

Collaboration-Based Design

Dirk Riehle, FAU Erlangen

ADAP D04

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

Agenda

1. Collaborations in UML
2. File system collaborations
3. Collaboration and class models
4. History and related concepts
5. The role object pattern

1. Collaborations in UML

Collaboration-Based Design

Collaboration-based design is

- An approach to modeling and implementing using collaborations

A collaboration specification (UML: Collaboration) is

- A model of how objects collaborate for a particular purpose

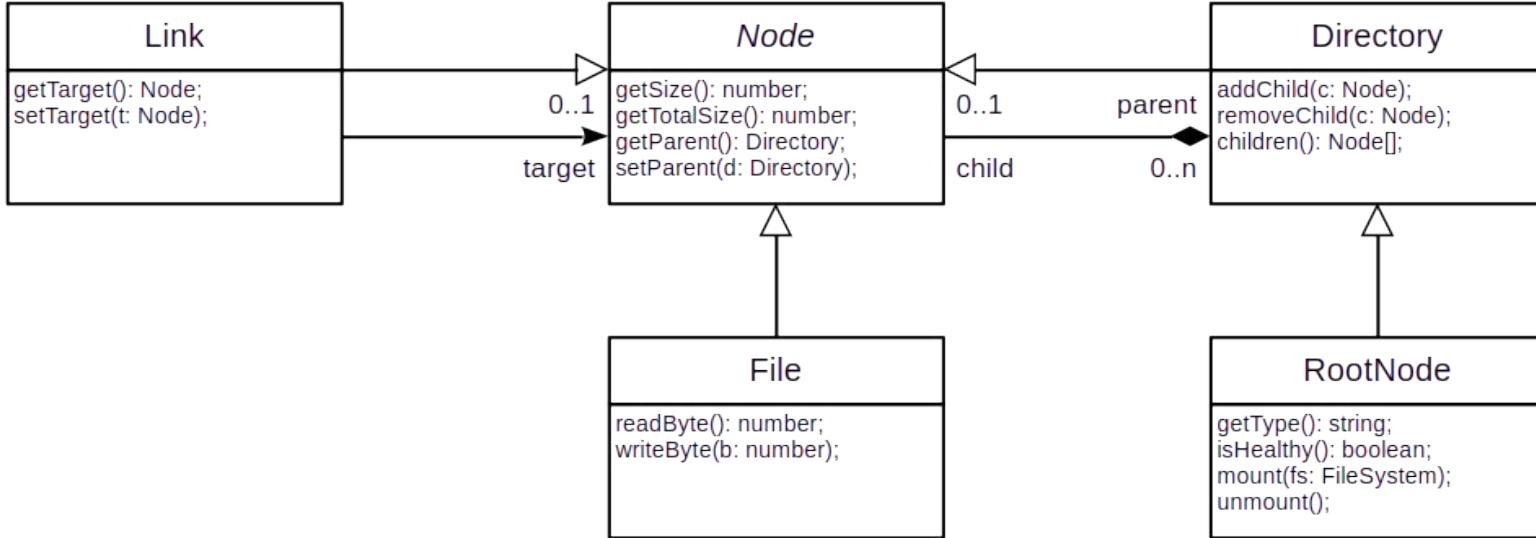
A collaboration instance (UML: Collaboration Use) is

- A set of objects collaborating according to a collaboration specification

A role specification (UML: Role) is

- A model of the behavior of one object in a collaboration instance

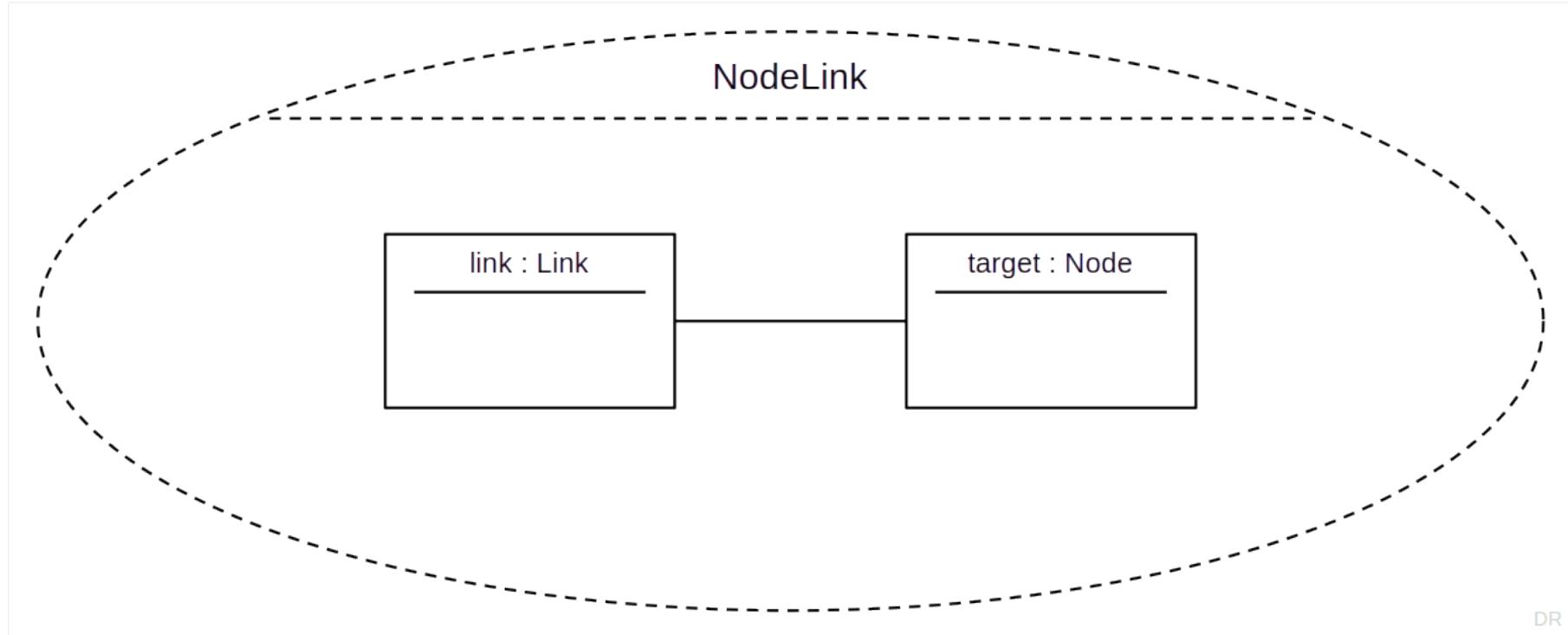
File System Example Revisited



DR

5

Collaboration Example (UML)



