Structural Patterns

Lect. PhD. Arthur Molnar

Babes-Bolyai University

Overview

Intro

- Concerned with how classes are composed to form larger structures.
- We have *class patterns* (inheritance), and *object patterns* (composition)
- Many of these patterns are related, and some of them we can find in others (hence their ordering)

Adapter

- What is an **adapter**? (non CS explanation)
- Why do we need them?
- Adapter allows classes with incompatible interfaces to work together (without source code changes)

Adapter pattern

Convert the interface of a class into another interface expected by clients.

Adapter

Motivating example:

- Let's consider a drawing editor for lines, polygons, ...
- The editor works with a *Shape* abstract base class
- Concrete elements subclass *Shape* (e.g. *LineShape*, *RectShape*, etc)
- *TextShape* is more interesting, as its implementation is more difficult
- Luckily (**!**), we've got a GUI library providing a *TextView* class - it's just what we need, but *Shape* and *TextView* don't know each other

Adapter

What to do, what to do?

1. Change *TextView* to conform to *Shape*? (why, **why not?**)
2. Introduce an adapter between the seemingly unrelated classes - enter *TextShape*

Adapter

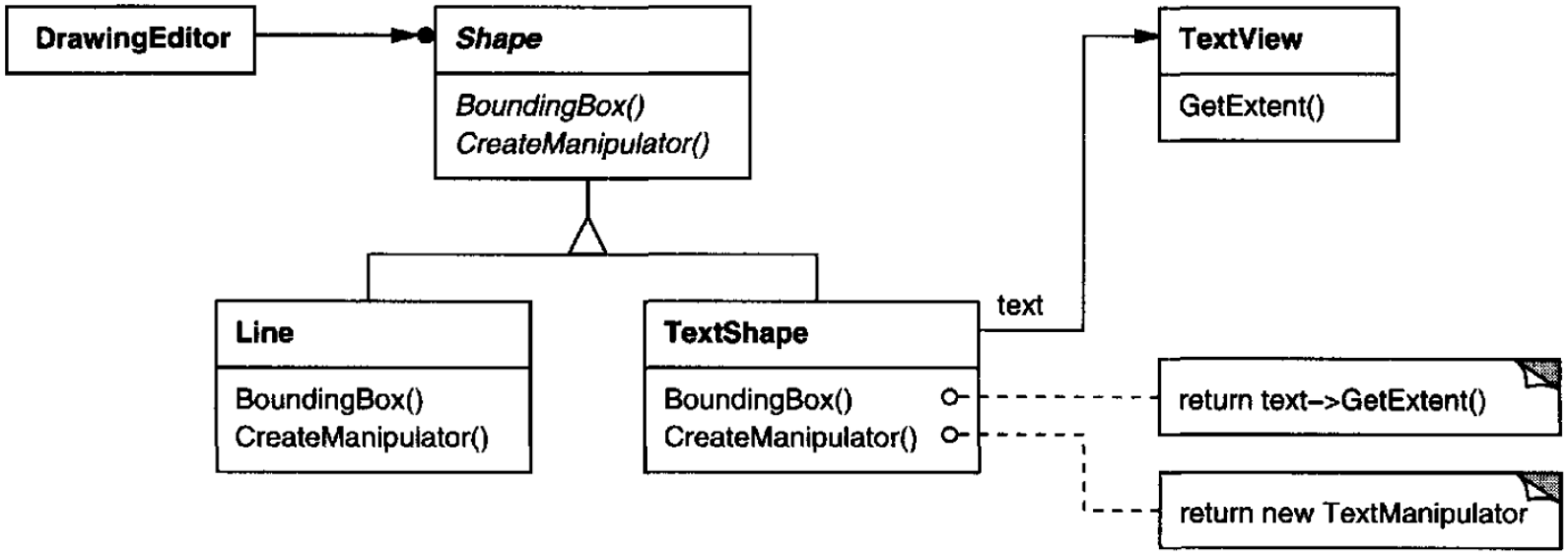

Figure: From[1]

Adapter

1. *BoundingBox()* messages are converted to *GetExtent()*
2. *CreateManipulator()* converted to the new *TextManipulator()* implementation
3. The difficulty in designing the adapter depends on the level of mismatch between **target** and **adaptee**

Adapter

Two possible implementations - **class adapter**

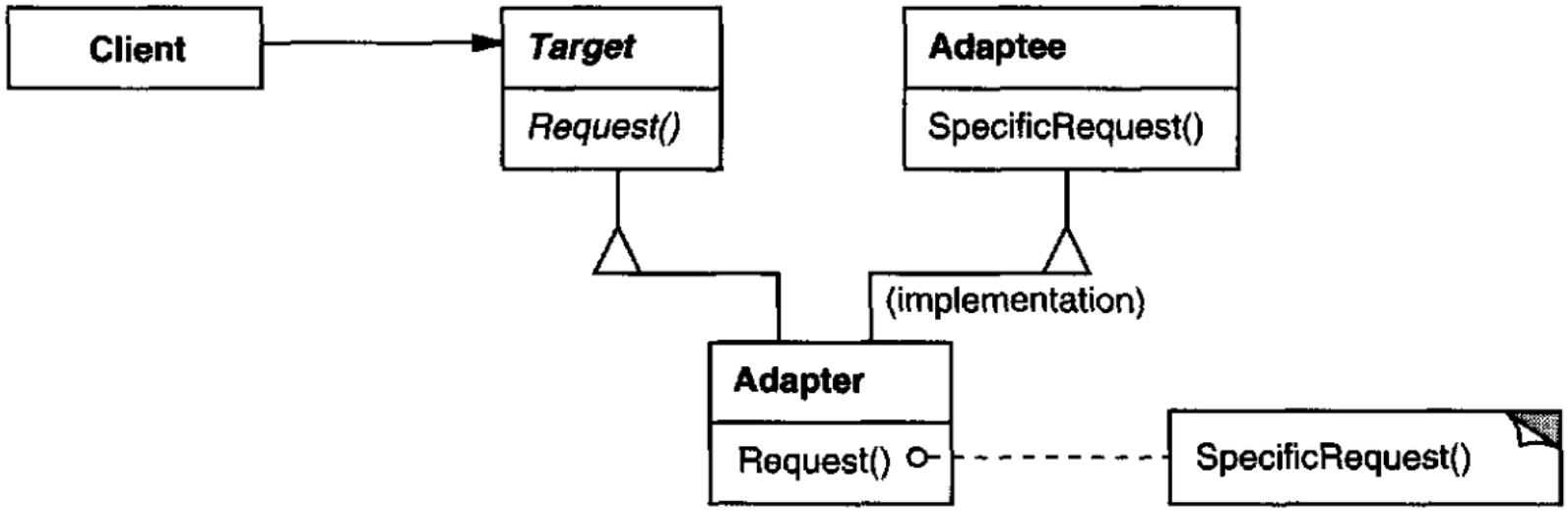

Figure: From[1]

Adapter

Two possible implementations - **object adapter**

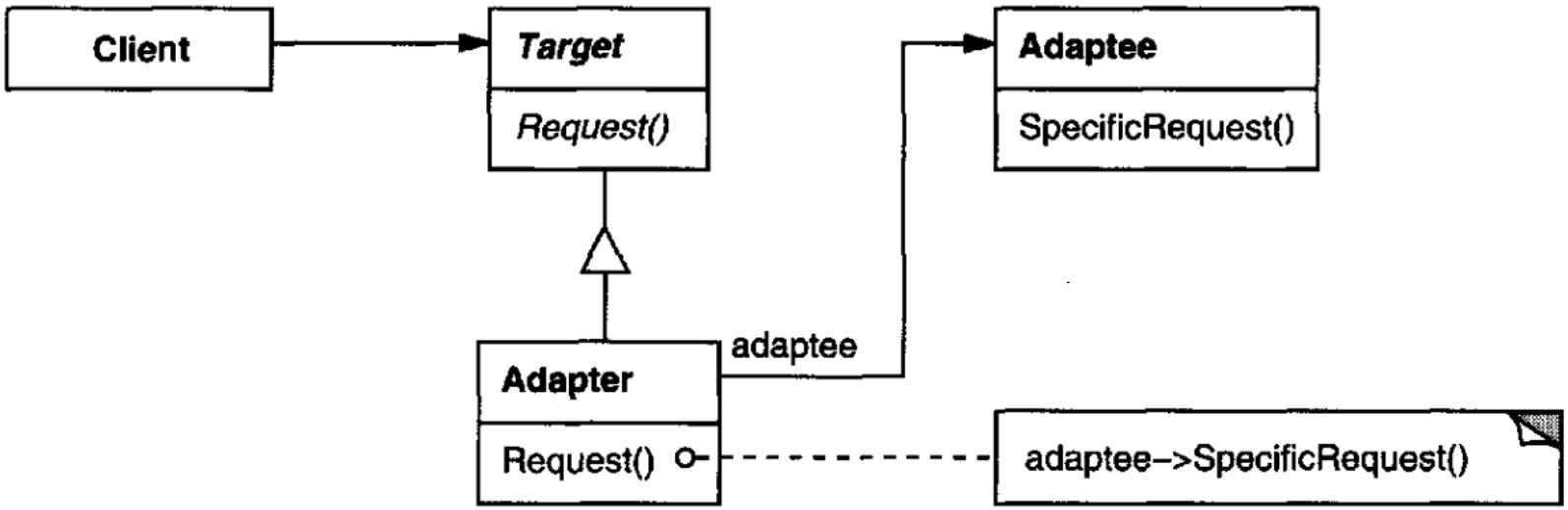

Figure: From[1]

