Structural Design Patterns

Software Modeling Foundations

Author: Eng. Carlos Andrés Sierra, M.Sc.
cavirguezs@udistrital.edu.co

Computer Engineer
Lecturer
Universidad Distrital Francisco José de Caldas

2024-III

Outline

1. Introduction
2. Patterns
   - Bridge
   - Composite
   - Proxy
   - Flyweight
   - Decorator*
   - Adapter*
   - Facade*
3. Conclusions

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

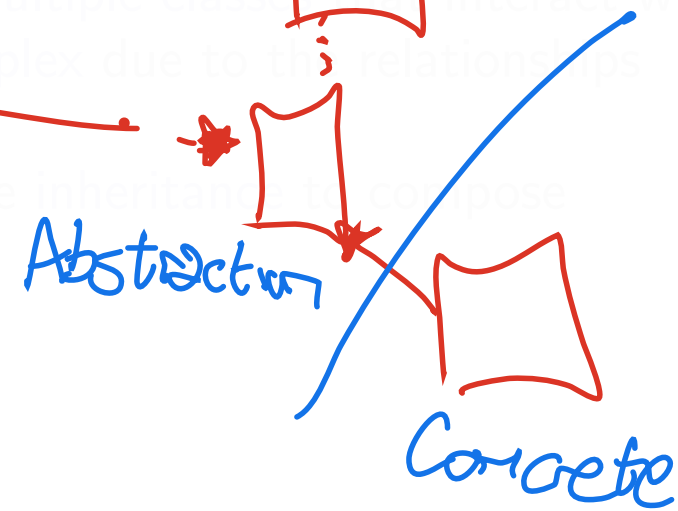
Outline

1. Introduction
2. Patterns
   - Bridge
   - Composite
   - Proxy
   - Flyweight
   - Decorator*
   - Adapter*
   - Facade*
3. Conclusions

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

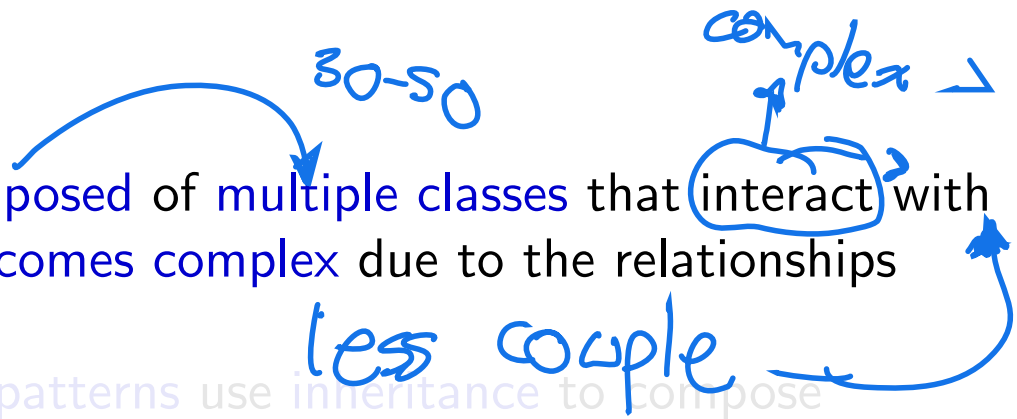
Basic Concepts

- **Intent**: Describe how objects are connected to each other. These patterns are related to the design principles of descomposition and generalization.
- **Motivation**:
  - **Problem**: A system is composed of multiple classes that interact with each other. The system becomes complex due to the relationships between these classes.
  - **Solution**: Structural class patterns use inheritance to compose interfaces or implementations.



Basic Concepts

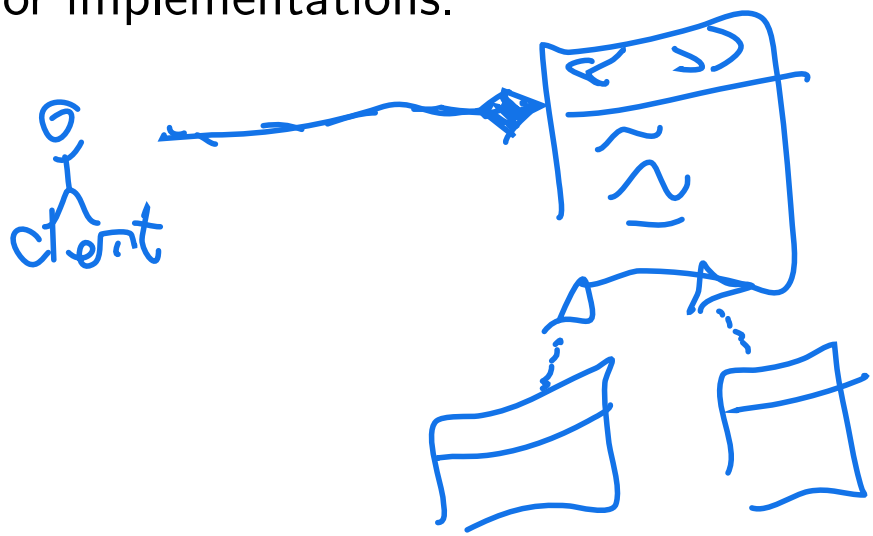
- **Intent**: Describe how objects are connected to each other. These patterns are related to the design principles of descomposition and generalization.
- **Motivation**:
  - **Problem**: A system is composed of multiple classes that interact with each other. The system becomes complex due to the relationships between these classes.
  - **Solution**: Structural class patterns use inheritance to compose interfaces or implementations.



UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Basic Concepts

- **Intent**: Describe how objects are connected to each other. These patterns are related to the design principles of descomposition and generalization.
- **Motivation**:
  - **Problem**: A system is composed of multiple classes that interact with each other. The system becomes complex due to the relationships between these classes.
  - **Solution**: Structural class patterns use inheritance to compose interfaces or implementations.



UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Outline

1 Introduction

2 Patterns
- Bridge
- Composite
- Proxy
- Flyweight
- Decorator*
- Adapter*
- Facade*

3 Conclusions

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Outline

1. Introduction
2. Patterns
   - Bridge
   - Composite
   - Proxy
   - Flyweight
   - Decorator*
   - Adapter*
   - Facade*
3. Conclusions

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Bridge Pattern — Concepts

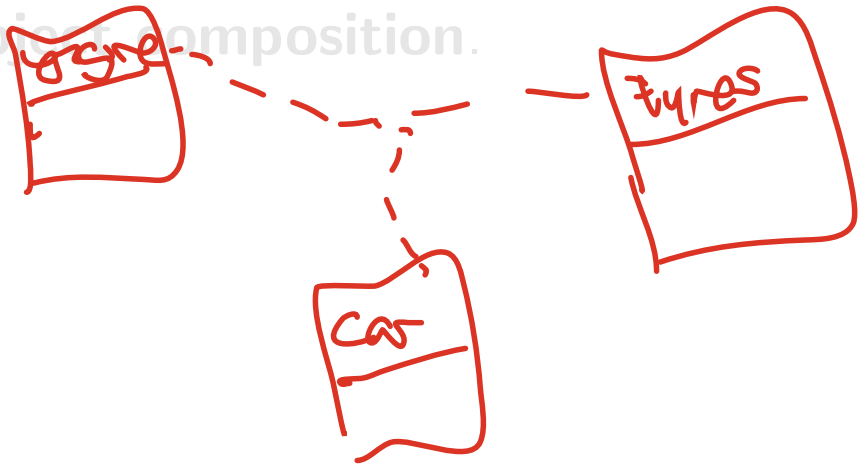
- When there is a **very large class**, this pattern lets **split** it into two separate hierarchies based on *abstraction* and *implementation*.
- Also, it helps when you want to combine two different but **related classes**, and you want to keep them independent.
- This **pattern** solves this problem avoiding **heritance** and trying to switch to **object composition**.

1. Multiple responsability
2. ~ 20 o more methods

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Bridge Pattern — Concepts

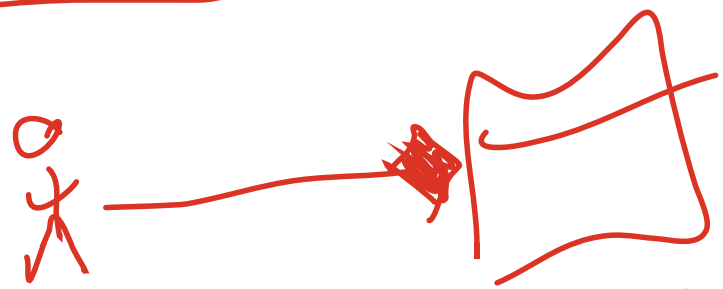
- When there is a **very large class**, this pattern lets **split** it into two separate hierarchies based on *abstraction* and *implementation*.
- Also, it helps when you want to combine two different but **related classes**, and you want to keep them independent.
- This **pattern** solves this problem avoiding **heritance** and trying to switch to **object composition**.



UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Bridge Pattern — Concepts

- When there is a **very large class**, this pattern lets **split** it into two separate hierarchies based on *abstraction* and *implementation*.
- Also, it helps when you want to combine two different but **related classes**, and you want to keep them independent.
- This **pattern** solves this problem avoiding **heritance** and trying to switch to **object composition**.



