");
    }
}
```

3. Class Invariants

A class invariant is

- A boolean condition that is true for any valid object

Permanent violation of the class invariant indicates a broken object

- Temporary violation is possible during method execution

Class invariants are constraints on the object's state space

- The class (implementation) must make sure its invariants are maintained

Class invariants are class-level components of a contract

Class Invariants Example

```
export class Node {
    protected baseName: string = "";
    protected permissions: number = 0;

    protected assertClassInvariants() {
        this.assertHasValidBaseName();
        this.assertHasValidPermissions();
    }
    ...
}

export class File extends Node {
    protected state: FileState = FileState.CLOSED;

    protected assertClassInvariants() {
        super.assertClassInvariants();
        this.assertHasValidFileState();
    }

    ...
}
```

3. Implementing Contracts

Defining Contracts

Where to define a contract?

- Class invariants in (class) interface
- Pre- and postconditions in public methods
- But not for protected / private methods

How to define?

- Using comments (documentation)
- Using class or method annotations
- Using asserts or assertion methods

Contracts are part of the public interface

Using Assertion Methods

Wrap assertions in assertion methods; they should be

- Side-effect free (no call to any mutation method)
- Throw an assertion-specific exception upon failure

Assertions can be programmed like any other method

- Reuse assertion code by parameterization
- Group assertions into larger assertion methods
- Inherit assertion methods along the class hierarchy

Implementing Preconditions With Assertion Methods

Preconditions guard the entry to a public client-facing method

- Call the corresponding assertion methods before the main method code
- Precondition assertion methods are a form of before method

Failing a precondition must leave the object in a valid state

- Because no mutation methods have been run yet
- The exception signals the client is at fault

Implementing Postconditions With Assertion Methods

Postconditions ensure valid exit of a public client-facing method

- Call the corresponding assertion methods after the main method code
- Postcondition assertion methods are a form of after method

Failing the postcondition implies the service couldn’t be performed

- The method should return the object to its method-entry state
- The exception signals contractor failure

Implementing Class Invariants With Assertion Methods

Class invariants ensure that the object is in a valid state

- Express the valid state space as a set of assertions
- Group all assertions into one assertion methods

Failing the class invariant implies the object is in an invalid state

- The how and why is probably unclear

Public Interface vs. Protected / Private Implementation

The contract only applies to the public interface

- All assertions are run before and after any implementation code

While inside the object's code, the public contract does not apply

- You can still use assertion methods for other purposes

4. Contract Violations

Contract Violations

Precondition failure

- The client did not fulfill the contract

Postcondition failure

- The contractor could not provide the service

Class invariants

- Something is wrong, really wrong

Recovery From Assertion Failure

Precondition failure

- Nothing to recover from; object remained in valid state

Postcondition failure

- Called method needs to return to initial valid state

Class invariant failure

- Onus is on client; needs to reset the object to a valid state

Basic Exceptions to Use [1]

Precondition

- IllegalArgumentException

Postcondition

- MethodFailureException

Class invariants

- InvalidStateException

[1] Javascript's "Error" classes are a misnomer; exceptions are not errors

Contracts and Control Flow

Contractor

- Use regular control flow (return) if nothing went wrong
- Use exception to indicate contract violation

Client

- Continue in regular control flow if nothing went wrong
- Either resume operations or escalate exception

More on this in lecture on error and exception handling

5. Contract Pragmatics

Contract Pragmatics

Focus on preconditions to guard execution

Contracts and Subtyping

Subclass methods may have less requirements (weaken preconditions)

- Example: Contravariant redefinition of argument types

Subclass methods may guarantee more (strengthen postconditions)

- Example: Covariant redefinition of return types

Homework

Homework Instructions

- Identify the names contracts from lecture and documentation
  - Implement preconditions, postconditions, and class invariants
  - Create corresponding component tests for the contract
- Identify the files contracts from lecture and documentation
  - Implement the corresponding preconditions
- Use the exception classes from common as explained in class
- Adapt your previous work to this homework as you see fit
- Commit homework by deadline to homework folder

Summary

1. Design by contract
2. Expressing contracts
3. Implementing contracts
4. Contract violations
5. Contract pragmatics

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