Adapter

1. Clients call the *Adapter*, and it calls *Adaptee* operations in turn
2. Class adapters commit to a concrete *Adaptee* class, less flexibility when we want to adapt *Adaptee* subclasses
3. Your mileage may vary based on difference between *Target* and *Adaptee*
4. Two-way adapters can be created, making both *Target* and *Adaptee* work with each other

Adapter

1. Pluggable adapters incorporates interface adaptation (more details in [1])

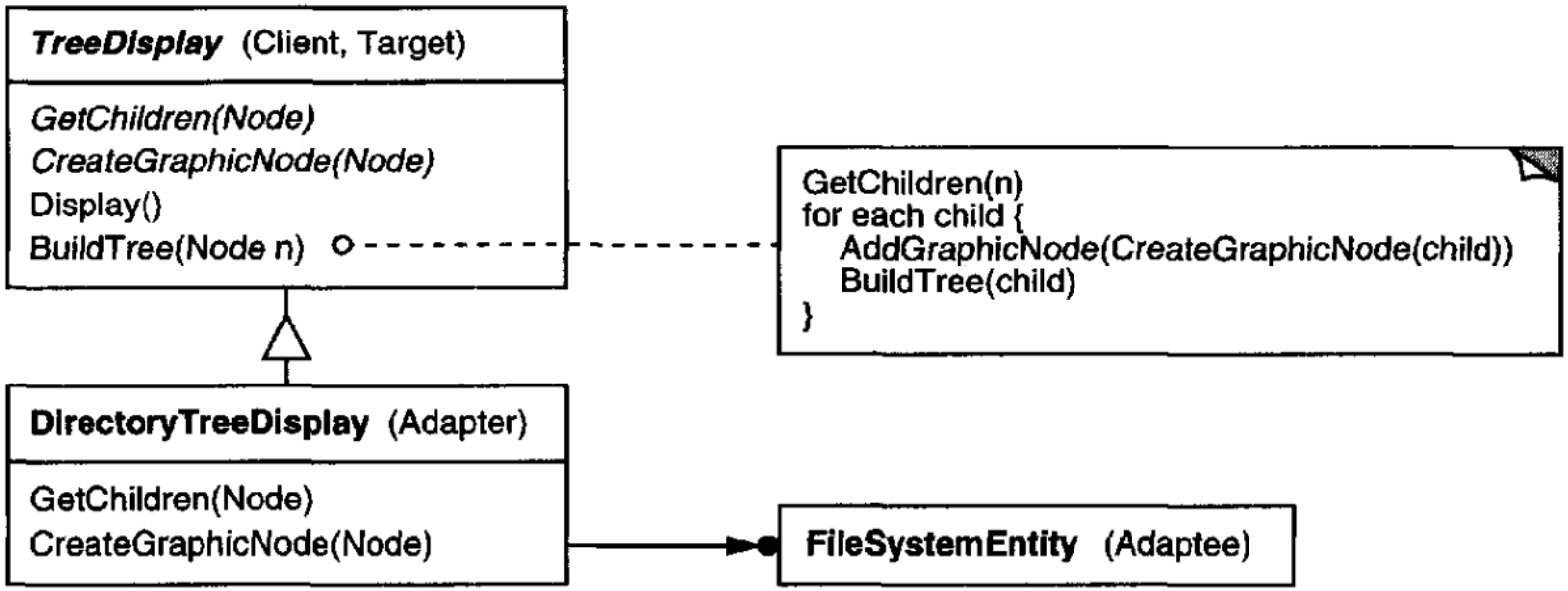

Figure: From [1]

Adapter example code

Source code

git: /src/ubb/dp/structural/Adapter

Bridge

- When an abstraction can have multiple implementations, we usually use inheritance, using *interfaces* or *abstract base classes*
- Inheritance glues abstraction and implementation together

Bridge

Motivating example:

- Implementation of a *Window* abstraction for a GUI toolkit
- We want it to work on multiple platforms (e.g. *X Window System* and *IBM Presentation Manager*)
- Define abstract *Window* class and subclass it:
  - Results in *XWindow* and *PMWindow*
  - Classes that extend *Window* have to be implemented in both frameworks
  - Client code is platform dependent

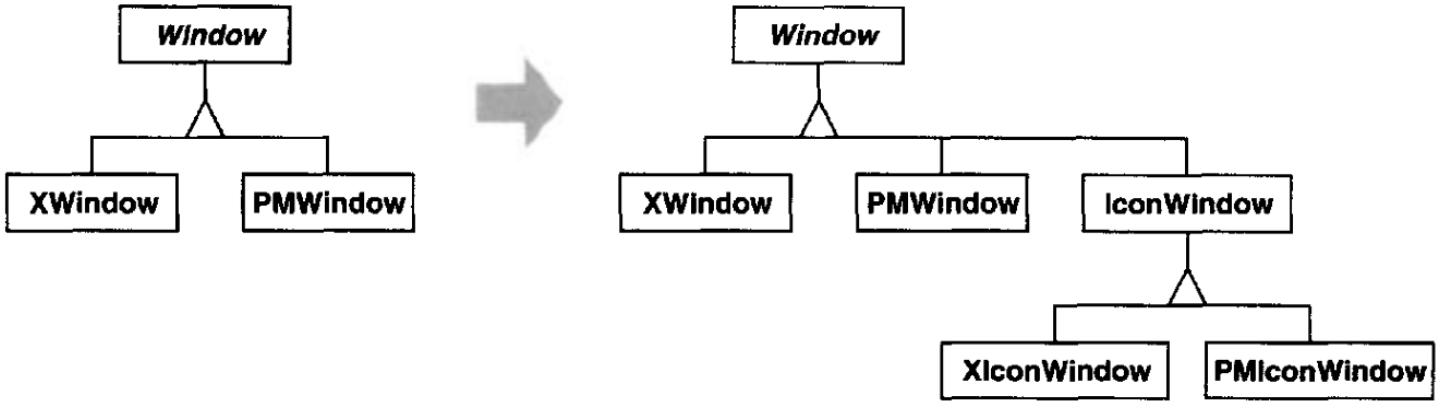

Figure: From [1]