Bridge Pattern — Classes Structure

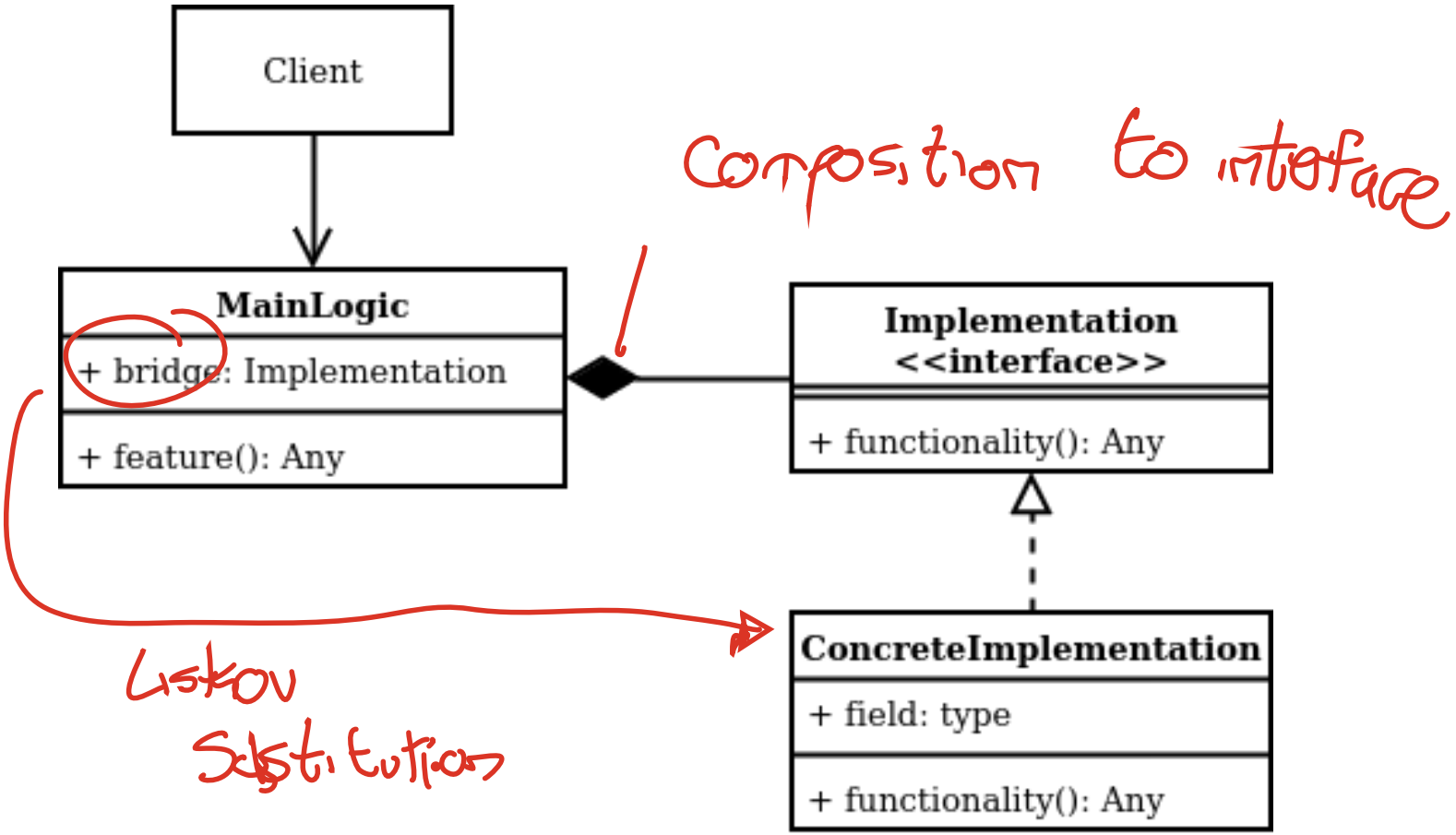
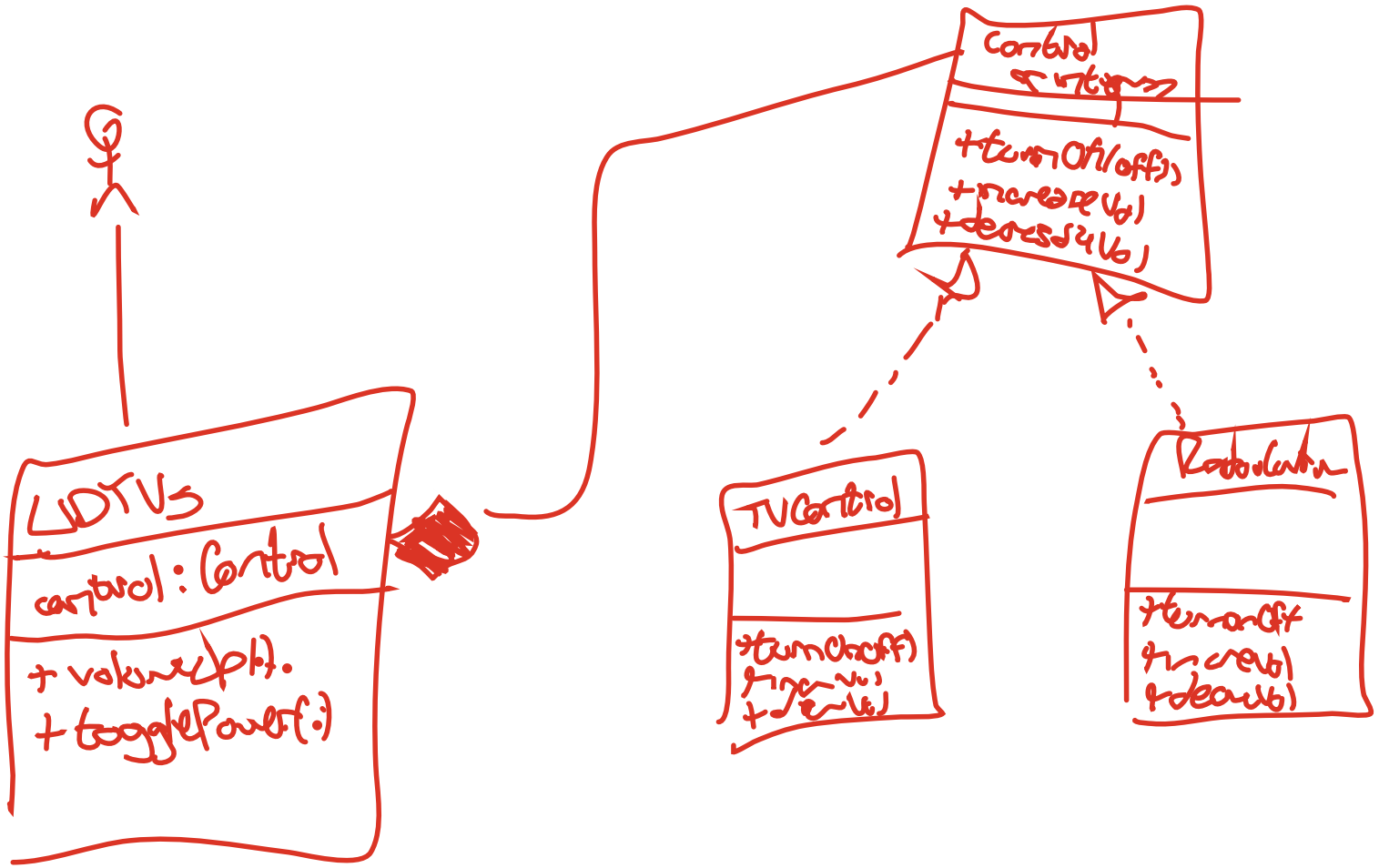


Figure: Bridge Pattern Class Diagram

Bridge Pattern Example: Remote Controls



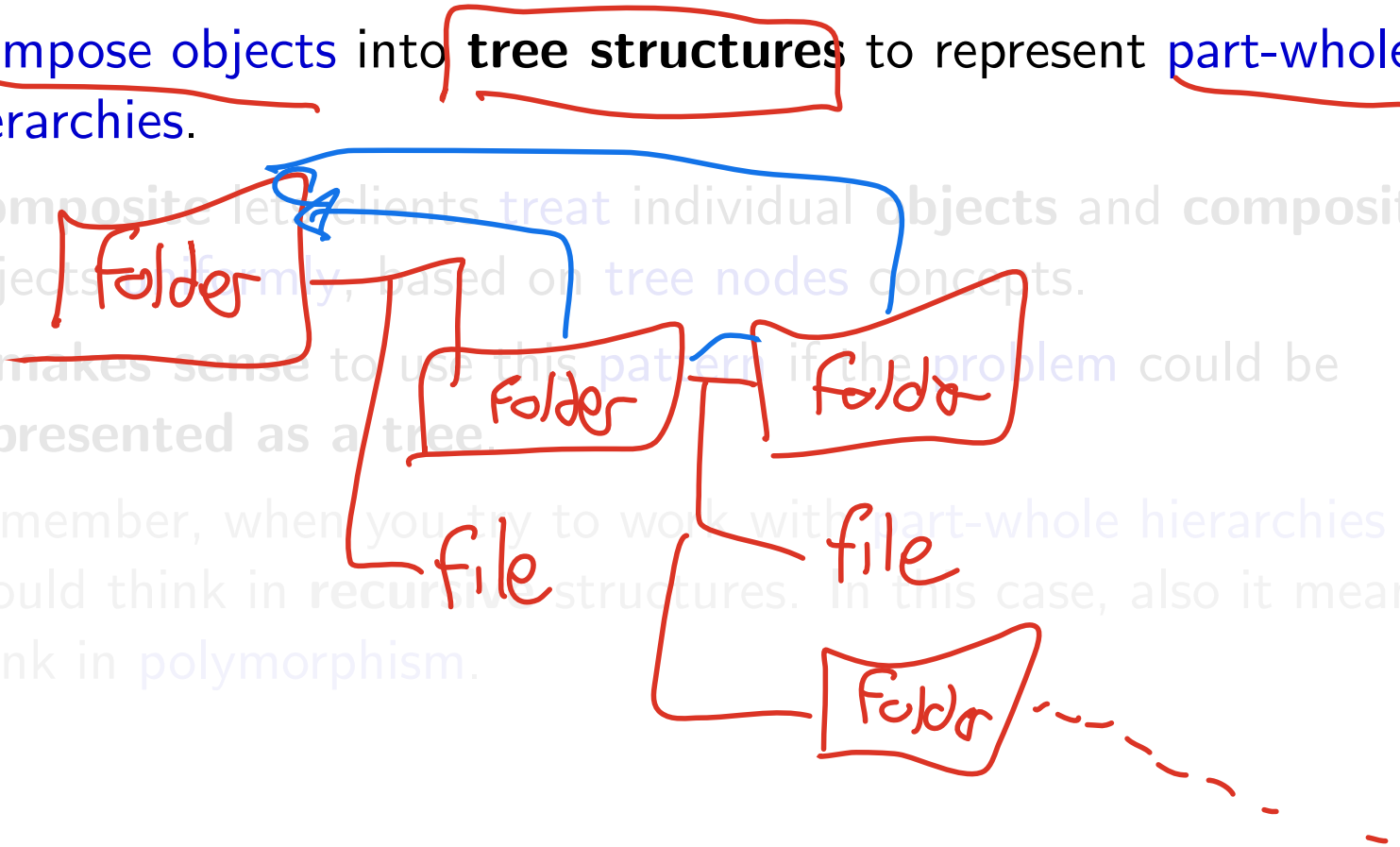
Outline

1. Introduction
2. Patterns
   - Bridge
   - Composite
   - Proxy
   - Flyweight
   - Decorator*
   - Adapter*
   - Facade*
3. Conclusions

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Composite Pattern — Concepts

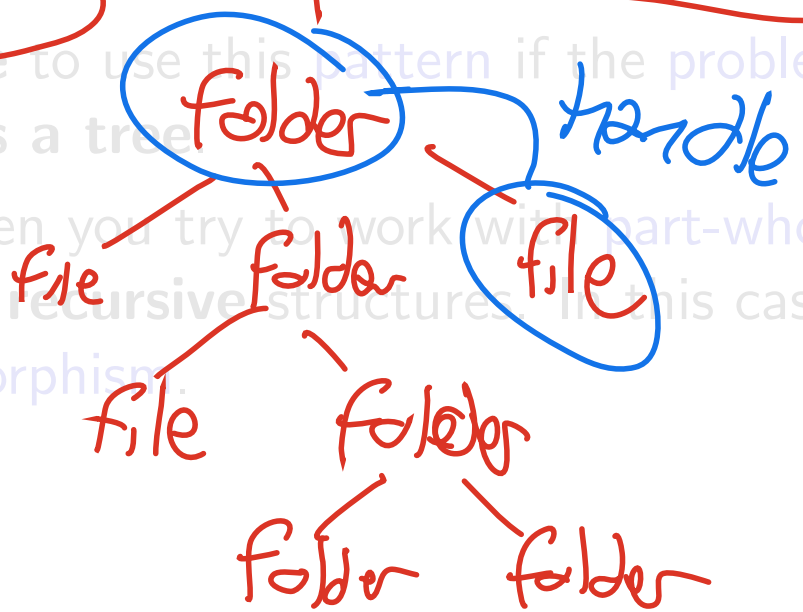
- Compose objects into **tree structures** to represent part-whole hierarchies.
- **Composite** lets clients treat individual **objects** and **compositions** of objects uniformly, based on tree nodes concepts.
- It **makes sense** to use this pattern if the problem could be **represented as a tree**.
- Remember, when you try to work with part-whole hierarchies you should think in **recursive** structures. In this case, also it means to think in polymorphism.



UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Composite Pattern — Concepts

- Compose objects into **tree structures** to represent part-whole hierarchies.
- **Composite** lets clients treat individual **objects** and **compositions** of objects uniformly, based on tree nodes concepts.
- It **makes sense** to use this pattern if the problem could be **represented as a tree**.
- Remember, when you try to work with part-whole hierarchies you should think in **recursive** structures. In this case, also it means to think in polymorphism.



UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Composite Pattern — Concepts

- Compose objects into **tree structures** to represent part-whole hierarchies.
- **Composite** lets clients treat individual **objects** and **compositions** of objects uniformly, based on tree nodes concepts.
- It **makes sense** to use this pattern if the problem could be **represented as a tree**.
- Remember, when you try to work with part-whole hierarchies you should think in **recursive** structures. In this case, also it means to think in polymorphism.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Composite Pattern — Concepts

- Compose objects into **tree structures** to represent part-whole hierarchies.
- **Composite** lets clients treat individual **objects** and **compositions** of objects uniformly, based on tree nodes concepts.
- It **makes sense** to use this pattern if the problem could be **represented as a tree**.
- Remember, when you try to work with part-whole hierarchies you should think in **recursive** structures. In this case, also it means to think in polymorphism.

Composite Pattern — Classes Structure

Looks like the *russian dolls*, the matryoshka.

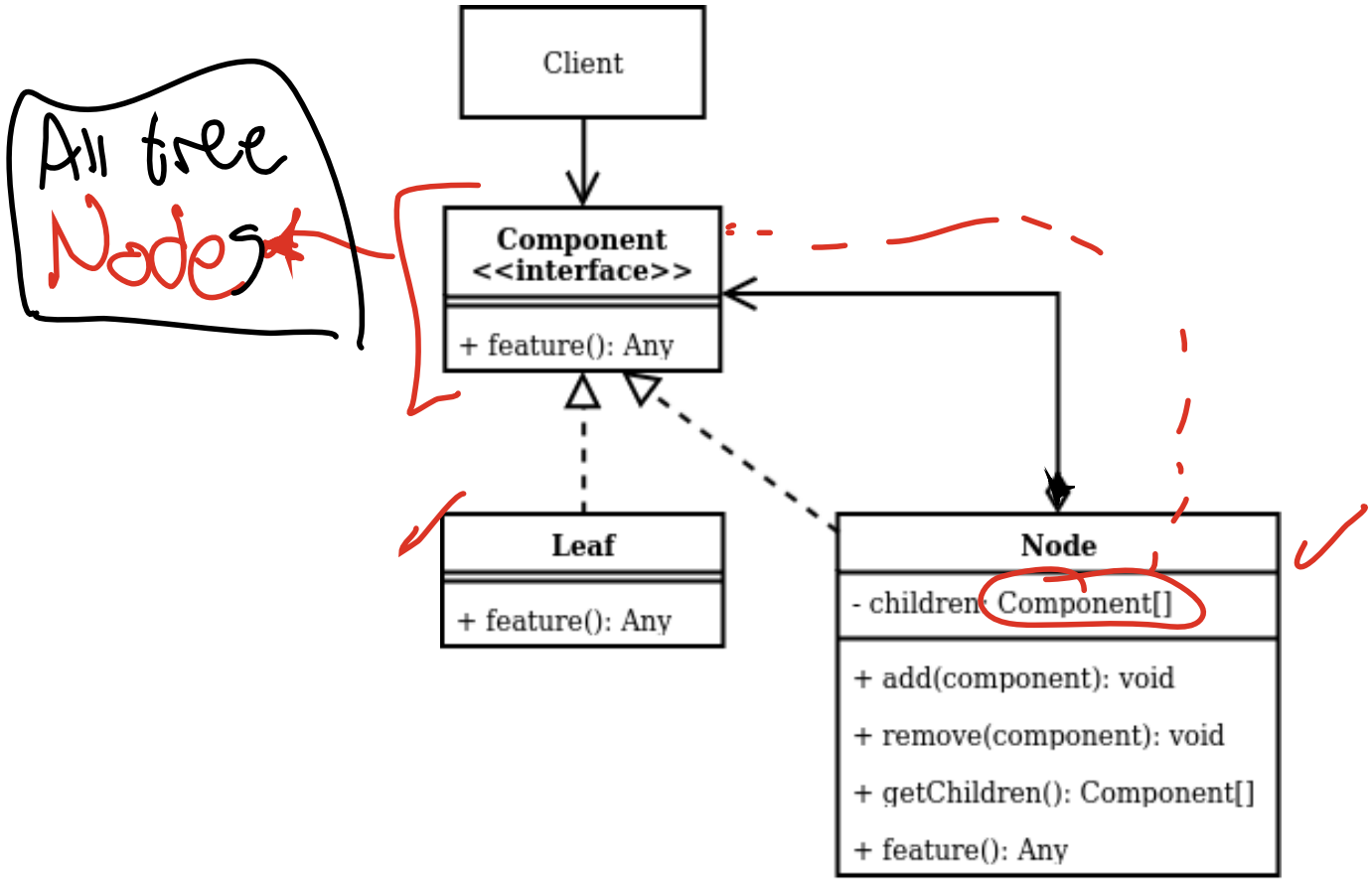
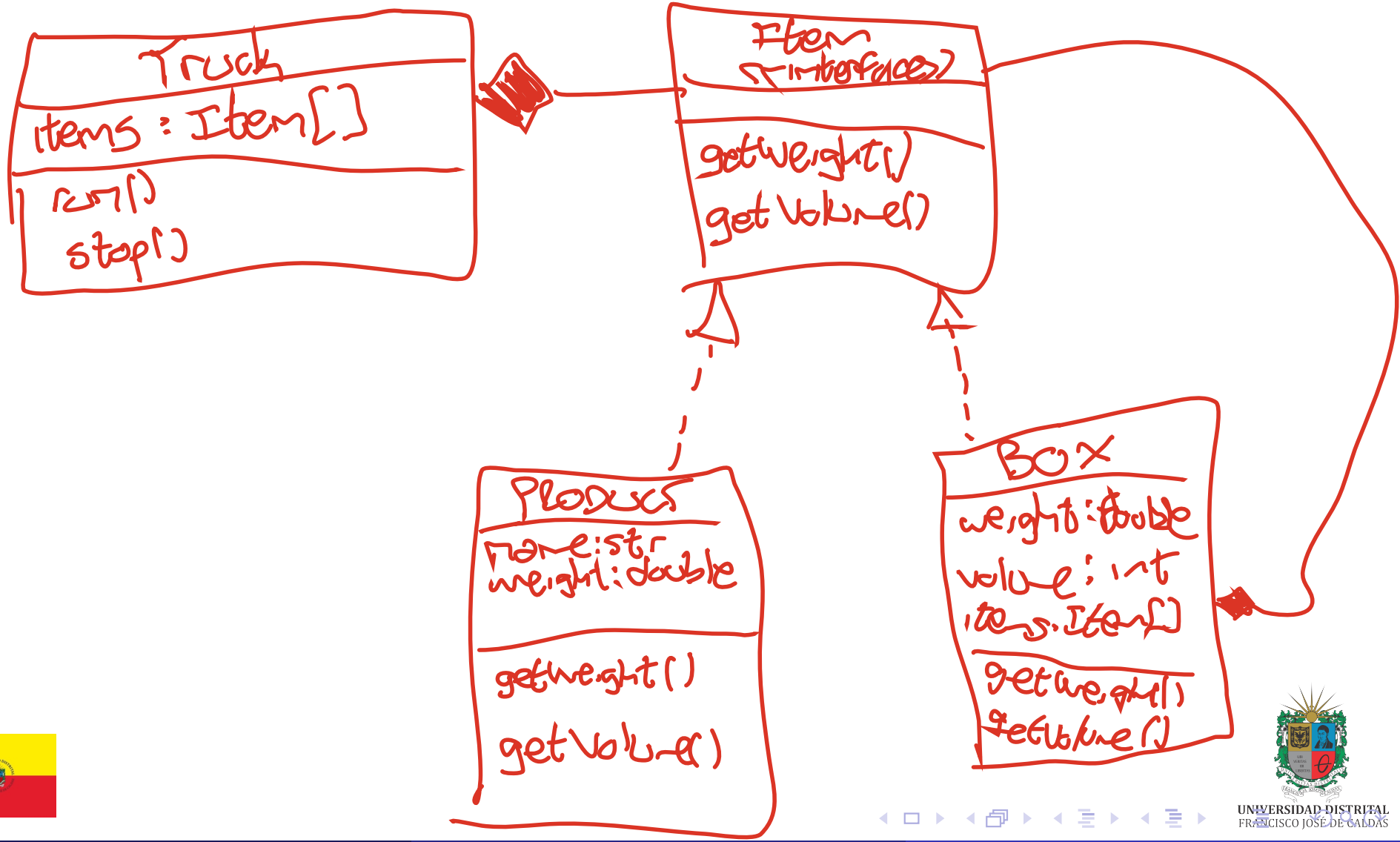