DR

Collaboration as Code

```
export class Link extends Node {  
    protected target: Node | null = null;  
  
    constructor(bn: string, pn: Directory, tn?: Node) {  
        super(bn, pn);  
  
        if (tn != undefined) {  
            this.target = tn;  
        }  
    }  
  
    public getTarget(): Node | null {  
        return this.target;  
    }  
  
    public setTarget(target: Node): void {  
        this.target = target;  
    }  
  
    ...  
}
```

```
export class Node {  
    ...  
}
```

Collaboration Role Binding Example (UML)



DR

8

Collaboration Roles as Code

```
export interface NodeLinkLink {
    getTarget(): NodeLinkTarget | null;
    setTarget(target: NodeLinkTarget): void;
}

export class Link extends Node implements NodeLinkLink {
    protected target: NodeLinkTarget | null = null;

    constructor(bn: string, pn: Directory, tn?: Node) {
        super(bn, pn);

        if (tn != undefined) {
            this.target = tn;
        }
    }

    public getTarget(): NodeLinkTarget | null {
        return this.target;
    }

    public setTarget(target: NodeLinkTarget): void {
        this.target = target;
    }

    ...
}
```

```
export interface NodeLinkTarget {
    // no methods
}

export class Node implements NodeLinkTarget {
    protected baseName: string = "";
    protected parentNode: Directory;

    constructor(bn: string, pn: Directory) {
        this.doSetBaseName(bn);
        this.parentNode = pn;
        this.initialize(pn);
    }

    protected initialize(pn: Directory): void {
        this.parentNode = pn;
        this.parentNode.addChild(this);
    }

    protected doSetBaseName(bn: string): void {
        this.baseName = bn;
    }

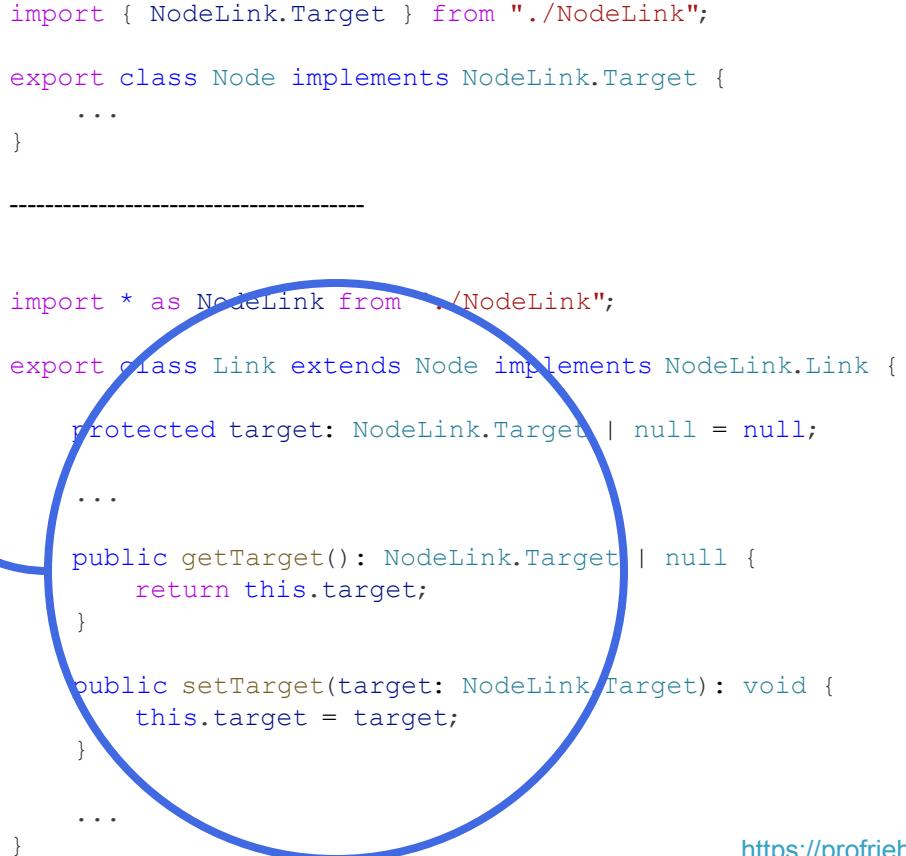
    ...
}
```

Collaboration as Code (as Language Construct)

```
export collaboration NodeLink {  
  
    role Link {  
  
        protected target: Target | null = null;  
  
        constructor(tn?: Target) {  
            if (tn != undefined) {  
                this.target = tn;  
            }  
        }  
  
        public getTarget(): Target | null {  
            return this.target;  
        }  
  
        public setTarget(target: Target): void {  
            this.target = target;  
        }  
  
    }  
  
    role Target {  
        // no methods  
    }  
  
}  
  
import { NodeLink.Target } from "./NodeLink";  
  
export class Node implements NodeLink.Target {  
    ...  
}  
-----  
  
import * as NodeLink from "./NodeLink";  
  
export class Link extends Node implements NodeLink.Link {  
  
    protected target: NodeLink.Target | null = null;  
  
    ...  
  
    public getTarget(): NodeLink.Target | null {  
        return this.target;  
    }  
  
    public setTarget(target: NodeLink.Target): void {  
        this.target = target;  
    }  
  
    ...  
}
```

Semantics of Collaboration “Implementation”

```
export collaboration NodeLink {  
  
    role Link {  
  
        protected target: Target | null = null;  
  
        constructor(tn?: Target) {  
            if (tn != undefined) {  
                this.target = tn;  
            }  
        }  
  
        public getTarget(): Target | null {  
            return this.target;  
        }  
  
        public setTarget(target: Target): void  
            this.target = target;  
        }  
  
    role Target {  
        // no methods  
    }  
}
```



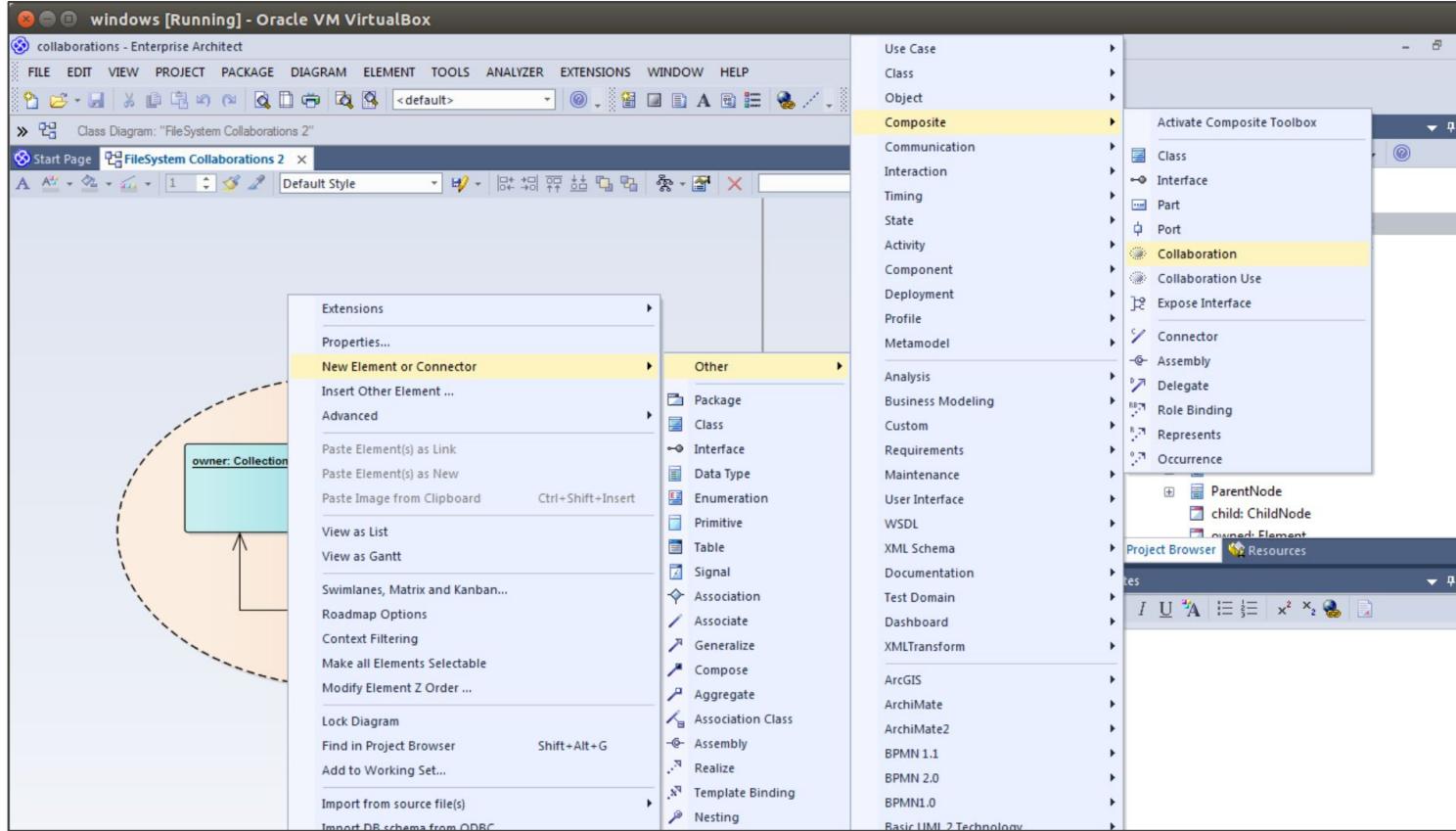
The diagram features two code snippets side-by-side, separated by a large red question mark. The left snippet is a UML-style collaboration diagram for `NodeLink`. It contains two roles: `Link` and `Target`. The `Link` role has a protected attribute `target` of type `Target` or `null`, initialized to `null`. It has a constructor that takes an optional `Target` parameter and sets it to `this.target`. It also has two methods: `getTarget()` which returns `this.target`, and `setTarget(target: Target)` which sets `this.target` to the provided `target`. The `Target` role has no methods. The right snippet is a TypeScript code listing. It starts with an import statement for `Target` from `./NodeLink`. Then it defines a class `Node` that implements the `Target` interface. The class body is partially shown with three dots. A horizontal dashed line separates this from another code block. Below the dashed line, there is an import statement for `Link` from `./NodeLink`. Following this is a class definition for `Link` that extends `Node` and implements the `Link` interface. This class has a protected attribute `target` of type `NodeLink.Target` or `null`, initialized to `null`. It also has a constructor, a `getTarget()` method that returns `this.target`, a `setTarget(target: NodeLink.Target)` method that sets `this.target` to the provided `target`, and an ellipsis at the end.

```
import { NodeLink.Target } from "./NodeLink";  
  
export class Node implements NodeLink.Target {  
    ...  
}  
  
-----  
  
import * as NodeLink from './NodeLink';  
  
export class Link extends Node implements NodeLink.Link {  
  
    protected target: NodeLink.Target | null = null;  
  
    ...  
  
    public getTarget(): NodeLink.Target | null {  
        return this.target;  
    }  
  
    public setTarget(target: NodeLink.Target): void {  
        this.target = target;  
    }  
    ...  
}
```

Benefits of Collaboration-Based Design

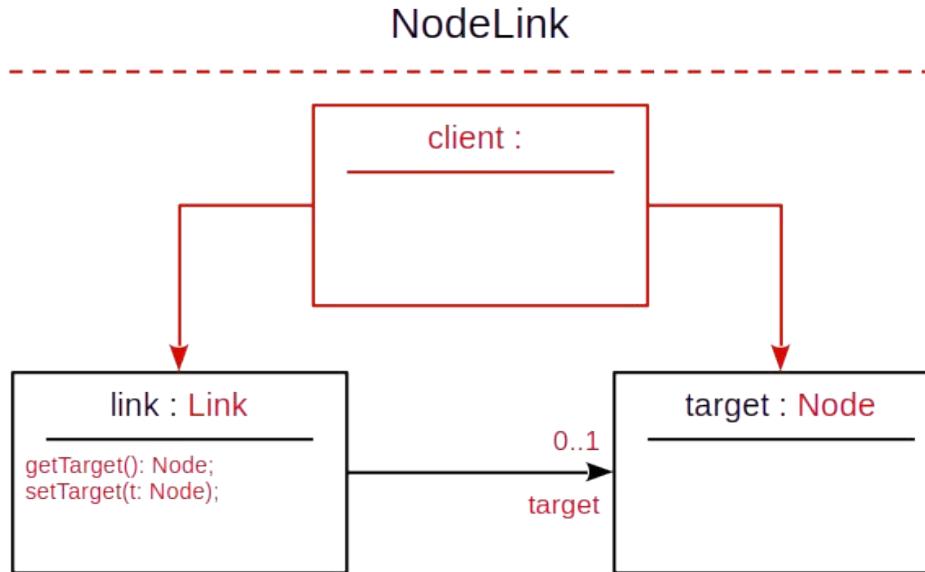
1. Separation of concerns
2. Better reusable models (potentially)

Reality Check: UML Collaboration in Sparx EA



2. File System Collaborations

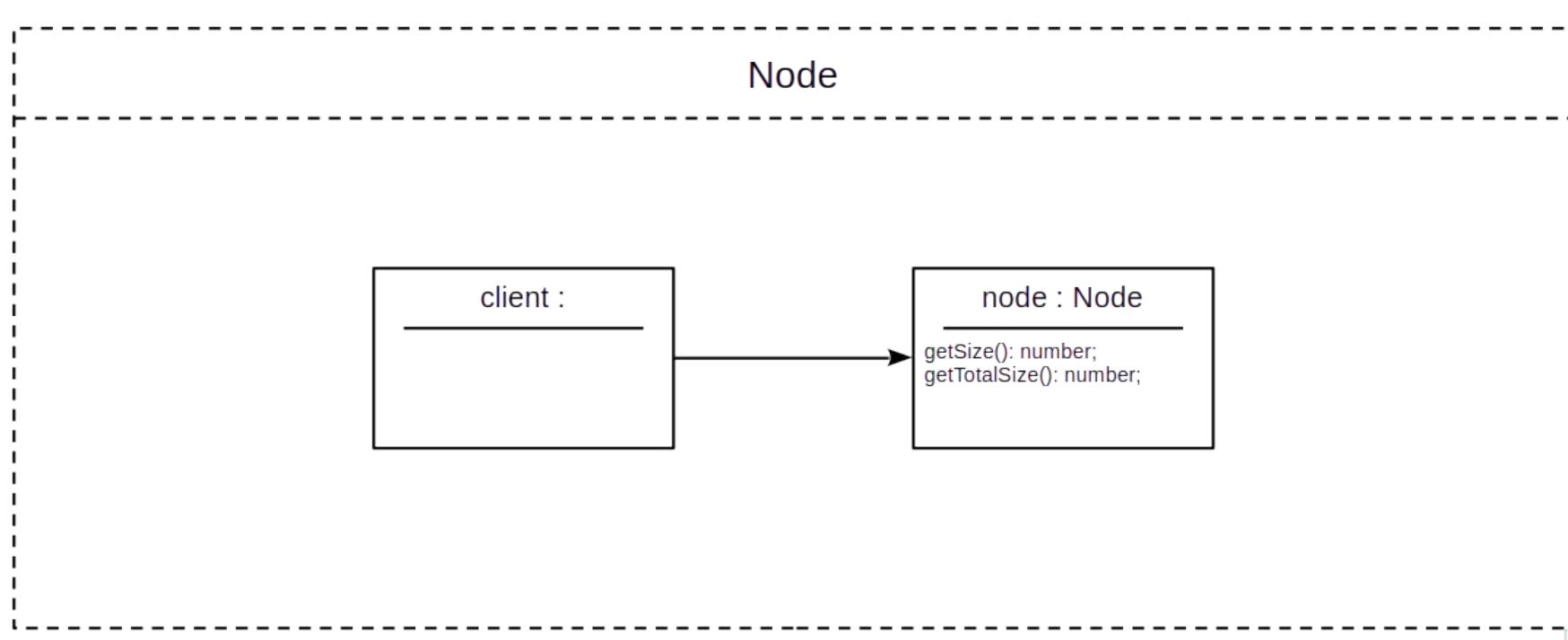
Node Link Collaboration (Riehle's UML Extension)



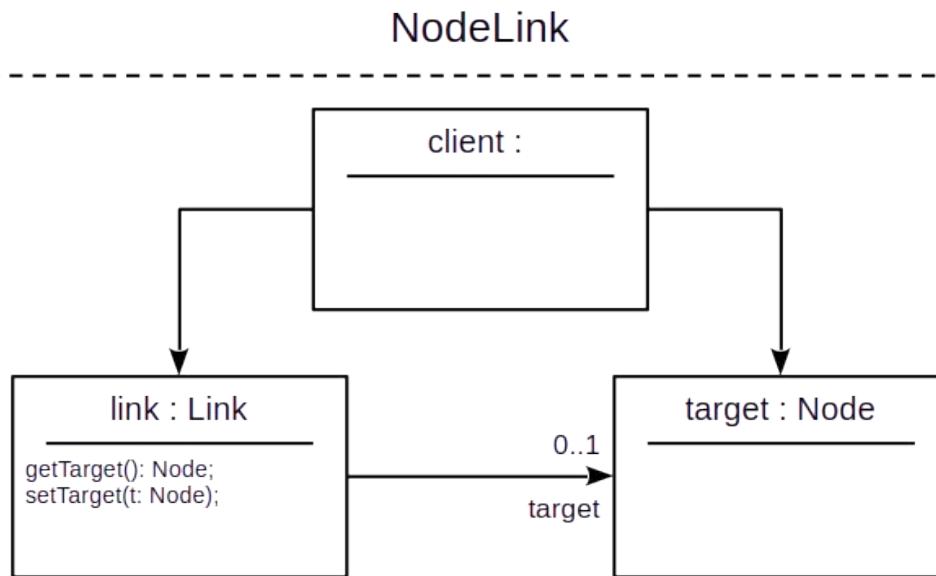
Collaborations in File System Example

1. Node collaboration
2. NodeLink collaboration
3. File collaboration
4. ParentChild collaboration (a.k.a. Composite pattern)
5. RootNode collaboration