Bridge

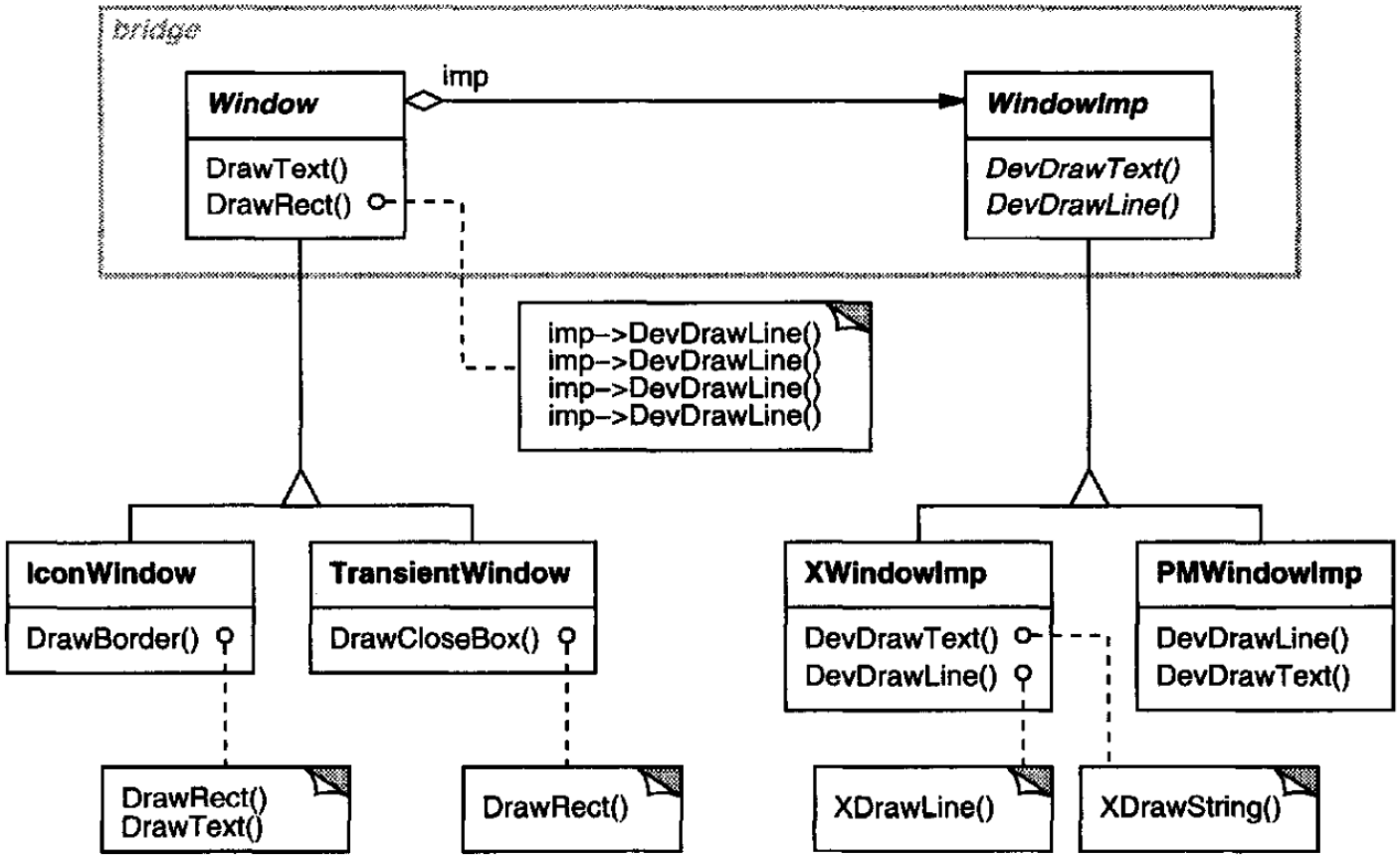

Figure: From [1]

Bridge

- *Bridge* separates the abstraction and its implementation into separate class hierarchies
- We have a *WindowImp* class as a platform agnostic root class
- *Window* subclass operations are implemented in terms of abstract operations in *WindowImp*.
- The **bridge** exists between *Window* and *WindowImp*, and it is between abstraction and implementation

Bridge

General case:

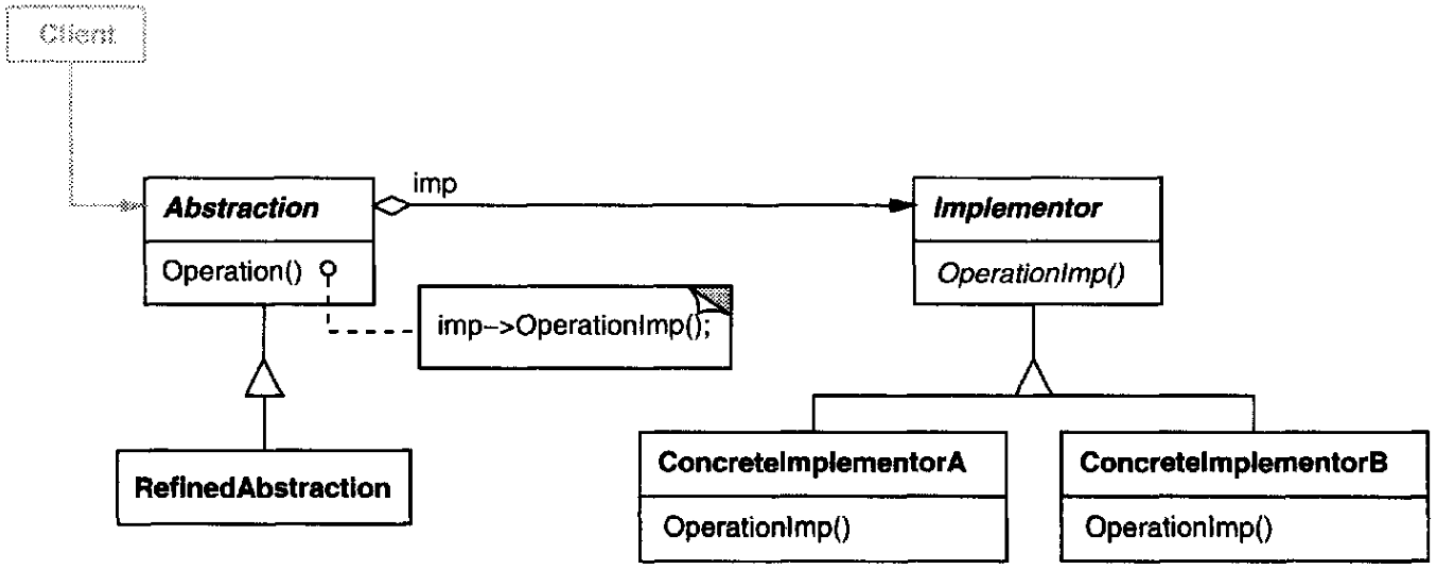

Figure: From [1]

Bridge

When and how to use:

- Decouple interface and implementations; this allows you to vary the implementation at run-time (e.g. use *Swing*, *JavaFX* or *SWT* windows)
- A proliferation of classes, such as in the first example
- Decision about which implementation to use can be taken using a *Factory* approach in the *Window* class constructor

Bridge example code

Source code

git: /src/ubb/dp/structural/Bridge

Composite

Composite pattern

Compose objects into tree structures to represent part-whole hierarchies. Clients treat compositions and individual objects uniformly.

Composite

Motivating example:

- Let's consider a graphical editor, supporting lines, shapes, text and pictures
- Components can be grouped to form larger components (**e.g.** shape built using multiple lines)
- Treating all components the same way simplifies *client* code greatly
- **The key:** use an abstract class to represent both *primitive components*, as well as *compositions*

Composite

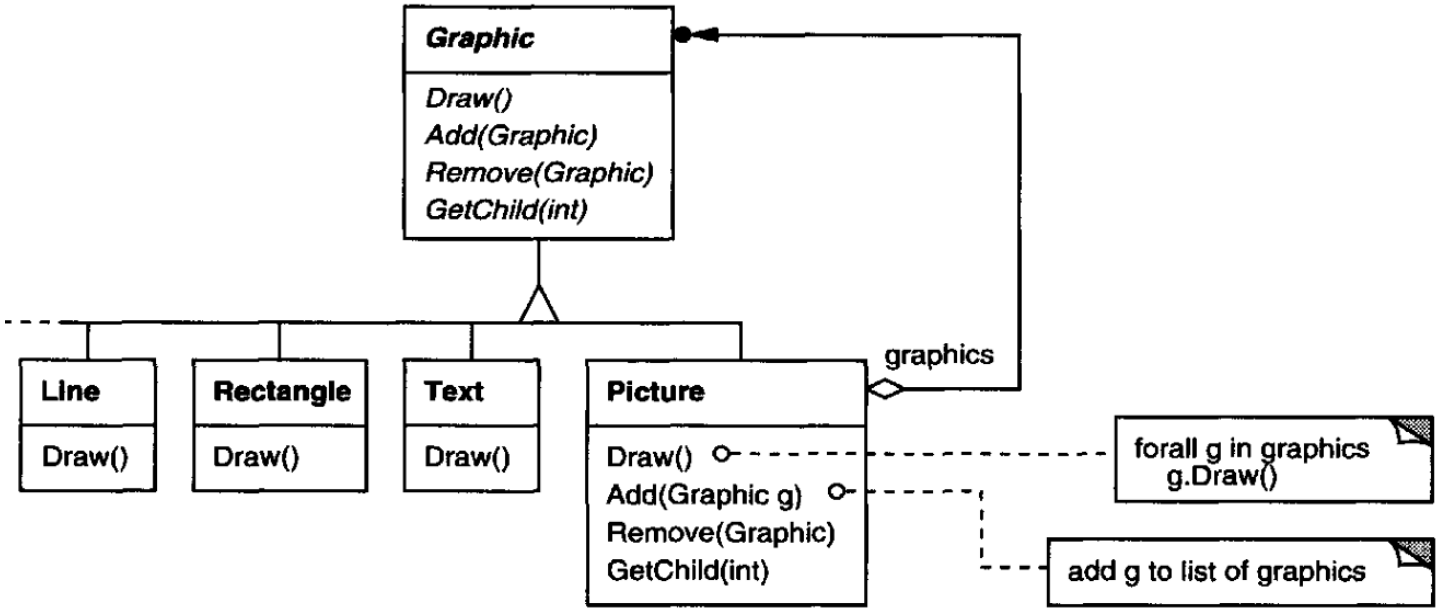

Figure: From [1]

Composite

Motivating example:

- *Graphic* class includes operations for management of its children
- *Line*, *Rectangle*, *Text* are primitive components, and can draw themselves using *Draw()*
- Primitive classes do not have children be definition
- *Picture* defines an aggregation of *Graphic* objects, and can be used recursively