Figure: Composite Pattern Class Diagram

Composite Pattern Example: Amazon Delivery Warehouse



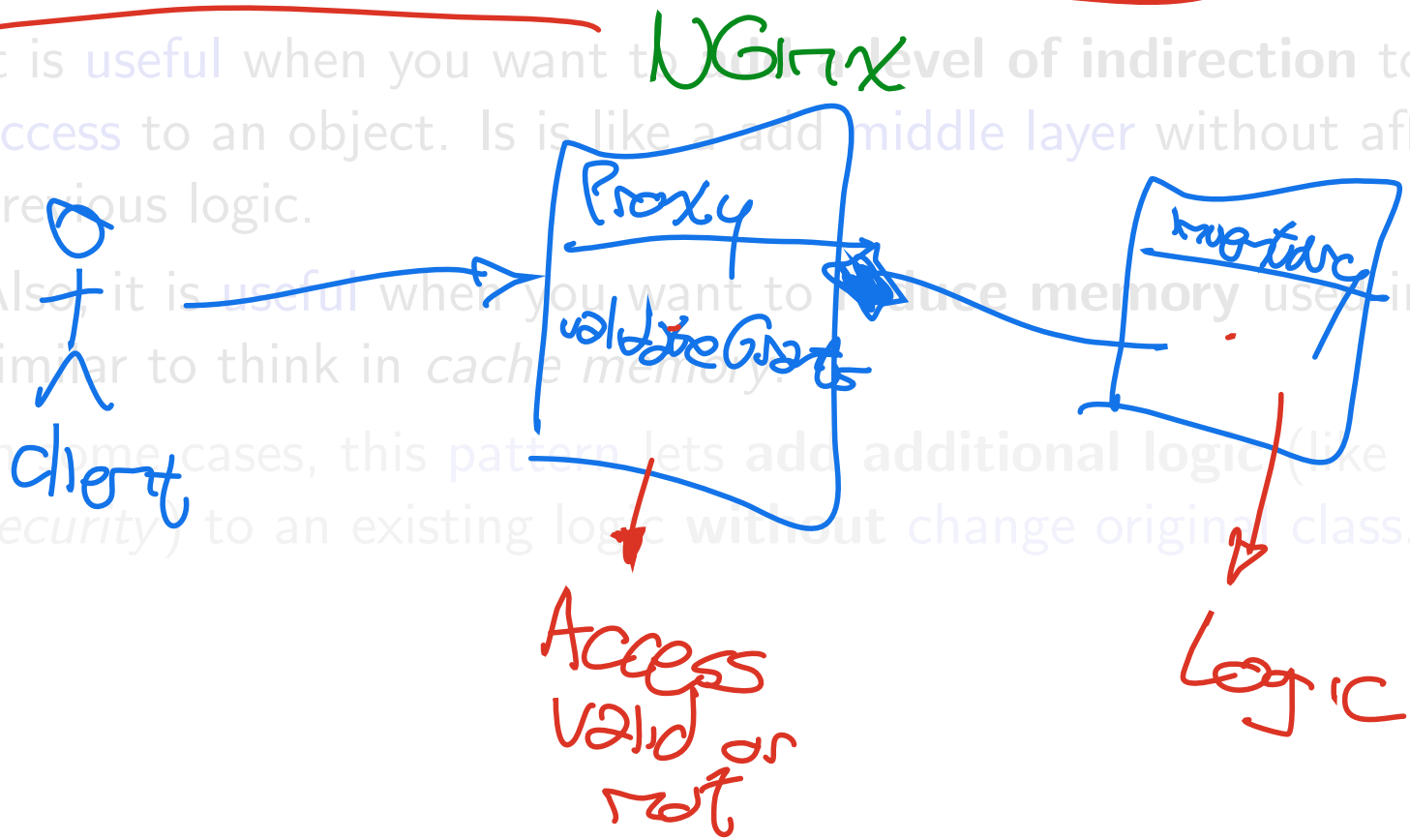
Outline

1. Introduction
2. Patterns
   - Bridge
   - Composite
   - Proxy
   - Flyweight
   - Decorator*
   - Adapter*
   - Facade*
3. Conclusions

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Proxy Pattern — Concepts

- This pattern lets to provide a **substitute** for an object. In this way, access could be controlled.
- It is useful when you want to add a **level of indirection** to control access to an object. Is is like a add middle layer without affect previous logic.
- Also it is useful when you want to **reduce memory** used in a service, similar to think in *cache memory*.
- In some cases, this pattern lets **add additional logic** (like *logging* or *security*) to an existing logic **without** change original class.



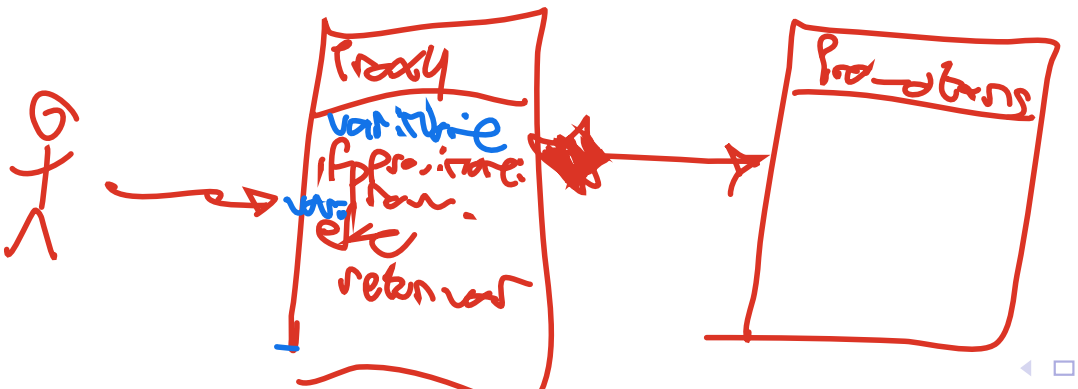
Proxy Pattern — Concepts

- This pattern lets to provide a **substitute** for an object. In this way, access coud be controlled.
- It is useful when you want to **add a level of indirection** to control access to an object. Is is like a add middle layer without affect previous logic.
- Also, it is useful when you want to **reduce memory** used in a service, similar to think in *cache memory*.
- In some cases, this pattern lets **add additional logic** (like *logging* or *security*) to an existing logic **without** change original class.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

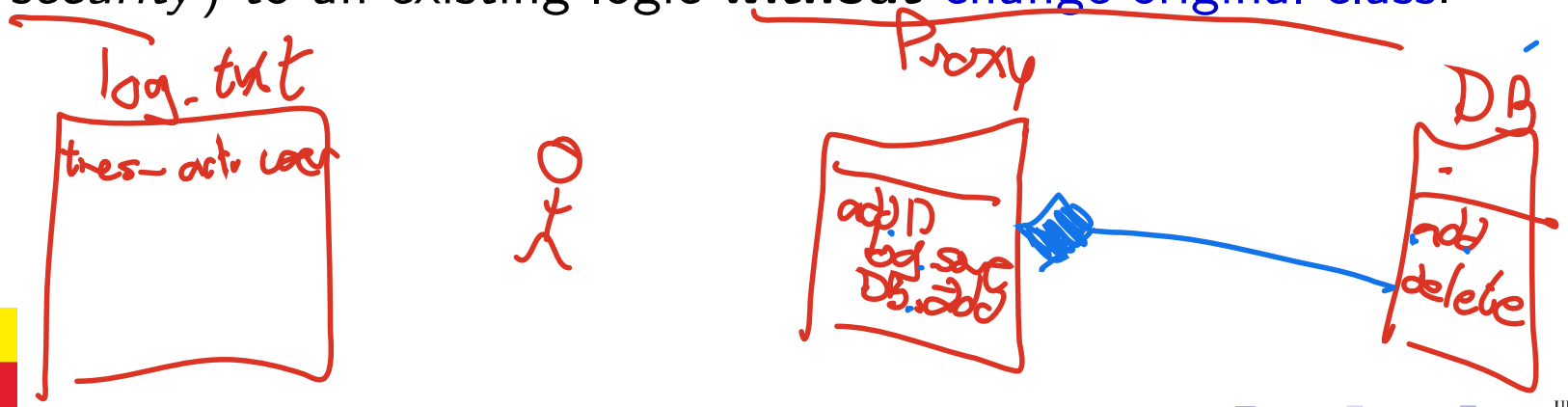
Proxy Pattern — Concepts

- This pattern lets to provide a **substitute** for an object. In this way, access coud be controlled.
- It is useful when you want to **add a level of indirection** to control access to an object. Is is like a add middle layer without affect previous logic.
- Also, it is useful when you want to **reduce memory** used in a service, similar to think in *cache memory*.
- In some cases, this pattern lets **add additional logic** (like *logging* or *security*) to an existing logic without change original class.



Proxy Pattern — Concepts

- This pattern lets to provide a **substitute** for an object. In this way, access coud be controlled.
- It is useful when you want to **add a level of indirection** to control access to an object. Is is like a add middle layer without affect previous logic.
- Also, it is useful when you want to **reduce memory** used in a service, similar to think in *cache memory*.
- In some cases, this pattern lets **add additional logic** (like *logging* or *security*) to an existing logic **without** change original class.



Proxy Pattern — Classes Structure

Do you remember Mini Me from *Austin Powers*?