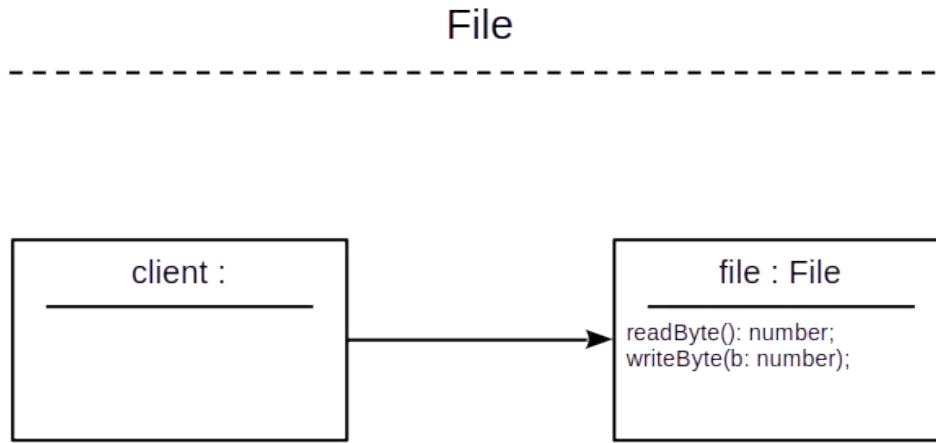
Node Collaboration



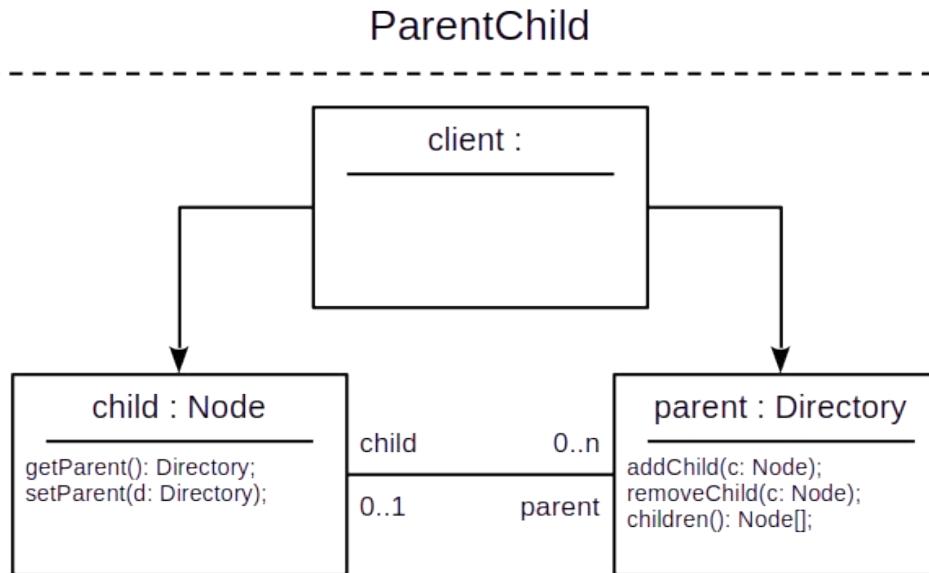
NodeLink Collaboration



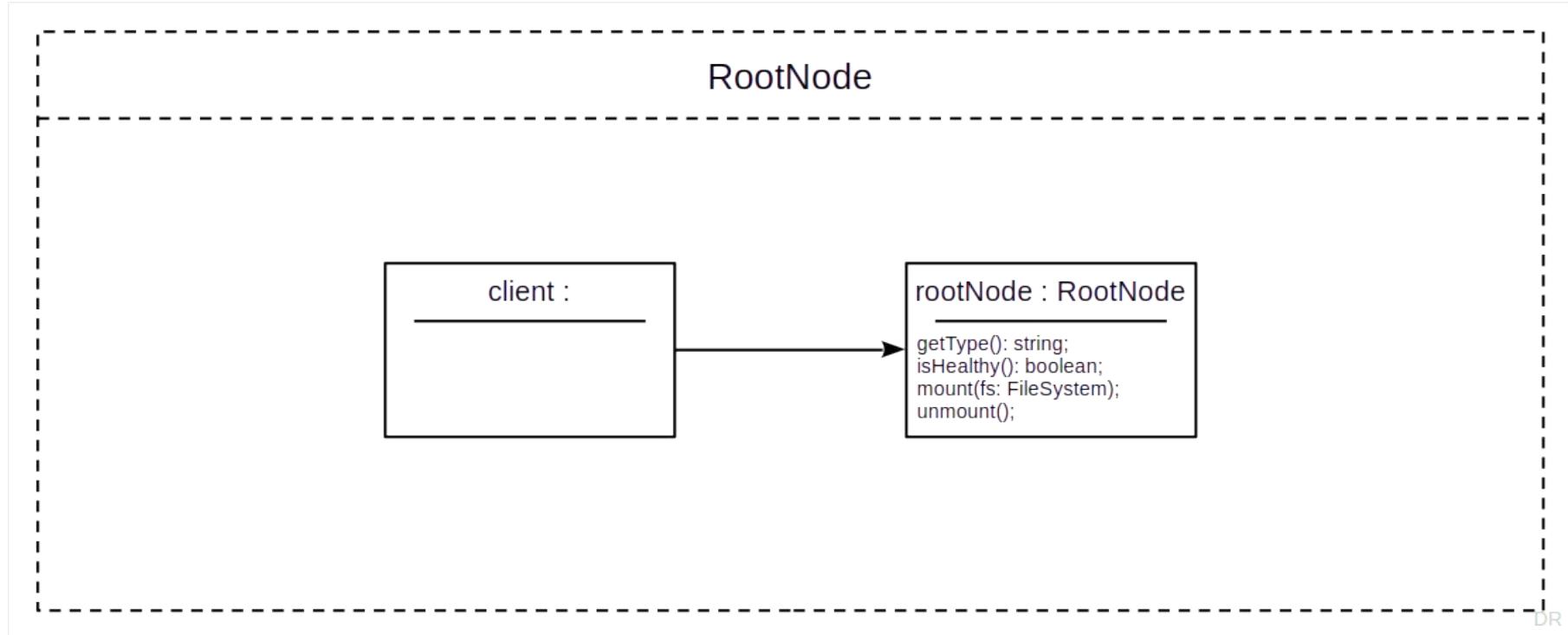
File Collaboration



Parent / Child Collaboration



RootNode Collaboration



Types of Collaborations

Primary service collaborations

- Typically simple client / service collaborations
- The client role often has no methods
- Are visible to the outside (use-clients)

Maintenance collaborations

- Maintain the domain logic within the model
- Often follow design pattern logic
- Often not visible to the outside (no use-clients)

3. Collaboration and Class Models

Collaborations / Roles vs. Class Models / Classes

Collaborations

- Focus on behavior

Roles

- Are part of collaborations
- Compose classes from parts
- Use use-client interface

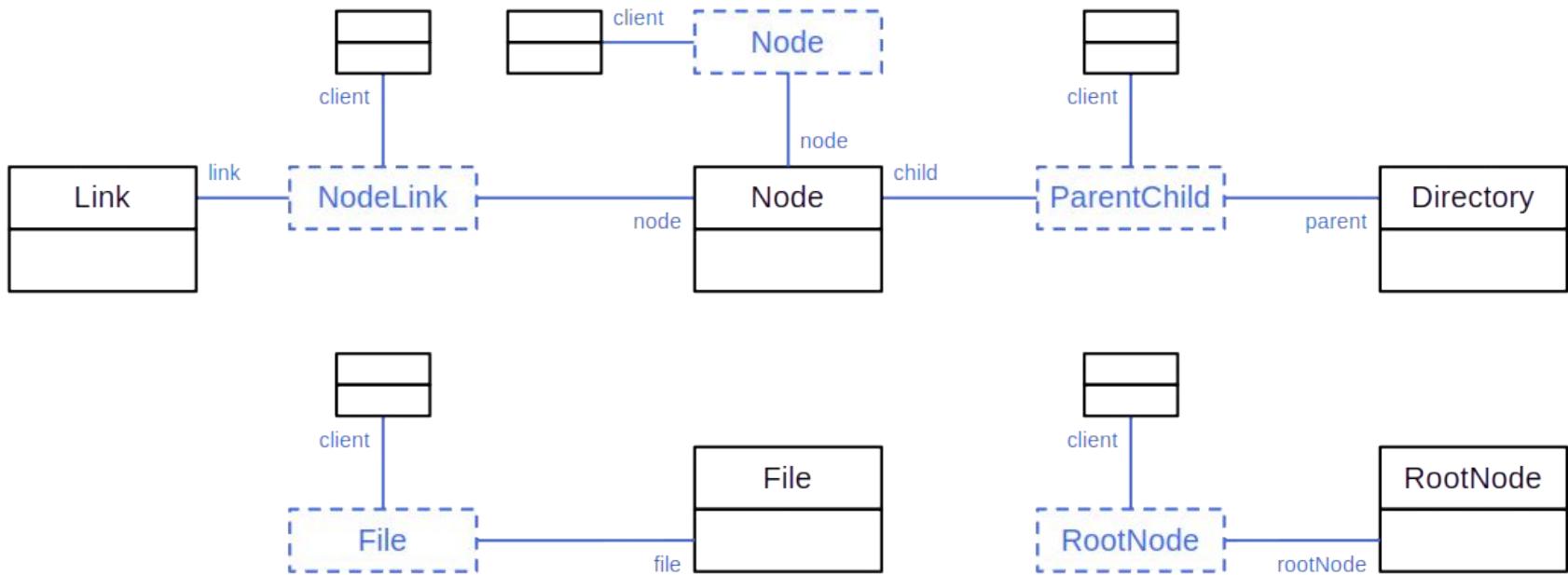
Class models

- Focus on structure

Classes

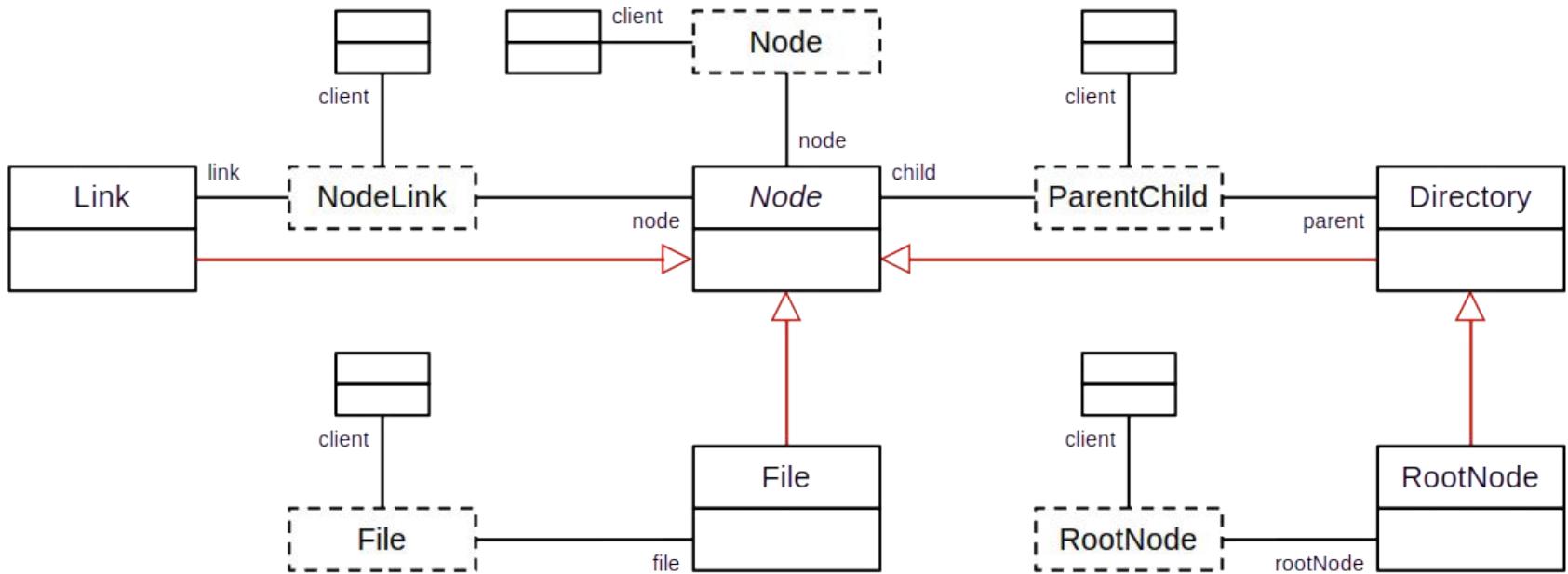
- Are part of class hierarchies
- Compose classes by inheritance
- Use inheritance interface

Collaboration Role to Class Assignments



DR

Collaborations with Class Hierarchy



DR

26

Role and Class Duality

A role specification ties a class into a collaboration context

- Control flow enters and leaves through a role

A class composes behavior in multiple contexts into one unit

- Role implementation is mapped onto implementation state

Levels of Abstraction

Design patterns

- No modeling support

Design templates

- UML Collaboration

Class models

- UML Class Model

4. History and Related Concepts

History of Collaboration Modeling Methods

1. Trygve Reenskaug, Role Modeling [1]
2. Dirk Riehle, Framework Design [2]
3. OMG, UML 2.0 Specification [3]

[1] Reenskaug, T., Wold, P., & Lehne, O. A. (1996). Working with Objects. Manning.

[2] Riehle, D. (2000). [Framework Design: A Role Modeling Approach](#). Dissertation, no. 13509. Swiss Federal Institute of Technology at Zurich (ETH Zurich), Zurich, Switzerland.

[3] OMG (2005). [Unified Modeling Language 2.0](#). OMG.

Related Concepts from Programming Language History

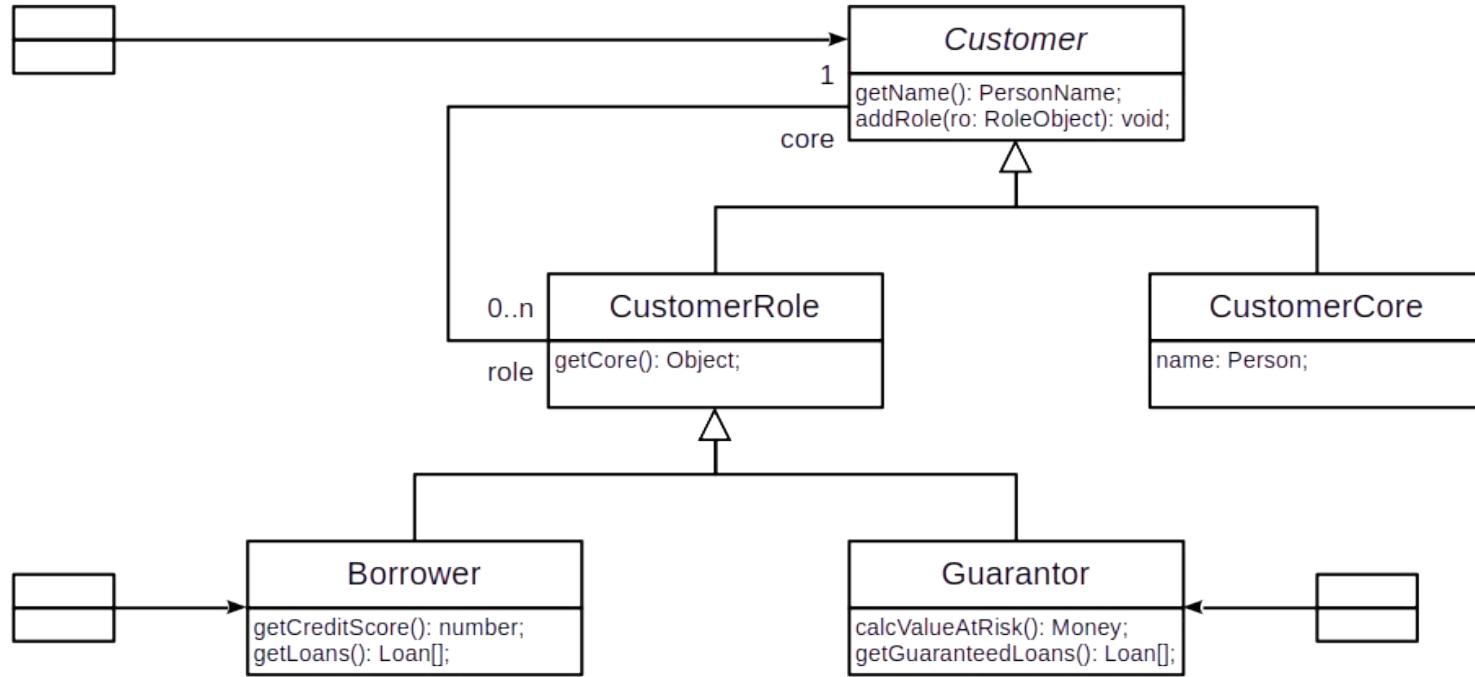
Past attempts

- Interfaces
- Protocols
- Mix-ins
- Traits

Common to all is that they ignore the collaboration, focus only on roles

5. The Role Object Pattern

Role Object Example



DR

[1] Bäumer, D., Riehle, D., Siberski, W. & Wulf, M. (2000). [Role Object](#). In Pattern Languages of Program Design 4 (PLoPD 4), pp. 15-32. Addison-Wesley.

33

<https://profriehle.com>

Role Object Pattern

Context

- A large modular system with core domain concepts and varied applications

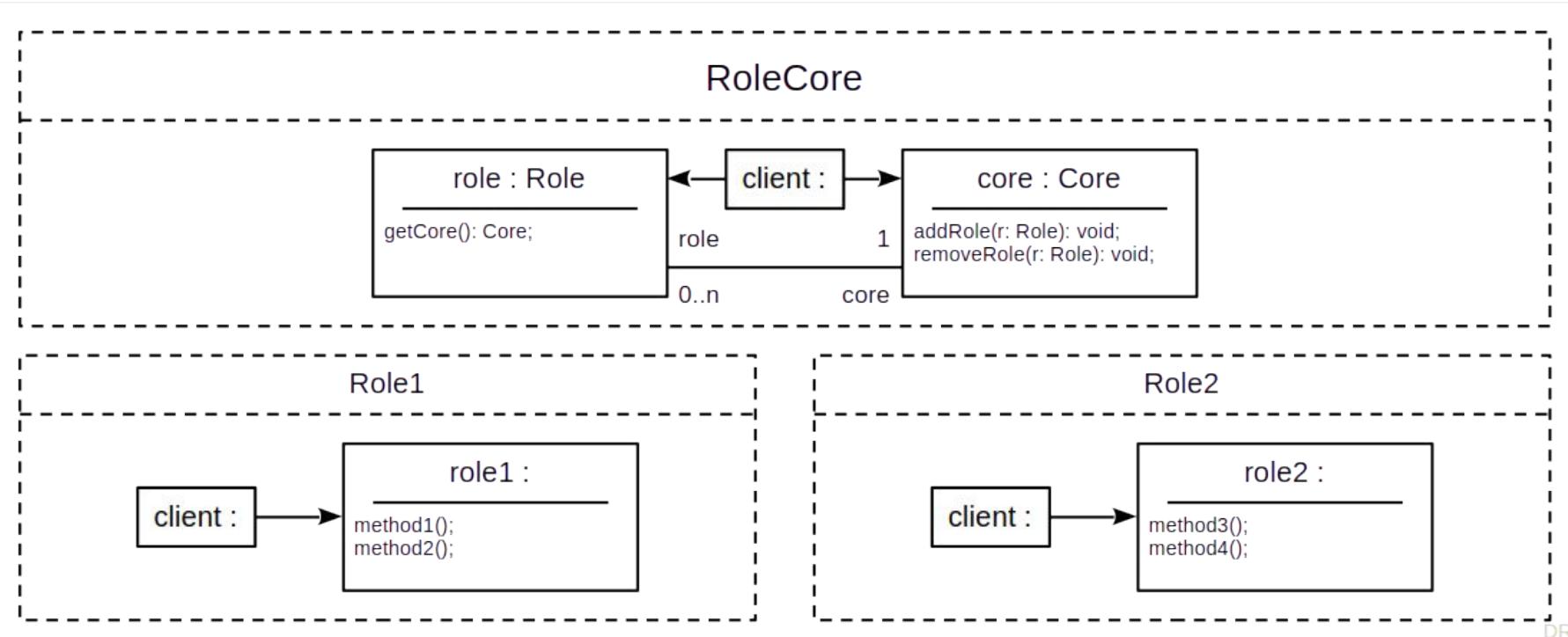
Problem

- A core concept needs to show a different face in each application

Solution

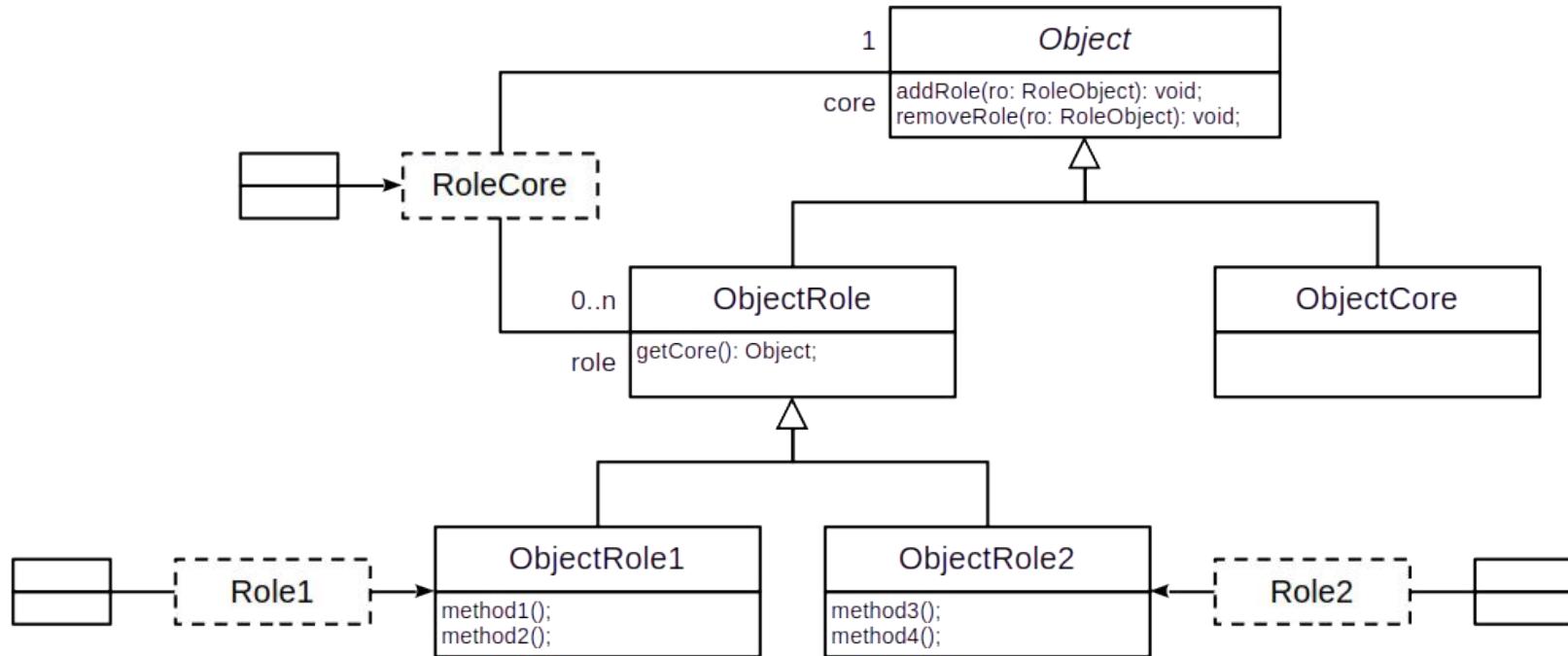
- Extend the core concept with separate role objects facing the application

Role Object Collaborations



DR

Role Object Structure Diagram



DR

Role Objects vs. Collaborations

The role object pattern

- Can be described using collaborations
- Allows for dynamic role playing
- Simplifies persistence

Summary

1. Collaboration in UML
2. File system collaborations
3. Collaboration and class models
4. History and related concepts
5. The Role Object pattern

Thank you! Any questions?

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

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

Legal Notices

License

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

Copyright

- © 2012, 2018, 2025 Dirk Riehle, some rights reserved