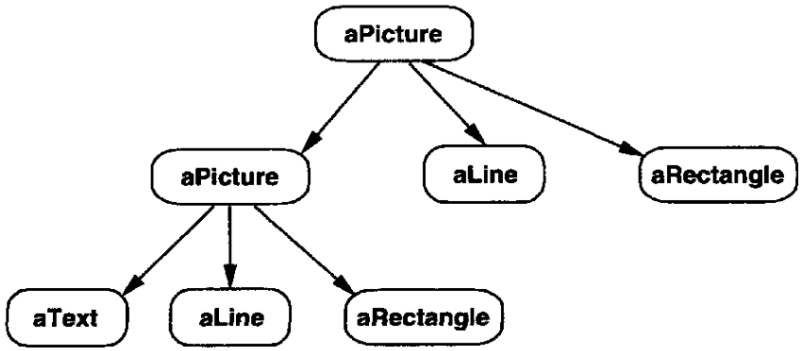

Figure: From [1]

Composite

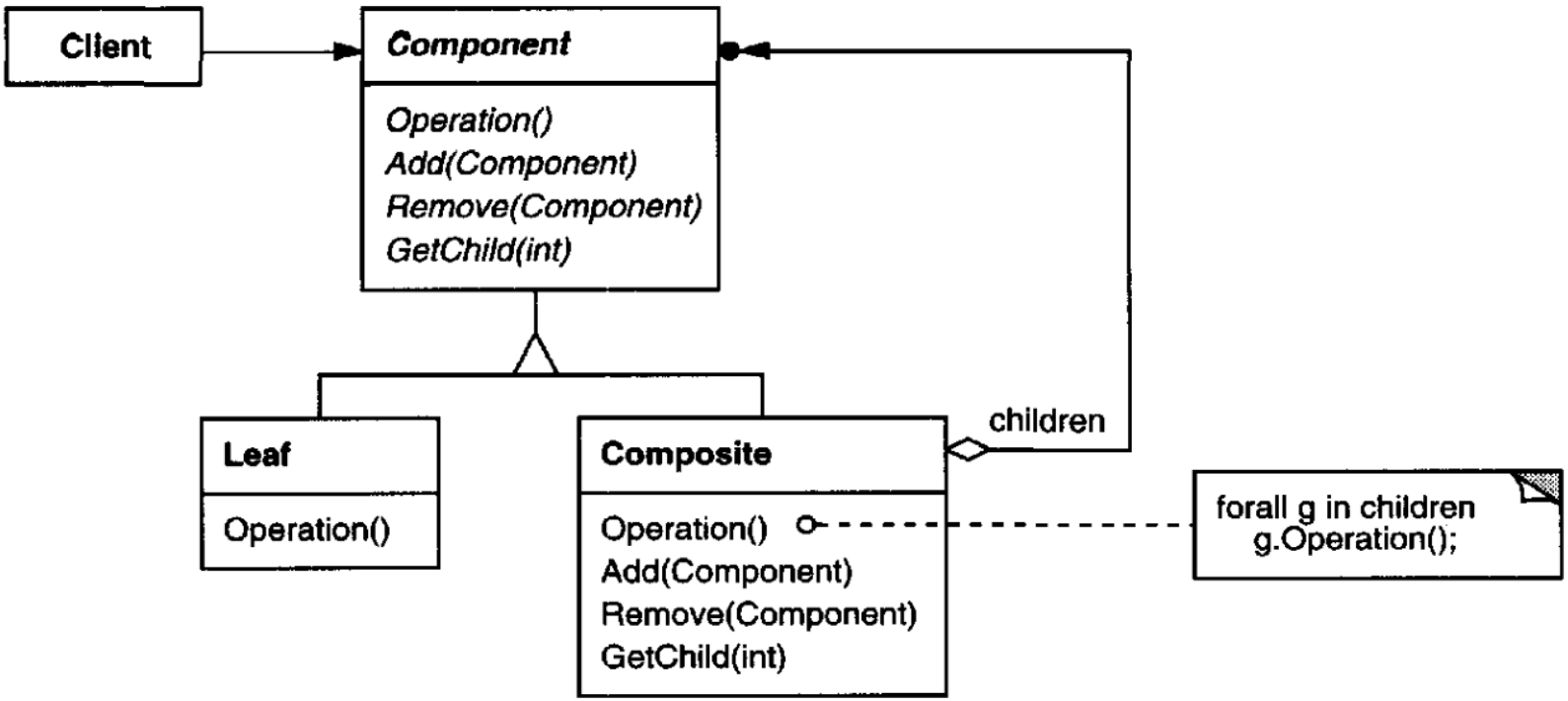

Figure: General case (from [1])

Composite

Consequences:

- \+ Simplifies clients, as they no longer care about the exact type of object they have
- \+ New leaves can be added without additional changes
- – Design might be too general, as you cannot restrict composite components (**e.g.** GUI widget hierarchies in the abstract factory pattern that cannot be mixed between platforms)

Composite

Implementation details:

- Children can also have a reference to parent, managed by *Component*
- Maintain the invariant of the parent-child relationship
- Where to define management of children?
  - **Component class**: transparent, as all classes are treated the same, but not safe, as operations on children don't make sense for leaves
  - **Composite class**: opaque, as it hidden by the component class, but safer
- Tension between maximizing the *Component* interface (generally good) and the types of leaves that can be added
- How do you know whether a component is a *Composite* without casting?

Composite example code

Source code

git: /src/ubb/dp/structural/composite

Decorator

Decorator pattern

Attach additional responsibilities to an object dynamically

- Dynamically means at runtime
- Most flexible, much more than inheritance

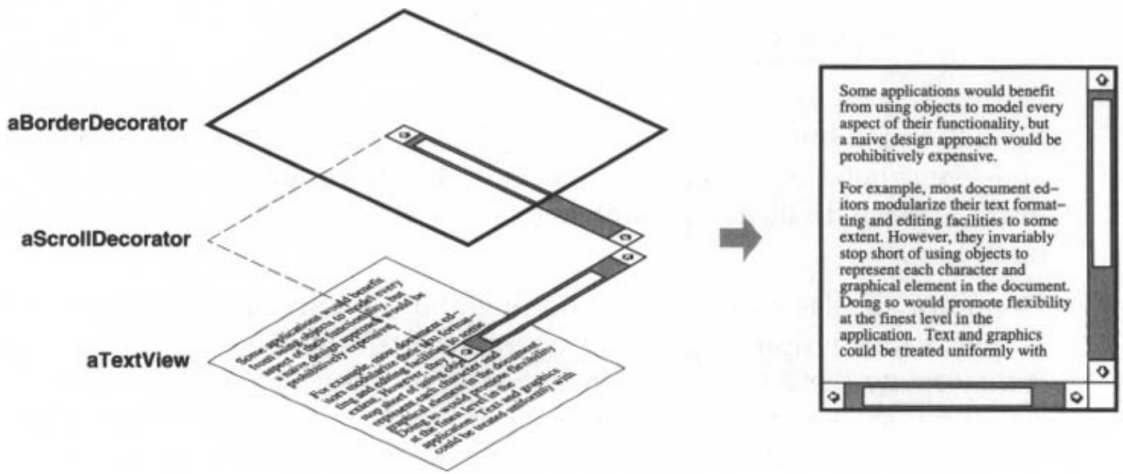

Figure: Decorator example (from [1])

Decorator

- Suppose we have a GUI component that does not support scrolling or borders (can you provide examples?)
- Sometimes we will need these additional behaviours, but not every time
- We wrap our component into a *decorator* that forwards components messages and adds its own behaviour
- Decorators are transparent to clients and can be chained recursively

Decorator

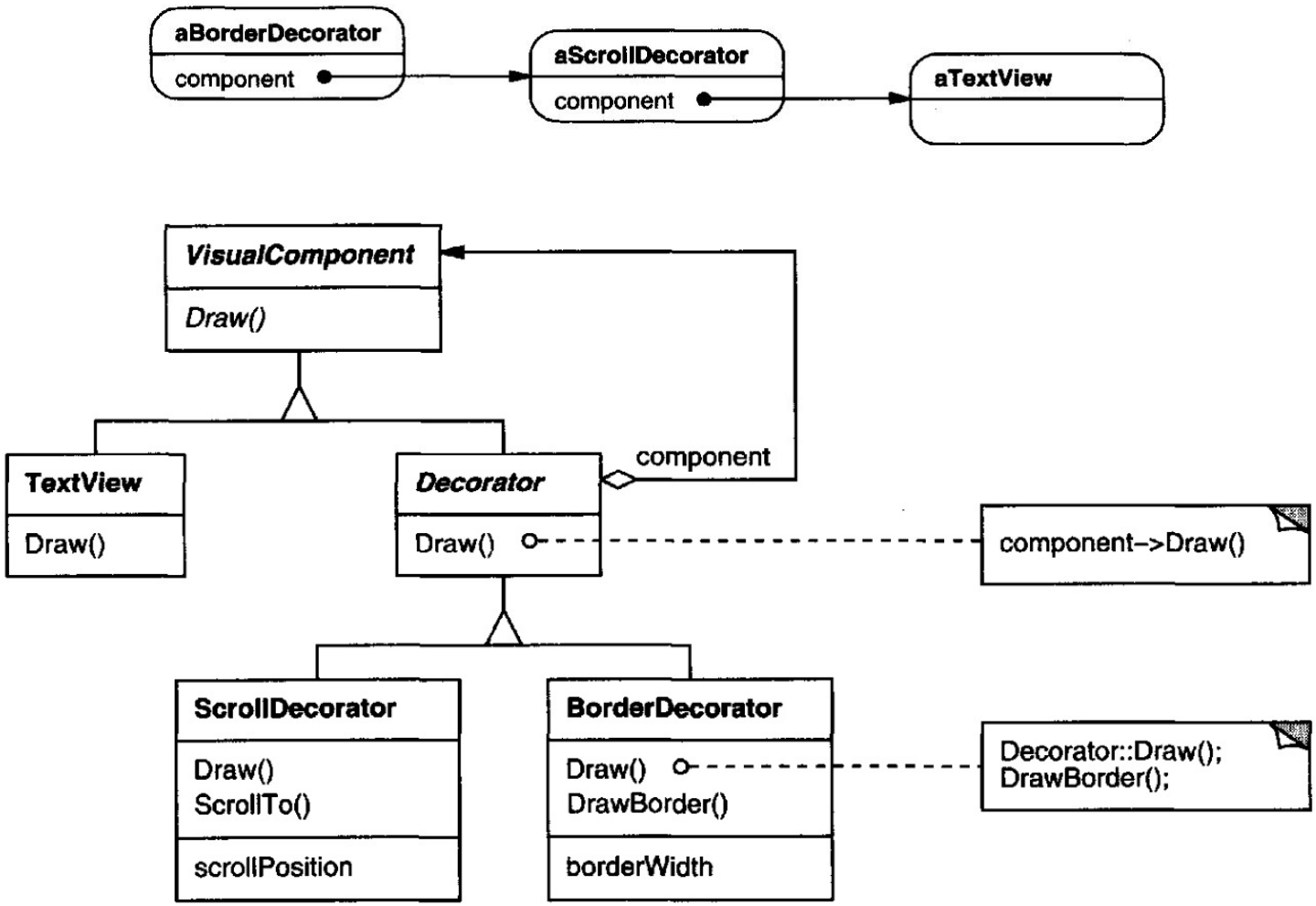

Figure: Decorator examples (from [1])

Decorator

When to use:

- Add responsibilities to individual objects transparently
- These responsibilities can be withdrawn dynamically
- When subclassing is impractical (**e.g.** result in a large number of classes, class definitions unavailable)

Decorator

+ More flexible than inheritance (**e.g.** *BorderBorderScrollablePanel* ?)
+ Only add what you need by composition
– Decorators are transparent but not equal to the decorated object (don't use object identity)

Decorator

Decorator versus *Strategy*:

- Decorator changes the skin
- Strategy changes the internals (**e.g.** a *List* class might implement the strategy pattern for sorting it)

Decorator example code

Source code

git: /src/ubb/dp/structural/DecoratorExampleComputer.java

Decorator example code

Source code

git: /src/ubb/dp/structural/DecoratorExamplePizza.java

Façade

Façade pattern

Provide a unified interface to a set of interfaces in a subsystem. Defines a higher-level interface through which the subsystem is easier to use

Façade

- Goal is to reduce apparent complexity
- Facade reduces the communication between systems - makes their interactions, and possibly the larger system, easier to understand

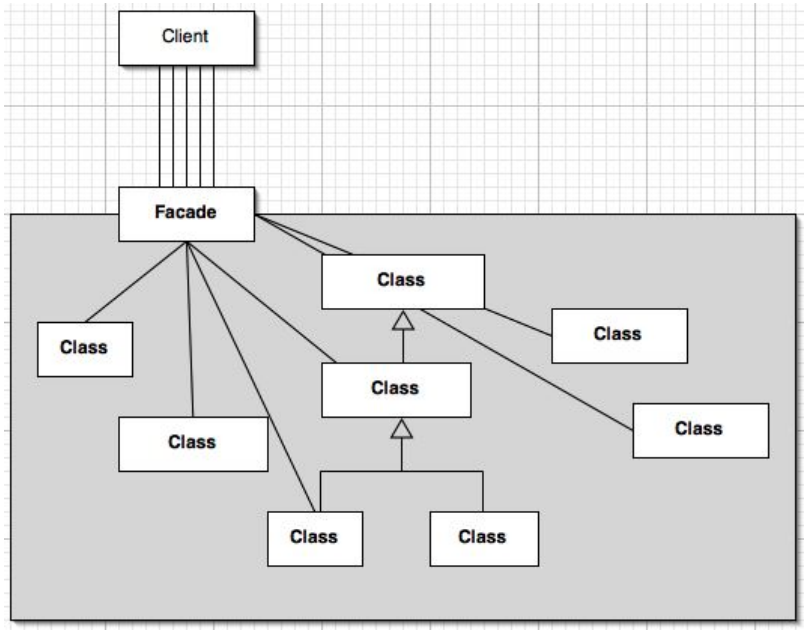

Figure: https://www.javaworld.com/article/2073463/fa–231-ade-clears-complexity.html

Façade

Example of a compiler:

- Compiler includes classes *Scanner*, *Parser*, **Node*, *NodeBuilder*, *CodeGenerator* and so on
- They all do something useful, and should be exposed
- If you implement an IDE plugin with syntax highlighting, auto-complete and incremental compiling all this comes in VERY handy
- What if you just want to compile the thing!?

Façade

The *Compiler* class is the system façade

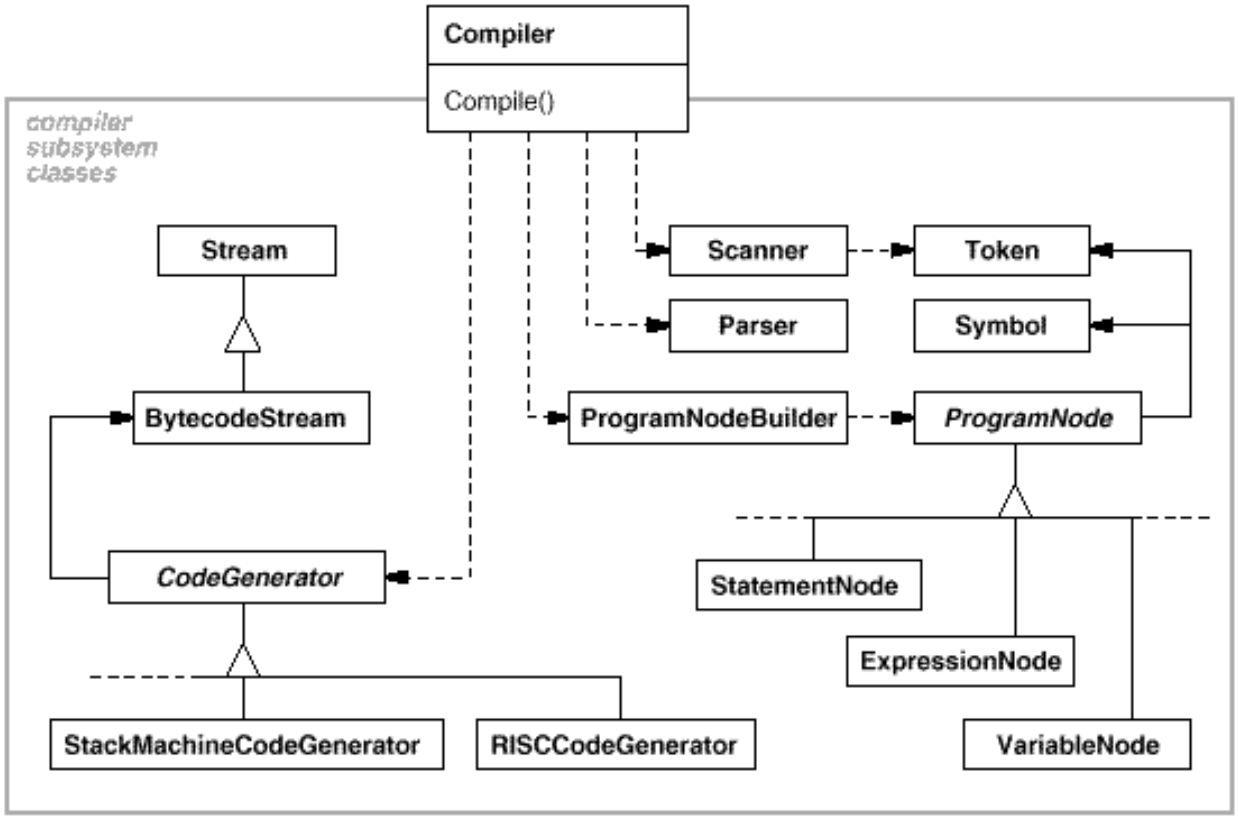

Figure: from [1]

Façade

When to use the façade pattern:

- Provide a simple, default view of a subsystem, "good enough" for most of its clients
- Reduce the number of dependencies between a subsystem and its clients
- Layer the subsystem - create façades as the entry point for each layer

Façade

Consequences:

- Shield clients from subsystems by providing a common access point for most (all?) subsystems
- Promote weak coupling, help organize a system
- You don't lose flexibility: all the nitty gritty is still there, if you need to use it

Implementation:

- You can create an abstract Façade, which you subclass depending on the view that is required by clients (**e.g.** one for compiling, one for syntax highlighting)

Façade example code

Compiler example source code

git: /src/ubb/dp/structural/FacadeCompilerExample.java

Source code

git: /src/ubb/dp/structural/FacadeComputerExample.java

Flyweight

Flyweight pattern

Share data to support a large number of instances efficiently.

Flyweight

- Everything is an object leads to a useful representation in OO languages
- If too many things are objects, you have too little memory ☺
- **e.g.** **CellRenderer* classes in Java are implemented as Flyweights

\+ Flyweight shares common attributes between instances to save memory

– More complex implementation, added coupling

Flyweight

Motivating example - a 3D shooter game with particle effects

- Naive implementation uses a complete instance for each particle
- However, certain particle classes can share state (e.g. all *bullets* look alike)

Flyweight

Naive implementation for particle system

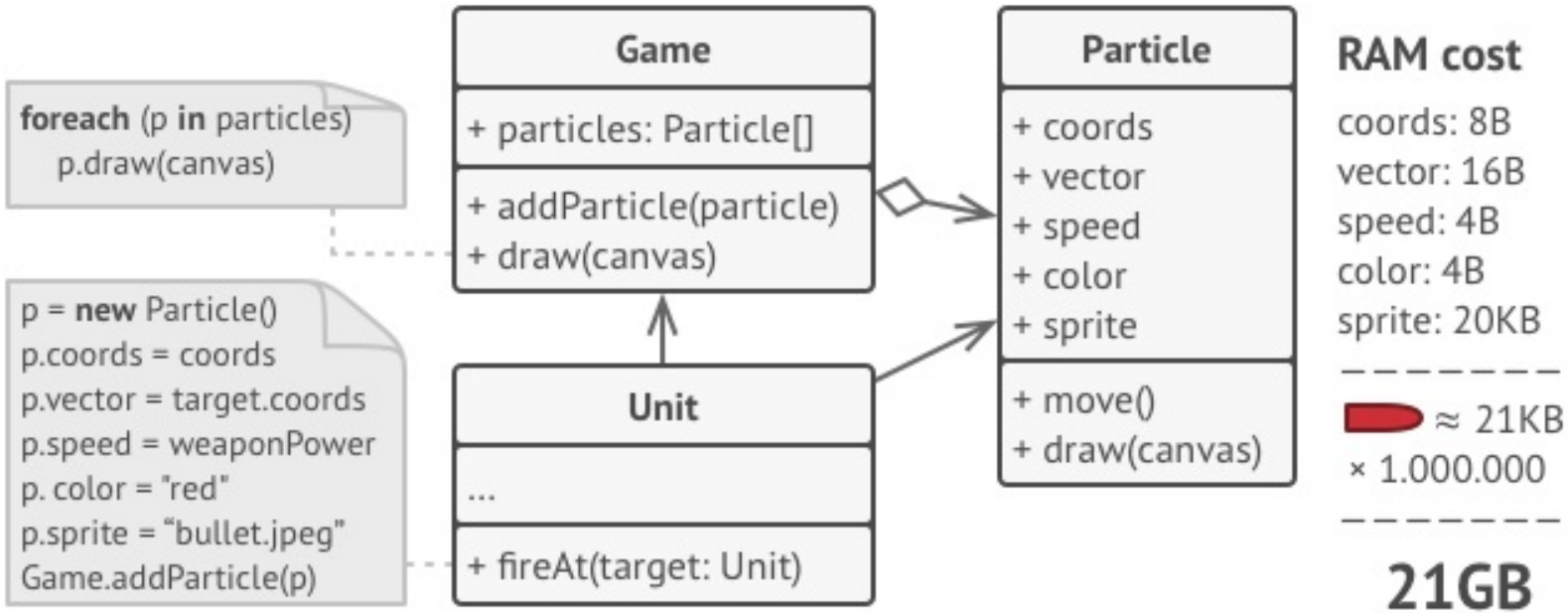

Figure: https://refactoring.guru/design-patterns/flyweight

Flyweight

Flyweight implementation:

- Realize that particle color and texture are constant for many particles
- Coordinates, movement vector are updated by the particle system

Flyweight divides instance state:

- **Intrinsic:** constant within the object, can be read but does not change
- **Extrinsic:** depends on flyweight context, is supplied from the outside