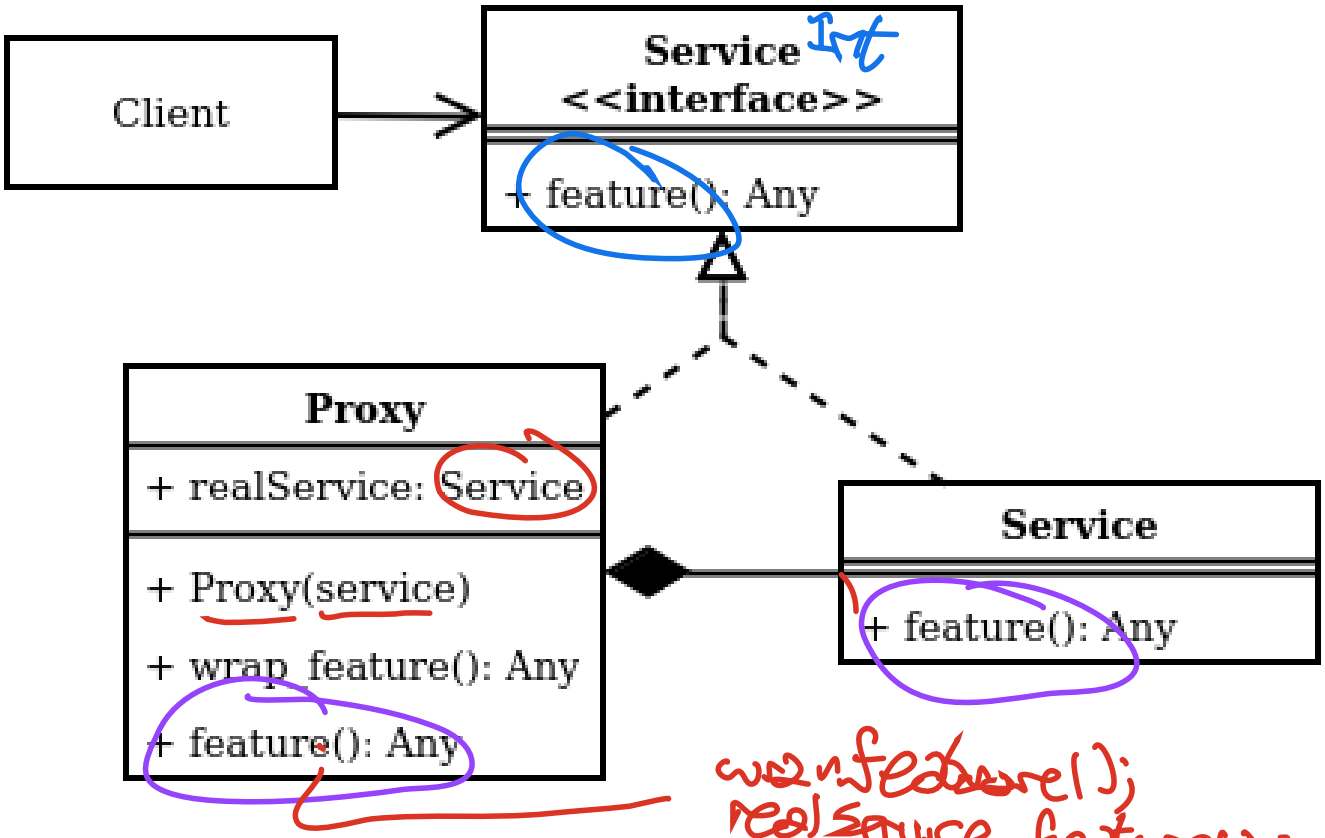
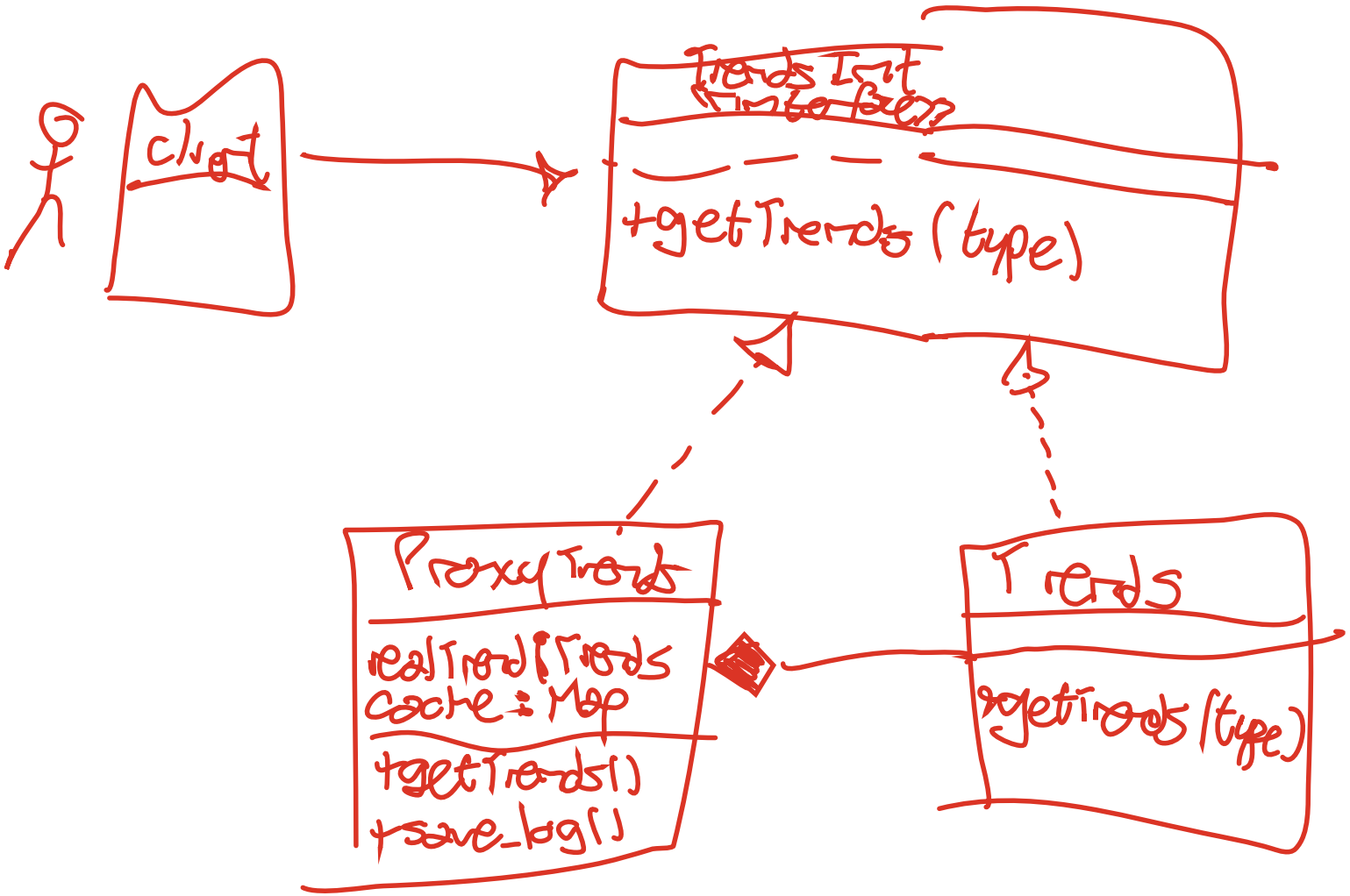


Figure: Proxy Class Diagram

Proxy Pattern Example: Cache Trends on a Social Networks



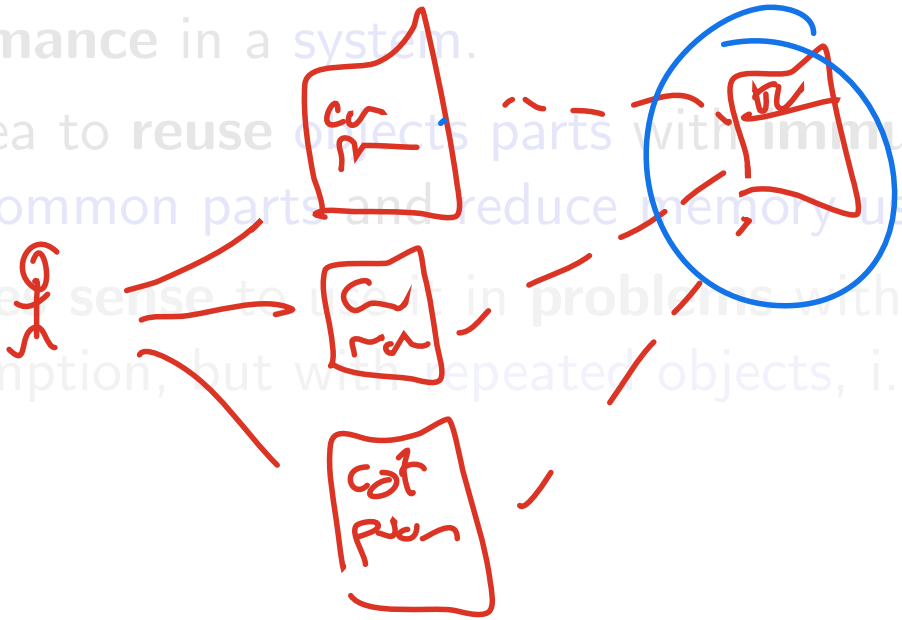
Outline

1. Introduction
2. Patterns
   - Bridge
   - Composite
   - Proxy
   - Flyweight
   - Decorator*
   - Adapter*
   - Facade*
3. Conclusions

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Flyweight Pattern — Concepts

- This pattern lets you use **sharing** to support large numbers of fine-grained objects efficiently.
- It is useful when you want to **reduce memory usage** and **increase performance** in a system.
- The idea to **reuse** objects parts with **immutable** state. This lets share common parts and reduce memory usage.
- It **makes sense to use** in **problems with high memory** comsumption, but with repeated objects, i.e. some *simulations*.



Flyweight Pattern — Concepts

- This pattern lets you use **sharing** to support large numbers of fine-grained objects efficiently.
- It is useful when you want to **reduce memory usage** and **increase performance** in a system.
- The idea to **reuse** objects parts with **immutable** state. This lets share common parts and reduce memory usage.
- It **makes sense** to use it in **problems** with **high memory** comsumption, but with repeated objects, i.e. some *simulations*.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

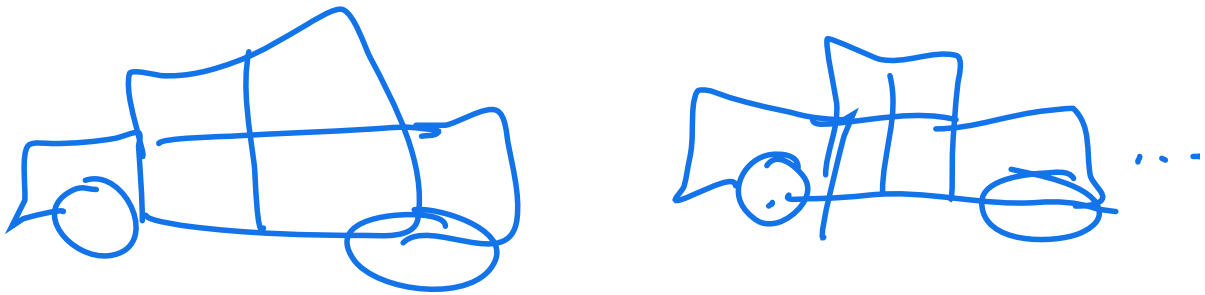
Flyweight Pattern — Concepts

- This pattern lets you use **sharing** to support large numbers of fine-grained objects efficiently.
- It is useful when you want to **reduce memory usage** and **increase performance** in a system.
- The idea to **reuse** objects parts with **immutable** state. This lets share common parts and reduce memory usage.
- It **makes sense** to use it in **problems** with **high memory** comsumption, but with repeated objects, i.e. some *simulations*.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Flyweight Pattern — Concepts

- This pattern lets you use **sharing** to support large numbers of fine-grained objects efficiently.
- It is useful when you want to **reduce memory usage** and **increase performance** in a system.
- The idea to **reuse** objects parts with **immutable** state. This lets share common parts and reduce memory usage.
- It **makes sense** to use it in **problems** with **high memory** comsumption, but with repeated objects, i.e. some *simulations*.



Flyweight Pattern — Classes Structure

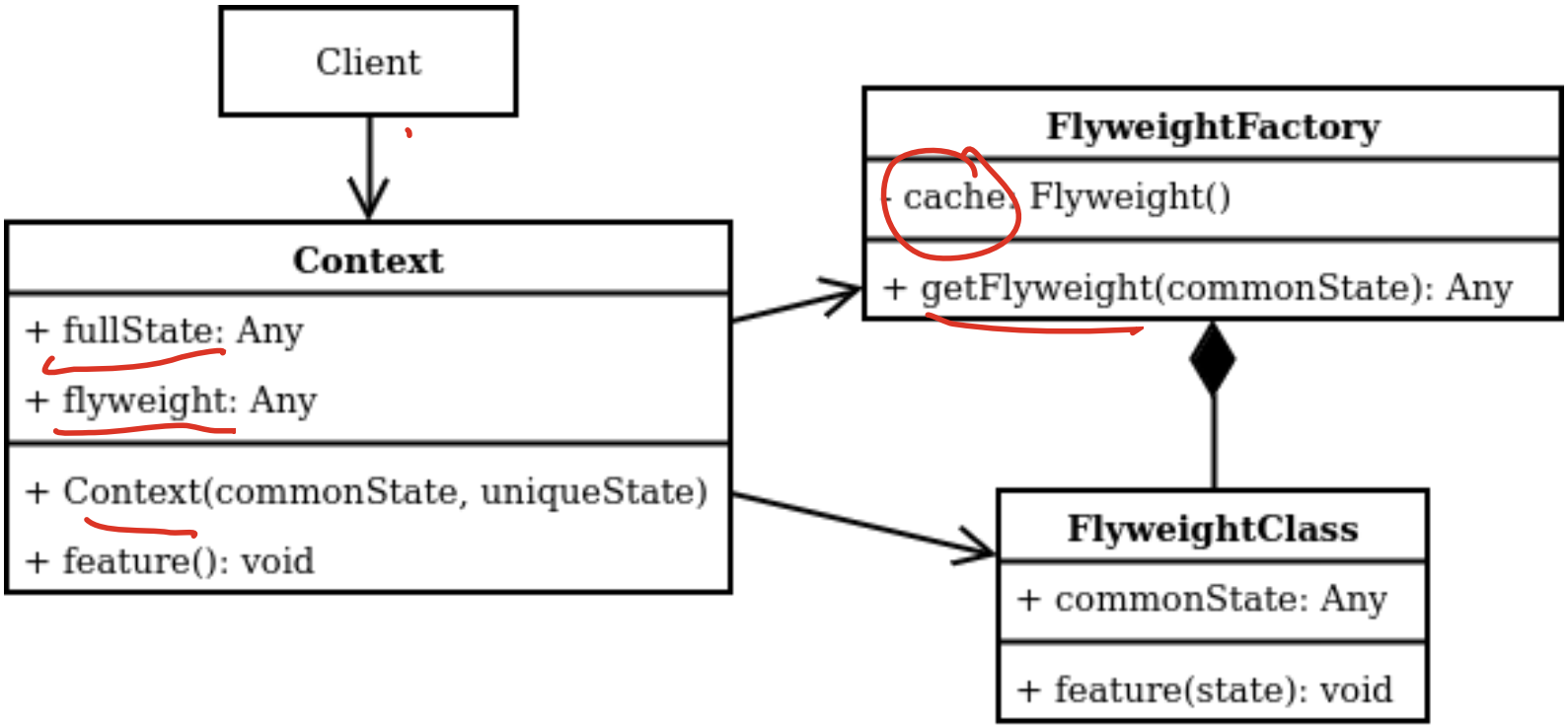
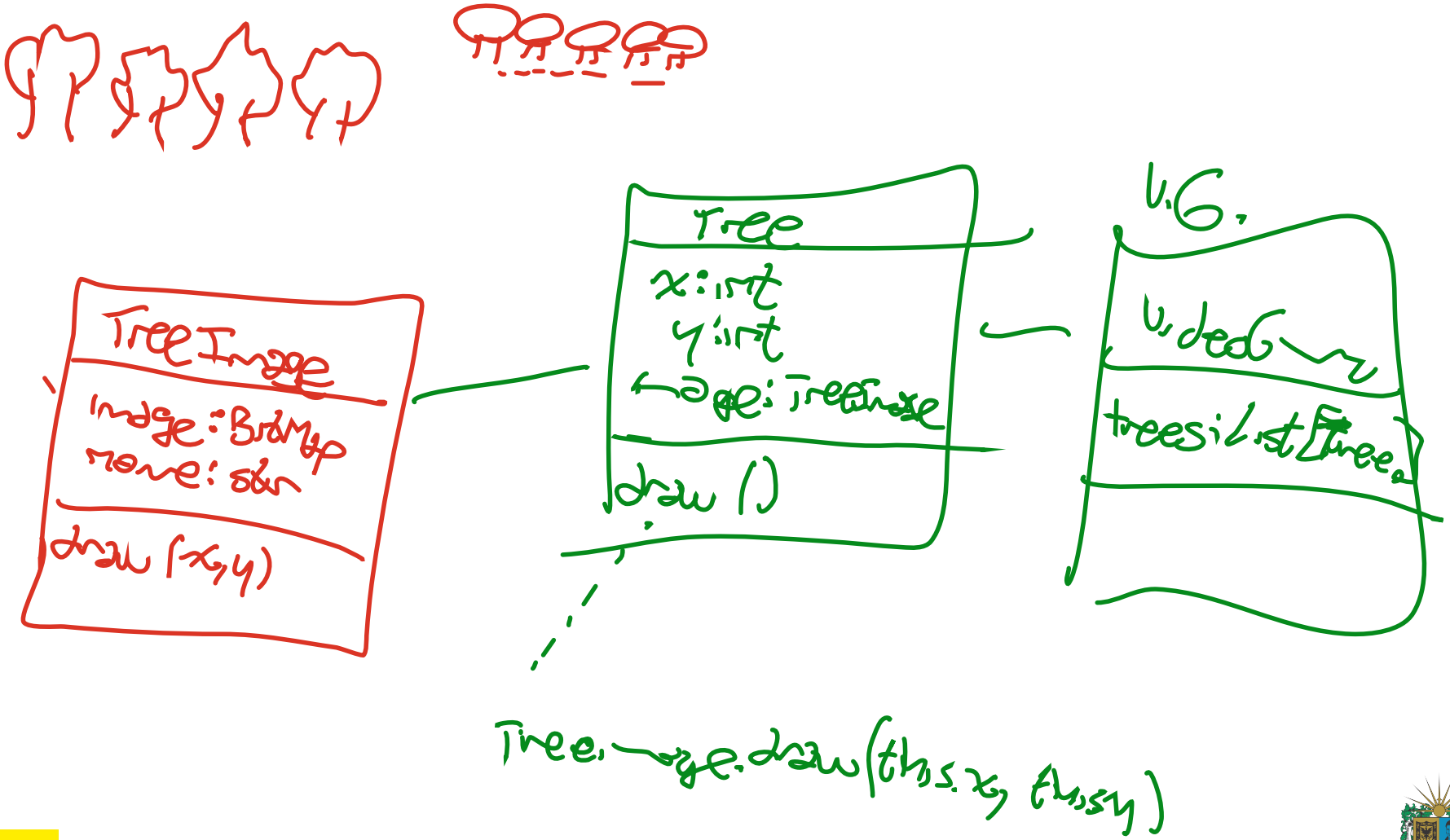


Figure: Flyweight Pattern Class Diagram

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Flyweight Pattern Example: Draw a Forest in a VideoGame



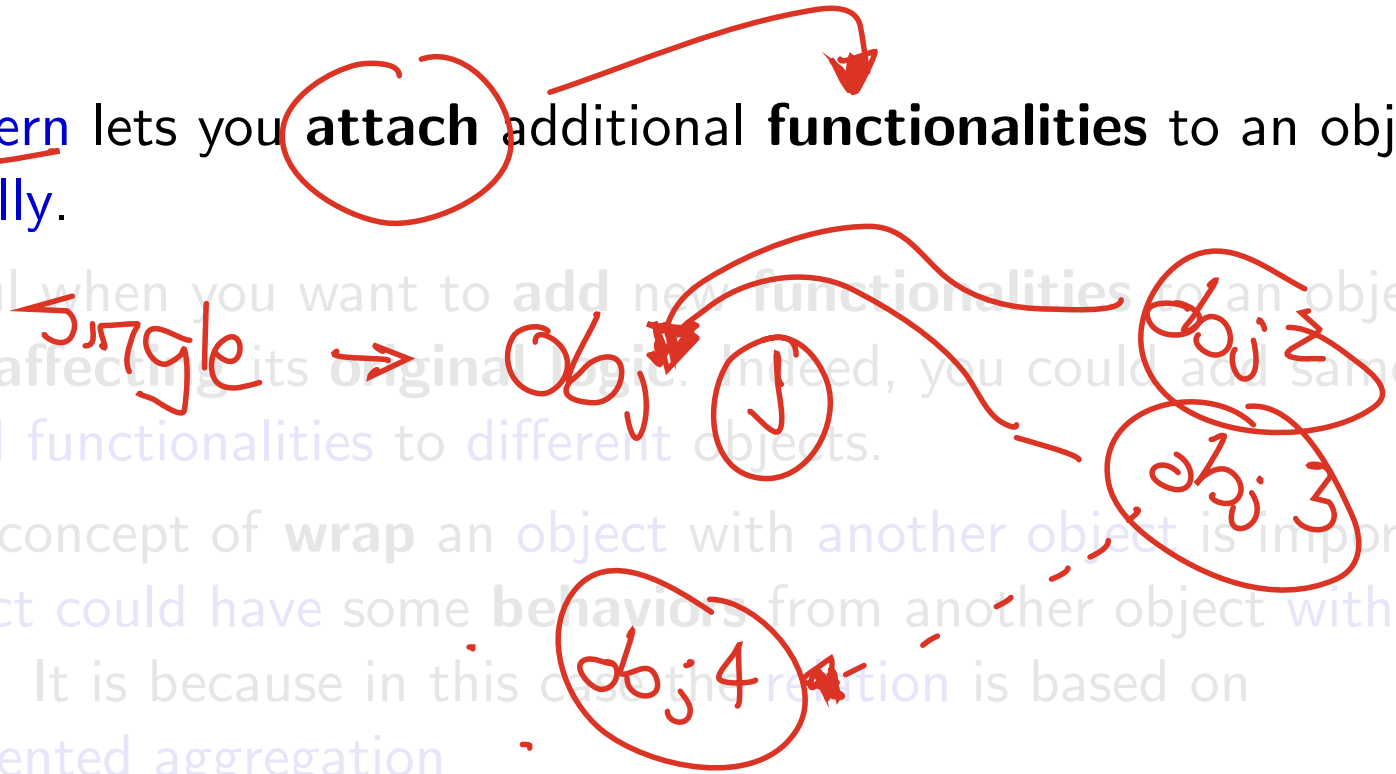
Outline

1. Introduction
2. Patterns
   - Bridge
   - Composite
   - Proxy
   - Flyweight
   - Decorator*
   - Adapter*
   - Facade*
3. Conclusions

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Decorator Pattern — Concepts

- This pattern lets you **attach** additional **functionalities** to an object dynamically.
- It is useful when you want to **add** new **functionalities** to an object **without affecting** its **original** object. Indeed, you could add same additional functionalities to different objects.
- Here the concept of **wrap** an object with another object is important. One object could have some **behaviors** from another object without heritance. It is because in this case the relation is based on object-oriented aggregation



UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Decorator Pattern — Concepts

- This pattern lets you **attach** additional **functionalities** to an object dynamically.
- It is useful when you want to **add** new **functionalities** to an object **without affecting** its **original logic**. Indeed, you could add same additional functionalities to different objects.
- Here the concept of **wrap** an object with another object is important. One object could have some **behaviors** from another object without heritance. It is because in this case the relation is based on object-oriented aggregation

Decorator Pattern — Concepts

- This pattern lets you **attach** additional **functionalities** to an object dynamically.
- It is useful when you want to **add** new **functionalities** to an object **without affecting** its **original logic**. Indeed, you could add same additional functionalities to different objects.
- Here the concept of **wrap** an object with another object is important. One object could have some **behaviors** from another object without heritance. It is because in this case the relation is based on object-oriented aggregation → weakness

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Decorator Pattern — Classes Structure

It is like Dr. Strange and his **Cloak of Levitation**.