Flyweight

Flyweight particle implementation

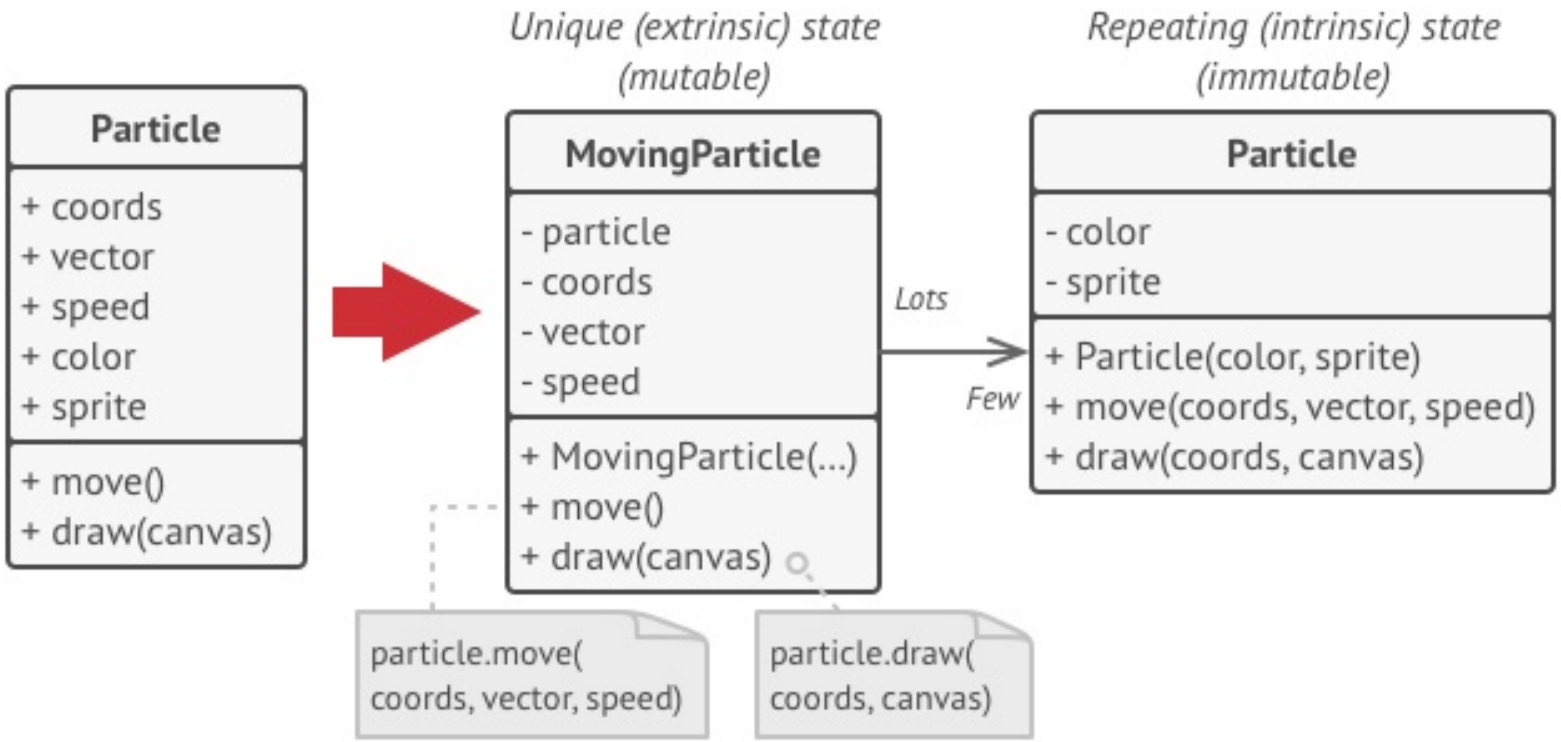

Figure: https://refactoring.guru/design-patterns/flyweight

Flyweight

Resulting savings

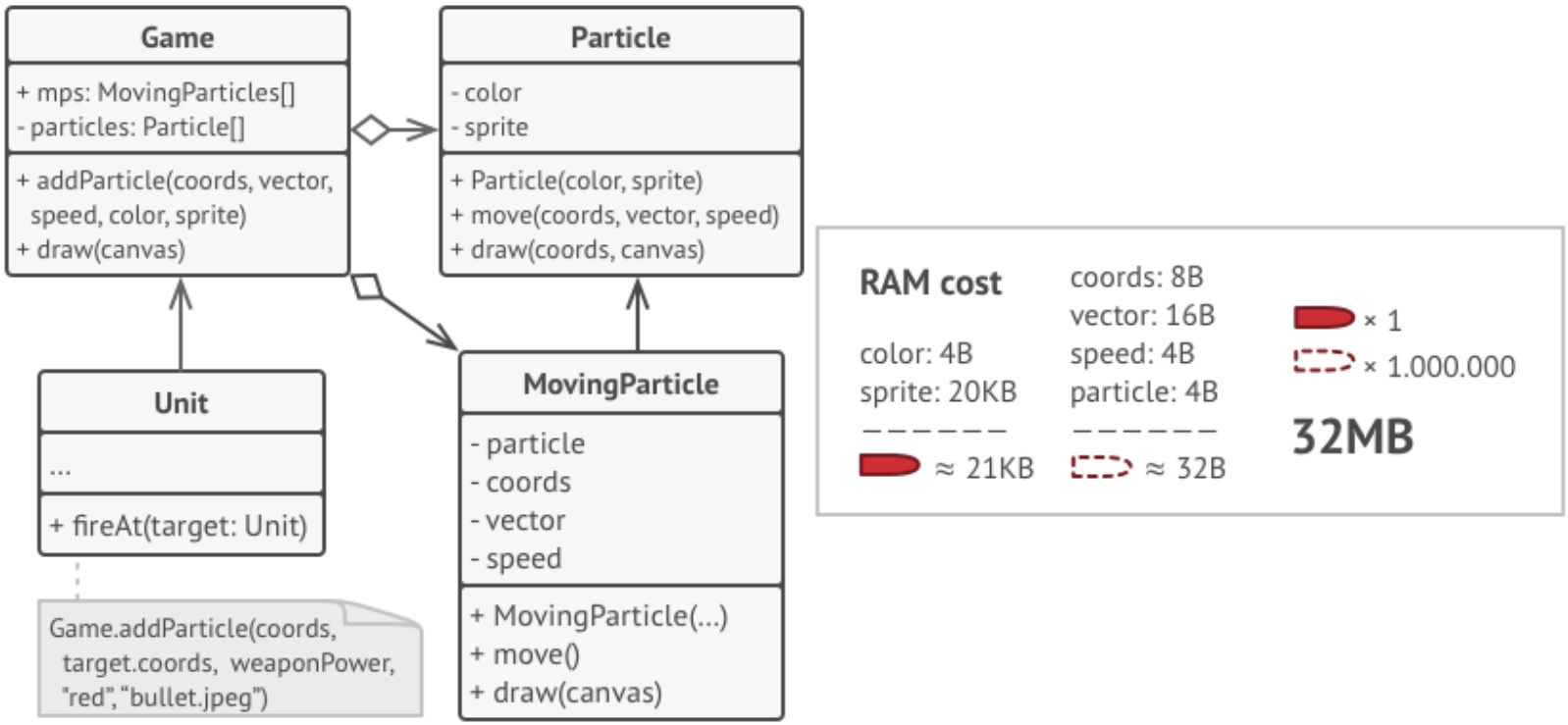

Figure: https://refactoring.guru/design-patterns/flyweight

Flyweight

Use flyweight when **all** these are true:

- Application uses a large number of objects
- Storage requirements are high
- Large groups of objects can be replaced by a small number of shared objects
- Application does not depend on object identity

– Flyweights **might** trade storage requirements with computation requirements (no such thing as free lunch)

– Flyweights **definitely** trade simplicity for storage requirements

Flyweight

Where do we store **extrinsic** state?

- In a different class, where else? ☺
- Extract extrinsic state to another object (**e.g.** *Context*)
- The class containing the extrinsic state together with the Flyweight represent a complete object
- Flyweight instances should be created using a **Factory** in order to centralize instance creation

Flyweight

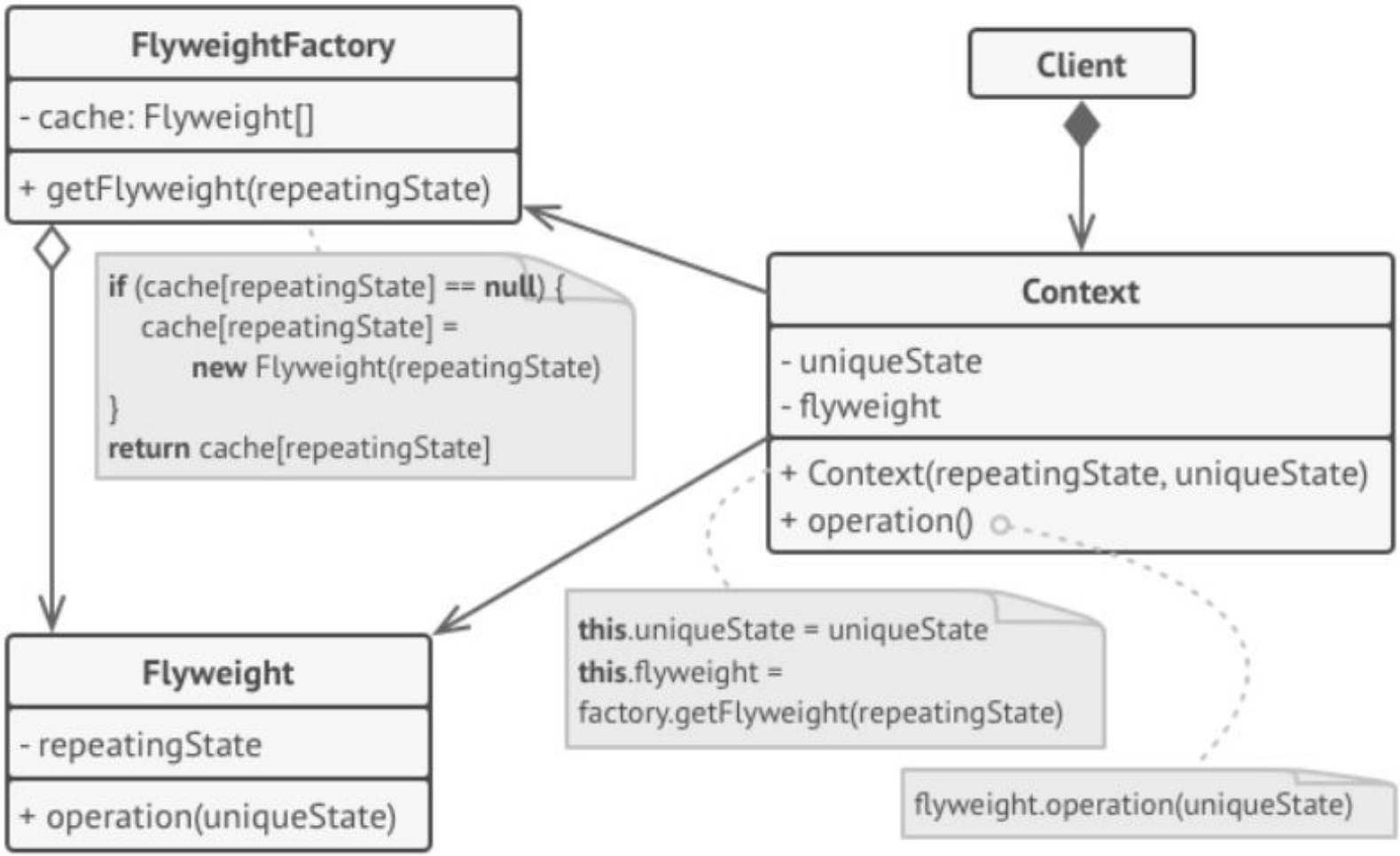

Figure: https://refactoring.guru/design-patterns/flyweight

Flyweight example code

Compiler example source code

git: /src/ubb/dp/structural/FlyweightTreeExample.java

Proxy

Proxy pattern

Provide a surrogate or placeholder for another object to control access to it.

Why would you do that?

- Lazily load expensive resources (**e.g.** email client, database BLOBS, large object hierarchies)
- Restrict access to a resource (**e.g.** check whether caller has the correct credentials for access)
- The same *proxy* class can be used for different subjects, by *programming to an interface*

Proxy

- The **proxy** object replaces the **subject**
- It forwards calls to the subject, when required

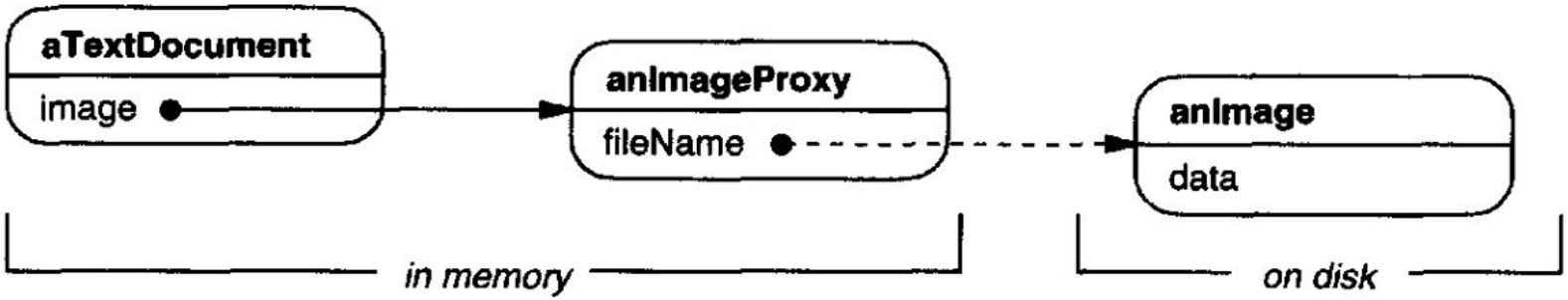

Figure: from [1]

Proxy

(Virtual) proxy example:

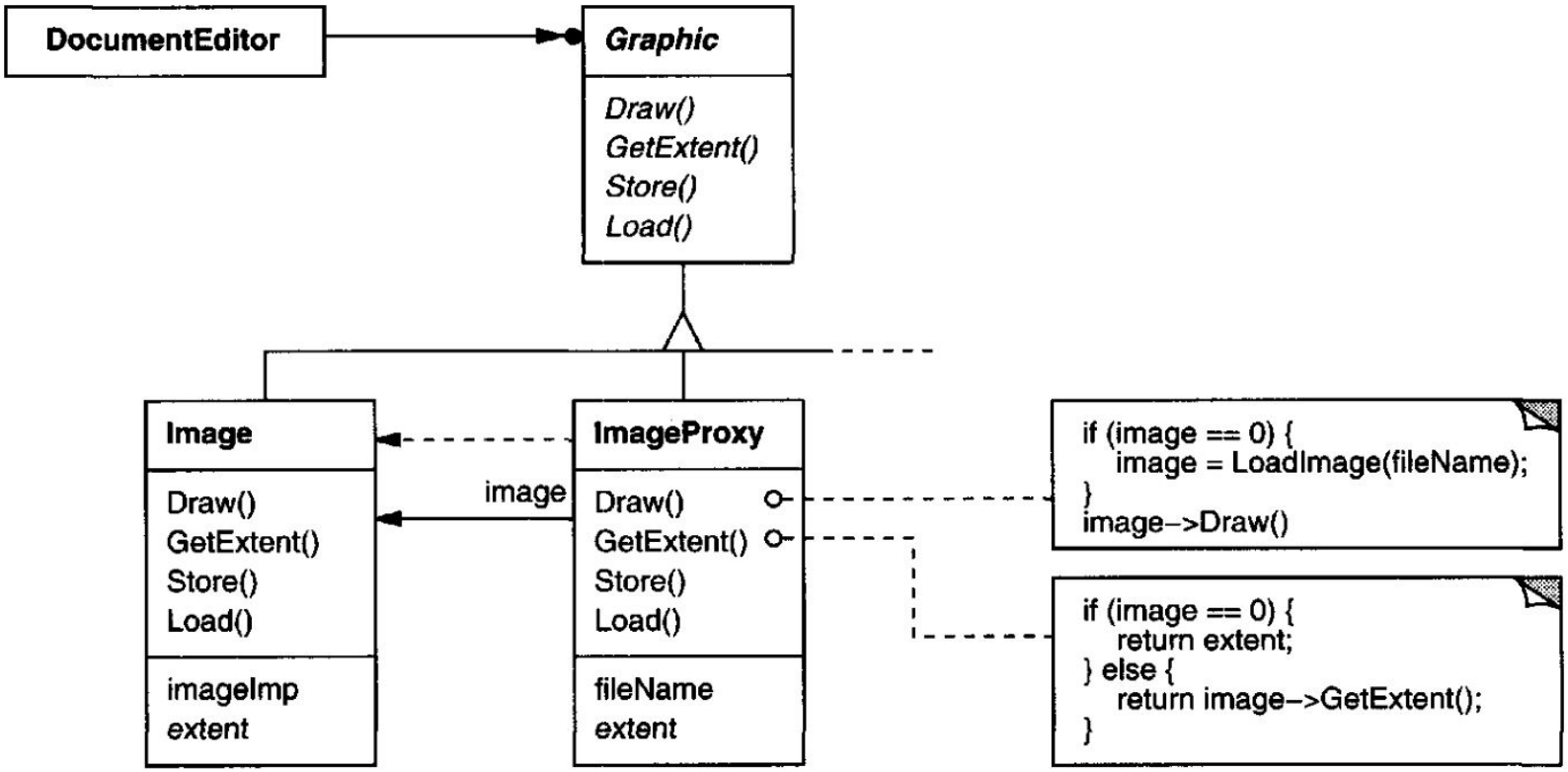

Figure: from [1]

Proxy

Proxy pattern implementations:

- **Remote proxy:** local representation for an object in a different address space (**e.g.** web service, database lazy loading)
- *Virtual proxy:* create expensive objects on demand
- *Protection proxy:* control access to objects
- *Smart reference:* smart pointers (and object locks etc.)

Proxy

Roles in the pattern

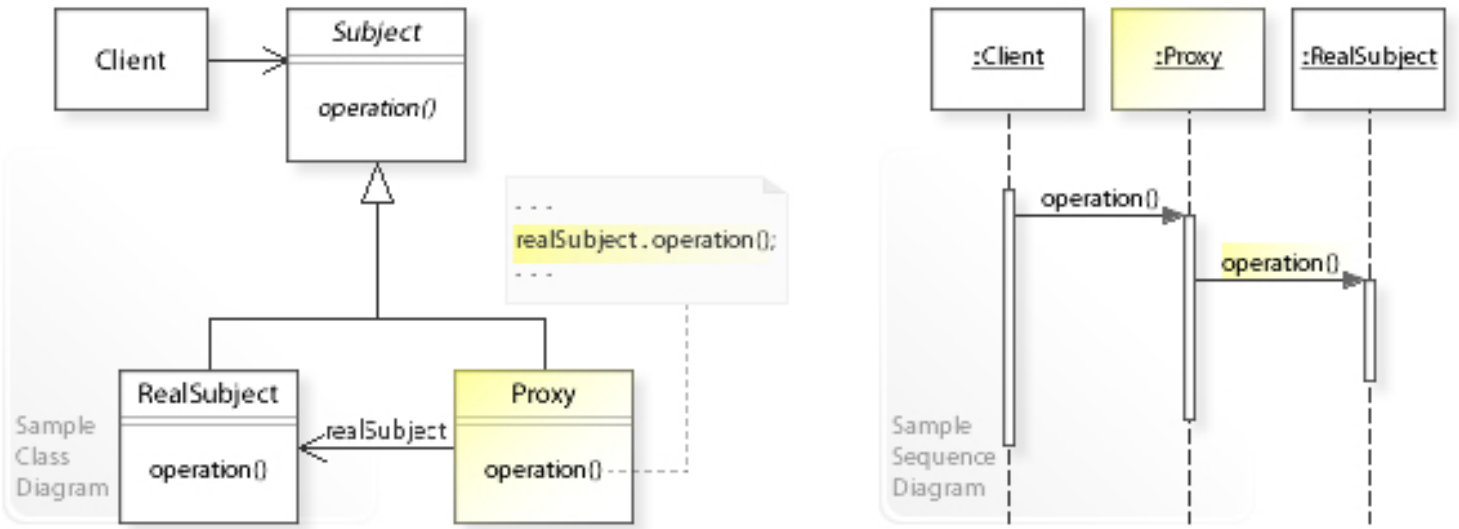

Figure: from [1]

Proxy pattern example code

Virtual proxy source code

git: /src/ubb/dp/structural/ProxyExampleImage.java

Protection proxy example code

git: /src/ubb/dp/structural/ProxyExampleProtection.java

Adapter versus Bridge - fight! ☺

- \+ Provide flexibility using indirection
- \+ Forward requests from a different interface
- – Adapter is usually employed after implementation, to connect distinct components, subsystems
- – Bridge is created as a conscious decision at design time

Composite versus Decorator

- \+ Composite and Decorator have similar structure
- – Composite structures classes to be used uniformly
- – Decorator allows you to add responsibilities by composition (without subclassing)

Decorator versus Proxy

- \+ Provide a level of indirection to an object
- – Proxy is not designed to add responsibilities
- – Proxy is not designed to be applied recursively