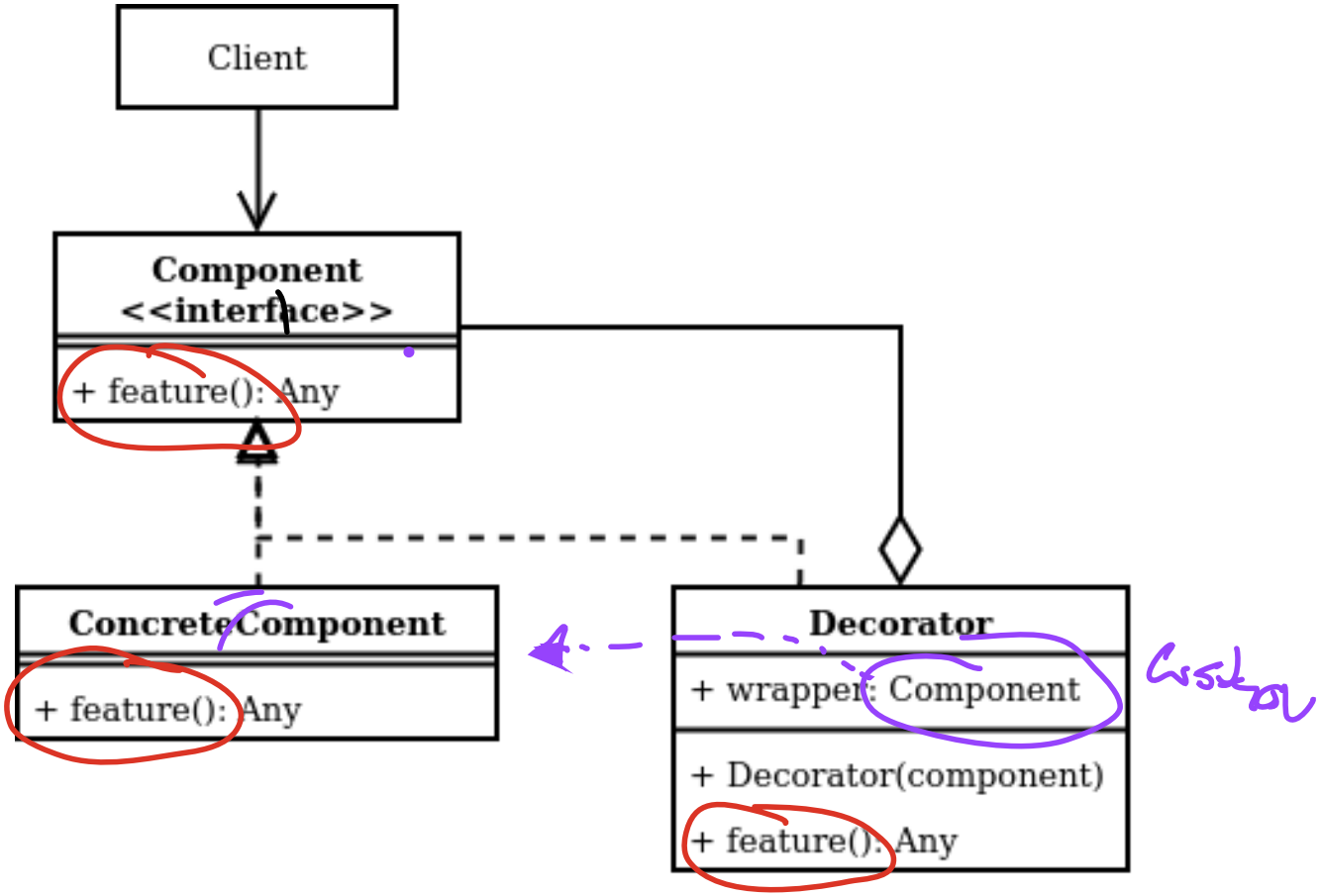
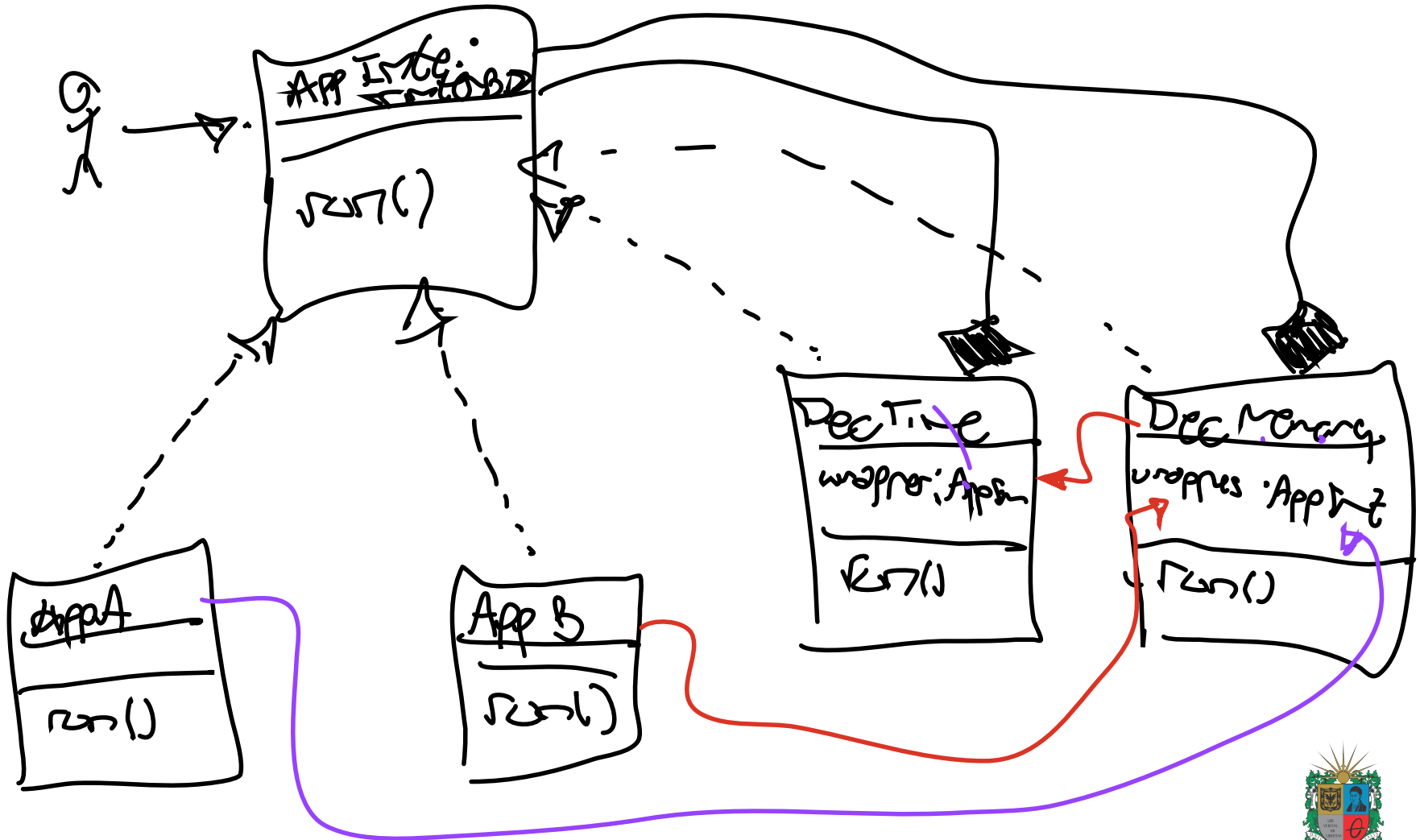


Figure: Decorator Pattern Class Diagram

Decorator Pattern Example: Monitoring an Application



Outline

1. Introduction
2. Patterns
   - Bridge
   - Composite
   - Proxy
   - Flyweight
   - Decorator*
   - Adapter*
   - Facade*
3. Conclusions

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Adapter Pattern — Concepts

- This pattern is pretty simple, it just attempts to **convert** the **interface** of a class into **another interface** clients expects.
- It is useful when you want to **reuse** an existing class, but its **not compatible** with the rest of your code, or at least where you need it.
- It is normal when you want to process **different data sources**, or to **upgrade** an existing system with new functionalities or technologies.
- In increases compatibility, and lets define an architecture based on **interfaces** and not on concrete classes.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Adapter Pattern — Concepts

- This pattern is pretty simple, it just attempts to **convert** the **interface** of a class into **another interface** clients expects.
- It is useful when you want to **reuse** an existing class, but its **not compatible** with the rest of your code, or at least where you need it.
- It is normal when you want to process **different data sources**, or to **upgrade** an existing system with new functionalities or technologies.
- In increases compatibility, and lets define an architecture based on **interfaces** and not on concrete classes.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Adapter Pattern — Concepts

- This pattern is pretty simple, it just attempts to **convert** the **interface** of a class into **another interface** clients expects.
- It is useful when you want to **reuse** an existing class, but its **not compatible** with the rest of your code, or at least where you need it.
- It is normal when you want to process **different data sources**, or to **upgrade** an existing system with new functionalities or technologies.
- In increases compatibility, and lets define an architecture based on **interfaces** and not on concrete classes.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Adapter Pattern — Concepts

- This pattern is pretty simple, it just attempts to **convert** the **interface** of a class into **another interface** clients expects.
- It is useful when you want to **reuse** an existing class, but its **not compatible** with the rest of your code, or at least where you need it.
- It is normal when you want to process **different data sources**, or to **upgrade** an existing system with new functionalities or technologies.
- In increases compatibility, and lets define an architecture based on **interfaces** and not on concrete classes.

Adapter Pattern — Classes Structure

Now technology is based in adapters to make everything **compatible**.

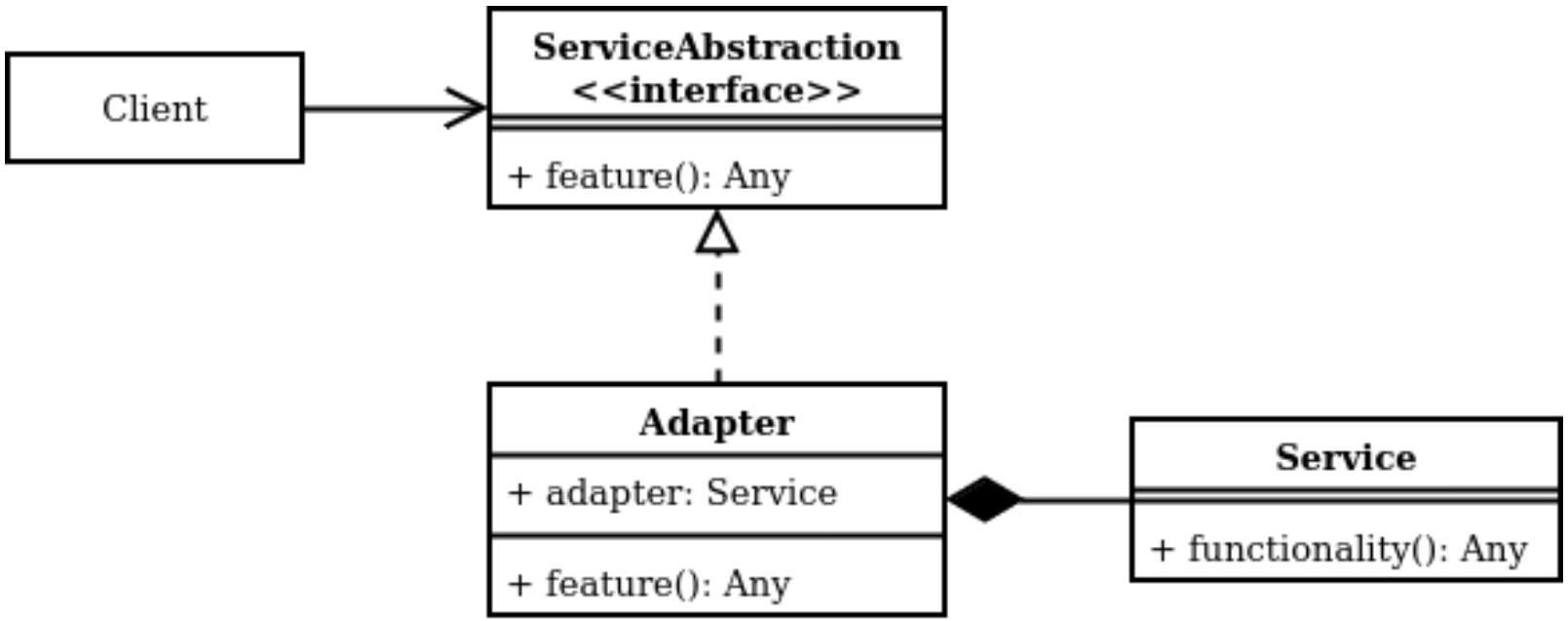


Figure: Adapter Pattern Class Diagram

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Adapter Pattern Example: Processing different File Sources

Outline

1. Introduction
2. Patterns
   - Bridge
   - Composite
   - Proxy
   - Flyweight
   - Decorator*
   - Adapter*
   - Facade*
3. Conclusions

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Facade Pattern — Concepts

- This pattern provides a **unified interface** to a set of classes that could be group into a subsystem.
- It is useful when you want to **define a high-level interface** that makes the subsystem **easier to use**. It means, hide any complex logicand let the client use a **simple interface**.
- It is normal when you want to **reduce** the **dependencies**. The client just interacts with the facade, and the **facade interacts with the subsystem**.
- You could add complexity at the subsystem and **client will not be affected**, it increases flexibility. At most, there will be more new functionalities to be exposed to the client.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Facade Pattern — Concepts

- This pattern provides a **unified interface** to a set of classes that could be group into a subsystem.
- It is useful when you want to **define a high-level interface** that makes the subsystem **easier to use**. It means, hide any complex logicand let the client use a **simple interface**.
- It is normal when you want to **reduce** the **dependencies**. The client just interacts with the facade, and the **facade interacts with the subsystem**.
- You could add complexity at the subsystem and **client will not be affected**, it increases flexibility. At most, there will be more new functionalities to be exposed to the client.

Facade Pattern — Concepts

- This pattern provides a **unified interface** to a set of classes that could be group into a subsystem.
- It is useful when you want to **define a high-level interface** that makes the subsystem **easier to use**. It means, hide any complex logicand let the client use a **simple interface**.
- It is normal when you want to **reduce** the **dependencies**. The client just interacts with the facade, and the **facade interacts with the subsystem**.
- You could add complexity at the subsystem and **client will not be affected**, it increases flexibility. At most, there will be more new functionalities to be exposed to the client.

Facade Pattern — Concepts

- This pattern provides a **unified interface** to a set of classes that could be group into a subsystem.
- It is useful when you want to **define a high-level interface** that makes the subsystem **easier to use**. It means, hide any complex logicand let the client use a **simple interface**.
- It is normal when you want to **reduce** the **dependencies**. The client just interacts with the facade, and the **facade interacts with the subsystem**.
- You could add complexity at the subsystem and **client will not be affected**, it increases flexibility. At most, there will be more new functionalities to be exposed to the client.

Facade Pattern — Classes Structure

You are the only one who knows how to find something in your bedroom.

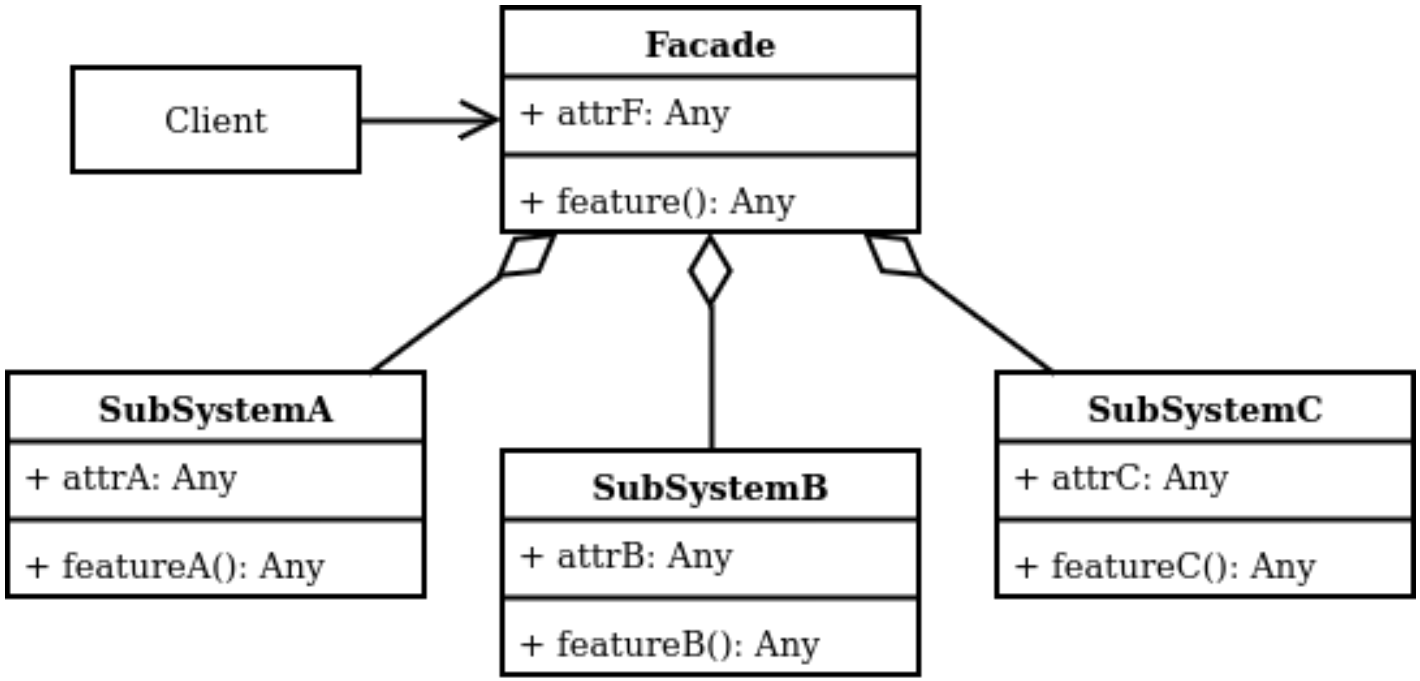


Figure: Facade Pattern Class Diagram

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Facade Pattern Example: Bank Account Management

Outline

1 Introduction

2 Patterns
- Bridge
- Composite
- Proxy
- Flyweight
- Decorator*
- Adapter*
- Facade*

3 Conclusions

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Conclusions

- **Structural patterns** are useful to describe how ocreate a clonebjects are connected to each other.
- They are related to the **design principles** of descomposition and generalization.
- You could fix a lot of problems with these **patterns** as a nice solution. However, be careful with the complexity of the solution.
- The idea is to make the **system** flexible, reusable, and easy to maintain.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Conclusions

- **Structural patterns** are useful to describe how ocreate a clonebjects are connected to each other.
- They are related to the **design principles** of descomposition and generalization.
- You could fix a lot of problems with these **patterns** as a nice solution. However, be careful with the complexity of the solution.
- The idea is to make the **system** flexible, reusable, and easy to maintain.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Conclusions

- **Structural patterns** are useful to describe how ocreate a clonebjects are connected to each other.
- They are related to the **design principles** of descomposition and generalization.
- You could fix a lot of problems with these **patterns** as a nice solution. However, be careful with the complexity of the solution.
- The idea is to make the **system** flexible, reusable, and easy to maintain.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Conclusions

- **Structural patterns** are useful to describe how ocreate a clonebjects are connected to each other.
- They are related to the **design principles** of descomposition and generalization.
- You could fix a lot of problems with these **patterns** as a nice solution. However, be careful with the complexity of the solution.
- The idea is to make the **system** flexible, reusable, and easy to maintain.

Outline

1. Introduction
2. Patterns
   - Bridge
   - Composite
   - Proxy
   - Flyweight
   - Decorator*
   - Adapter*
   - Facade*
3. Conclusions

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Thanks!

Questions?

Repo: *https://github.com/EngAndres/ud-public/tree/main/courses/software-modeling*

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS