

Softwareudvikling 2018

software development

Design Patterns

Boris Düdder, Datalogisk Institut
30.4.2018

UNIVERSITY OF COPENHAGEN



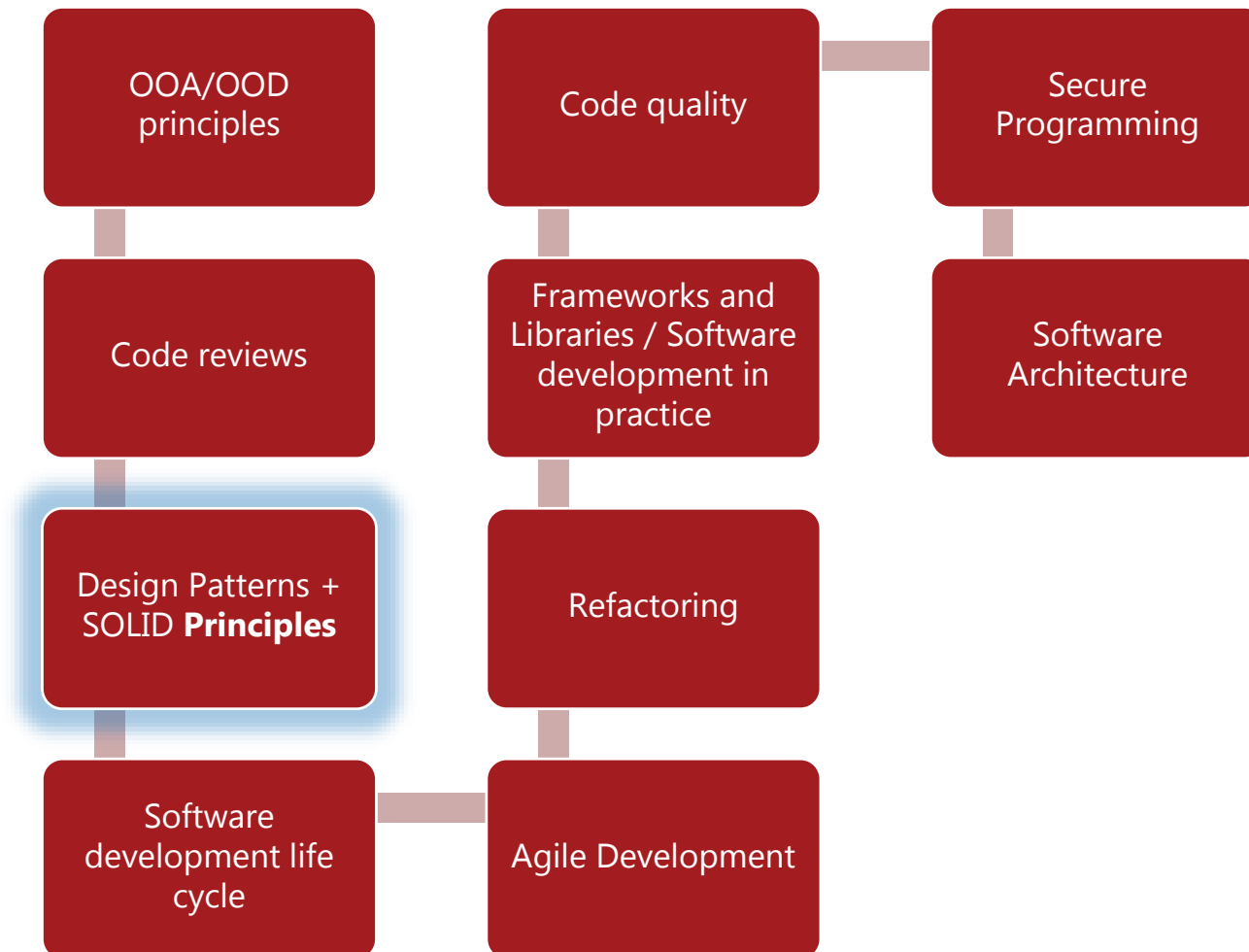
Literature

- Erich Gamma, et al. Design Patterns: Elements of Reusable Object-Oriented Software, 1st Edition, Addison-Wesley Professional, 1994
- Robert C. Martin and Micah Martin. Agile Principles, Patterns, and Practices in C#, Pearson Education, Inc., 2007

Learning Goals

- To understand generalized solutions of often occurring problems
- To abstract from a concrete problem to a context less solution
- To identify problems that can be solved by applying design patterns to this context

Course outline block 4: Lectures



Design patterns

- Design patterns provide an introduction into the techniques for designing software systems.
- Design patterns in software development were presented for the first time in:
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:
Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995, ISBN 0201633612
- Note: This book is sometimes difficult to understand due to many forward references.

Idea of design patterns

- The problems, which occur when designing software, are always similar.
Example: When changing a value, several windows must be updated (e.g. multiple instances of the file manager of the operating system).
- There are always similar solutions for similar problems.
- A good solution is developed for each identified problem.
- This solution is generalized to an approach and is described as a so-called **design pattern**.

(For the above example: design pattern *observer*)

Idea of design patterns

- While developing a specific software, the design pattern will be transferred into the context of the specific task.

Advantages of using design patterns

- A shortened development period,
since a known solution approach is used
- An improved quality of software,
since a suitable solution approach is used
- An improved intelligibility,
since the solution approach has been documented in
advance and is known to many developers
- A kind of *standardization* of the created software

Types of design patterns

Classification based on the considered structuring unit:

- Classes and their relationships at **type** level:
Class-related patterns
- Objects and their relationships during **execution**:
Object-related patterns

(In both cases, the pattern itself is described as a structure of classes.)

Types of design patterns

Classification based on the problem solved by the design pattern:

- **Structural** connection of classes or objects:

Structural patterns

- **Interaction** between objects:

Behavioral patterns

- **Creation** of objects:

Creational patterns

Types of design patterns

	Structural patterns	Behavioral patterns	Creational patterns
Class-related patterns	Class Adapter pattern	Interpreter pattern Template method pattern	Factory method pattern
Object-related patterns	Object Adapter pattern Decorator pattern Composite pattern Facade pattern Bridge pattern Flyweight pattern Proxy pattern	Strategy pattern Mediator pattern Observer pattern Chain of responsibility pattern Command pattern Iterator pattern Memento pattern State pattern Visitor pattern	Abstract factory pattern Singleton pattern Builder pattern Prototype pattern

- In software development presented design patterns

Description of design patterns

- Information about the purpose:
 - Which problem should be solved by the pattern?
 - What should be achieved by using the pattern?
 - What is the pattern doing?
- Information on the scope of application:
 - In which situations does the pattern fit?
- Description of structure and behavior:
 - Representation of the structure by an **UML class diagram**
 - Clarification of the structure by an **UML object diagram**
 - Representation of the behavior by **UML sequence diagram**
- Examples for the implementation:
 - Implementation in C#

Class Adapter pattern

- An **adapter**
enables the connection of items with different
interfaces

Class Adapter pattern

- General observations:
 - Adapters (in the physical world) are required when two completed components have to be connected.
 - An adapter is a comparatively cheap connector and especially cheaper than specially adapted components.
 - Observations for software:
 - Many classes are created independently of specific problem solutions.
 - Classes provide interfaces in the form of the applied methods.
 - Methods expect certain interfaces from the used objects.
 - Completed classes are expected to produce a common solution for a new problem.
 - Problem: The provided interfaces do not match.
- ➡ **Solution:** Implementation of a (simple) connection between two classes
- An Adapter is a class,
- which uses the methods of a completed class
 - to provide those methods,
 - which are expected by the using classes.

Example Class Adapter pattern

- Problem:
 - An application needs features for color management .
 - The application is **completed** and requires a **specified** interface:

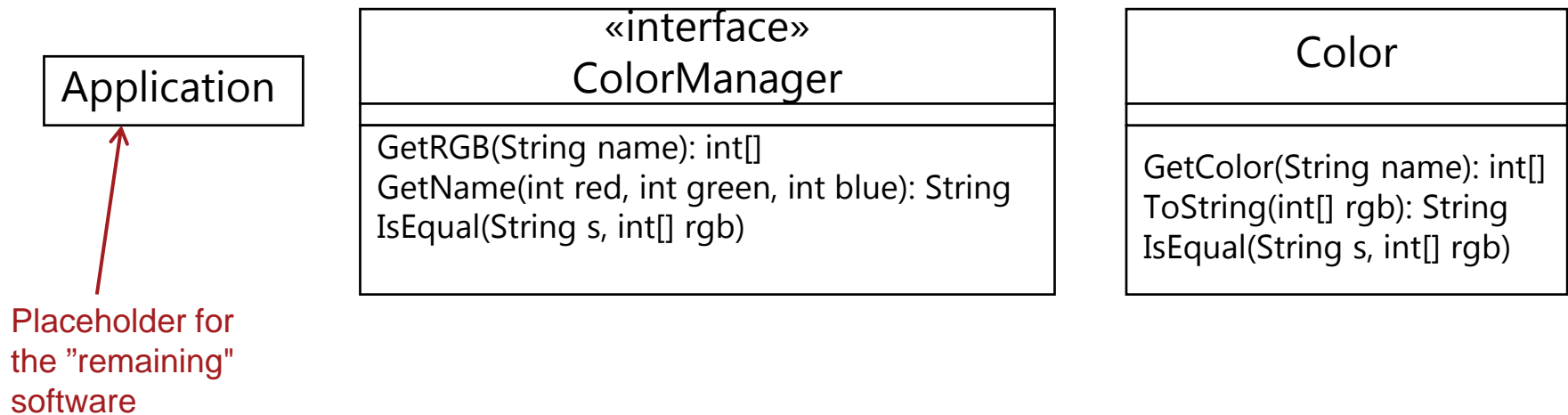
```
public interface ColorManager {  
    String GetName(int red, int green, int blue);  
    int[] GetRGB(String name);  
    boolean IsEqual(String s, int[] rgb);  
}
```

- Support for implementation:
 - There is an implemented, **tested and practically proven** class:

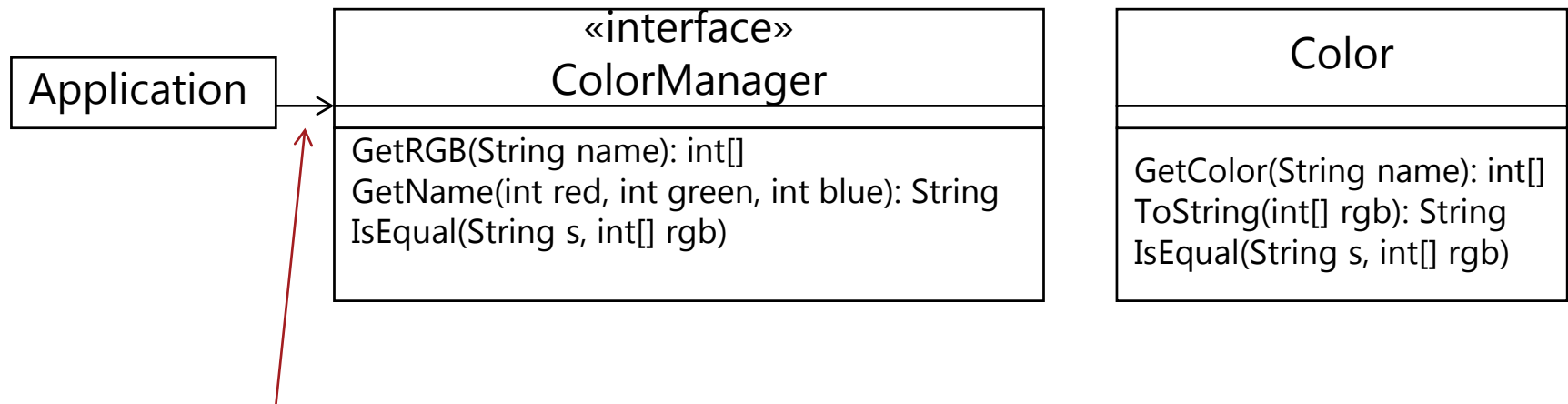
```
public class Color {  
    public int[] GetColor(String name) { ... }  
    public String ToString(int[] rgb) { ... }  
    public boolean IsEqual(String s, int[] rgb) { ... }  
}
```

- Solution:
 - Since the already implemented classes should not be changed, a ColorAdapter class is implemented.

Visualization Class Adapter pattern



Visualization Class Adapter pattern

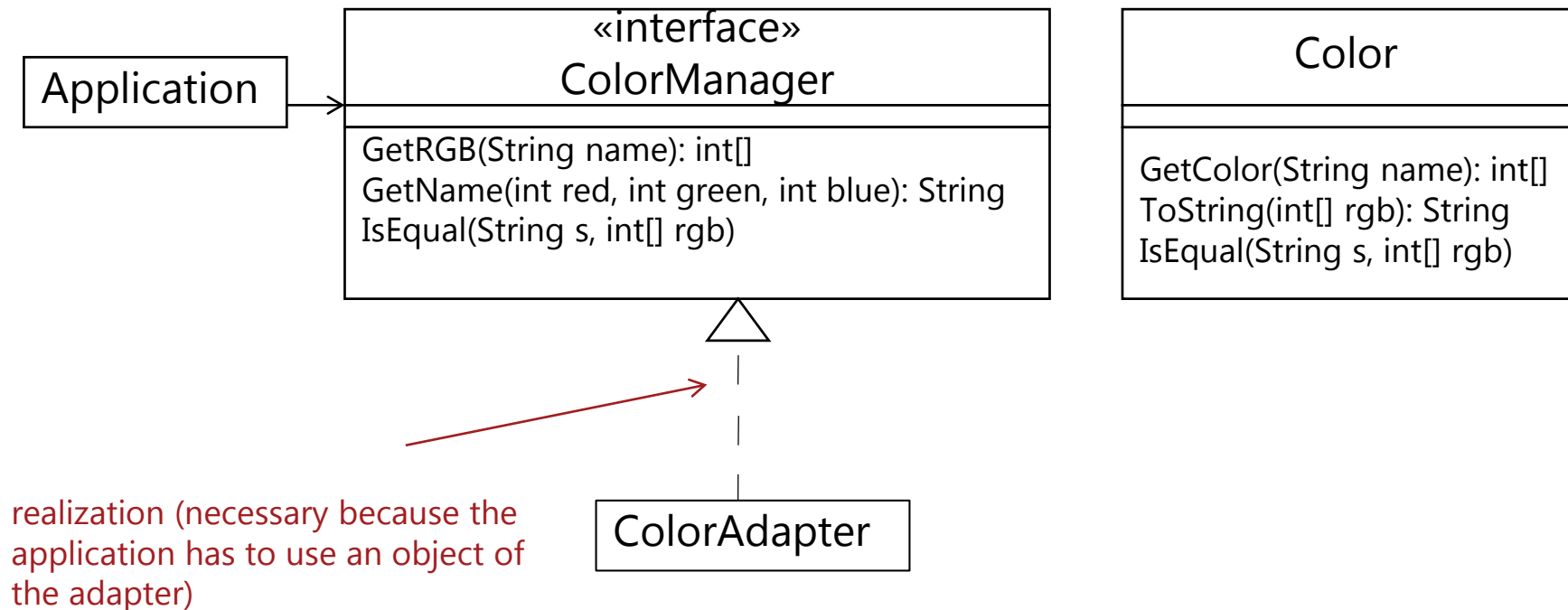


(directed) association

Meaning: the application knows an object compatible with the **ColorManager** interface and can use it

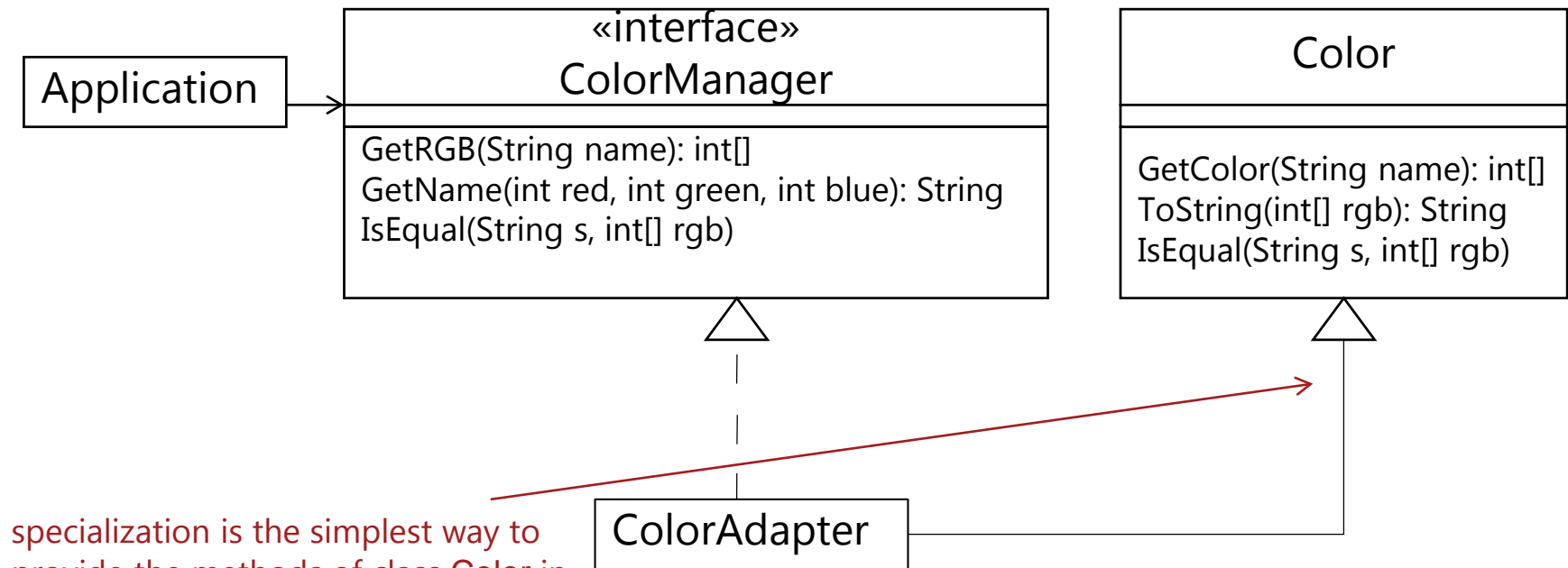
Visualization Class Adapter pattern

- Realization
establishes an implementation of the interface by a class.



Visualization Class Adapter pattern

- Specialization
establishes the transfer of properties through a hereditary relationship



specialization is the simplest way to provide the methods of class **Color** in the adapter

Implementation

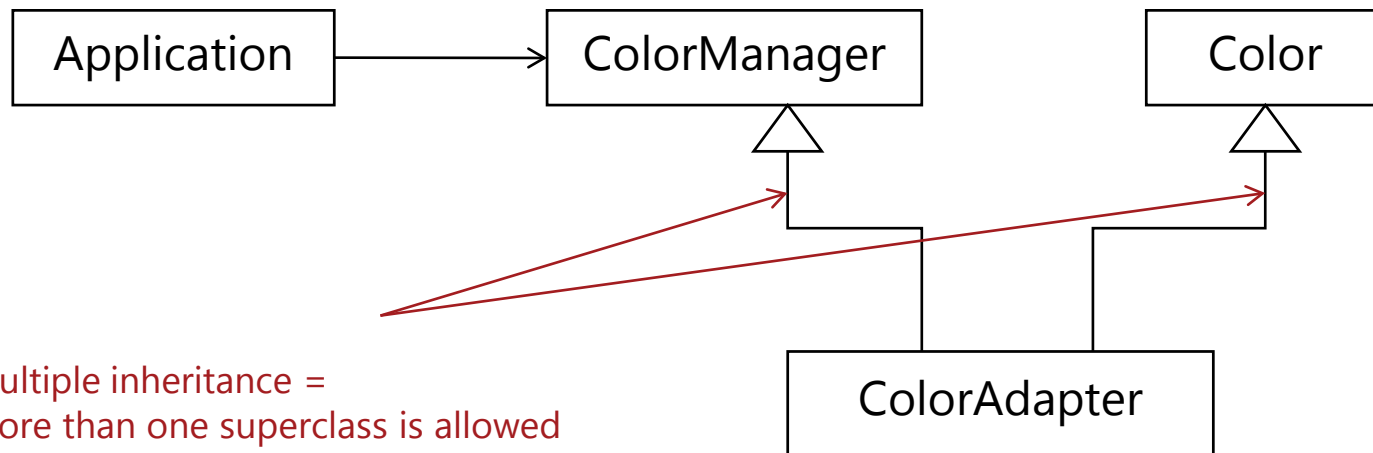
```
public class ColorAdapter
extends Color : ColorManager {
    public int[] GetRGB(String name) {
        return GetColor(name);
    }
    public String GetName(int red, int green, int blue) {
        int[] c = {red, green, blue};
        return GetName(c);
    }
    //Implementation of IsEqual is not necessary since it has already been
    //inherited
}
```

Summary

- Presented design pattern: **Class Adapter pattern**
- The adapter class inherits the class, which is to be used:
 - The relationship between the classes is established at the structural level.
 - The adapter has (by inheritance) all properties of the class, which is to be used and therefore has to be adjusted
(In particular, there may also be more public methods than the adapter requires)
- The adapter class implements the target interface:
 - All methods of the interface are generally called by using the methods of the inherited class.
- The application uses an adapter class object, to which a reference (with the type of the implemented interface) is accessed.

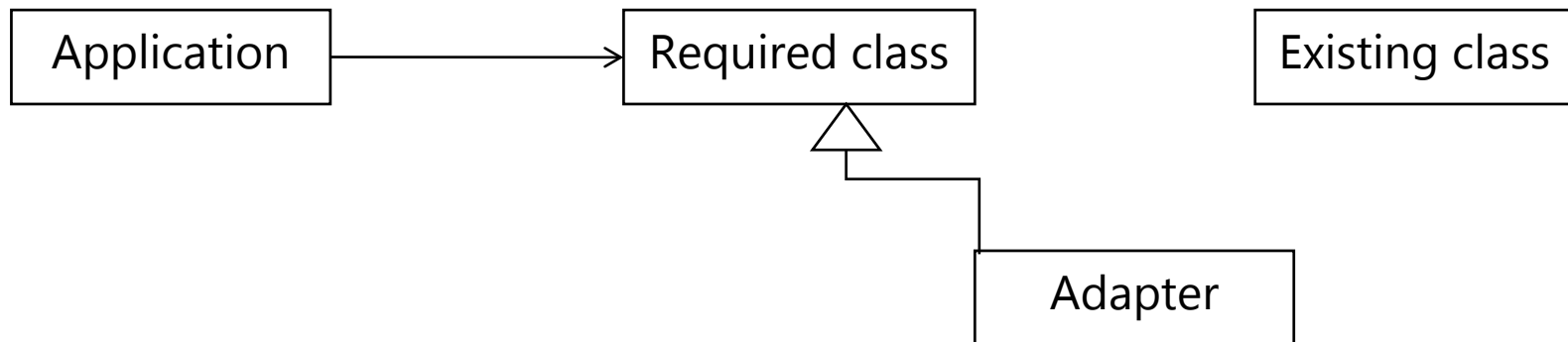
Summary

- Class adapter can also be used, if the required interface is a class:
 - The adapter class inherits from both classes.
 - The programming language used for implementation must then allow multiple inheritance (e.g. C++)



Class Adapter pattern - problems

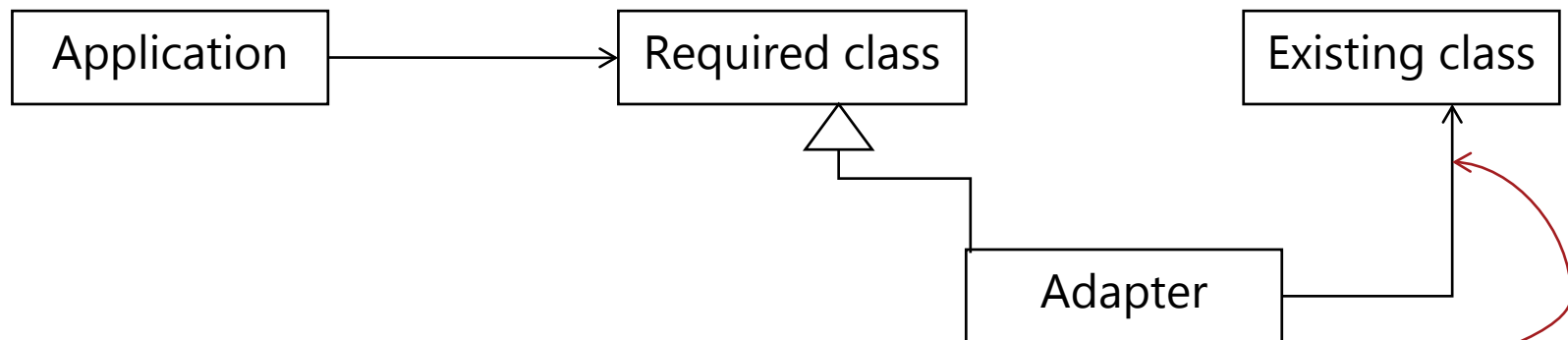
- Assumptions:
 - The required interface is only available as a class and the programming language used does **not** allow multiple inheritance.
- Consequence:
 - The existing class can not be inherited.
➡ **Class adapter is not an applicable solution.**



Object Adapter pattern as a solution

- The adapter uses an object of the existing class.
- The adapter has to create and manage this object itself.
- The methods of the adapter are implemented by calling the methods of the object.

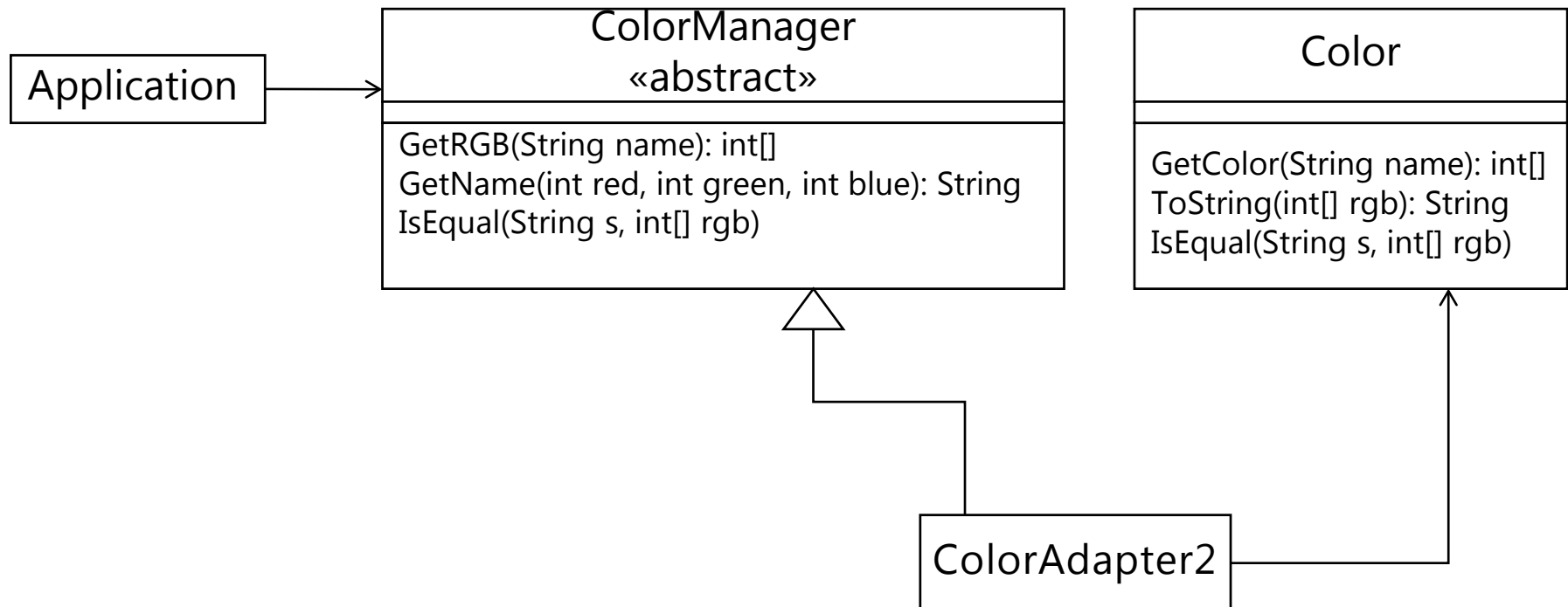
➡ Object Adapter pattern



Association:

An adapter object knows and uses
an object of the existing class.

Example Object Adapter pattern



Implementation

```
public class ColorAdapter2
```

```
: ColorManager {
```

```
    private Color theObject;
```

← Attribute for realizing the association

```
    public ColorAdapter2(Color c) {
```

← Constructor to get/to produce the object

```
        theObject = c;
```

```
    }
```

```
    public int[] getRGB(String name) {
```

```
        return theObject.GetColor(name);
```

← Use of the object

```
    }
```

```
    public String GetName(int red, int green, int blue) {
```

```
        int[] c = {red, green, blue};
```

```
        return theObject.GetName(c);
```

```
    }
```

```
    public boolean IsEqual(String s, int[] c) {
```

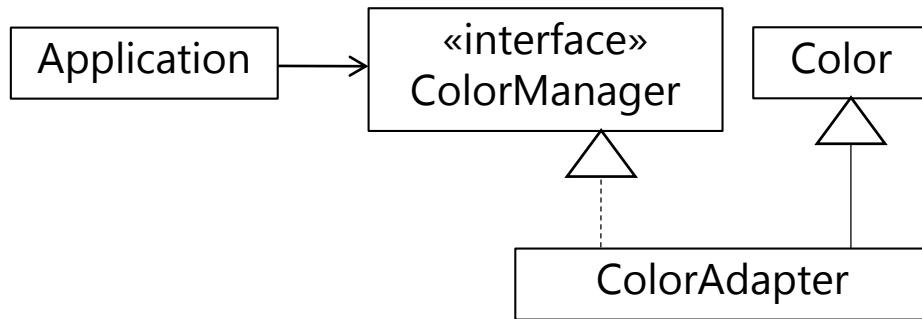
```
        return theObject.IsEqual(s, c);
```

```
    }
```

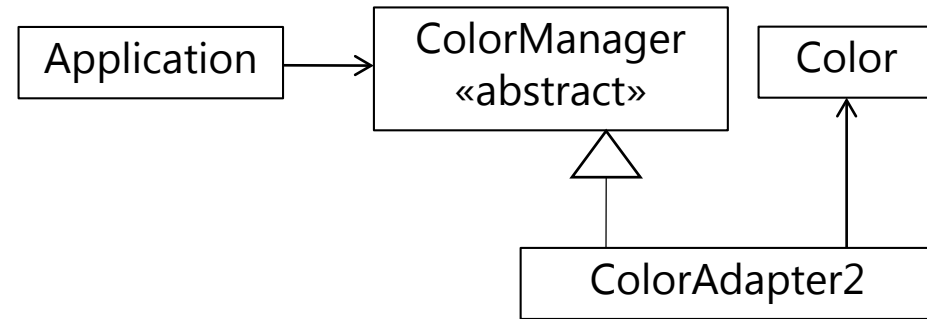
```
}
```

Comparison of the object structure

Class Adapter pattern

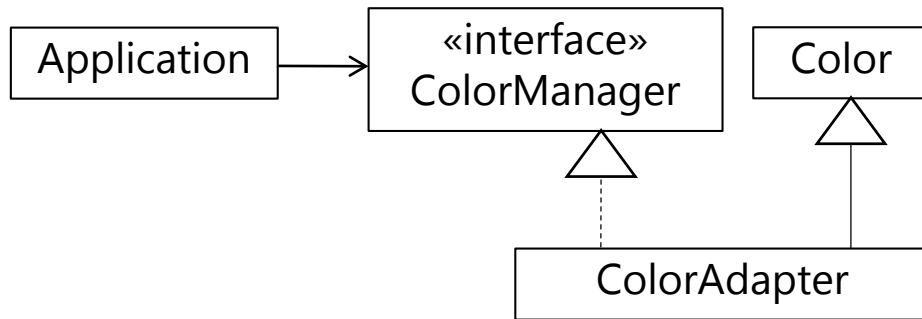


Object Adapter pattern

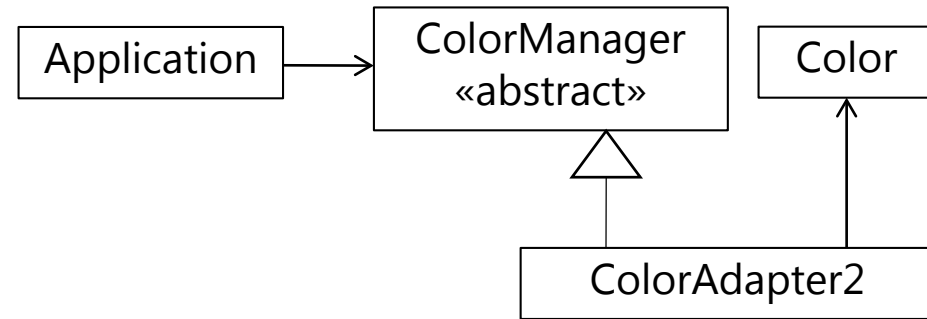


Comparison of the object structure

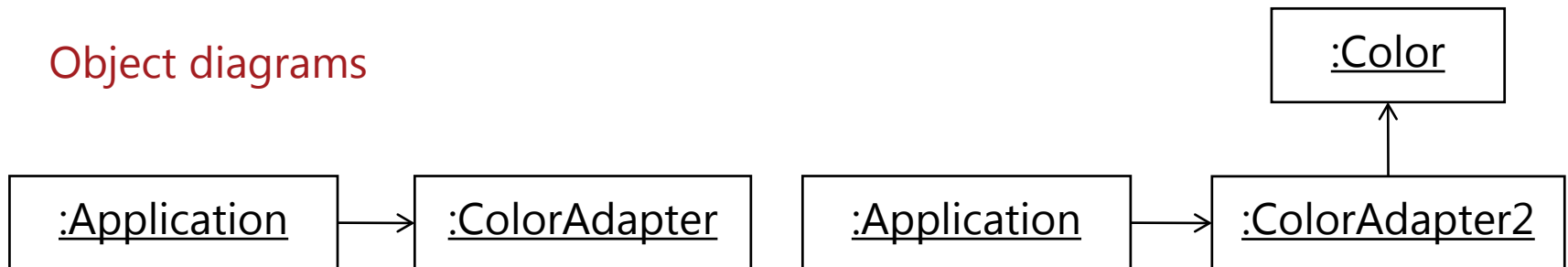
Class Adapter pattern



Object Adapter pattern

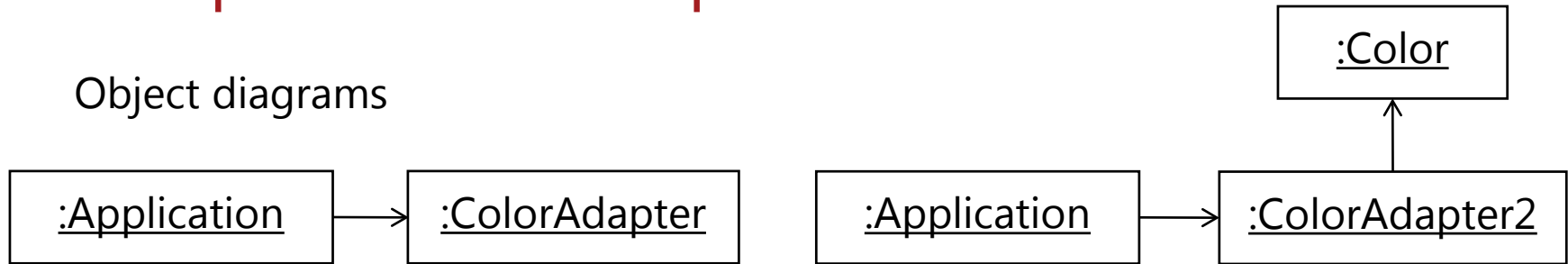


Object diagrams

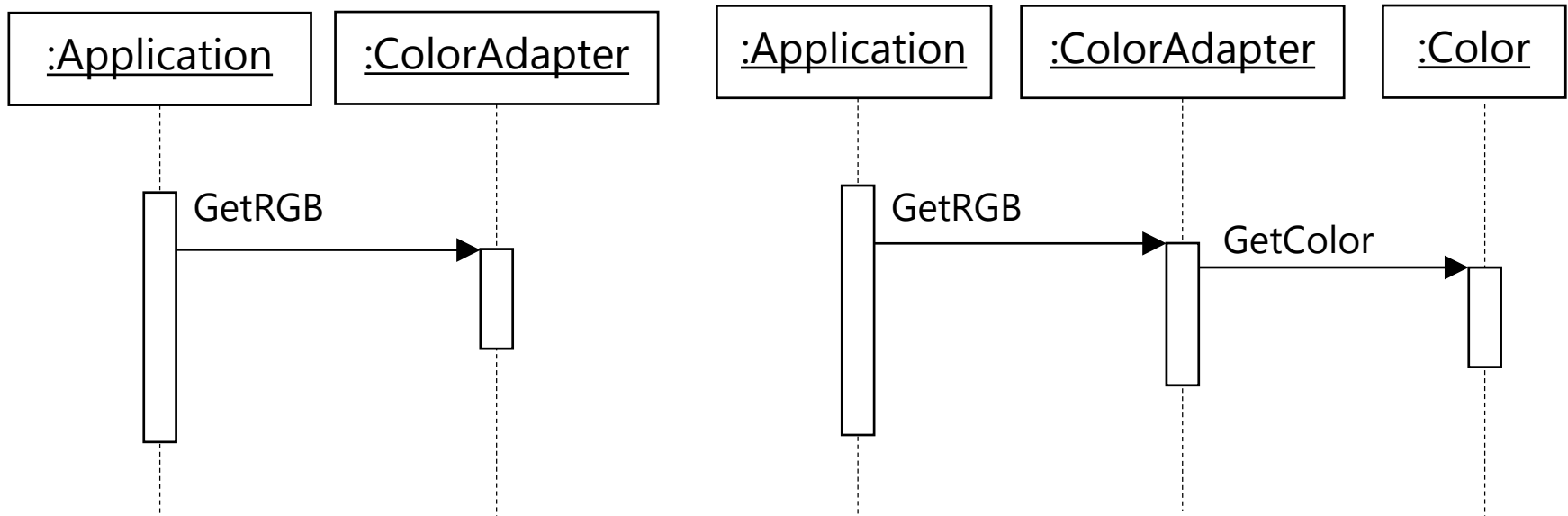


Comparison of the procedures

Object diagrams



Sequence diagrams



Comparison Class Adapter – Object Adapter

Comparison of the object diagrams shows

- Class Adapter pattern
 - consists of only one object,
 - which can carry out the requested tasks itself.
- Object Adapter pattern
 - comprises several objects,
 - of which the actual adapter produces only the appropriate interface
 - and distributes the actual "work" to other objects (**delegation**).

Actually, the adapter object in the object adapter pattern does little by itself, but only knows objects, which can fulfill the required tasks together!

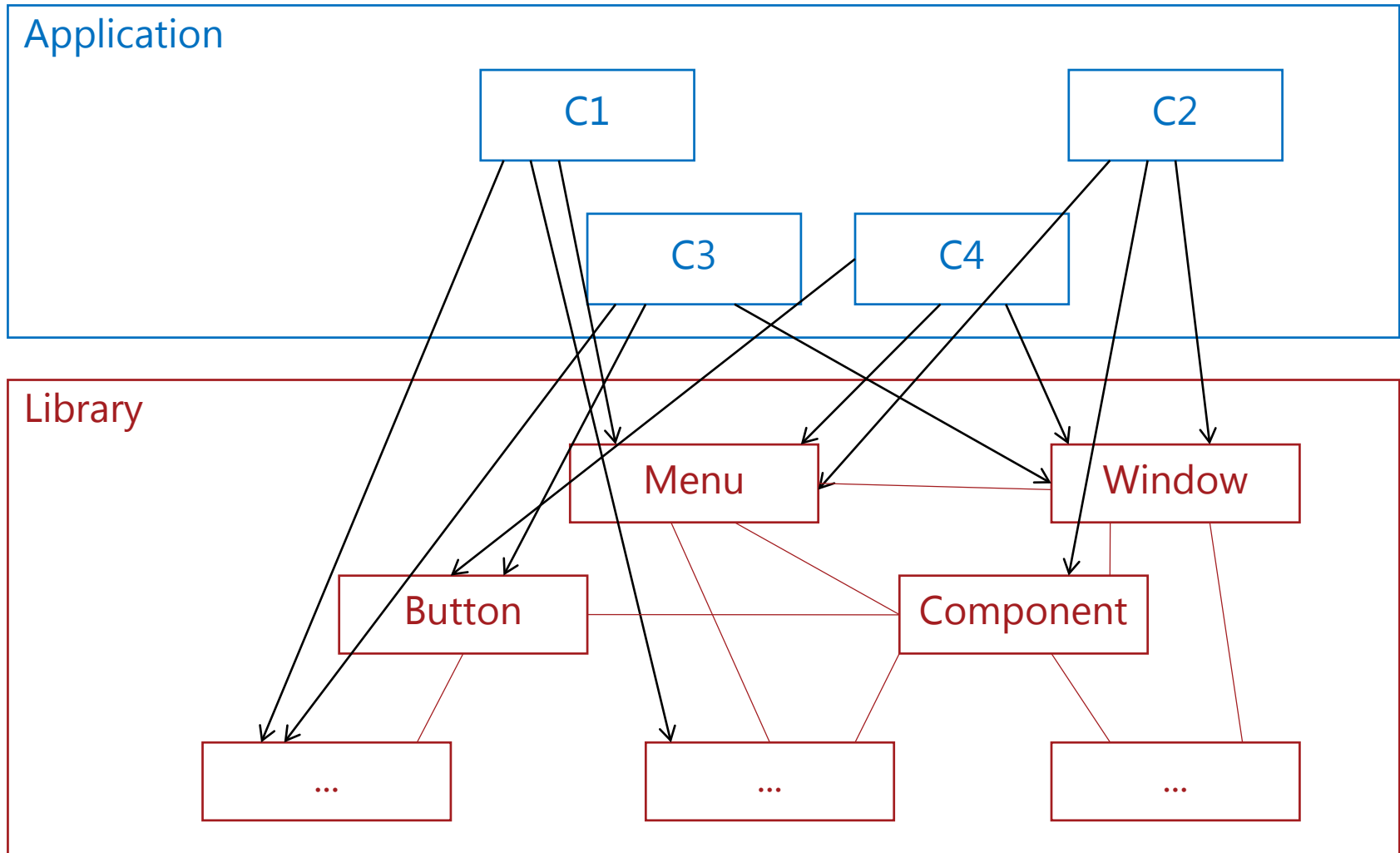
Summary: Adapter pattern

- An adapter is used when ready-to-use solutions should be used, which do not fit exactly to the given requirements.
- In particular, adapters are used when classes from libraries are to be used.
- An adapter is always a simple component.
- Without knowledge of the development history an adapter may not be recognized in the finished program code.

Facade pattern

- A facade pattern allows to hide complex interfaces.
- Example: In a software application, graphical user interfaces for input and output of information have to be implemented in different places, thus in methods of different classes.

Class structure of the example



Class structure of the example

- Each of the classes C1, C2, C3, C4 uses several classes of the library.
- All developers of C1, C2, C3, C4 need the competence to use the library.
- Arrangements between the developers are necessary, because all parts of the graphical user interface of an application should have a similar appearance.
- If the *similar* appearance is to be changed, changes have to be made at many points in the application.
- If the library is to be replaced by another graphics library, changes in the classes C1, C2, C3, C4 have to be made.

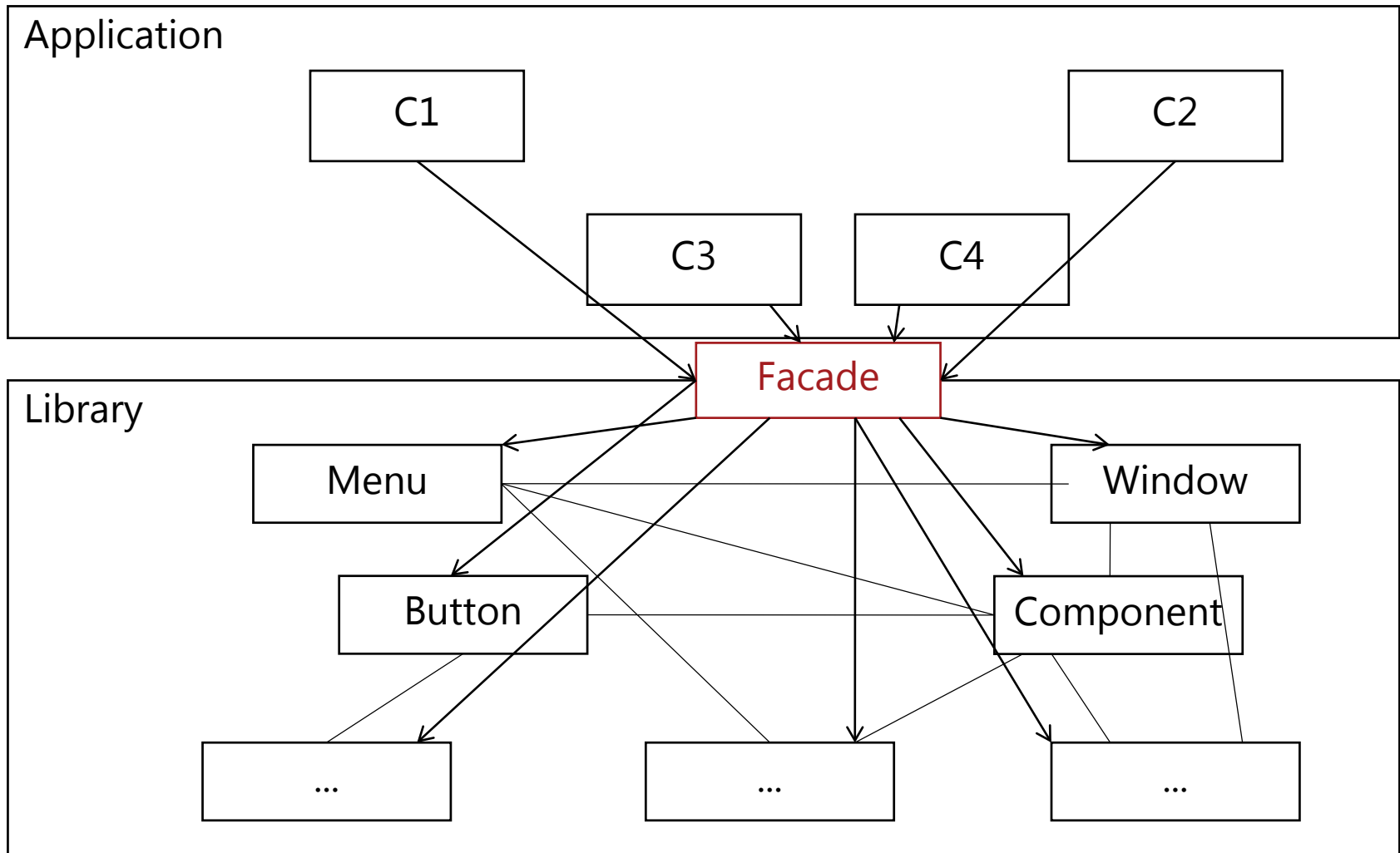
Class structure of the example

- **Solution:**

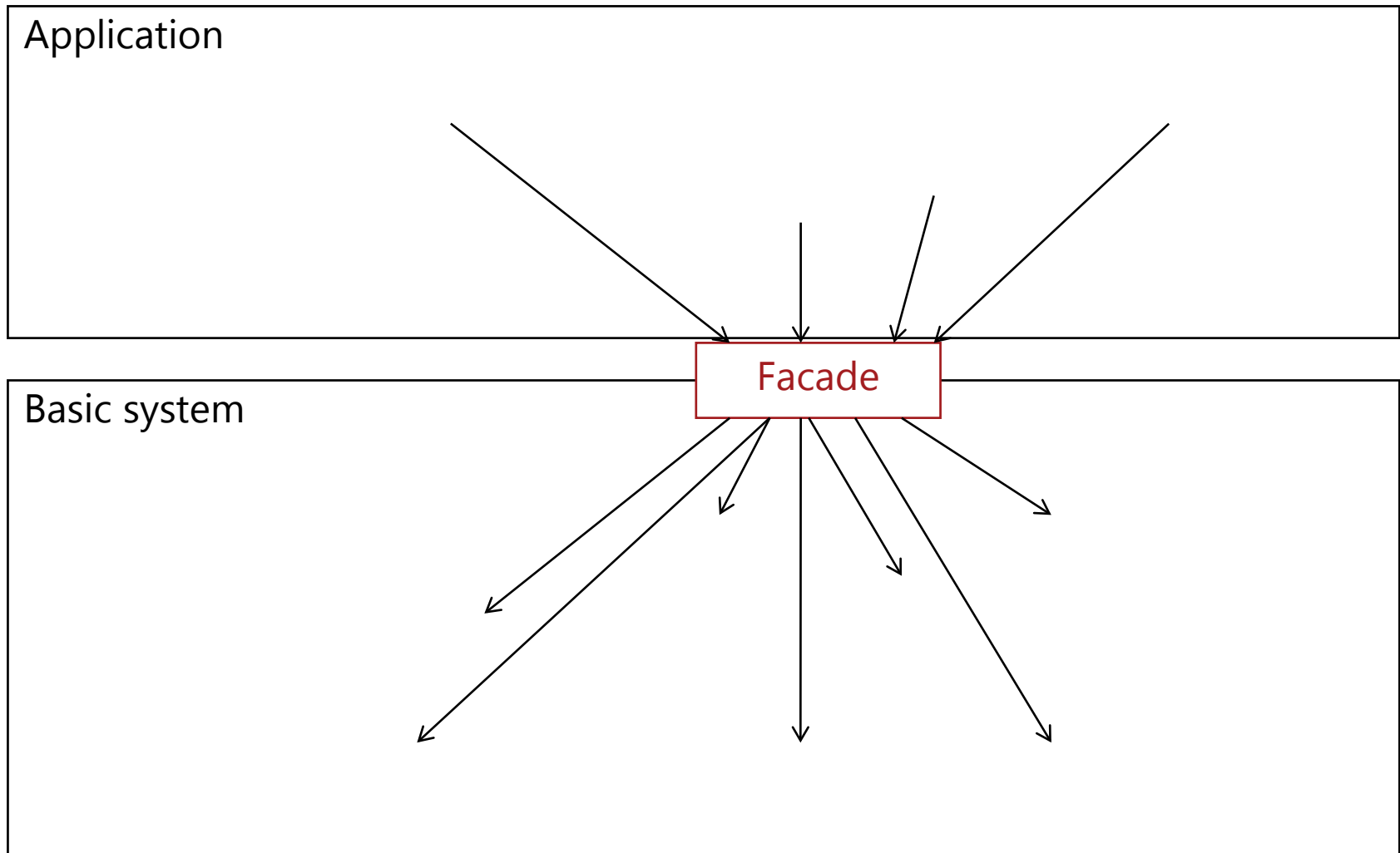
Introduction of a (compact) intermediate layer, which

- provides specialized methods that specifically support the work with the library for the classes C1, C2, C3, C4
- thus simplifying the access to the library
- and thereby conceals the library (thus building a **facade** in front of the library).

Facade pattern



Facade pattern: general presentation



Evaluation of the facade pattern

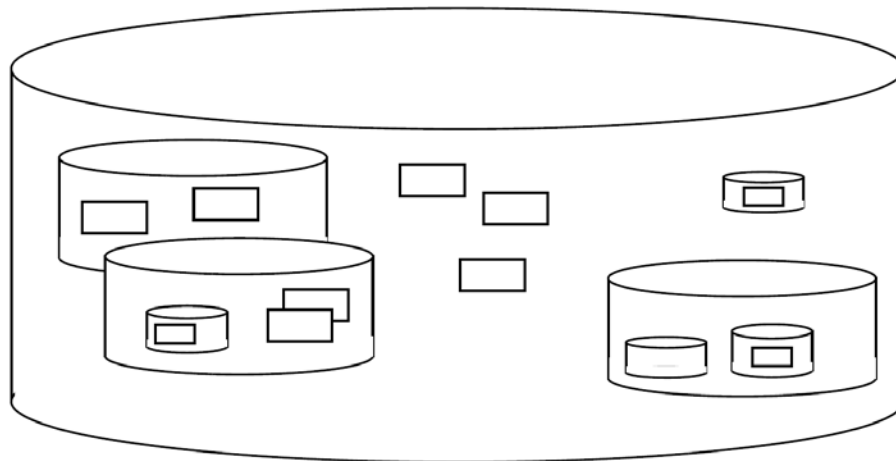
- Advantages:
 - The access to the basic system is simplified.
 - The application is decoupled from the basic system.
 - For a basic system, there may be several facades.
 - The classes of the basic system do not know the facade.
 - The use of the basic system is also possible without a facade.
 - The bundling of calls in the facade can improve the performance.
- Disadvantages:
 - Because of the additional hierarchy level the structure becomes more complex.
 - The performance can also get worse due to the additional call level.

Evaluation of the facade pattern

- Difference to the adapter pattern:
 - A facade creates a new, simpler interface.
 - A facade is created during the development of the application, while an adapter links specified interfaces with already implemented classes.

Composite pattern

- A **Composite pattern** allows the creation of tree-like structures composed of heterogeneous components.
- Example: File structure of an operating system



Components:



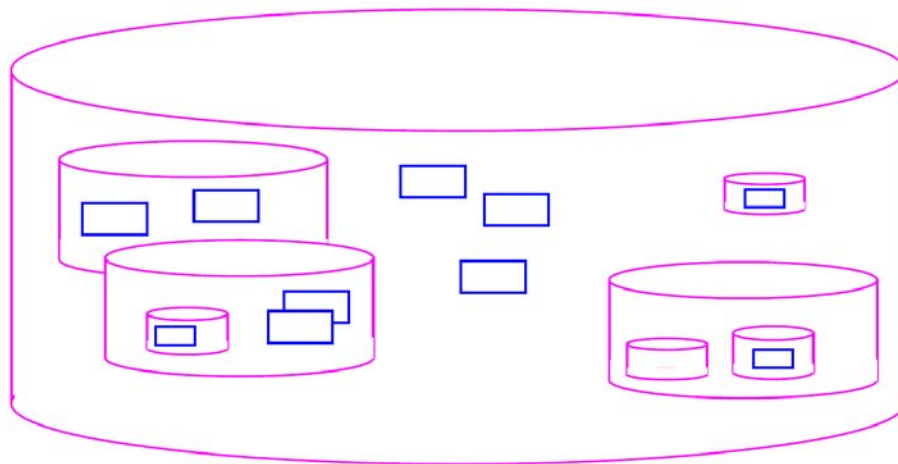
File



Register

Composite pattern

- A **Composite pattern** allows the creation of tree-like structures composed of heterogeneous components.
- Example: File structure of an operating system



Components:



File

Leaf



Register

Composite

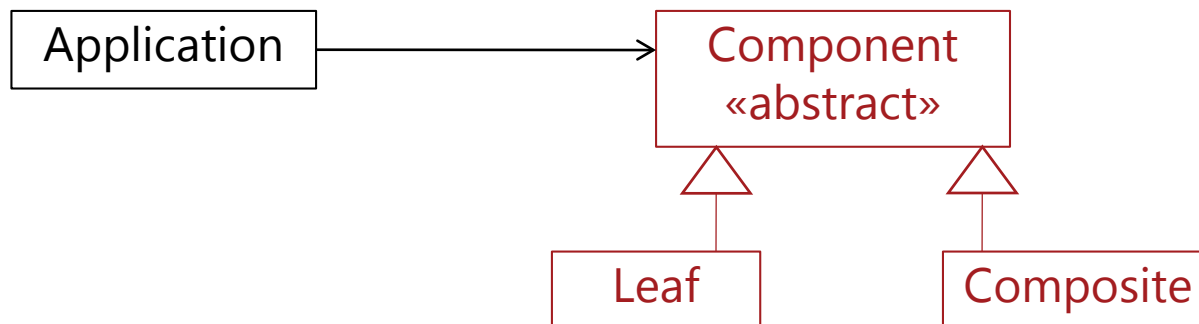
Analysis of the file structure

- There are two types of components:
 - Leaves, which can not contain any other components.
 - Composites, which again can contain components.
- Component is the upper term for leaf or composite.
- The structure builds up tree-like (=recursive) of (both types of) components.
- The leaves form the leaf objects of the tree structure.
- The composites form inner knots of the tree structure.

Analysis of the file structure

- There are two types of components:
 - Leaves, which can not contain any other components.
 - Composites, which again can contain components.
- Component is the upper term for leaf or composite.
- The structure builds up tree-like (=recursive) of (both types of) components.

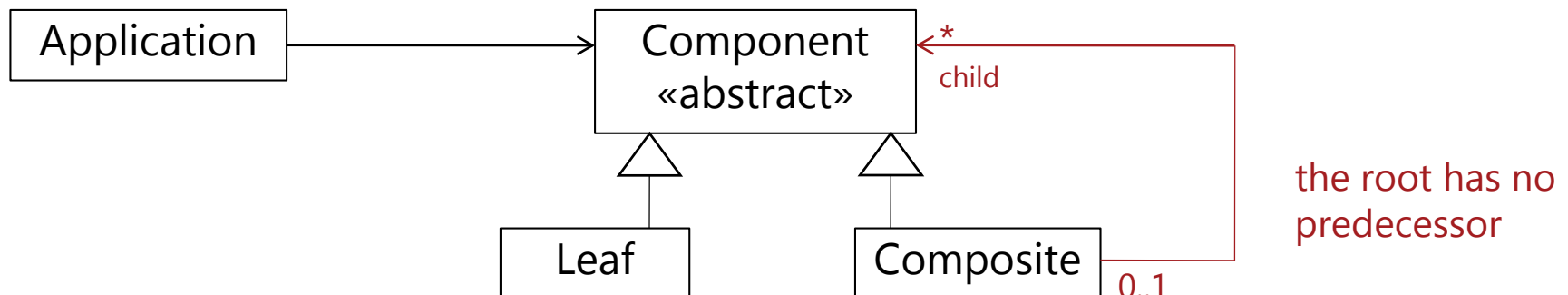
Modeling as a class diagram:



Analysis of the file structure

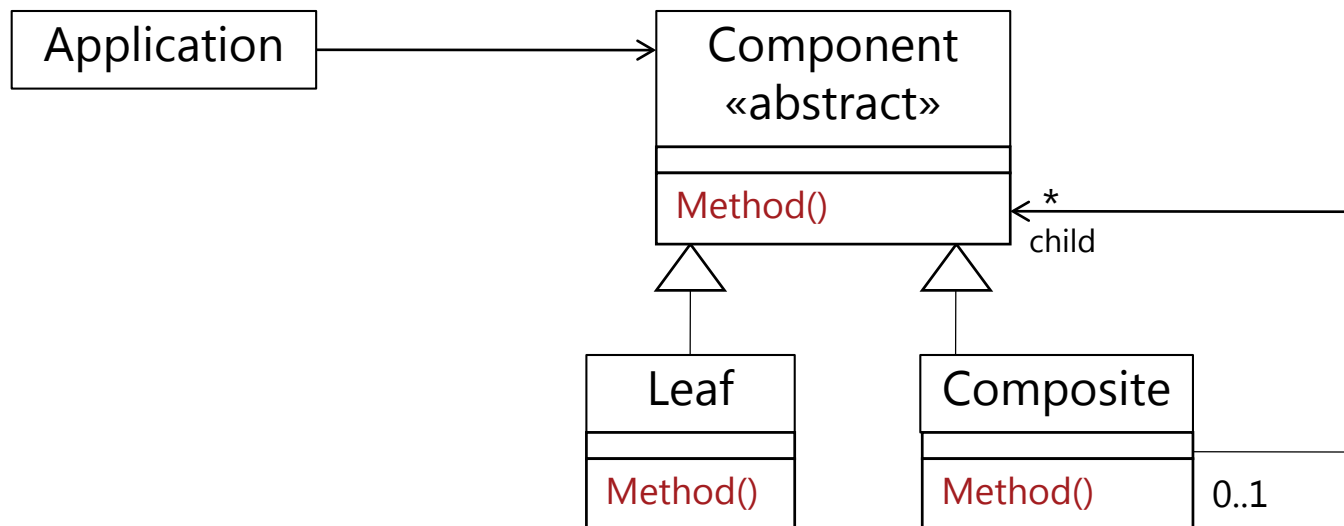
- There are two types of components:
 - Leaves, which can not contain any other components.
 - Composites, which again can contain components.
- Component is the upper term for leaf or composite.
- The structure builds up tree-like (=recursive) of (both types of) components.

Modeling as a class diagram:



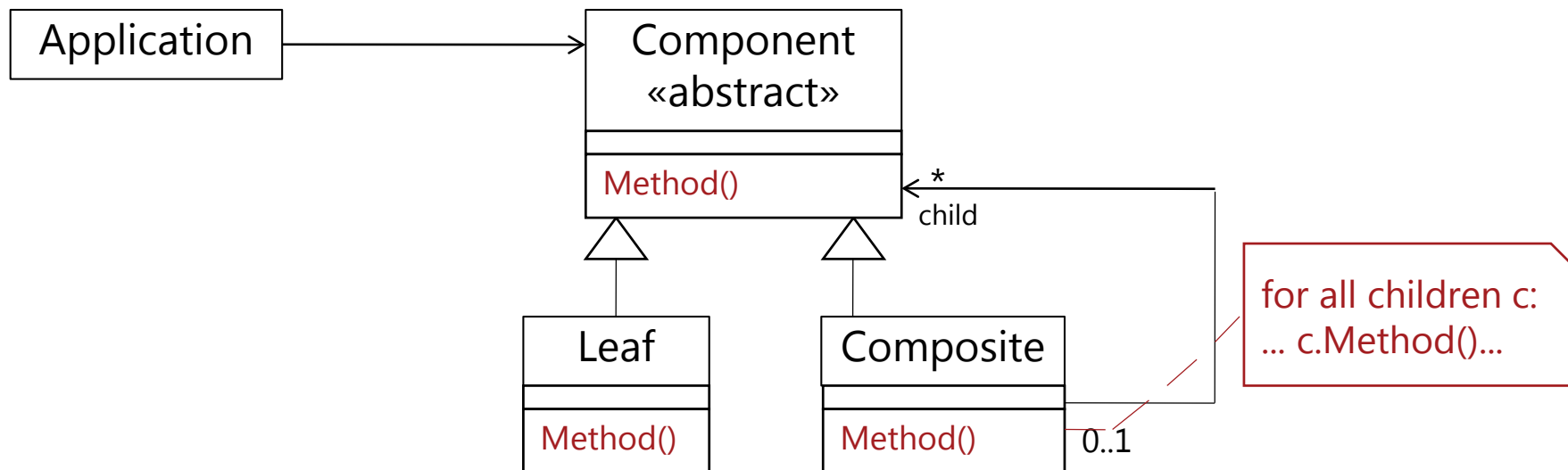
Analysis of the file structure

- The abstract component has to offer the methods used for
 - Leaves and
 - Composites, which can be called.
- Method calls on composites must be transferred to the children.

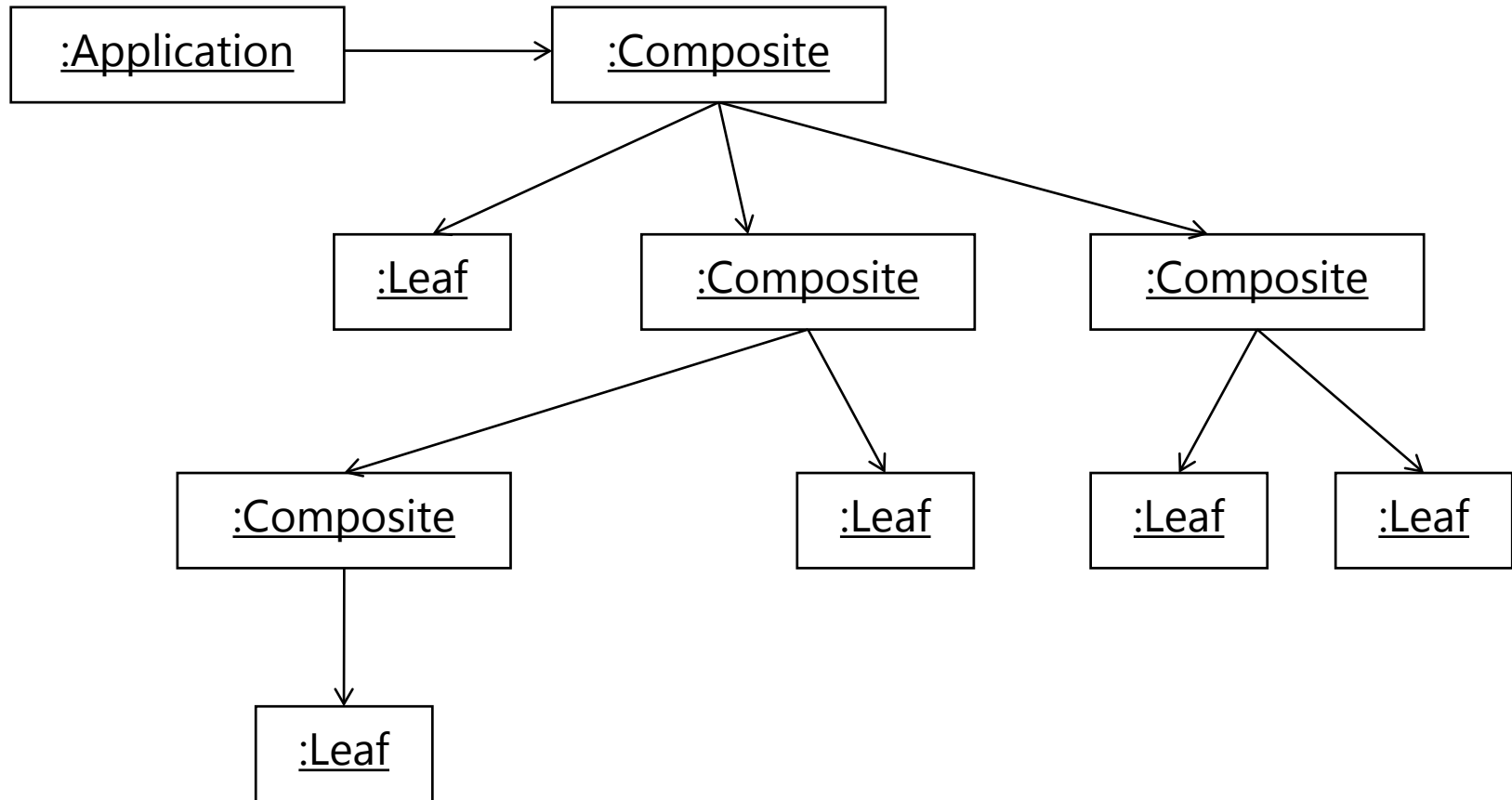


Analysis of the file structure

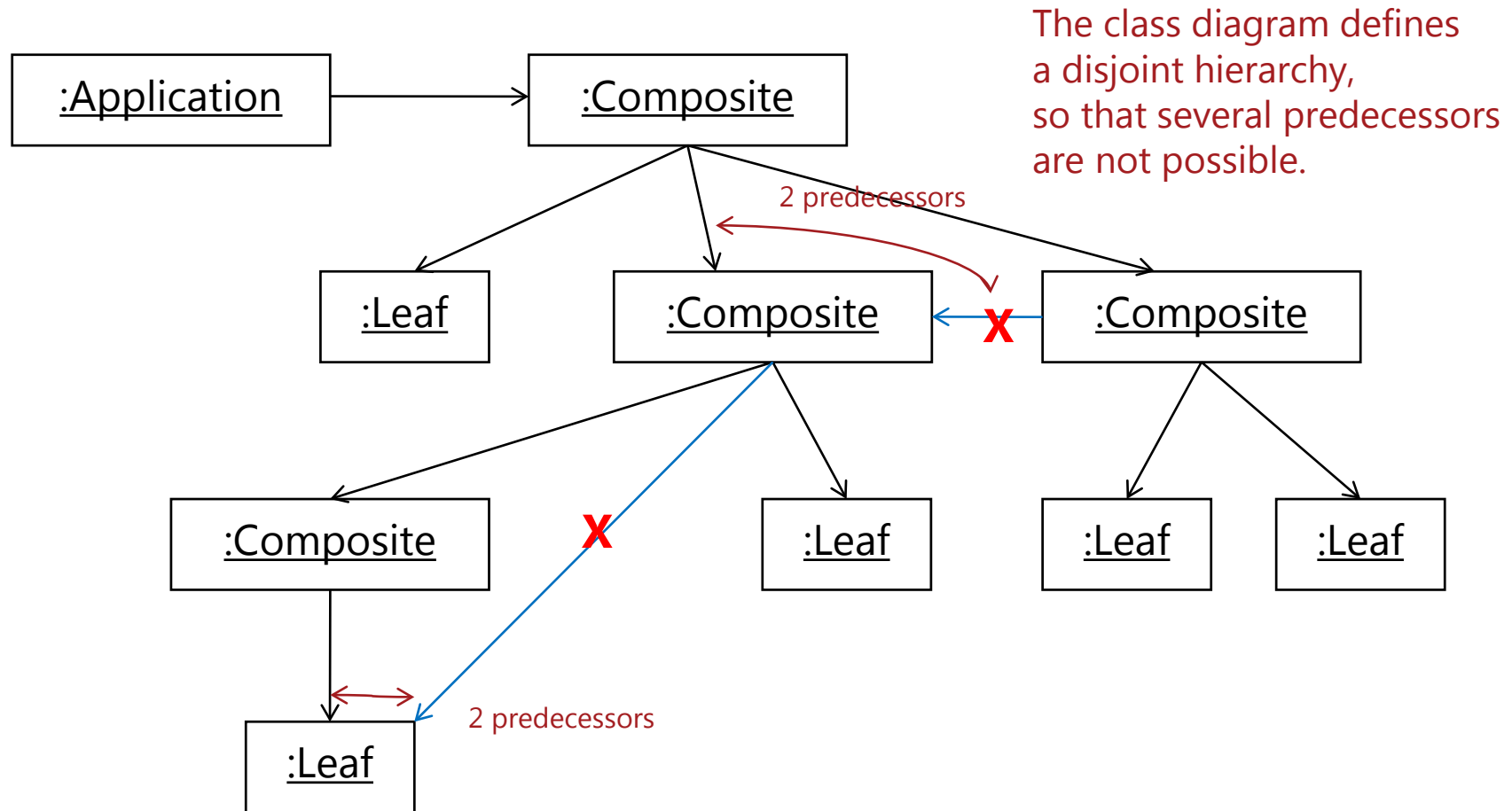
- The abstract component has to offer the methods used for
 - Leaves and
 - Composites, which can be called.
- Method calls on composites must be transferred to the children.



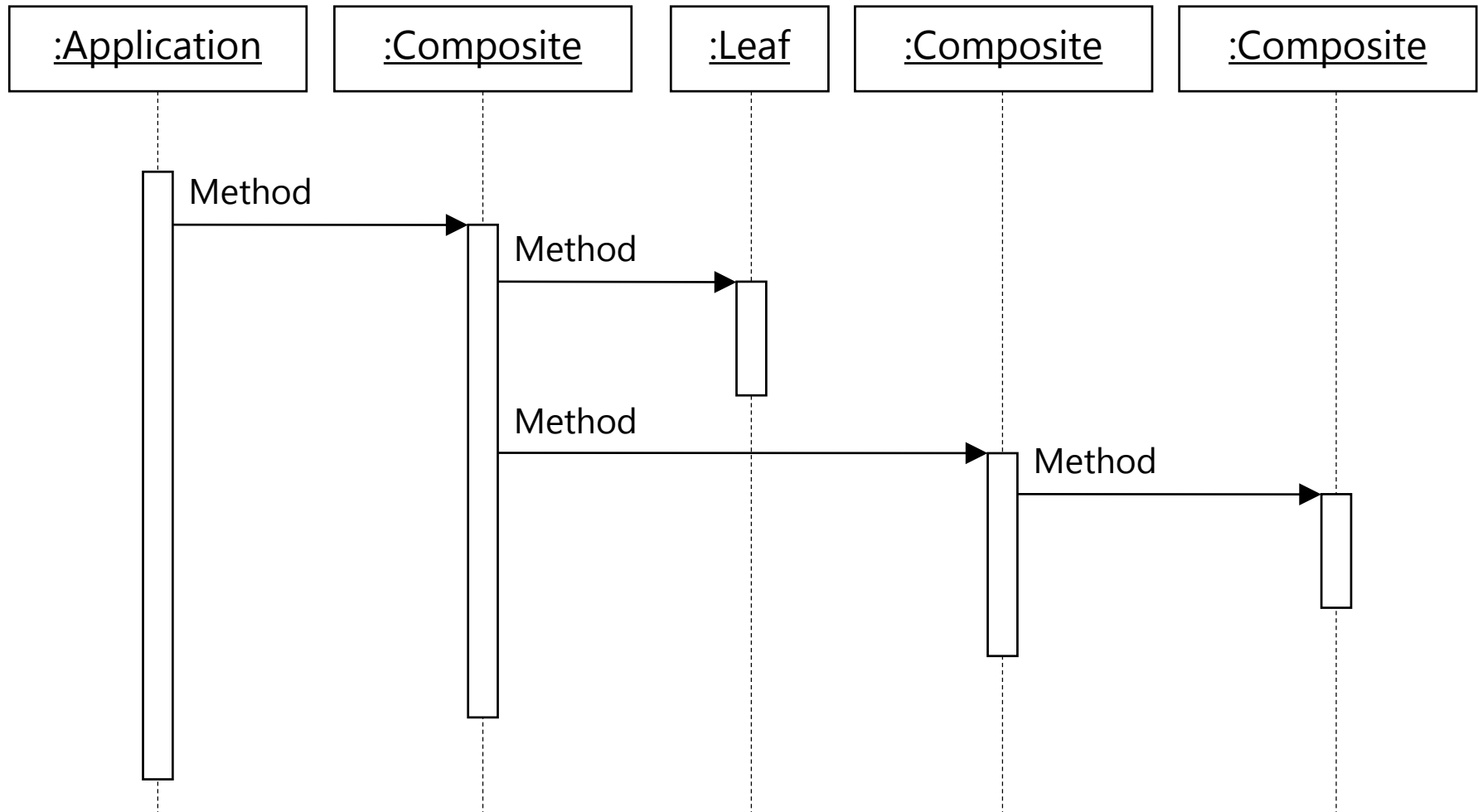
Example of a corresponding object diagram



Example of an object diagram

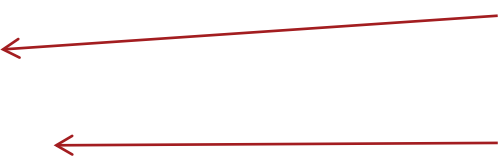


Example of behavior



Example of an implementation of the Composite pattern

```
public abstract class Component
    protected String content;
    public abstract String Get();
    public abstract void Add(Component c);
}
```



example for an attribute

example for methods

Example of an implementation of the Composite pattern

```
public abstract class Component
{
    protected String content;
    public abstract String Get();
    public abstract void Add(Component c);
}

public class Leaf : Component {
    public Atom (String s) { content = s; }
    public String Get() { return content; }
    public void Add(Component c) {};
}
```

Example of an implementation of the Composite pattern

```
public abstract class Component
    protected String content;
    public abstract String Get();
    public abstract void Add(Component c);
}
```

```
public class Leaf : Component {
    public Atom (String s) { content = s; }
    public String Get() { return content; }
    public void Add(Component c) {};
}
```

```
public class Composite : Component {
    private List<Component> children = new List<Component>();
    public Composite(String s) { content = s; }
    public String Get() {
        String allContents = content;
        for (Component c: children) { allContents += c.Get(); }
        return allContents;
    }
    public void Add(Component c){ children.add(c); }
}
```

Attribute to manage children
(realizes * from diagram)



Example of an implementation of the Composite pattern

```
public abstract class Component
```

```
    protected String content;
```

```
    public abstract String Get();
```

```
    public abstract void Add(Component c);
```

```
}
```

```
public class Leaf : Component {
```

```
    public Atom (String s) { content = s; }
```

```
    public String Get() { return content; }
```

```
    public void Add(Component c) {};
```

```
}
```

```
public class Composite : Component {
```

```
    private List<Component> children = new List<Component>();
```

```
    public Composite(String s) { content = s; }
```

```
    public String Get() {
```

```
        String allContents = content;
```

```
        for (Component c: children) { allContents += c.Get(); }
```

```
        return allContents;
```

```
}
```

```
    public void Add(Component c){ children.Add(c); }
```

```
}
```

Method is useful for both subclasses



Delegates call to children



Example of an implementation of the Composite pattern

```
public abstract class Component
{
    protected String content;
    public abstract String get();
    public abstract void add(Component c);
}

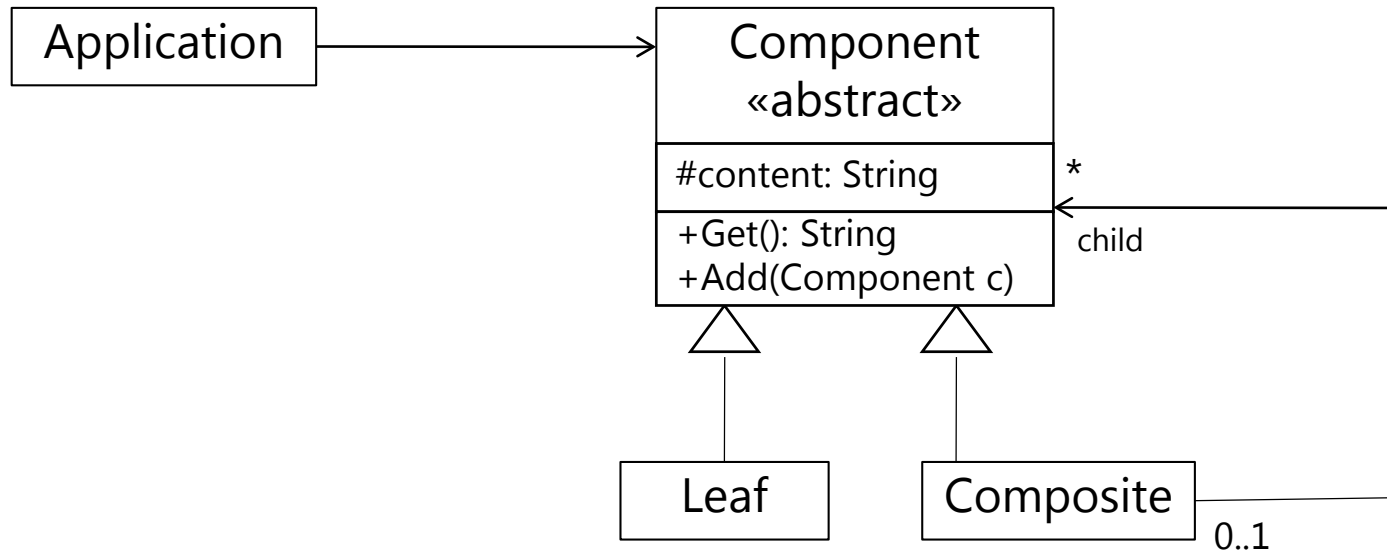
public class Leaf : Component {
    public Atom (String s) { content = s; }
    public String Get() { return content; }
    public void Add(Component c) {};
}

public class Composite : Component {
    private List<Component> children = new List<Component>();
    public Composite(String s) { content = s; }
    public String Get() {
        String allContents = content;
        for (Component c: children) { allContents += c.Get(); }
        return allContents;
    }
    public void Add(Component c){ children.Add(c); }
}
```

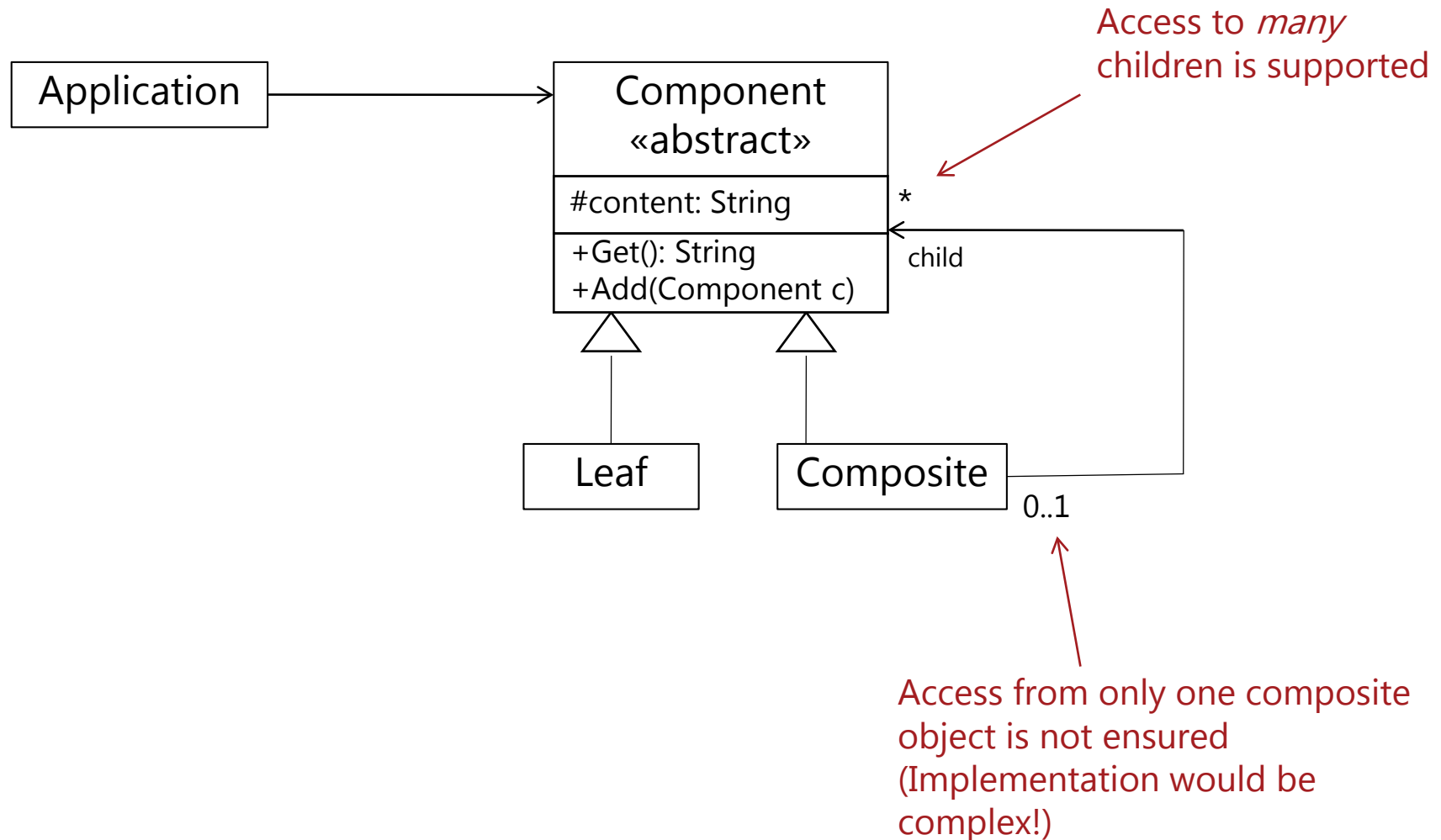
Method is only useful for one subclass



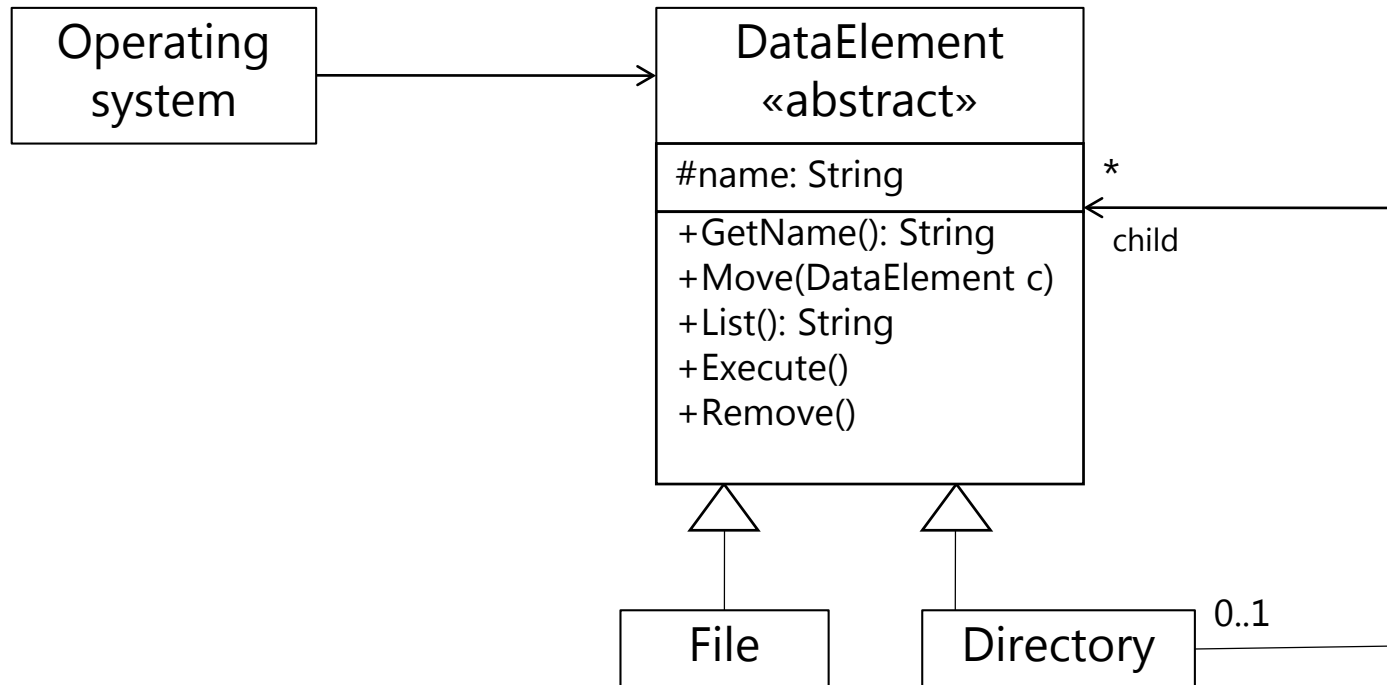
Class diagram for the example



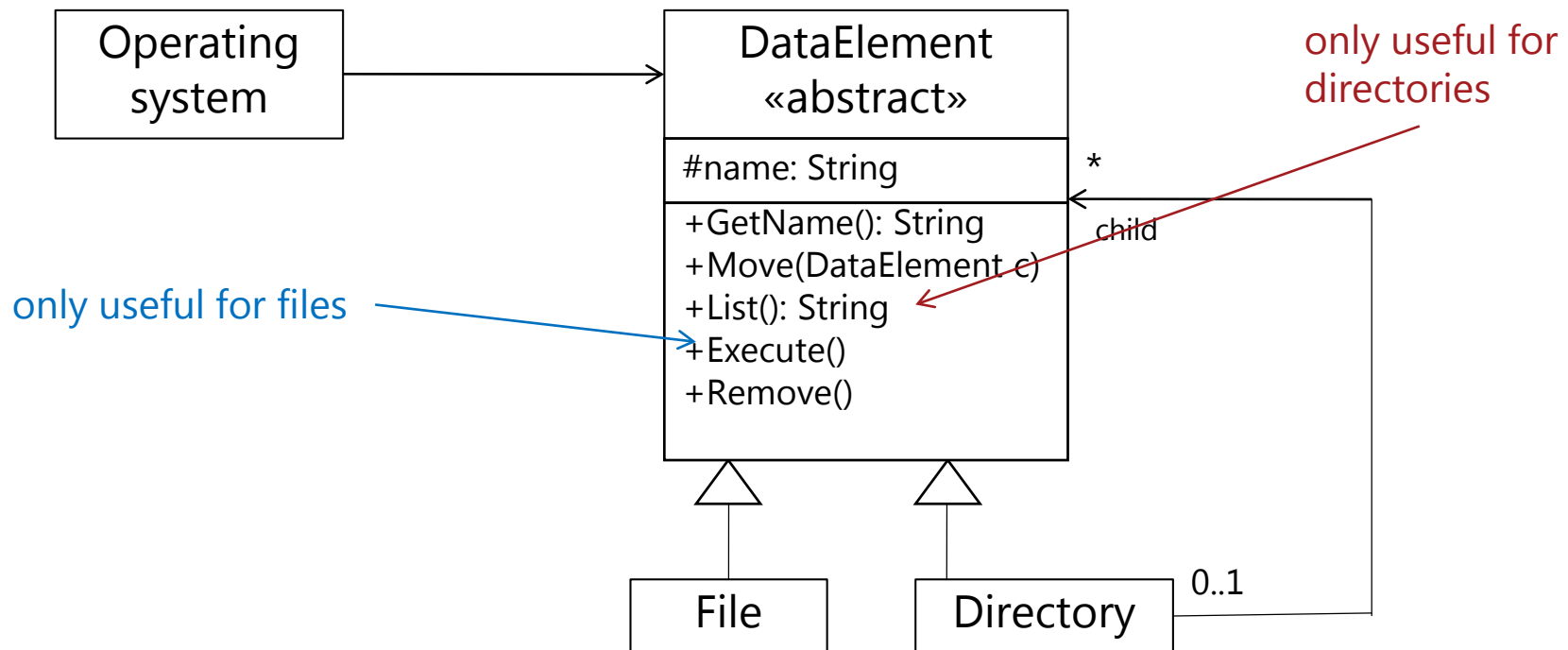
Annotations on implementation



Class diagram for a file system (example)



Class diagram for a file system (example)



Summary – Composite pattern

- Advantages:
 - Leaves and composites are treated uniformly.
 - Several types of leaves or several types of composites are possible.
 - Further leaves or composites can easily be supplemented.
 - It is not necessary to check the type of a component.
 - The constructed object structure is unlimited.
 - A tree is formed, which is composed of **specialized, heterogeneous** nodes.
- Disadvantages:
 - The common interface for leaves and composites results in methods, which do not trigger reasonable actions on all objects.
 - The structure can become **too general**, since composites can be restricted only with difficulty:
 - Number of children
 - Kind of children
 - Disjoint structure

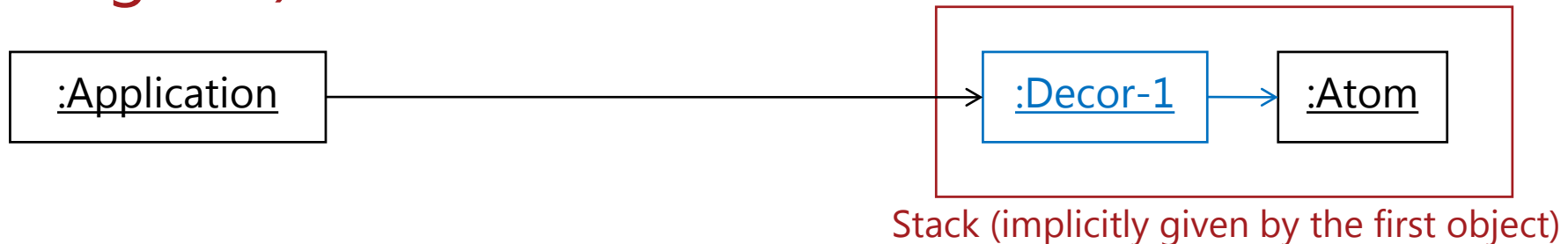
Decorator pattern

- A **Decorator pattern** allows the creation of heterogeneous stack structures.
- Preliminary note:
 - The class diagram leaves the impression, that the Decorator pattern is a simplified form of the Composite pattern.
 - **But:**
Decorator patterns are used with an entirely different purpose.

Decorator pattern

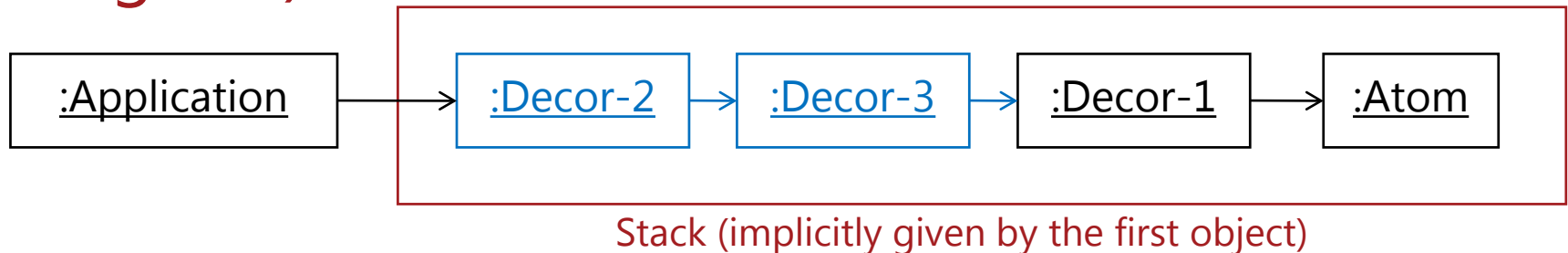
- A **Decorator pattern** allows the creation of heterogeneous stack structures.
- Approach:
 - A stack consists of structurally different elements,
 - the last element without a successor and
 - preceding elements, which have exactly one direct successor.
 - The different elements offer different attributes and different behavior.
 - But the attributes and behavior refer with regards to content always to all the following elements.
 - Each additional element expands (= decorates) the existing elements.
 - The entire stack has the same semantics like its second element.
 - Requirement:
All elements must offer the same interface to the outside.

Visualization Decorator pattern (object diagram)



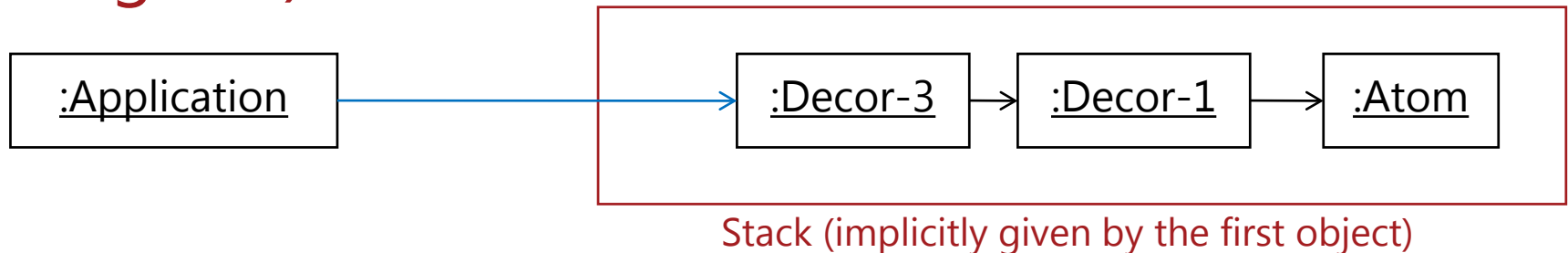
- Application scenarios for such stacks:
 - Further properties (= previous items) should be added dynamically to an output object (= the last element).
 - With the addition of a property, the overall behavior of the stack has to be changed, too.

Visualization Decorator pattern (object diagram)



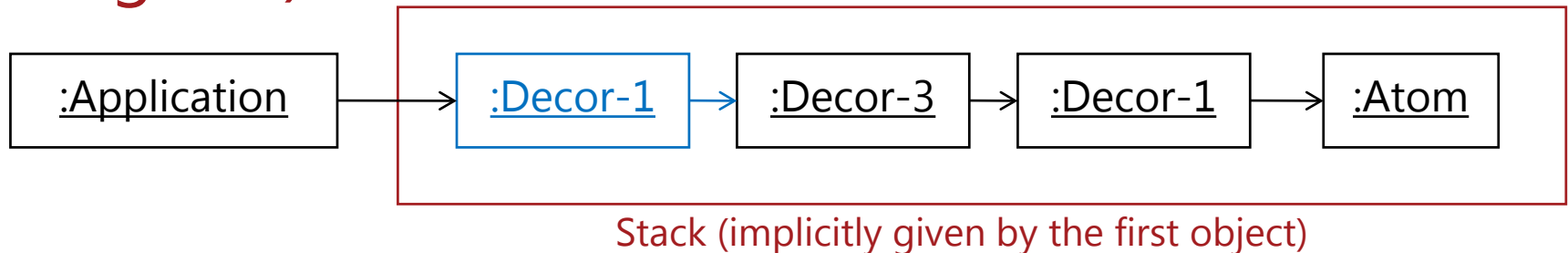
- Application scenarios for such stacks:
 - Further properties (= previous items) should be added dynamically to an output object (= the last element).
 - With the addition of a property, the overall behavior of the stack has to be changed, too.
 - The order, in which properties can be added, should be arbitrary.

Visualization Decorator pattern (object diagram)



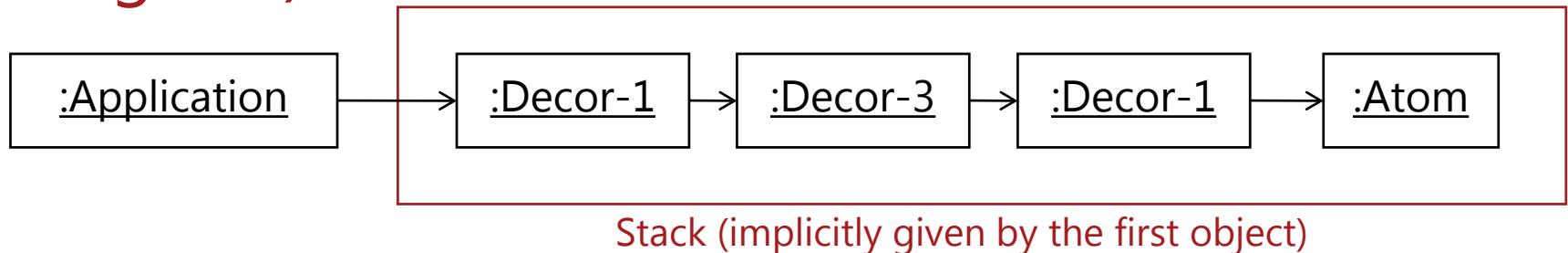
- Application scenarios for such stacks:
 - Further properties (= previous items) should be added dynamically to an output object (= the last element).
 - With the addition of a property, the overall behavior of the stack has to be changed, too.
 - The order, in which properties can be added, should be arbitrary.
 - The properties should also be able to be removed.

Visualization Decorator pattern (object diagram)



- Application scenarios for such stacks:
 - Further properties (= previous items) should be added dynamically to an output object (= the last element).
 - With the addition of a property, the overall behavior of the stack has to be changed, too.
 - The order, in which properties can be added, should be arbitrary.
 - The properties should also be able to be removed.
 - A property may also be added multiple times.

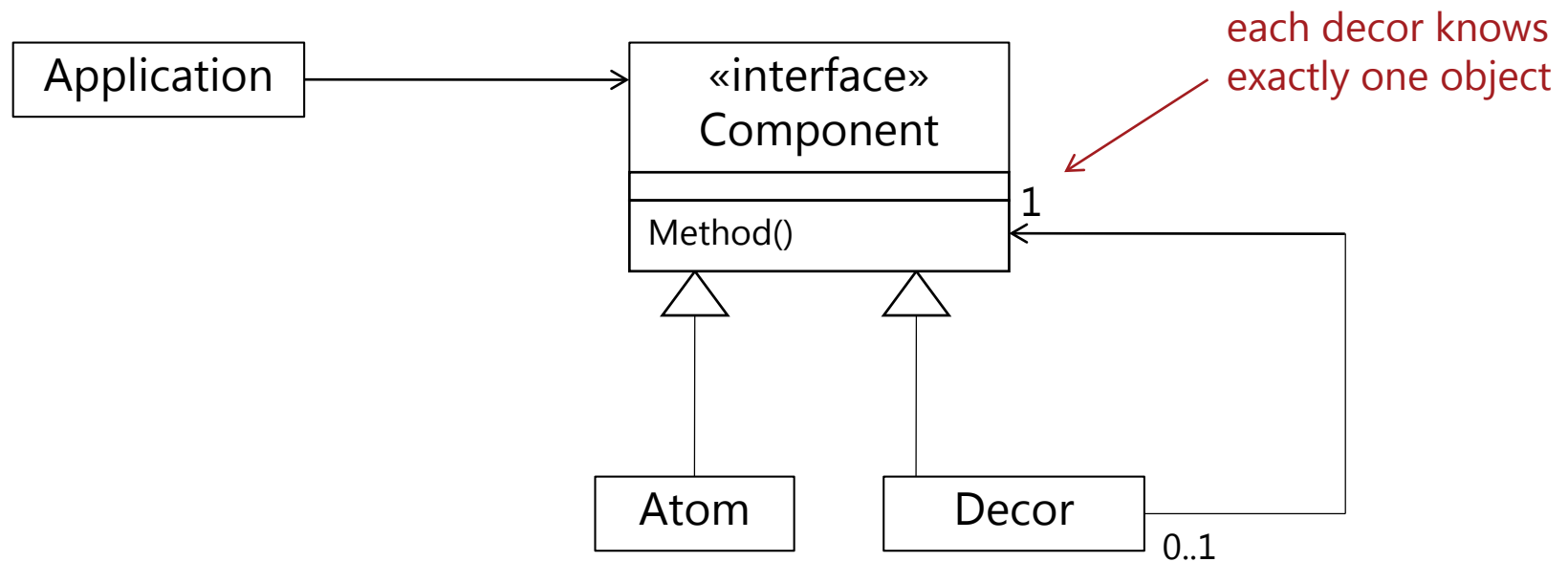
Visualization Decorator pattern (object diagram)



- Application scenarios for such stacks:
 - Further properties (= previous items) should be added dynamically to an output object (= the last element).
 - With the addition of a property, the overall behavior of the stack has to be changed, too.
 - The order, in which properties can be added, should be arbitrary.
 - The properties should also be able to be removed.
 - A property may also be added multiple times.
 - Further properties are to be supplemented easily during the development: All objects have to meet the same interface.

Application scenarios for such stacks

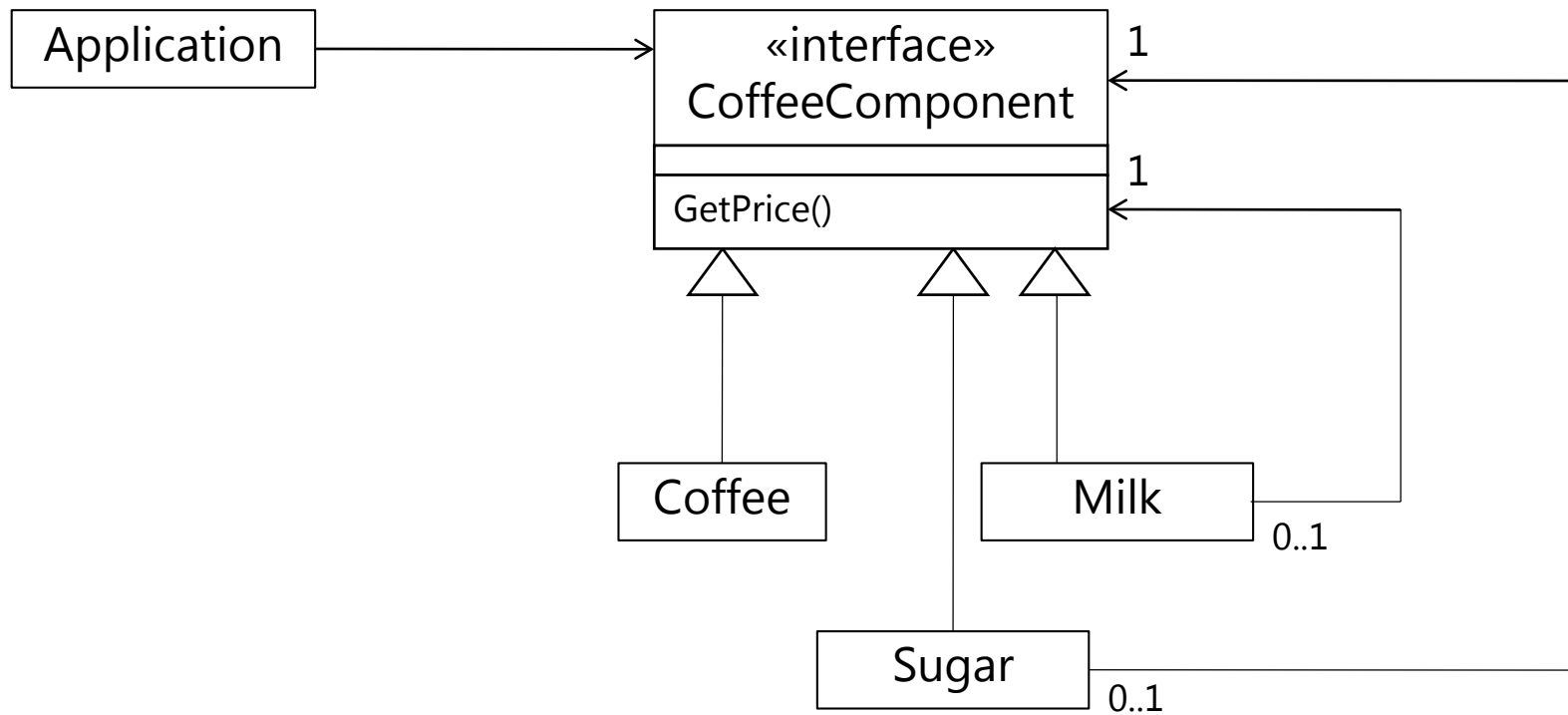
- Further properties are to be supplemented easily during the development: All objects have to meet the same interface.



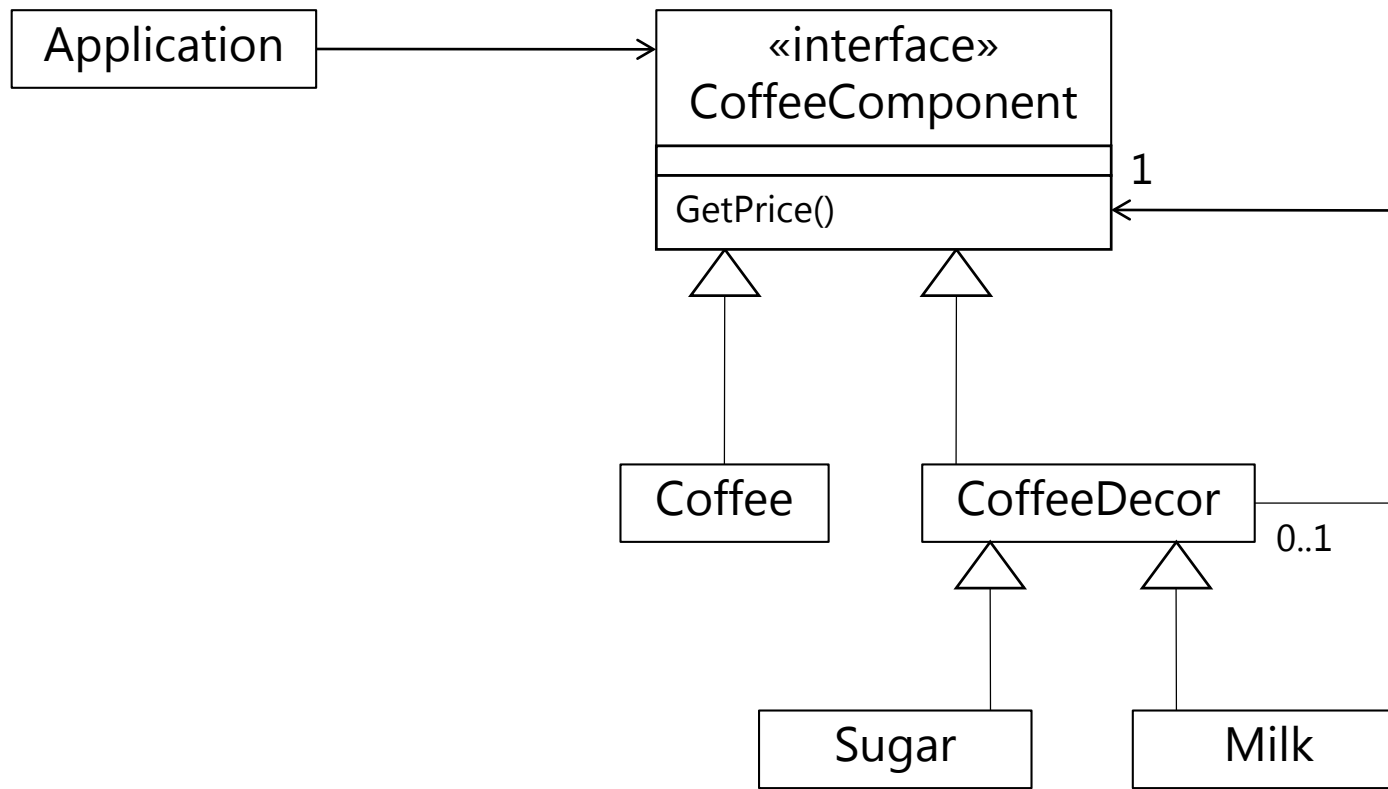
Examples for the use of the Decorator pattern

- GUI components are «decorated»:
 - Frames are added to the text field or to the graphics area.
 - Toolbars are added to a window.
 - Horizontal and vertical scrollbar is added to text field.
- Files (Stream classes)
 - File is a sequence of bytes (view close to hardware).
 - By decoration, behavior can be added, which interprets several bytes together.
 - By decoration, behavior can be added, which changes the file processing (e.g. buffers upon reading and writing).
- Non-technical example: Coffee is (maybe several times) decorated by
 - Sugar, cream, milk, chocolate, ...
 - And in doing so, each decor results in a specific price increase.

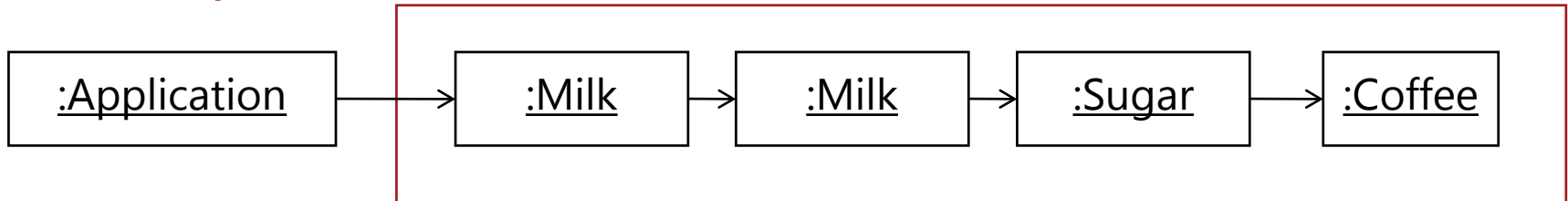
Visualization (example: price calculation for coffee)



Visualization (example: price calculation for coffee) - Alternative implementation



Visualization (example: price calculation for coffee)

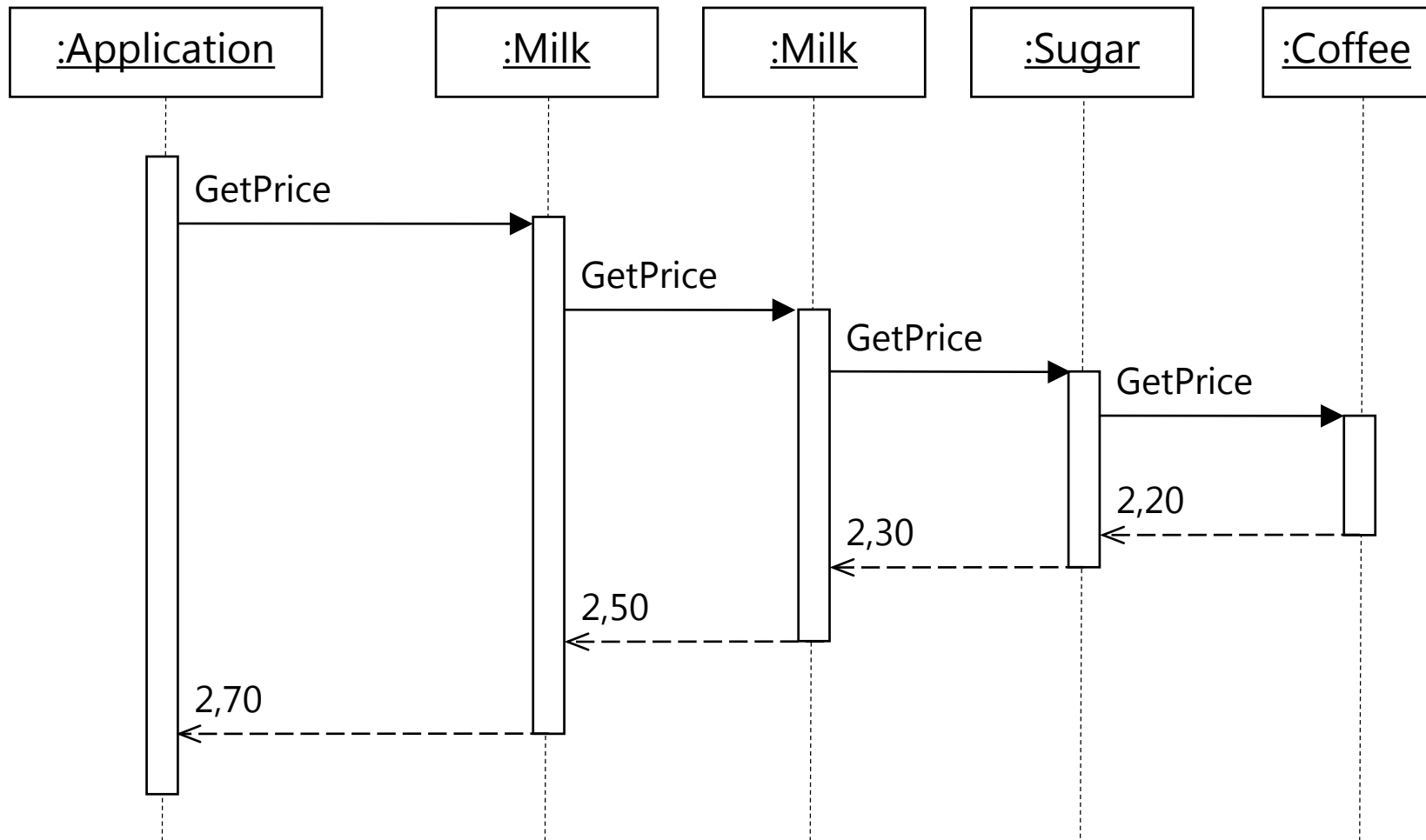


```
public class Milk : CoffeeComponent{  
...  
public double GetPrice() { return 0,20 + next.getPrice(); }  
}
```

```
public class Sugar : CoffeeComponent{  
...  
public double GetPrice() { return 0,10 + next.GetPrice(); }  
}
```

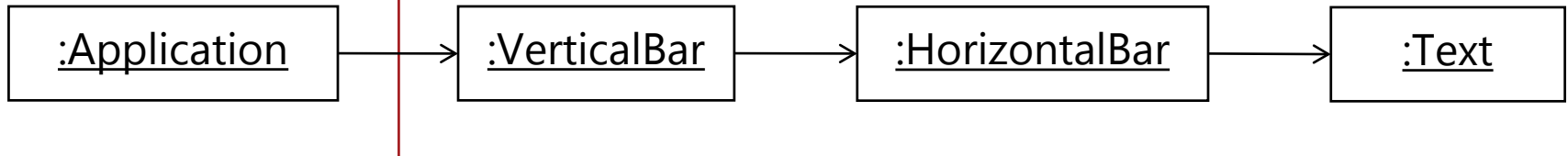
```
public class Coffee : CoffeeComponent{  
...  
public double GetPrice() { return 2,20; }  
}
```

Visualization (example: price calculation for coffee) – Method calls

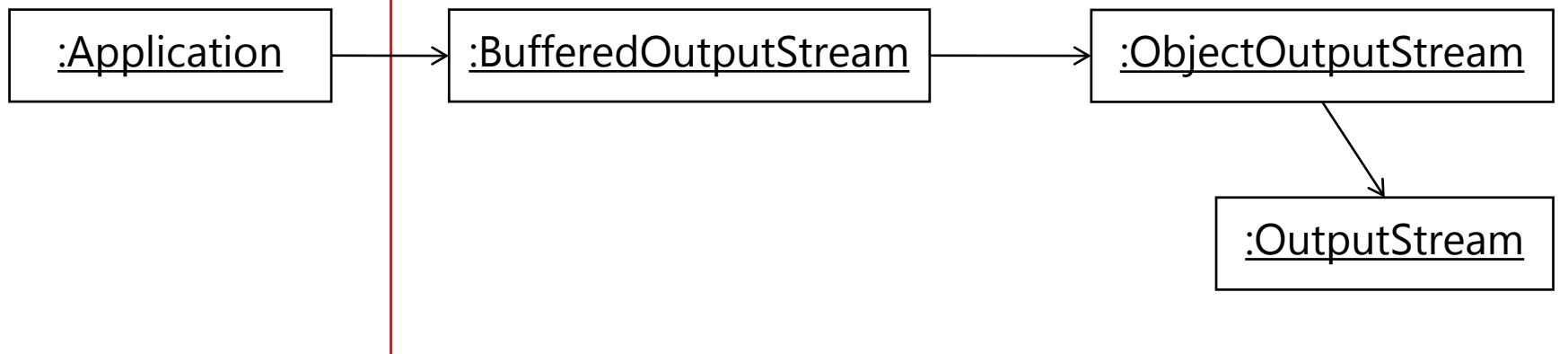


Visualization (further examples)

Decorated windows



Decorated streams



Summary – Decorator pattern

- Advantages:
 - Atoms and decors are treated uniformly.
 - Several types of atoms or decors are possible.
 - New atom or decor classes can easily be added.
 - An atom is always associated with decors. It is not necessary to convert already existing objects.
 - It is not necessary to check the type of a component.
 - The constructed object structure is unlimited.
 - A stack is created, which is made up of specialized, heterogeneous nodes.

Previously examined: examples of structure patterns

	Structural patterns	Behavioral patterns	Creational patterns
Class-related patterns	Class Adapter pattern	Interpreter pattern Template method pattern	Factory method pattern
Object-related patterns	Object Adapter pattern Decorator pattern Composite pattern Facade pattern Bridge pattern Flyweight pattern Proxy pattern	Strategy pattern Mediator pattern Observer pattern Chain of responsibility pattern Command pattern Iterator pattern Memento pattern State pattern Visitor pattern	Abstract factory pattern Singleton pattern Builder pattern Prototype pattern

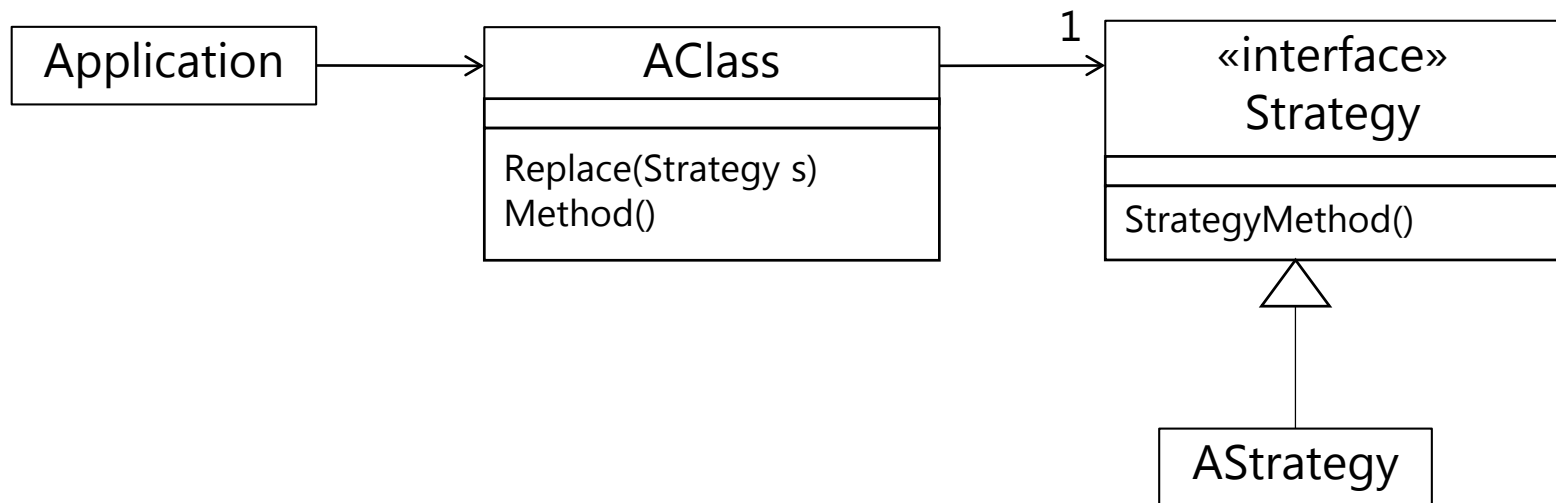
Now follows an example of a behavioral pattern: the **Strategy pattern**

Strategy pattern

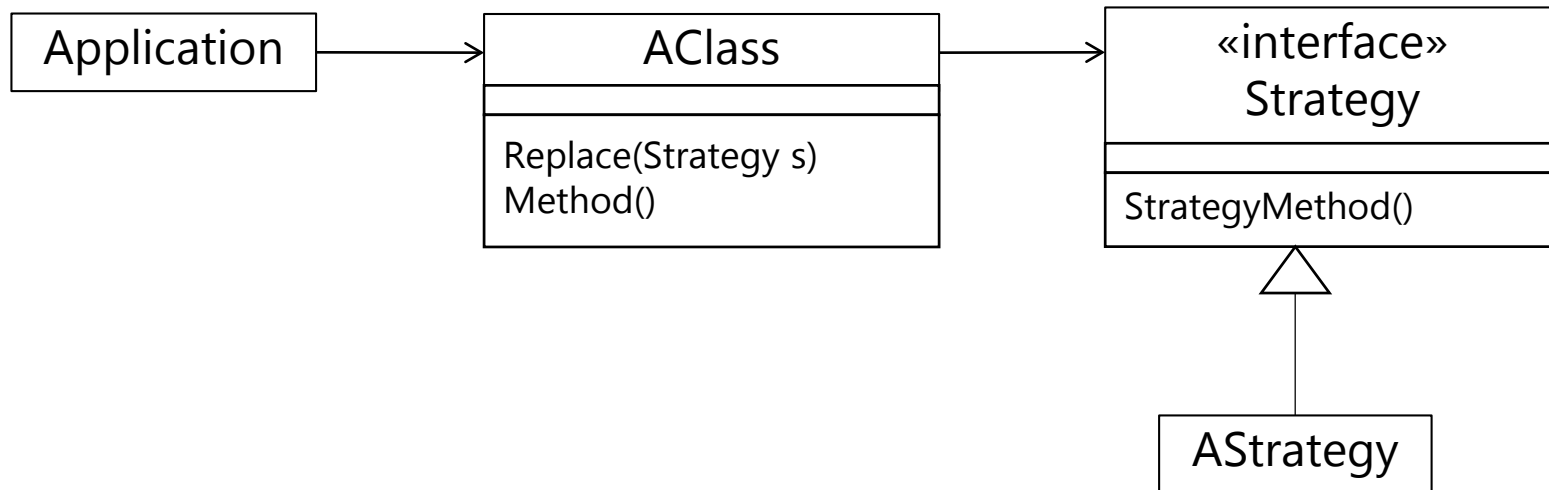
- A **Strategy pattern**
allows to change the behavior of the methods of an object during execution.
- Examples:
 - Arranging graphical elements in a window:
Next to each other, among each other, in a grid,
 - Design of text formatting:
Left-aligned, right-aligned, centered, ...
- **But:**
 - Various types of behavior should be provided, without expanding the class of the executing object.
 - The behaviour is supposed to be established only after the declaration of the class of the executing object.

Strategy pattern – idea of construction

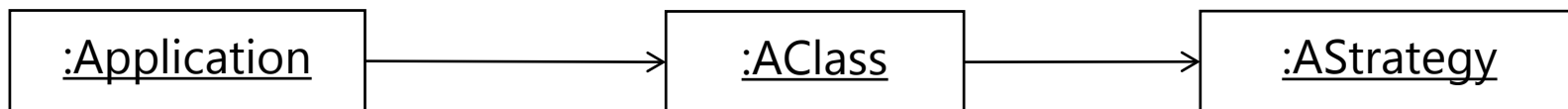
- The behavior , which has to be changed, is encapsulated in a separate **strategy** class, which maintains a given interface.
- The behavior of a strategy object is accessed during the execution.
- If changes are made, the strategy object is replaced.



Strategy pattern – structure of objects



Object diagram

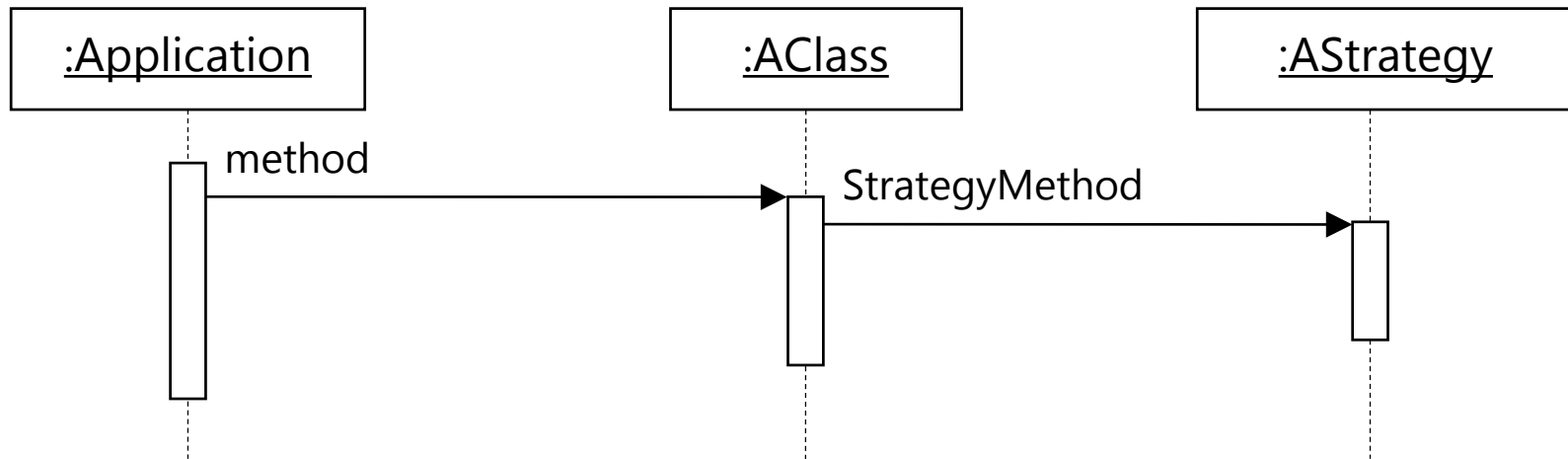


Strategy pattern – structure of objects

Object diagram



Sequence diagram



Summary – Strategy pattern

- The use of the pattern strategy is in some object-oriented programming languages the only way to exchange the behaviour during execution.
- A strategy class also offers the possibility of combining the interchangeable behavior with status information.

Iterator pattern

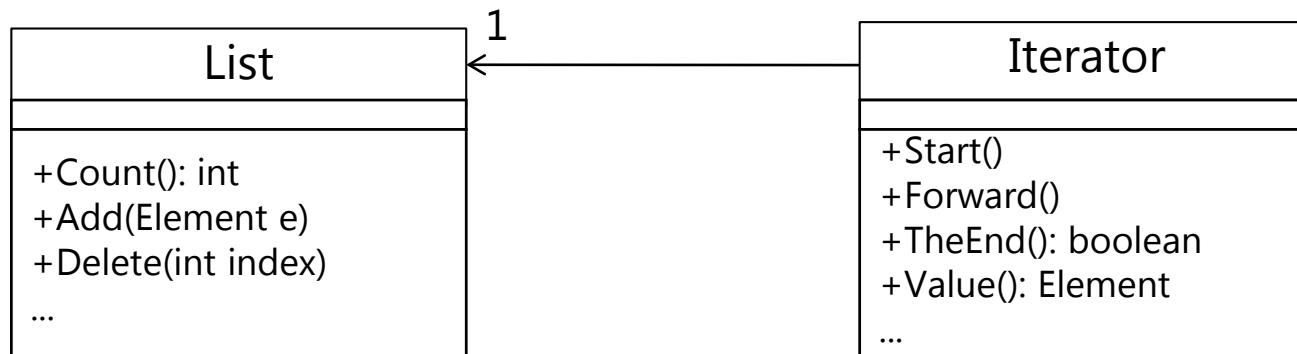
- An **Iterator pattern** allows sequential access to the elements of a composite object (aggregate, e.g. list or tree) without revealing its underlying structure.
- Examples:
 - Sequential throughput through a list
 - Sequential throughput through a binary tree
 - Sequential throughput through a composite structure (n-ary tree with different types of nodes)
- **But:**
 - The exact structure of the aggregate should have no meaning for the throughput.
 - Several passages through an aggregate should be possible at the same time.
 - Various types of throughputs are to be provided, without inflowing the aggregate class.

Iterator pattern

- Idea:

Responsibility for the throughput is not in the aggregate object, but is taken over by an **iterator** object.

- Example list: general structure



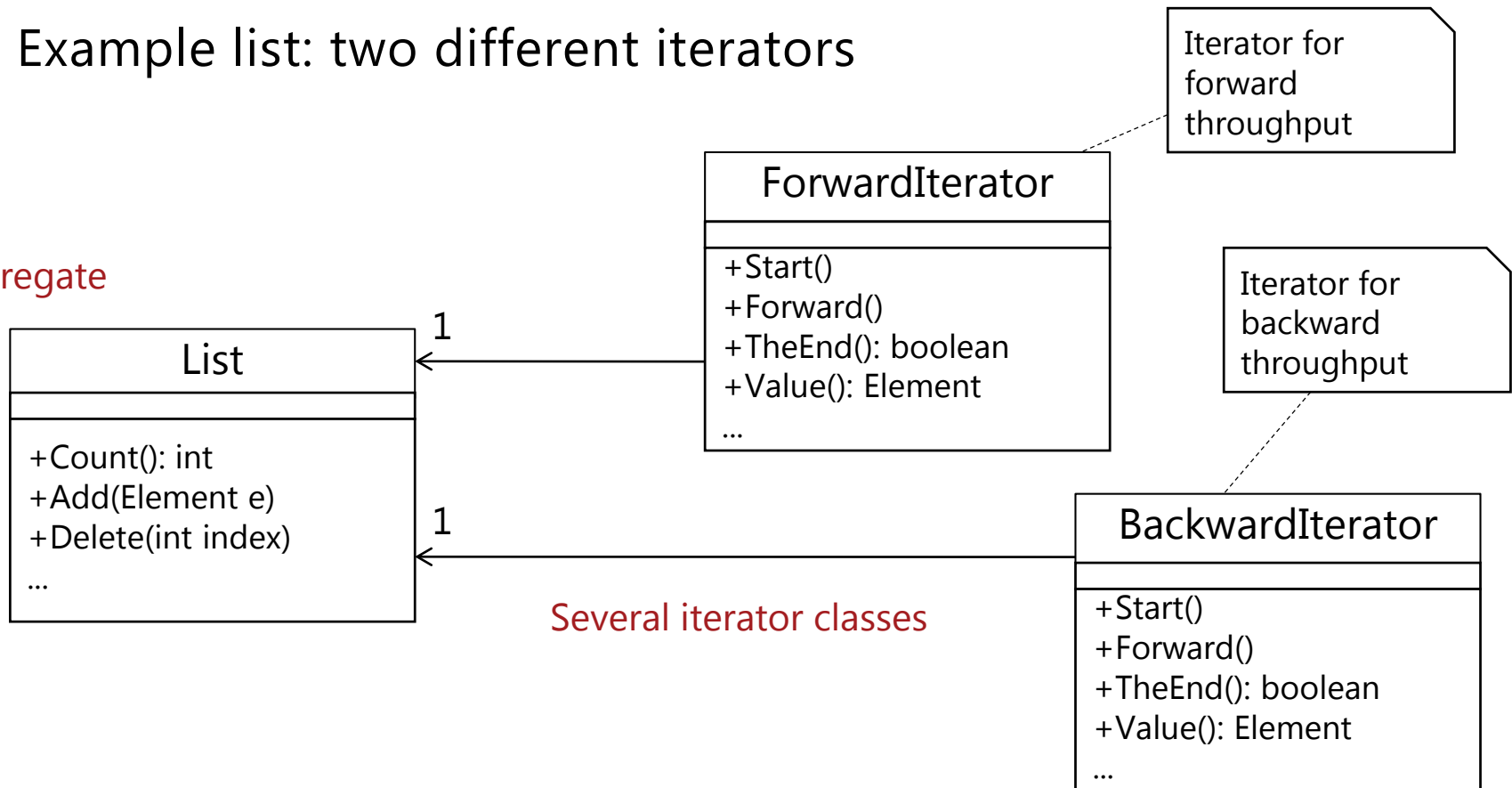
Iterator pattern

- Idea:

Responsibility for the throughput is not in the aggregate object, but is taken over by an **iterator** object.

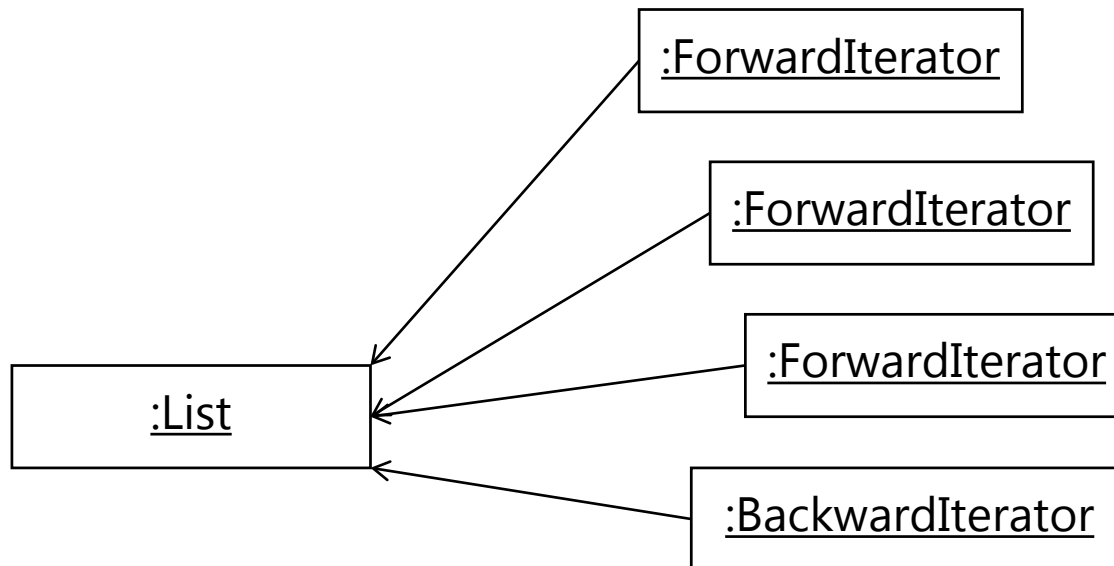
- Example list: two different iterators

Aggregate



Iterator pattern

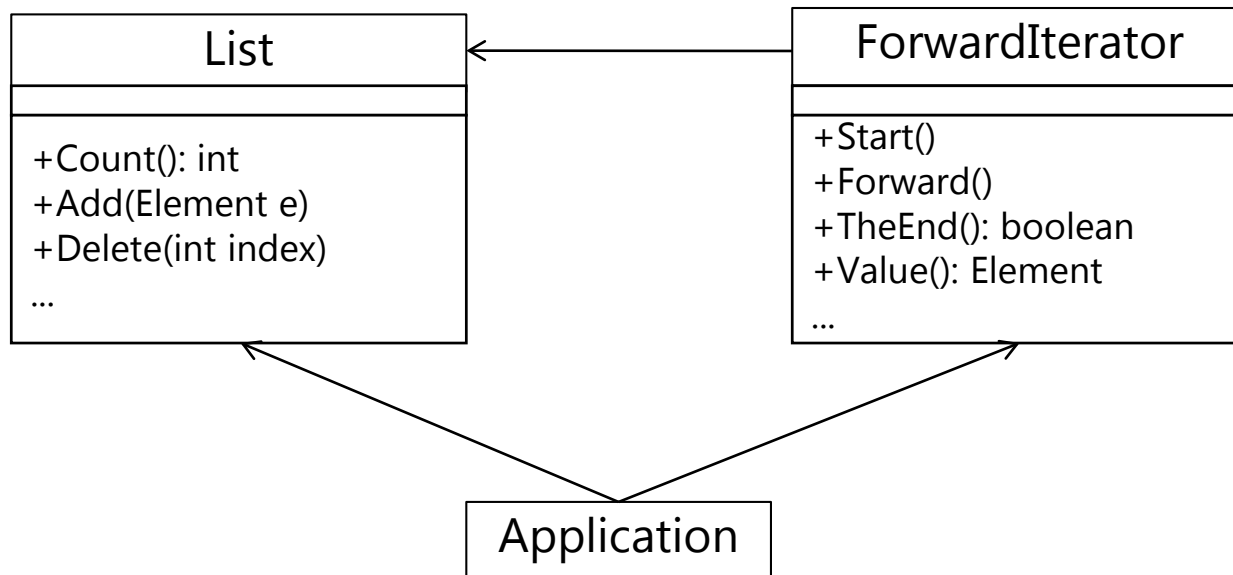
- Idea:
Responsibility for the throughput is not in the aggregate object, but is taken over by an **iterator** object.
- Example list: object diagram



Several iterator classes

Iterator pattern

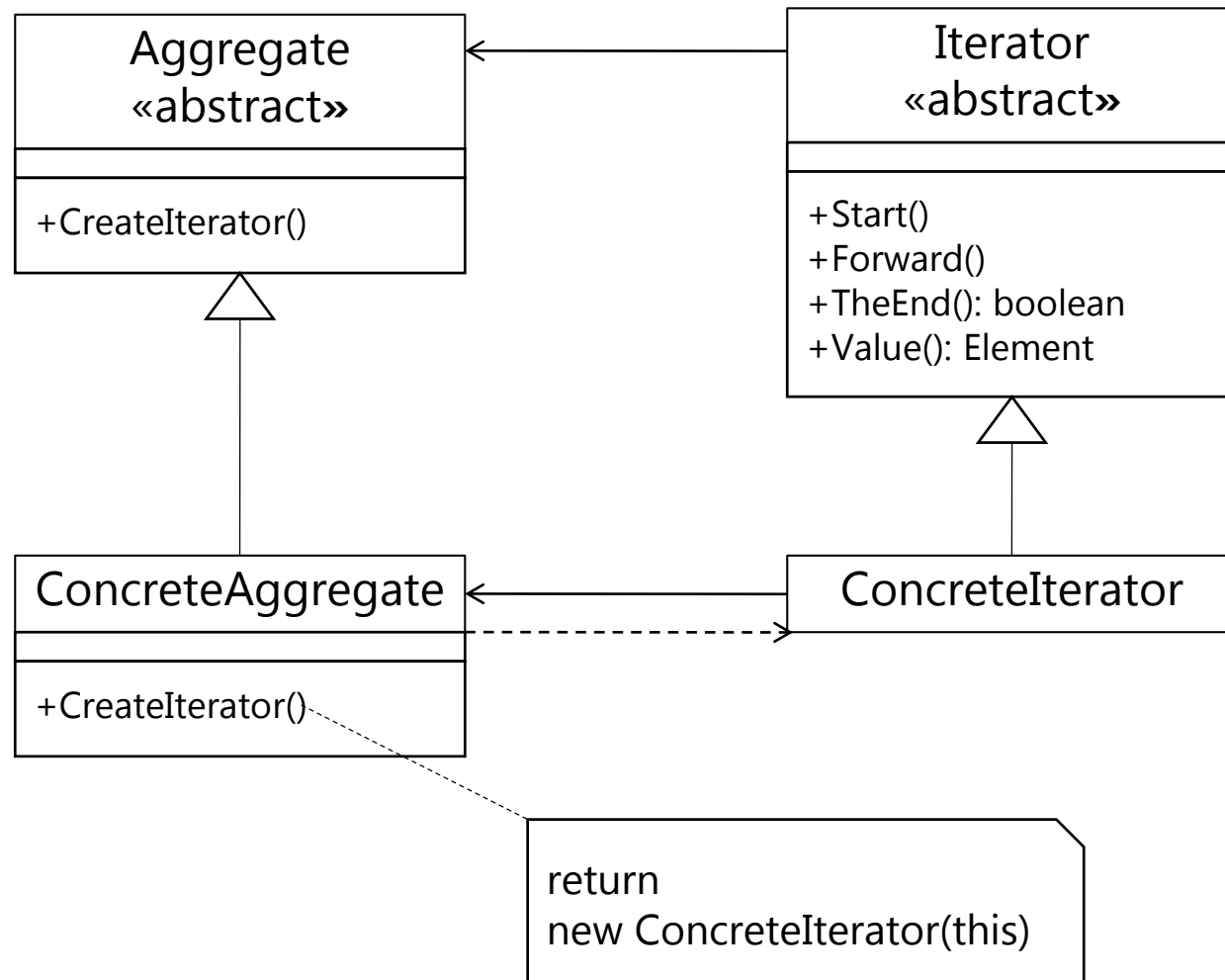
- The application must ensure that the aggregate and iterator match each other, e.g. connecting List and ForwardIterator.



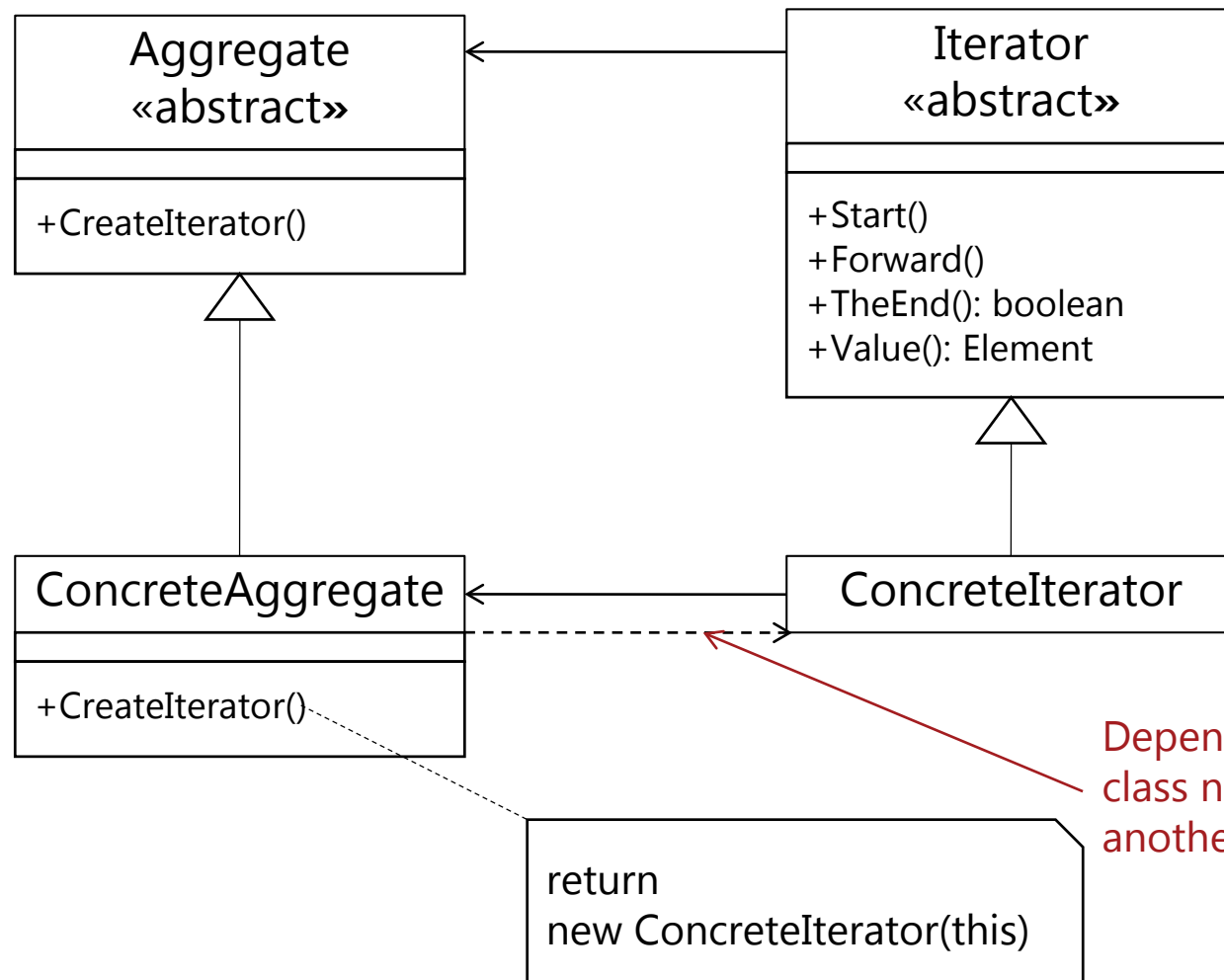
Iterator pattern

- But:
 - The application would need to know, which data structure the aggregate has. Otherwise, no reasonable allocation could be made.
 - The iterator could only access public methods of the aggregate.
- Therefore the following improvement:
 - The aggregate implements a general interface, which makes the aggregate **iteratable**.
 - The iterator implements a uniform interface for iterators, so that all iterators can be used equally.
 - Each iteratable aggregate has its **own iterator**, which is provided by the aggregate.
 - Thus the aggregate and the iterator are closely interlinked. The iterator can obtain a preferred access to the aggregate, thereby performing an (efficient) implementation.

Class diagram Iterator pattern (general view)



Class diagram Iterator pattern (general view)




Iterator pattern - evaluation

- Advantages:
 - The implementation of the data structure underlying the aggregate remains hidden.
 - The aggregate objects can be exchanged without modifying the application, since the access via the iterator remains unchanged.
- Disadvantages:
 - Iterators require additional programming effort.
 - Iterators may need to be informed of changes to the aggregate.

Iterator pattern - implementation

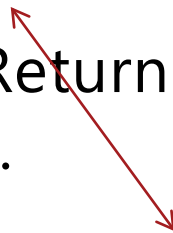
```
interface IEnumerable<> {  
    IEnumerator<T> GetEnumerator();  
    // Returns an enumerator over a set of elements of  
    type T.  
}
```



The interface only requires that there is a method, which returns an enumerator.

Iterator pattern - implementation

```
interface IEnumerable<T> {  
    IEnumerator<T> GetEnumerator();  
    // Returns an enumerator over a set of elements of  
    // type T.  
}
```

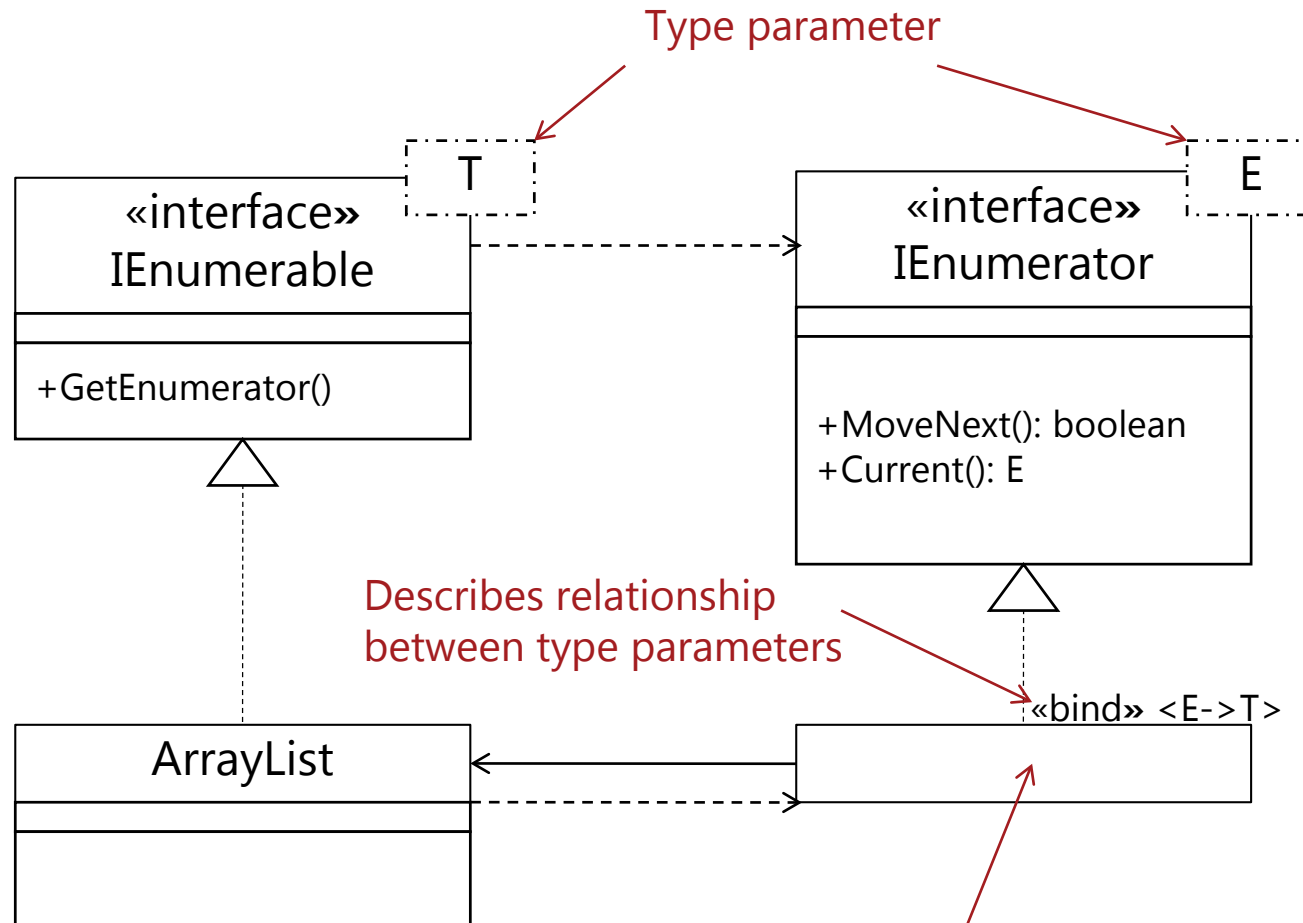


```
interface Iterator<E> {  
    public E Current; // get current element in scope  
    public bool MoveNext(); // move to next element  
    ...  
}
```

Iterator pattern - implementation

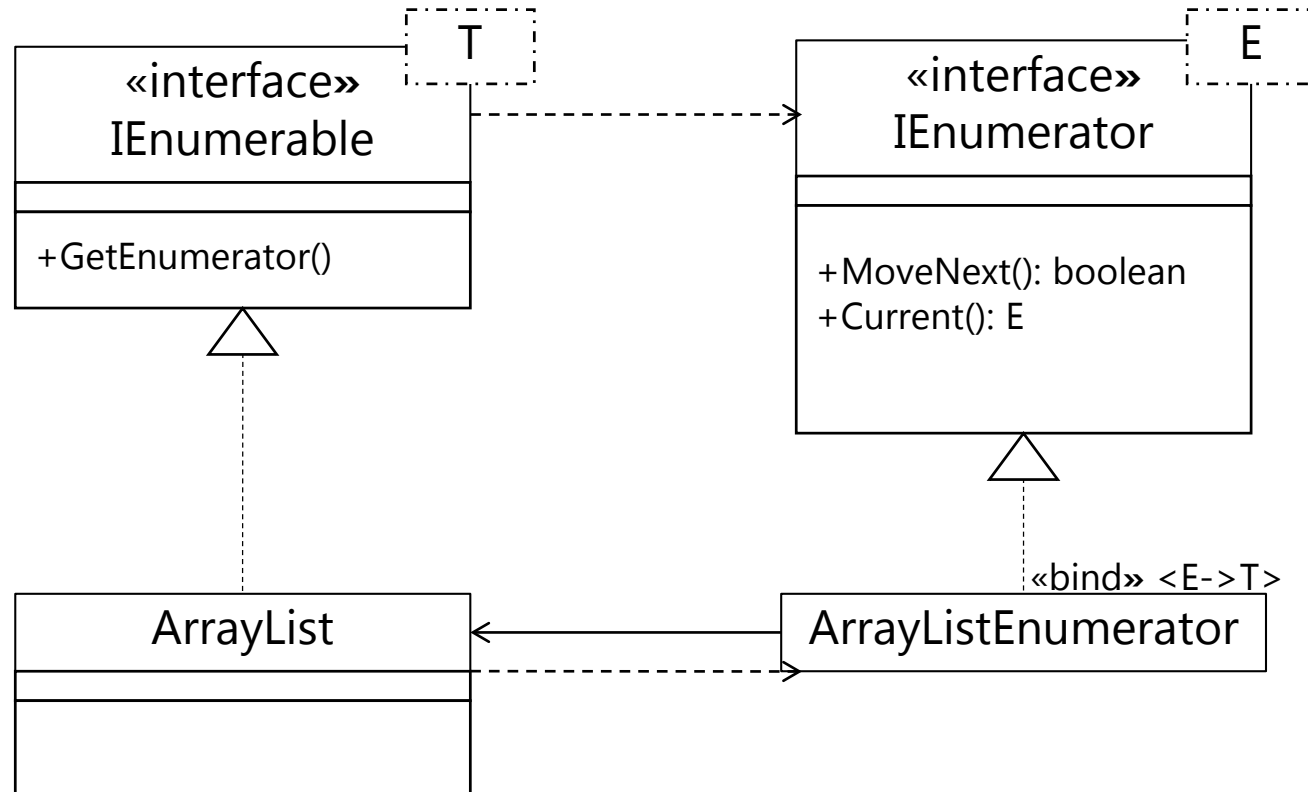
- The **interface** `IEnumerable<T>` is implemented amongst others by the following classes: `List<T>`, `LinkedList<T>`, `Queue<T>`, `Stack<T>`, `SynchronizedCollection<T>`
- Each of these classes provides an `GetEnumerator()` method using an `IEnumerator<T>` object.
- Some classes provide additional iterators

Iterator pattern - implementation



Name unknown:
The knowledge that the object implements
the `IEnumerator` interface is sufficient

Iterator pattern - implementation



Iterator pattern – example for application

- General method for searching on any structures, which implement Iterable:

```
<T> bool check(IEnumerable<T> struc, T value) {  
    IEnumerator<T> it = struc.GetEnumerator();  
    while (it.MoveNext()) {  
        if (value.equals(it.current)) return true;  
    }  
    return false;  
}
```

Supplies enumerator
object

Checks whether the
enumerator has reached the
end of struc

Supplies the current element
from struc

Iterator pattern – example for application

- Example shows the advantages of the Iterator pattern:
 - The data structure underlying the implementation of the aggregate has no meaning for traversing the aggregate with the iterator.
 - Various aggregates can be treated equally.
 - Aggregates can be exchanged without changing the application, since the access via the iterator remains unchanged.
 - Iterator object is external to the iterated data structure, so that the progress of the iterator can be determined by a method call from the outside (e.g., MoveNext ()).
 - Implementation options:
 - The traversing: The throughput takes place in the aggregate.
 - A copy of the data structure of the aggregate is created within the iterator and this copy will be run through during the traversal.
 - A need for clarification when implementing an iterator:
How do changes affect the iterated data structure?

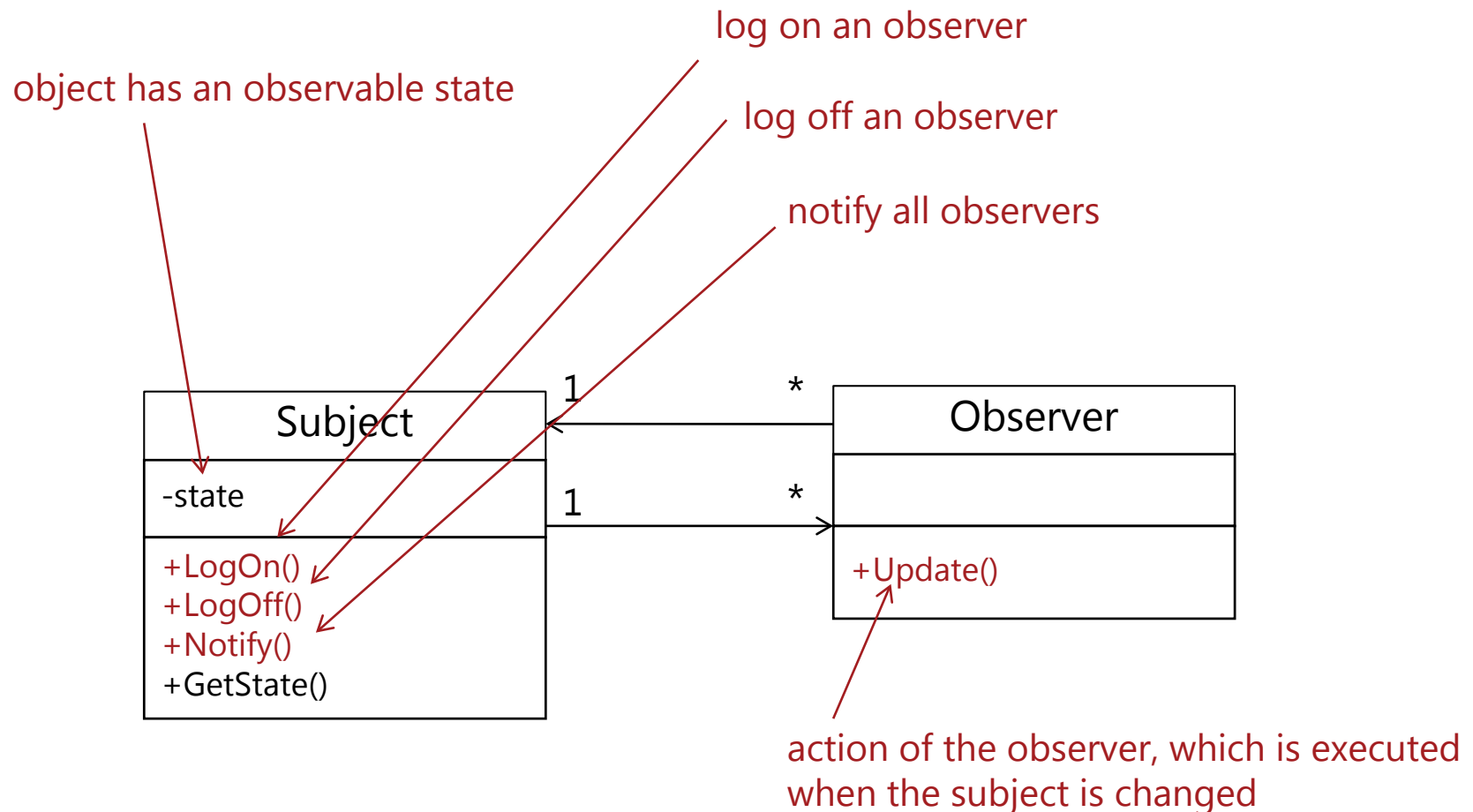
Observer pattern

- An **Observer**
allows the detection of changes to objects.
- Examples:
 - Arrival of a new order
 - Log on to a new user
 - Import of new data with other software
 - In the real world, observing is an active activity by the observers.
 - Many observers can notice the same change immediately.
 - But then **all** observers are constantly concerned with the process of observing.

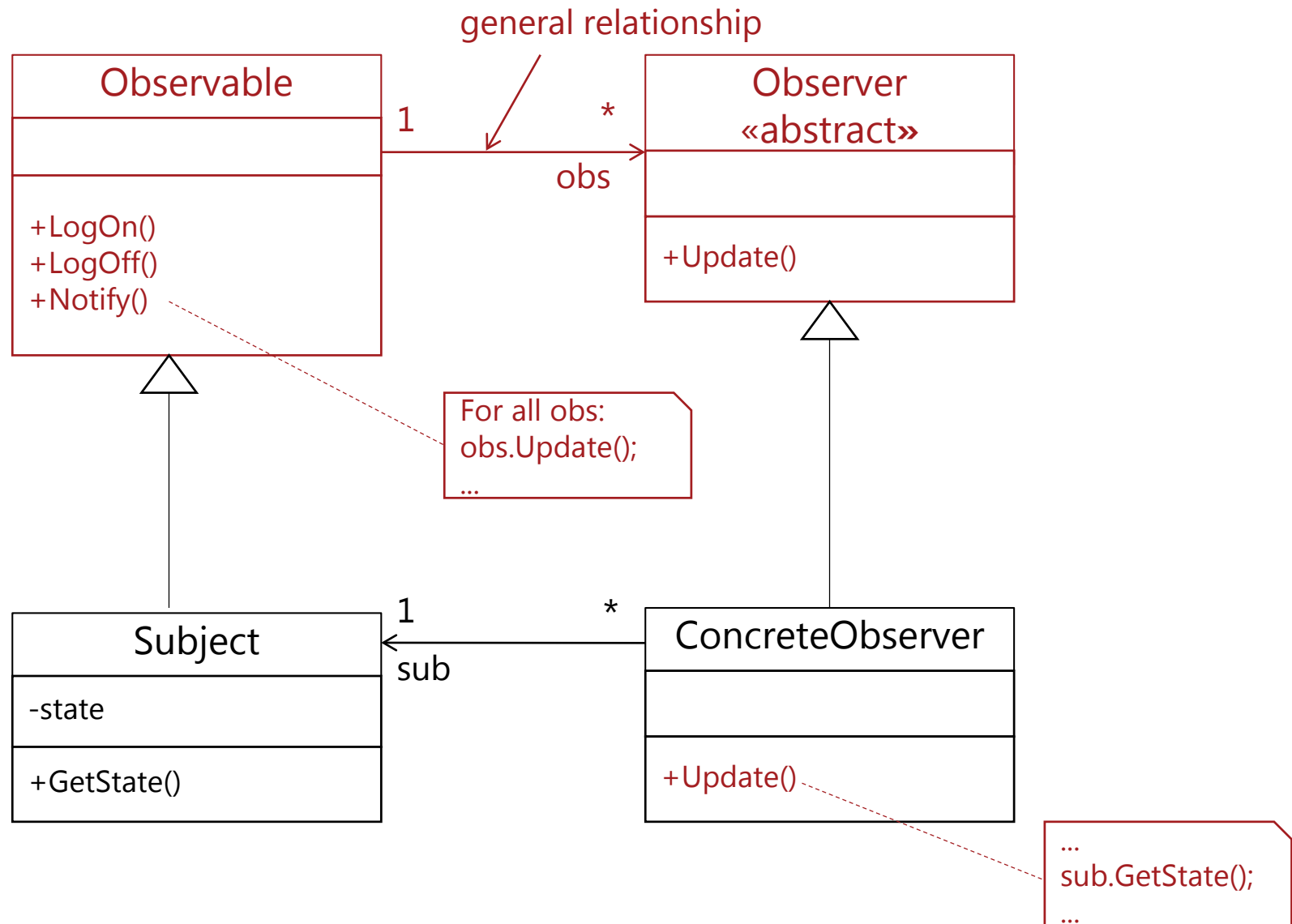
Objective of the Observer pattern

- Changes to another object – the subject - are to be made known to interested objects – the observers - quickly and
- at the same time with little effort.
- For this purpose, a property of programmed solutions is used:
 - Objects in the program cooperate reliably.
 - In the observer pattern, the (observed) subject and the observer cooperate.
 - Equations in the real world would be:
 - Amazon notifies about a new book of the favorite author.
 - The thief informs the detective about his theft.
- **Idea:**
 - The (observed) subject allows the logging on and off of observers.
 - Observers are waiting passively for notification by the subject.
 - The subject has a notification mechanism and informs all registered observers that an event has occurred.
 - The observer can obtain information about the subject after the notification of an event, which has occurred, thus completing the observation.

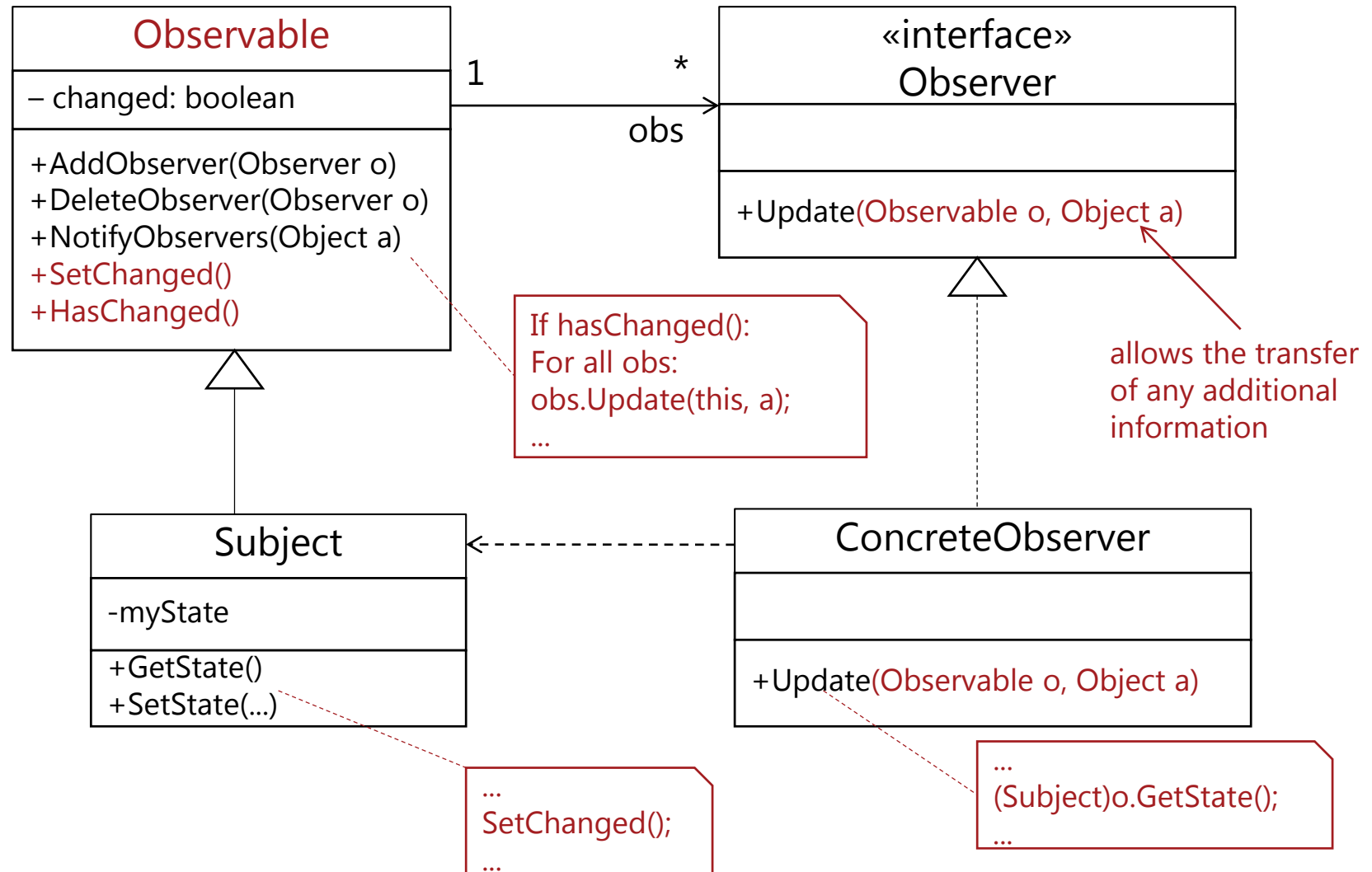
Structure of the Observer pattern



Structure of the Observer pattern – general



Structure of the Observer pattern – realization



Observer pattern: realization

- Methods of the class Observable

- | | |
|-----------------|--|
| AddObserver | checks in an observer, which implements the Observer interface |
| DeleteObserver | checks out an observer |
| SetChanged | sets changed-attribute, which indicates whether the subject was changed |
| HasChanged | returns the value of the changed-attribute |
| NotifyObservers | notifies all observers, but only if the subject has actually been changed:
The changed-attribute allows a decoupling of change and notification |

- Methods of the interface Observer

- | | |
|--------|--|
| Update | <ul style="list-style-type: none">• is called by the NotifyObservers method and has to implement the actions that are to be performed when a change occurs in the specific observer.• since the observed subject hands itself as a parameter, the observer does not need an attribute to remember it! |
|--------|--|



The flow will now be somewhat more complex, its description too!

(The observer pattern is a behavioral pattern!)

Description of the order of method calls

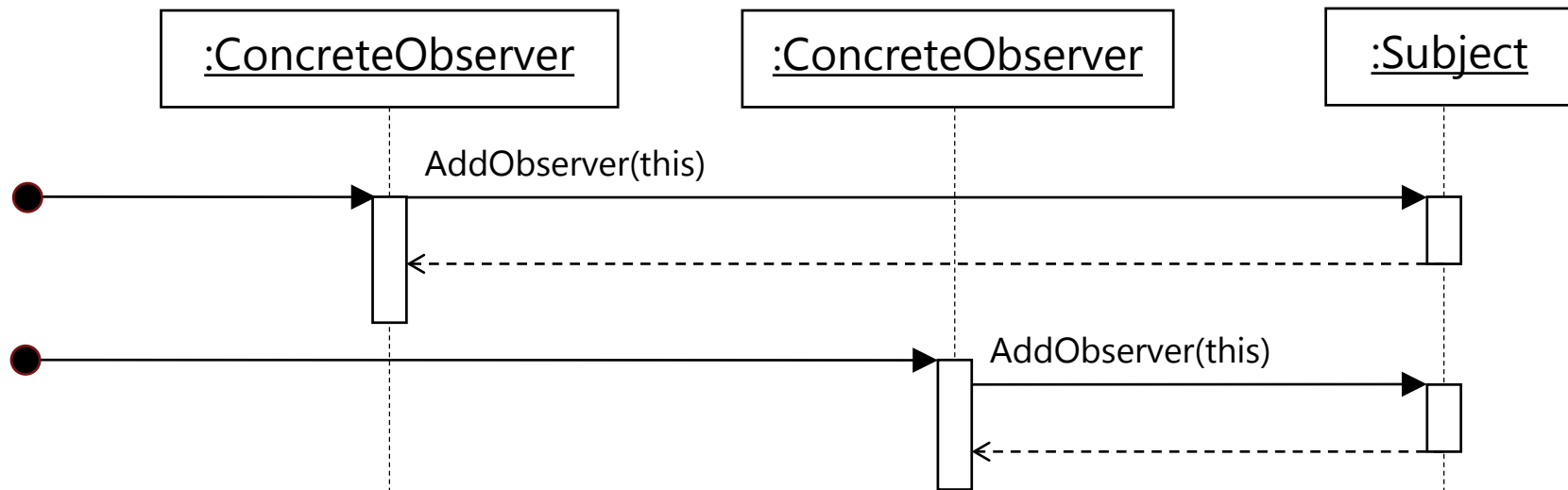
- Sequence diagram

- It shows the flow of communication between objects.
- It shows the temporal sequence of the communication steps.
- It only displays the call of methods.

One speaks also generally of message exchange:

- *Calling a method* m for an object obj is interpreted as *sending a message* to obj .
- The *termination* of m (and possibly the return of a value) are understood as the *answer* of obj .

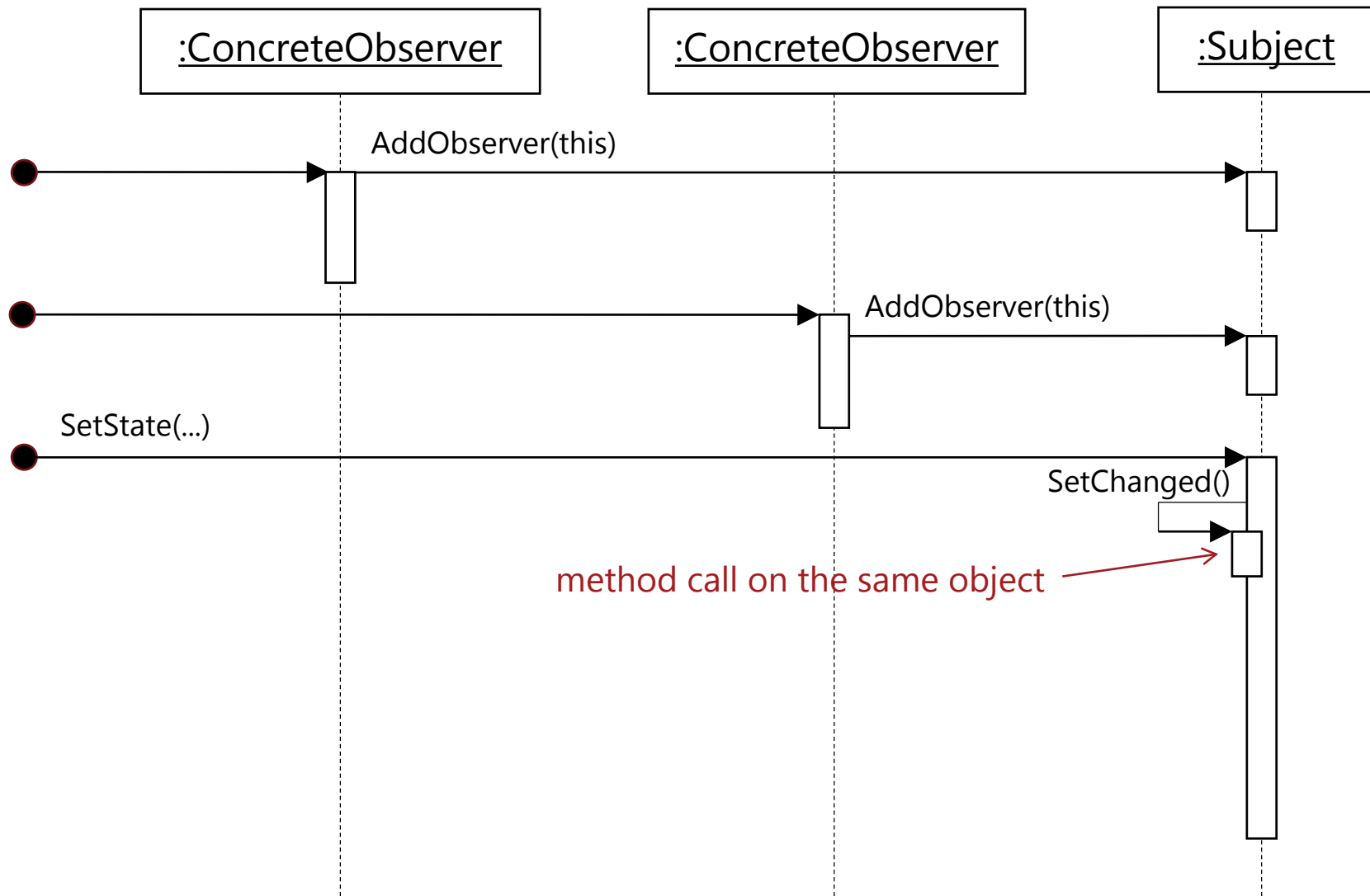
Sequence diagram for describing the flow of the observer pattern



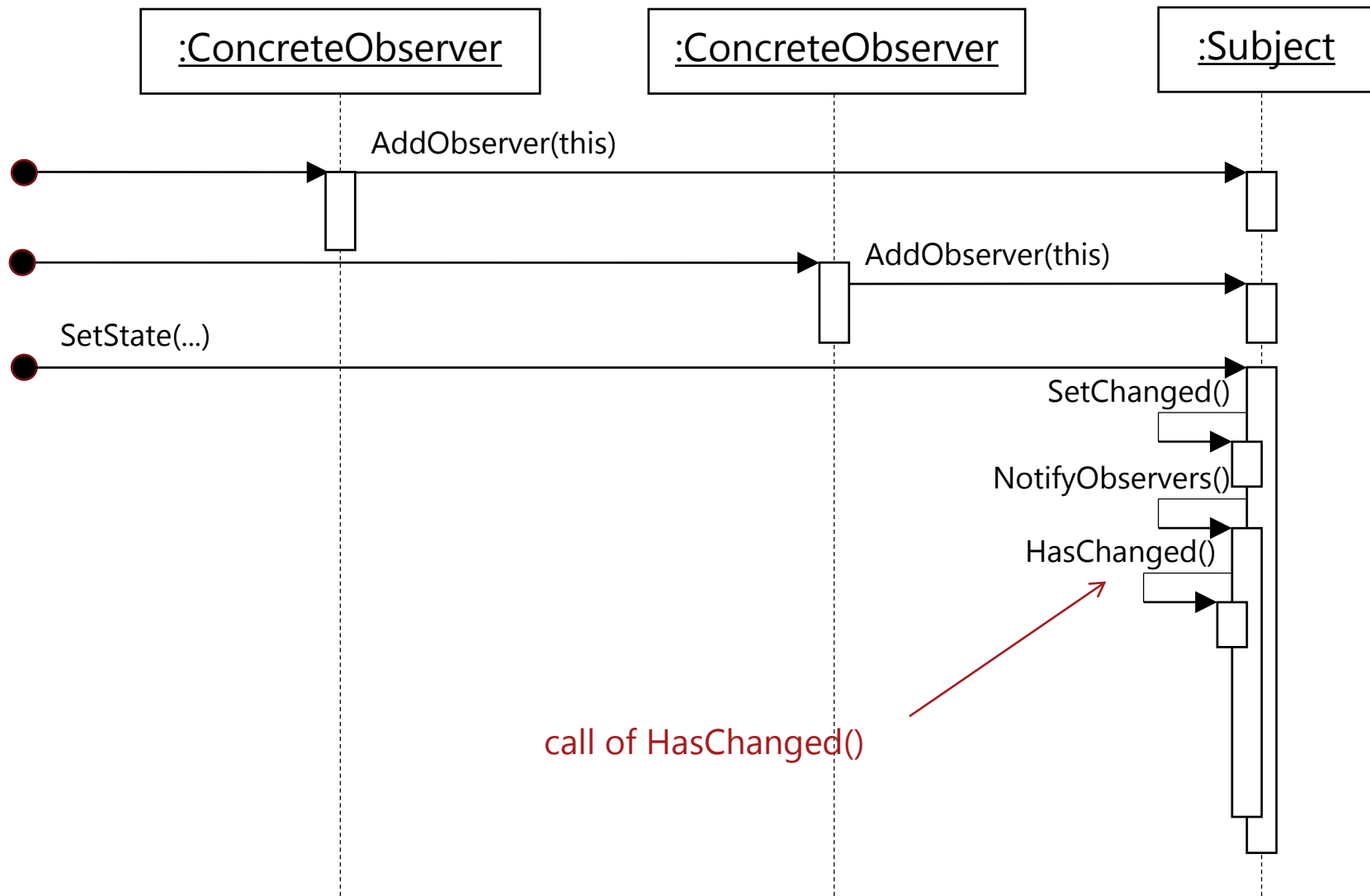
current situation:

- both Observer-objects have registered themselves as observers
- now follows: state of the subject changes, observers are notified

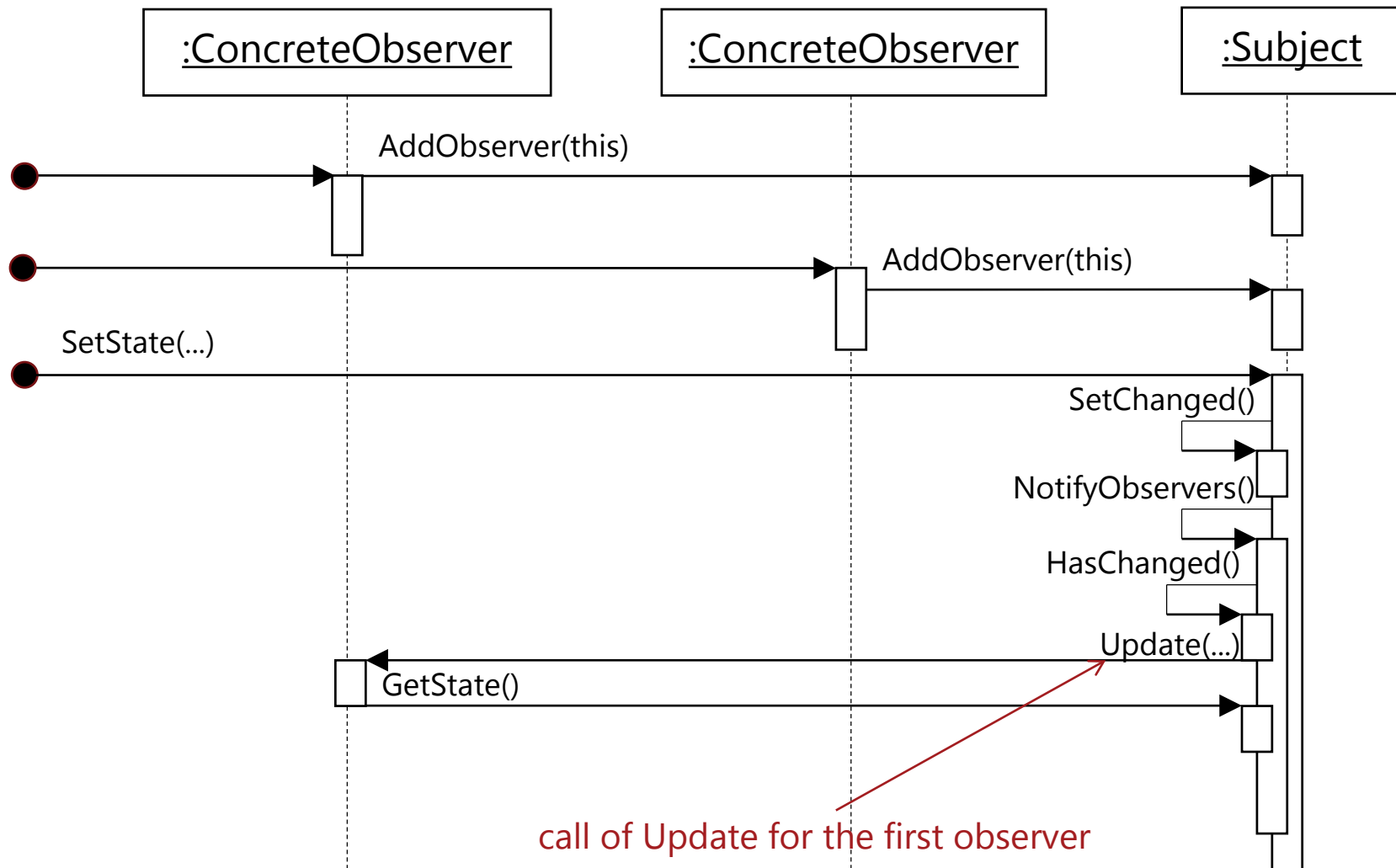
Sequence diagram for describing the flow of the observer pattern



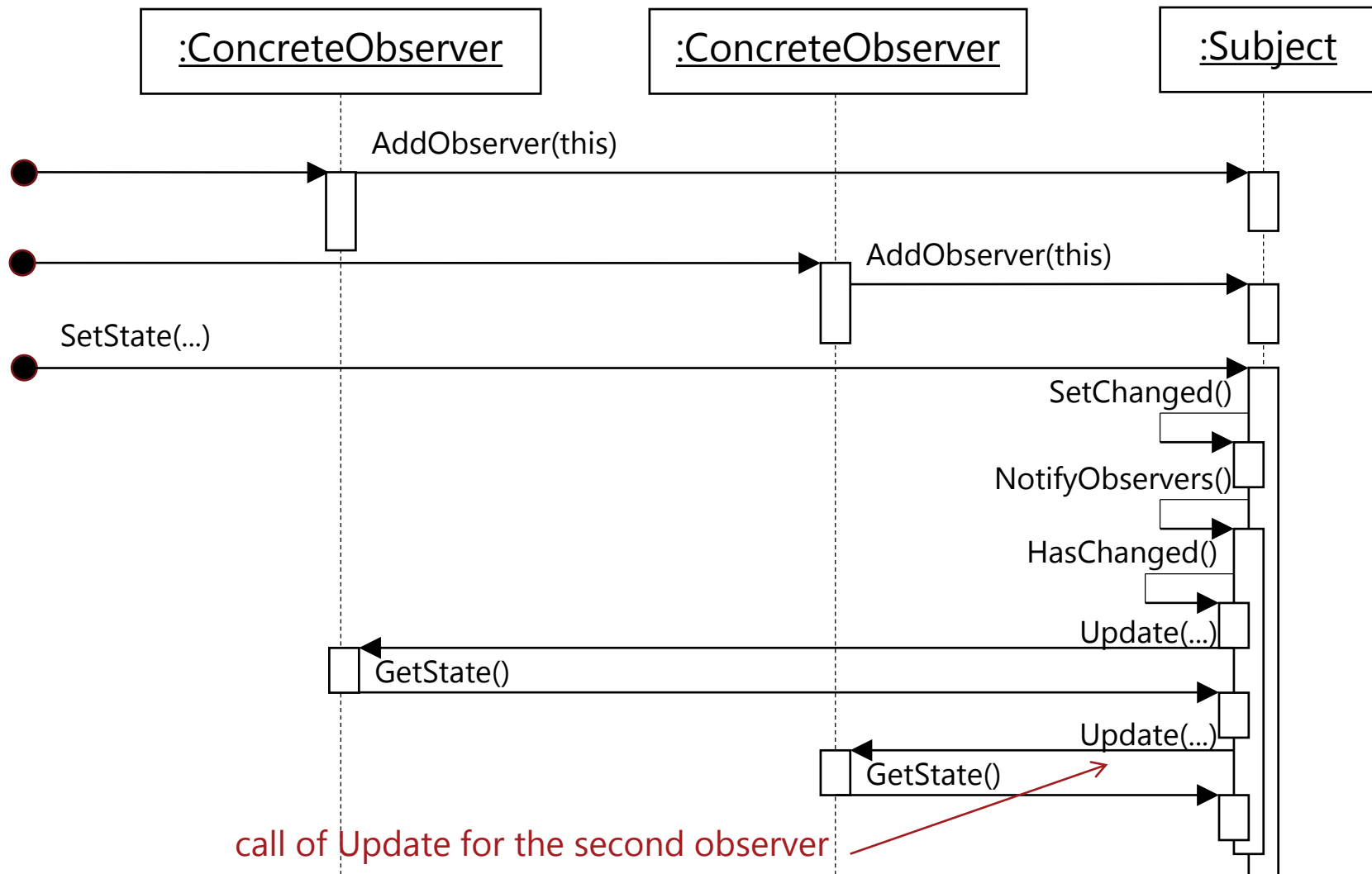
Sequence diagram for describing the flow of the observer pattern



Sequence diagram for describing the flow of the observer pattern

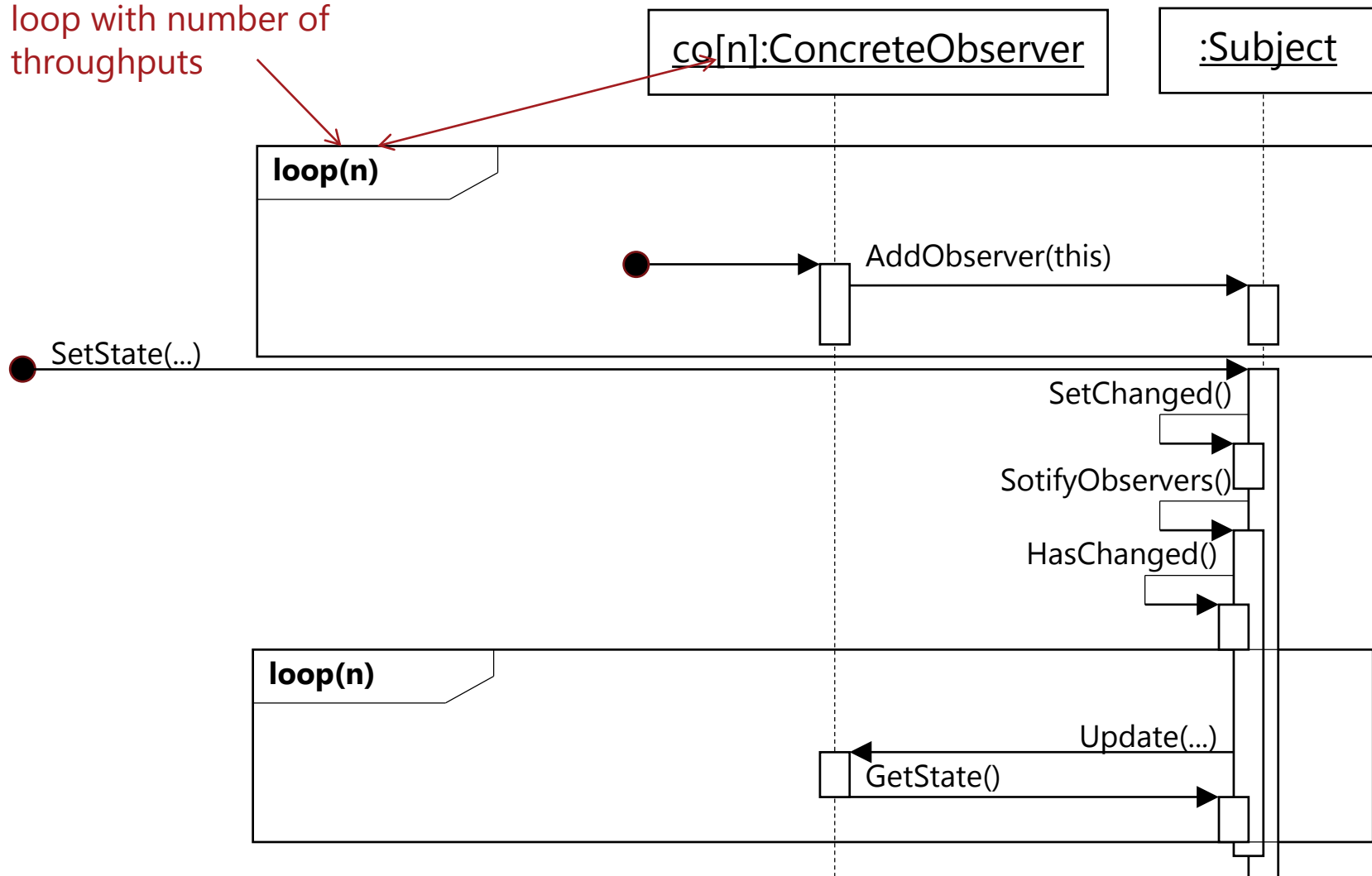


Sequence diagram for describing the flow of the observer pattern



General description of the flow of the observer pattern

loop with number of throughputs



Findings from class and sequence diagrams

- The implementation of the observer pattern requires little effort, since essential sequences can be defined by predefined classes.
- Any number of observers can be supplied with information during execution.
- The pattern is used, in particular, in the design of graphic surfaces:
 - The graphical elements of the interface, which the user can manipulate, are observed: menus, switch buttons, text fields, ...
 - Observers are program sections, which are supposed to react when manipulated by the user.

Events in C#

- Lightweight implementation of observer pattern
- A “protected delegate”
 - Owning class gets full access
 - Consumers can only hook or unhook handlers
 - Similar to a property – supported with metadata
- Used throughout the frameworks
- Very easy to extend

Event Sourcing

- Define the event signature

```
public delegate void EventHandler(  
    object sender, EventArgs e);
```

- Define the event and firing logic

```
public class Button : Control  
{  
    public event EventHandler Click;  
  
    protected void OnClick(EventArgs e) {  
        if (Click != null) {  
            Click(this, e);  
        }  
    }  
}
```


Event Handling

- Define and register event handler

```
public class MyForm : Form
{
    Button okButton;

    public MyForm() {
        okButton = new Button();
        okButton.Text = "OK";
        okButton.Click += new EventHandler(OkButtonClick);
    }

    private void OkButtonClick(object sender, EventArgs e) {
        MessageBox.Show("You pressed the OK button");
    }
}
```

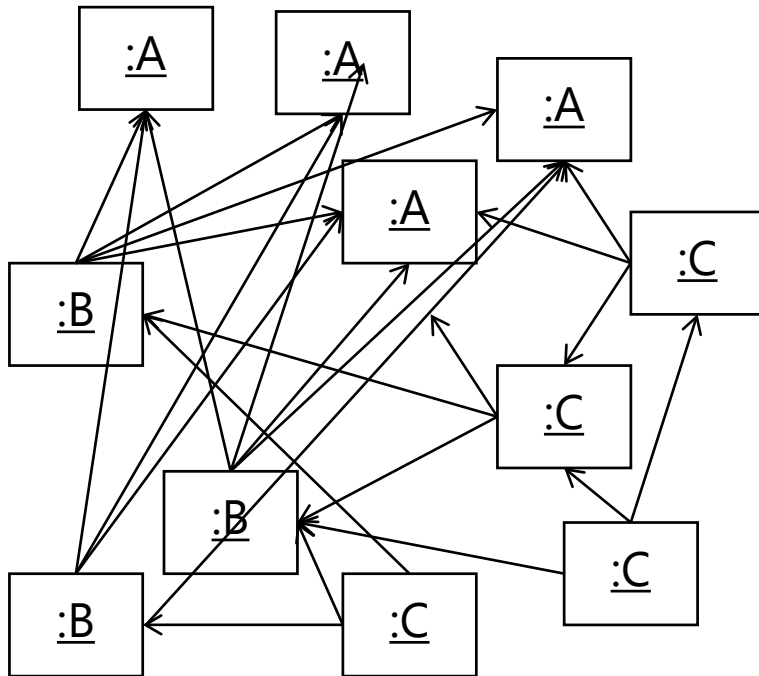
Summary Observer pattern

- Advantages:
 - The observer pattern specifies a protocol on which the information exchange between objects is oriented.
 - The observer pattern decouples the observed subject from its observers. As a result, further observer classes can easily be created. During the execution, the number of observers is dynamic and not limited.
 - The notification mechanism can be implemented independently of the specific problem.
- Disadvantages:
 - If an observer object observes different subjects, the identification of the notifying object can be problematic.

Mediator pattern

- A **mediator** promotes the loose coupling of objects by avoiding an explicit relationship between the involved objects.
- Remarks:
 - The distribution of the behavior to many objects is usually the basis for the good changeability and reusability of object-oriented systems.
 - **But:** Too many relationships between too many objects reduce the changeability and reusability because building such object structures is complex.
- **Idea:**
 - An object operates as an intermediary between the other objects.
 - Communication between all involved objects always takes place via the intermediary.
 - Further objects can be connected very easily.

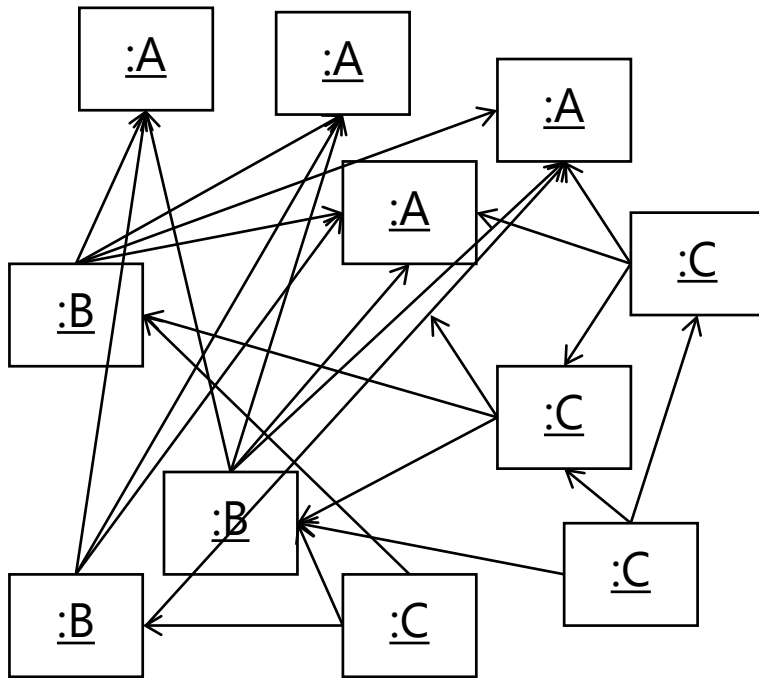
Situation for the use of the mediator pattern



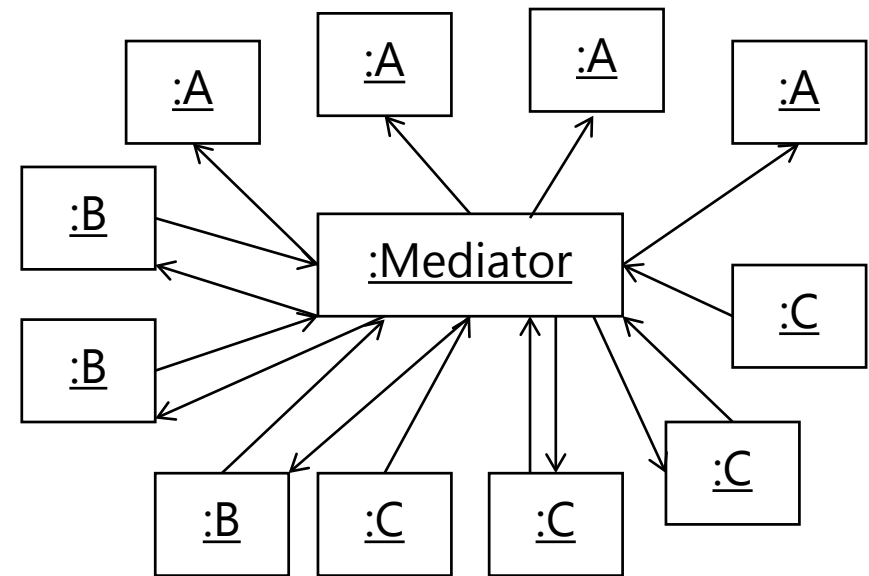
Object diagram

Situation for the use of the mediator pattern

Initial situation



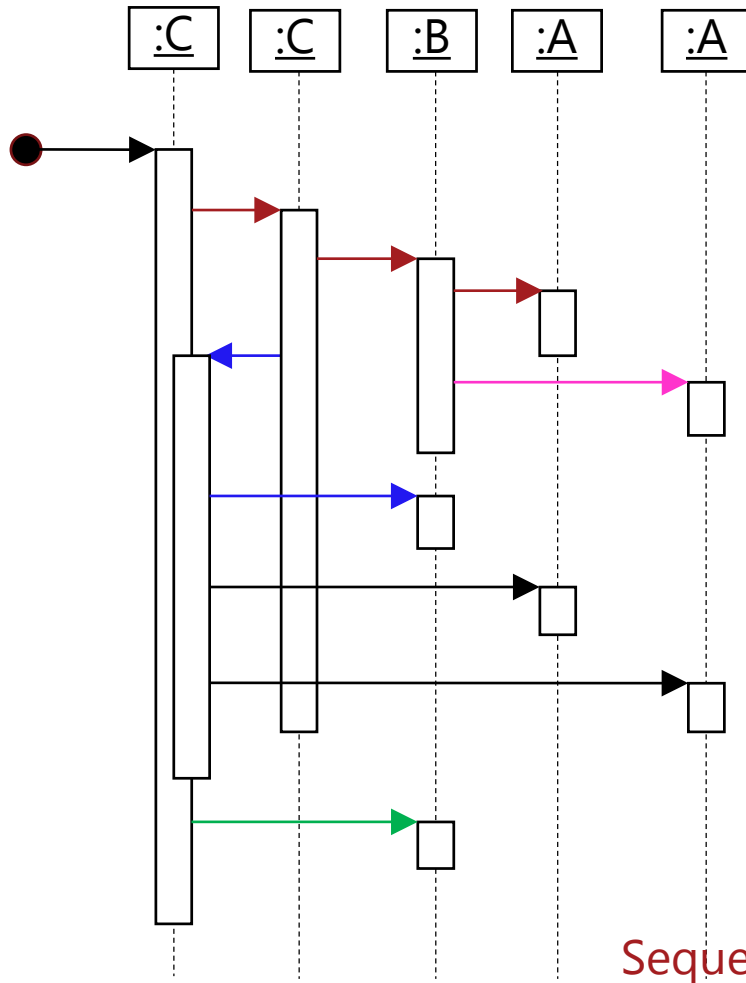
Modified structure by using the mediator pattern



Object diagram

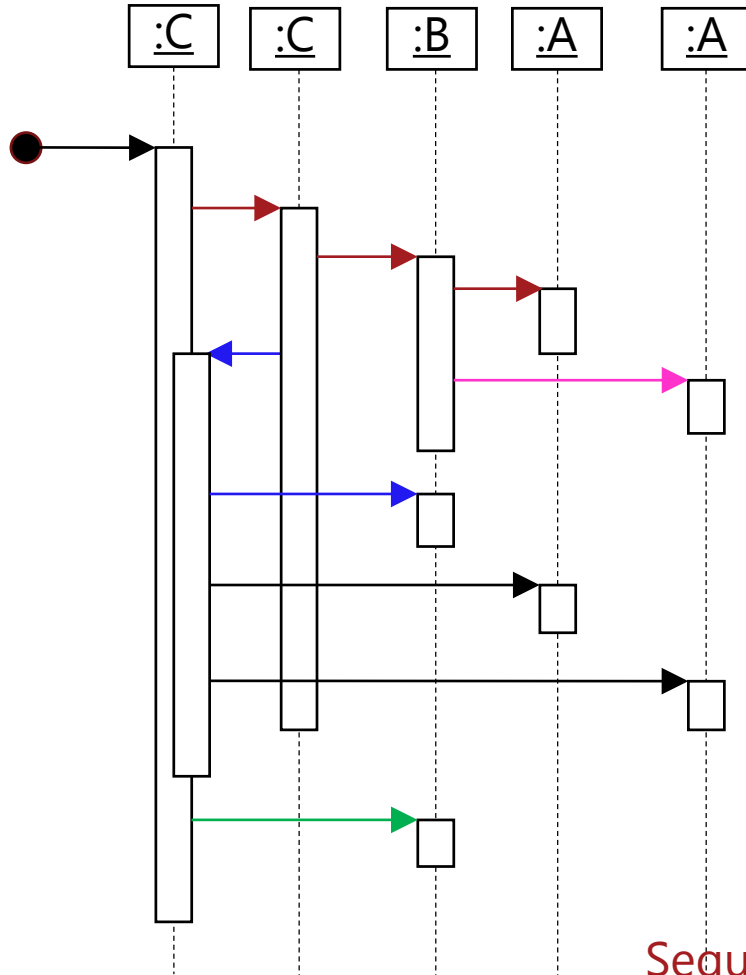
Activities

Initial situation



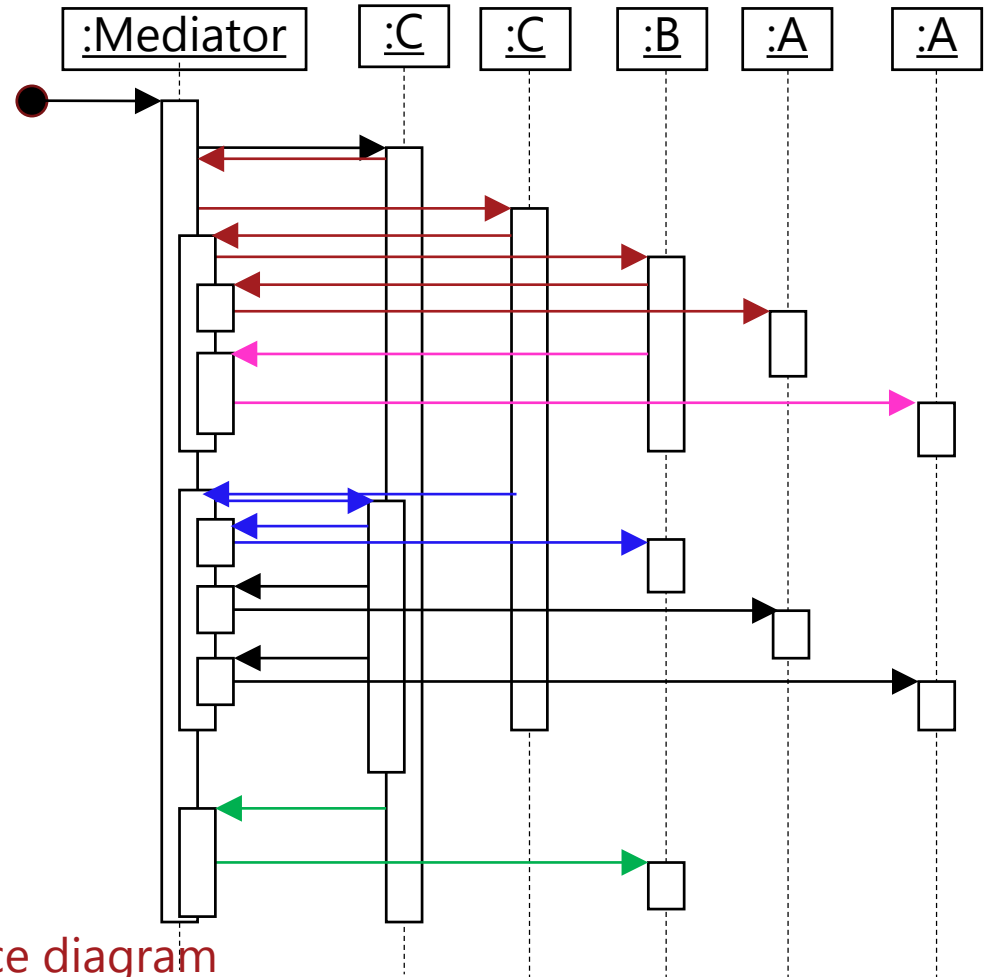
Activities

Initial situation

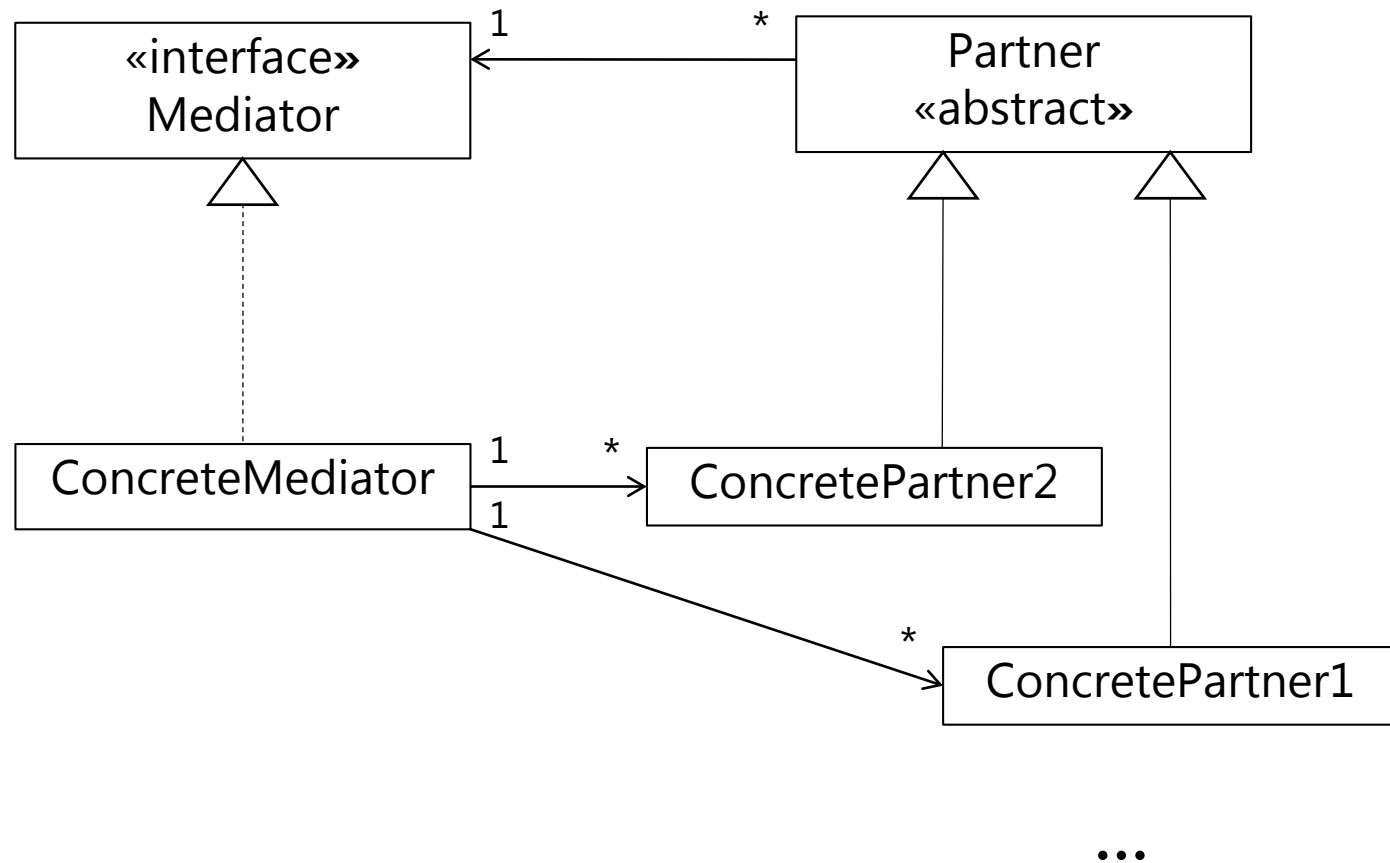


Sequence diagram

Modified structure by using the mediator pattern



General structure of a mediator pattern: class diagram



Summary Mediator pattern

- Advantages:
 - The mediator pattern simplifies the protocol between objects: All partner objects call only the methods of the mediator.
 - The mediator pattern abstracts from the collaboration between the objects: All partner objects know only the mediator.
 - The mediator pattern decouples the objects of the system: Further objects as well as further partner classes can be easily integrated.
- Disadvantages:
 - The mediator fulfills a central task: The complexity of the interaction is replaced by the complexity of the mediator.
- Remarks:
 - Comparison with the facade pattern:
 - A facade provides a more convenient interface to simplify the use.
 - A facade supports a unidirectional protocol between classes.
 - A mediator supports multidirectional protocol to simplify the collaboration of objects.
 - The observer pattern can be used for support: The mediator observes partners to determine their communication needs.

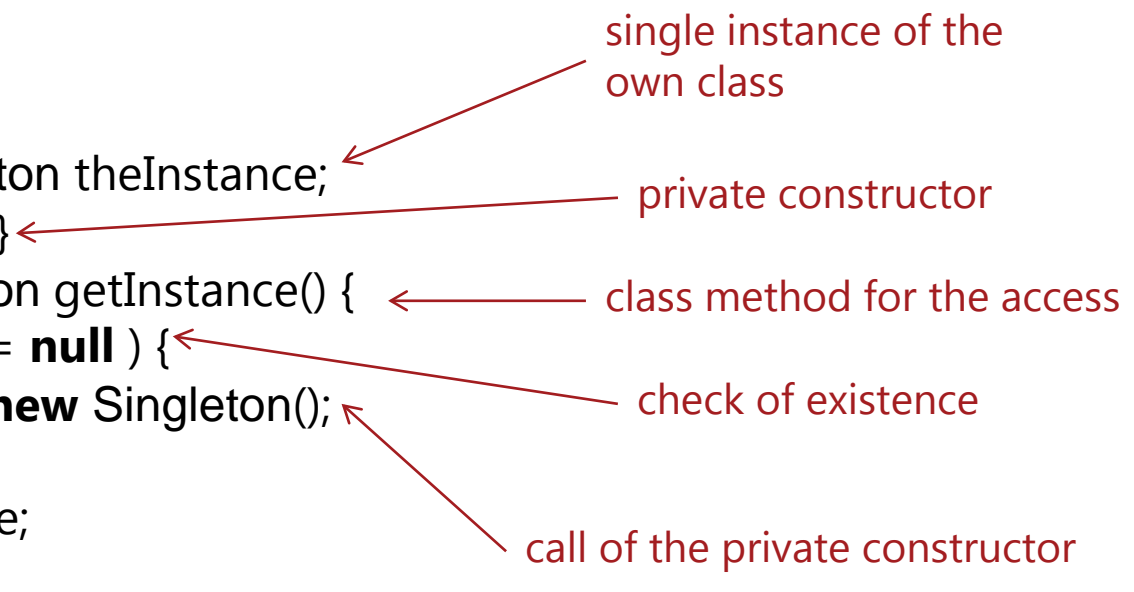
Singleton pattern

- A **Singleton** ensures that there is only one single object from a class.
- Motivation:
 - There are classes, of which only one object can exist at runtime.
 - Example: Assigning unique keys, such as order numbers, customer numbers, ...
- **Idea:**
 - The class itself ensures that there is only one object:
 - The implementation depends on the possibilities of the programming language.
 - Access to constructors must be restricted, (e.g. by preventing access: The constructors are agreed privately!
 - Instead of the constructor, special static methods control the creation of only one object.

Implementation

Standard variant:

```
public class Singleton {  
    private static Singleton theInstance;  
    private Singleton() { }  
    public static Singleton getInstance() {  
        if ( theInstance == null ) {  
            theInstance = new Singleton();  
        }  
        return theInstance;  
    }  
    ...  
}
```



single instance of the own class

private constructor

class method for the access

check of existence

call of the private constructor

Problem:

- Concurrent processes (threads) can overlap during production!

Implementation

Solution with synchronization (exclusion of competing accesses):

```
public class Singleton {  
    private static Singleton theInstance;  
    private Singleton() { }  
    public static Singleton getInstance() {  
        lock(theInstance) {  
            if ( theInstance == null ) {  
                theInstance = new Singleton();  
            }  
            return theInstance; }  
        }  
    ... }  
}
```

theInstance is used for forming a lock



sequentializes method calls from different threads



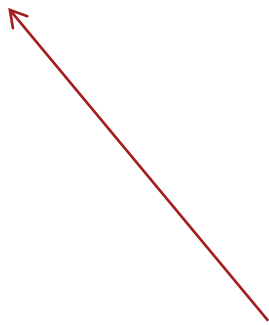
Problems:

- The synchronization slows down the execution!
- However, the synchronization is only required during the first method flow!

Implementation

Solution with prematurely created instance:

```
public class Singleton {  
    private static readonly Singleton theInstance = new Singleton();  
    private Singleton() { }  
    public static Singleton getInstance() {  
        return theInstance;  
    }  
    ...  
}
```



creates an instance when loading the class

Problems:

- This solution is only possible if, as in the example, no values are entered into the generation, which had to be calculated beforehand!

Implementation

Flexible and high-performance solution:

```
public class Singleton {  
    private Singleton() { }  
    private static class SingletonHolder {  
        public static readonly Singleton instance = new Singleton();  
    }  
    public static Singleton getInstance() {  
        return SingletonHolder.instance;  
    }  
    ...  
}
```

inner class creates instance of Singleton



inner class is not created until getInstance is called



Implementation

Another flexible and high-performance solution in Java(!):

```
public enum Singleton {  
    INSTANCE;  
    ...  
}
```

More information:

http://en.wikipedia.org/wiki/Singleton_pattern#The_Enum-way

So there are several different solutions for the quite simple Singleton pattern in one programming language

But: Singleton pattern should only be used if there is a risk of multiple instances being generated - e.g. in concurrent processes.

Abstract factory pattern

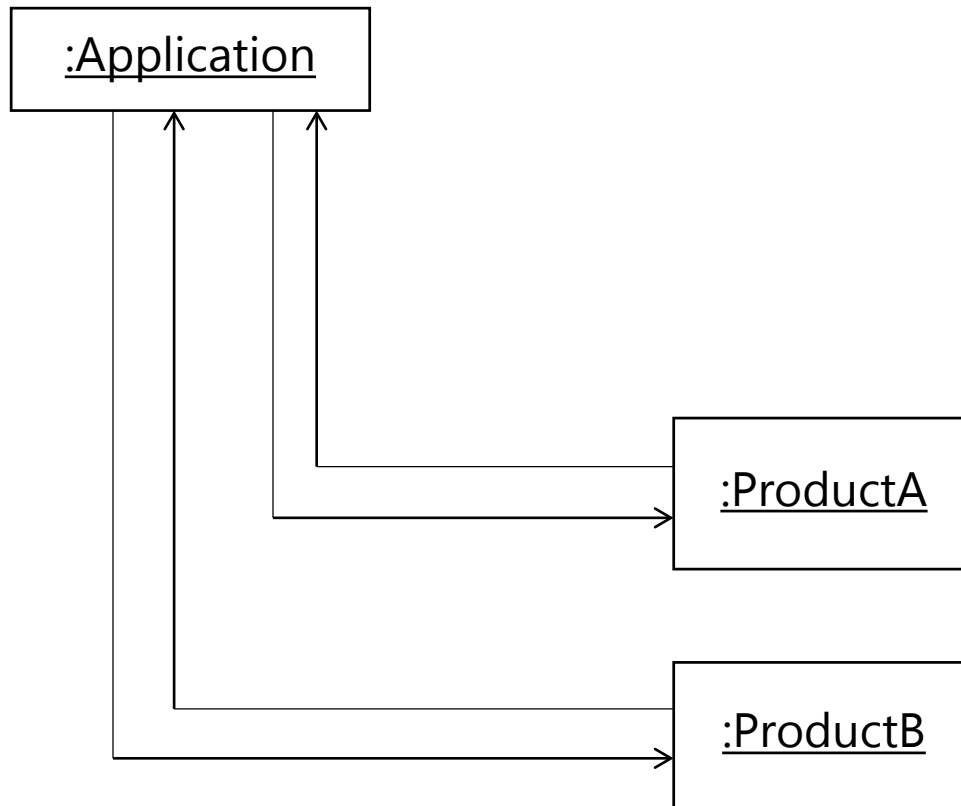
- An **Abstract Factory** allows the use of the same procedures for different families of objects.
- Motivation:
 - A software product can be used with the largely identical processes in different contexts. The same parts should be maintained unchanged.
- **Idea:**
 - The software consists of a constant application core and other components, which occur in different variants.
 - Only components that match, that is, belong to a family of products, are selected for a configuration.
 - The components required for a configuration are generated by a special component, the factory, as needed.

Abstract factory pattern

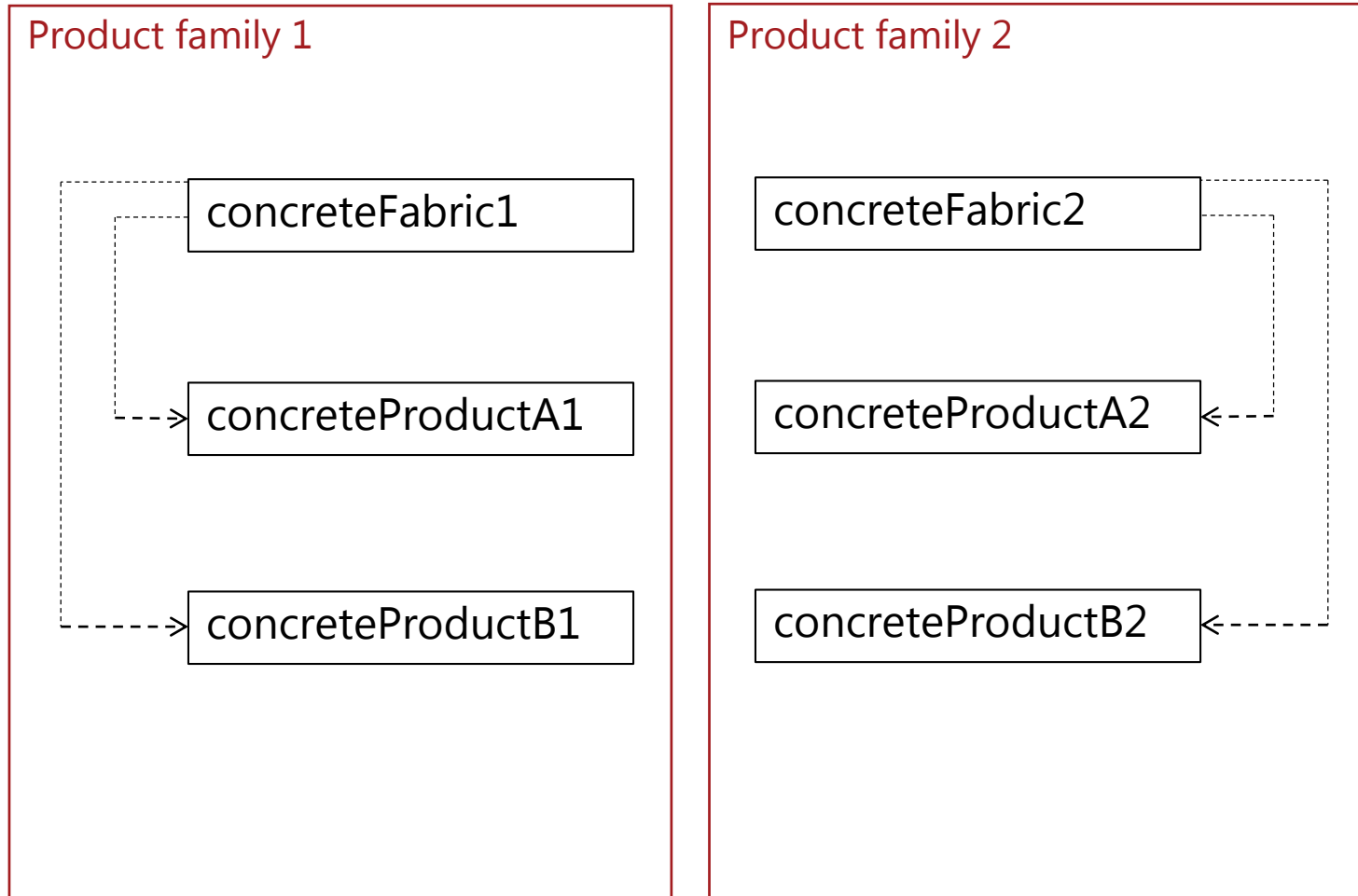
Examples:

- Aim: different user interfaces for the same software product
Solution: several families with classes for graphic presentations; one of these families is selected and the associated objects are created
- Aim: use of a software product in different application areas
Solution: several families with classes for the user interface with different presentation; one of these families is selected and the associated objects are generated
- Aim: use of a software product based on different techniques for data storage
Solution: several families with classes for the storage of values; one of these families is selected and the associated objects are generated

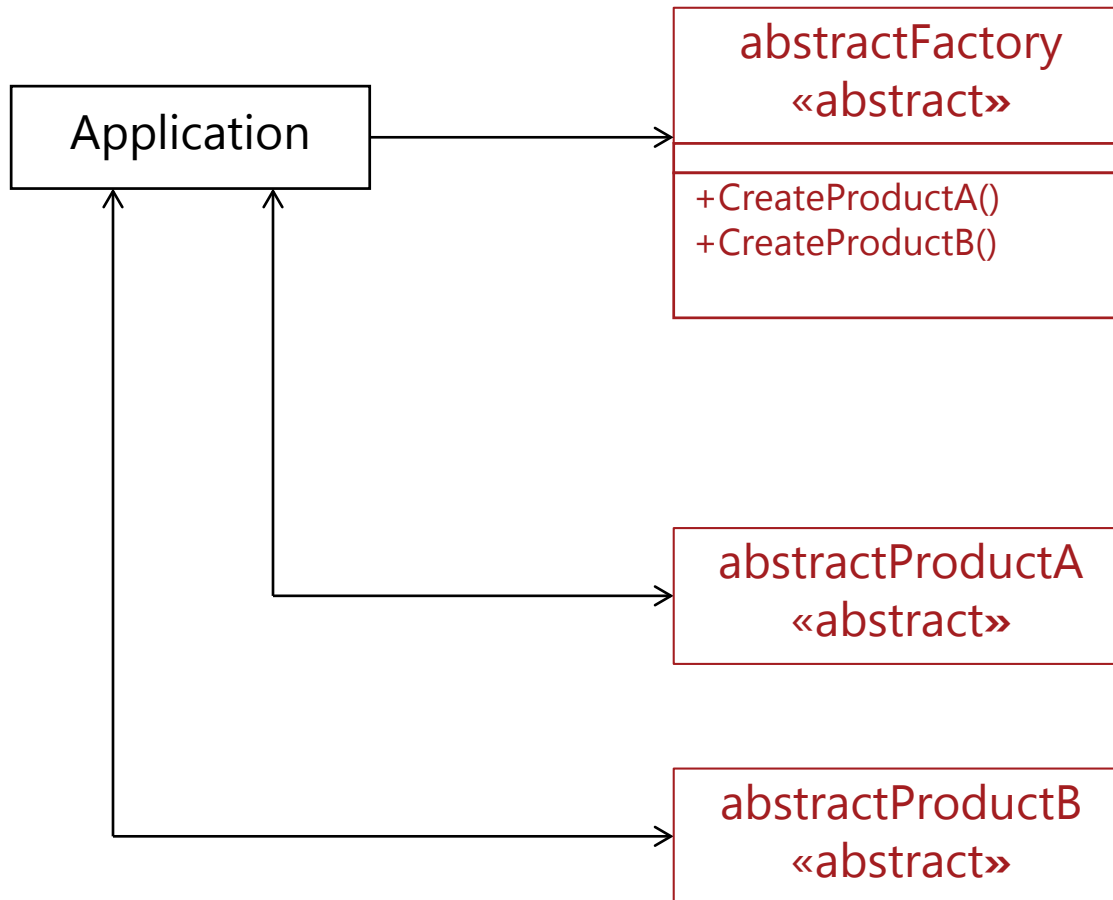
General structure (objects of application)



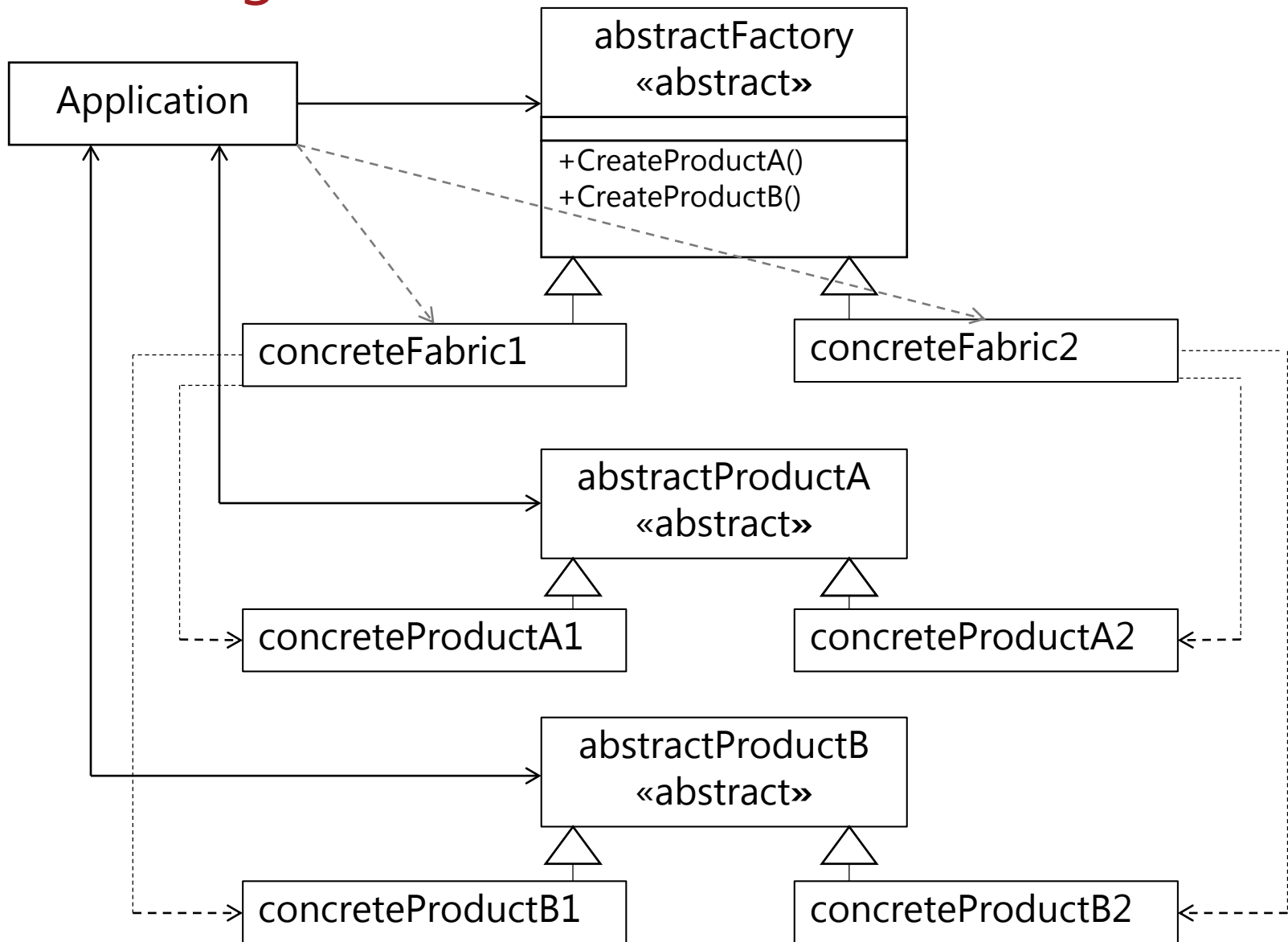
General structure (class diagram product families)



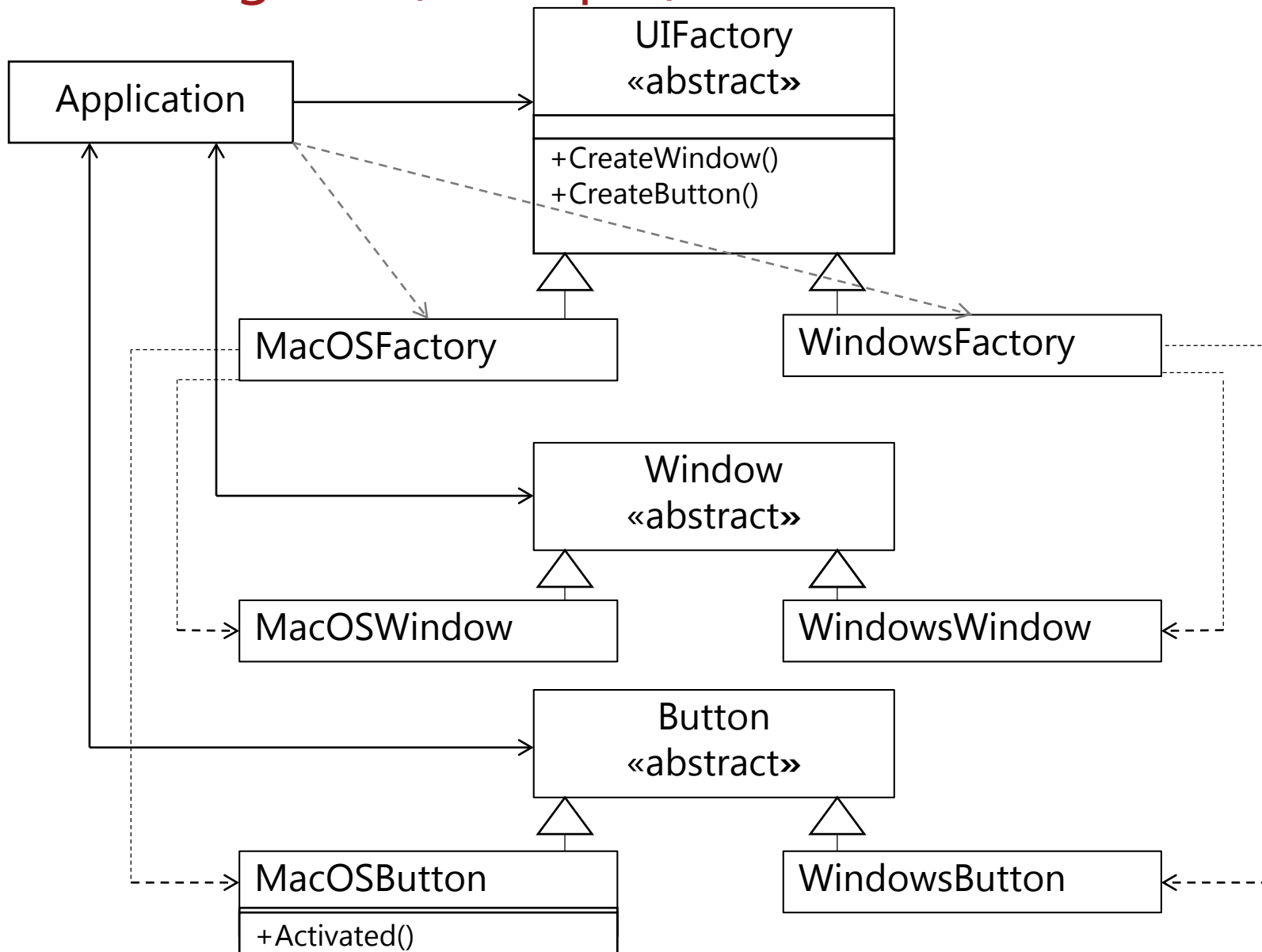
General structure (classes)



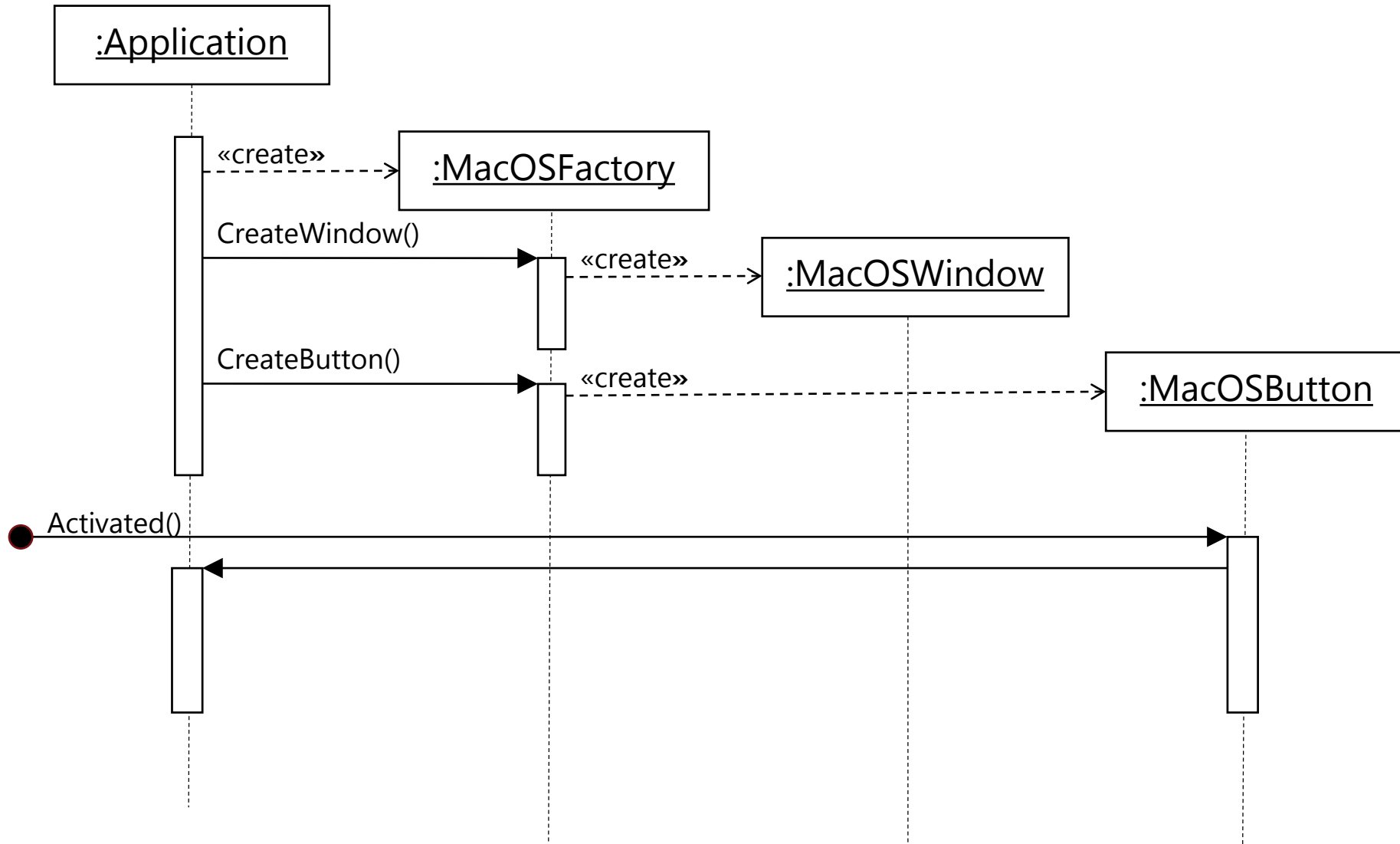
Class diagram



Class diagram (example)



Sequence diagram (example)



Summary – Abstract factory pattern

- Advantages:
 - The Abstract factory pattern simplifies the customization of a software product by exchanging groups (families) of objects.
 - The adjustment is made dynamically at runtime.
 - Further product families can easily be added in the framework given by the interfaces.
- Disadvantages:
 - The pre-recognition of a situation which is sustainably supported by an abstract factory is difficult.
 - The construction of an abstract factory is complex. In particular, a suitable description of the scope of the product family must be carried out as preparation.
 - The creation of an abstract factory is only profitable if actually several products can be identified in different families.



What are the concepts behind Design Patterns?

- Abstraction from concrete classes and objects
- Are there more?

SOLID principles

SOLID Stands For

- **Single responsibility**
 - **Open-closed**
 - **Liskov substitution**
 - **Interface segregation**
 - **Dependency inversion**
-
- The principles, when applied together, intend to make easy to maintain and extend over time system

S – SRP - Single responsibility principle

- Every class, function, variable should define a single responsibility, and that responsibility should be entirely encapsulated by the context
- It is important to keep a class focused on a single concern is that it makes the class more robust

Single Responsibility Principle (SRP)



"There should never be more than one reason for a class to change."

Each responsibility should be a separate class, because each responsibility is an axis of change.

A class should have one, and only one, reason to change.

If a change to the business rules causes a class to change, then a change to the database schema, GUI, report format, or any other segment of the system should not force that class to change.

- **Axis of Change**
- **Separation of Concern**
- **Test-Driven Development (TDD)**
- **Logical Separation of Namespaces And Assemblies.**

Robots on an assembly line are streamlined for the individual tasks they perform. This makes maintaining, upgrading, and replacing them easier and less expensive.

O –OCP - Open/closed principle

- Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification that is, such an entity can allow its behavior to be extended without modifying its source code
- This is especially valuable in a production environment, where changes to source code may necessitate code reviews, unit tests, and other such procedures to qualify it for use in a product

Open/Closed Principle (OCP)



« software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.”

-Bertrand Meyer

- **Meyer's Open/Closed Principle**

- Implementation of a class should only be modified to correct errors.
- Changes or new features require that a different class be created (Interfaces)

- **Polymorphic Open/Closed Principle**

- All member variables should be private.
- Global variables should be avoided.

Change Happens

"All systems change during their life cycles. This must be borne in mind when developing systems expected to last longer than the first version."

- Ivar Jacobson

L – LSP - Liskov substitution principle

- It is a particular definition of a subtyping relation, called behavioral subtyping
- If S is a subtype of T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of that program

Liskov Substitution Principle (LSP)



"Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program"

"Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it."

As you extend objects the original functionality of the elements that makeup the object should not change.

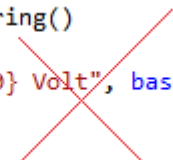
Any Base 10 calculator should produce a result of "4" when you press "2+2=" regardless of the age or sophistication of the device. The original functionality of objects should preserved as you build on them.

- **Polymorphism**
- **Test-Driven Development (TDD)**
- **Avoid Run-Time Type Information (RTTI)**

Objects should be replaceable with subtypes

```
public virtual String Content
{
    get;
    protected set;
}
```

```
public override string ToString()
{
    return String.Format("{0} Volt", base.ToString());
}
```



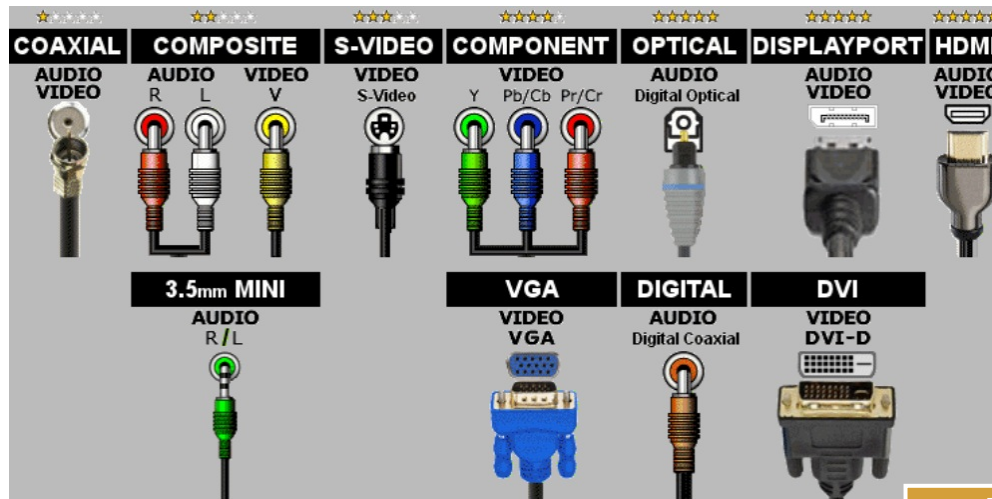
```
public override String Content
{
    get
    {
        return base.Content;
    }
    protected set
    {
        base.Content = value;
        NotifyPropertyChanged("Content");
    }
}
```

When overriding a class you must be careful that the way you extend it does not cause it to no longer work in place of the class it inherited from

I – ISP - Interface segregation principle

- It states that no client should be forced to depend on methods it does not use
- ISP splits interfaces which are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them

Interface Segregation Principle (ISP)



“Many client specific interfaces are better than one general purpose interface”

“Clients should not be forced to depend upon interfaces that they do not use.”

It is difficult to use a device that produces “HDMI Audio/Video” with a “Digital Optical” sound player.

- Design by contract (DbC) / Design to Interfaces
- Test-Driven Development (TDD)

Many specific interfaces are better than one general

```
public interface IAudioOptical
{
    Stream AudioOptical { get; }
}
```

```
/// <summary>
/// RGBHV Video HDTV VGA
/// </summary>
public interface IVideoRGBHV
{
    /// <summary>
    /// Red
    /// </summary>
    [ConnectorColorAttribute(ConnectorColorType.Red)]
    IRCA Red { get; }

    /// <summary>
    /// Green
    /// </summary>
    [ConnectorColorAttribute(ConnectorColorType.Green)]
    IRCA Green { get; }

    /// <summary>
    /// Blue
    /// </summary>
    [ConnectorColorAttribute(ConnectorColorType.Blue)]
    IRCA Blue { get; }

    /// <summary>
    /// Horizontal Sync
    /// </summary>
    [ConnectorColorAttribute(ConnectorColorType.Yellow)]
    IRCA HorizontalSync { get; }

    /// <summary>
    /// Vertical Sync
    /// </summary>
    [ConnectorColorAttribute(ConnectorColorType.White)]
    IRCA VerticalSync { get; }
}
```

```
public interface IAudioVideoHDMI
{
    /// <summary>
    /// Pin 1 TMDS Data2+
    /// Pin 2 TMDS Data2 Shield
    /// Pin 3 TMDS Data2-
    /// </summary>
    Stream Data2 { get; }

    /// <summary>
    /// Pin 4 TMDS Data1+
    /// Pin 5 TMDS Data1 Shield
    /// Pin 6 TMDS Data1-
    /// </summary>
    Stream Data1 { get; }

    /// <summary>
    /// Pin 7 TMDS Data0+
    /// Pin 8 TMDS Data0 Shield
    /// Pin 9 TMDS Data0-
    /// </summary>
    Stream Data0 { get; }

    /// <summary>
    /// Pin 10 TMDS Clock+
    /// Pin 11 TMDS Clock Shield
    /// Pin 12 TMDS Clock-
    /// </summary>
    Stream Clock { get; }

    /// <summary>
    /// Pin 13 CEC
    /// </summary>
    Stream CEC { get; }

    /// <summary>
    /// Pin 14 Reserved (HDMI 1.0-1.3c), HEC Data- (Optional, HDMI 1.4+ with Ethernet)
    /// </summary>
    Stream HEC { get; }

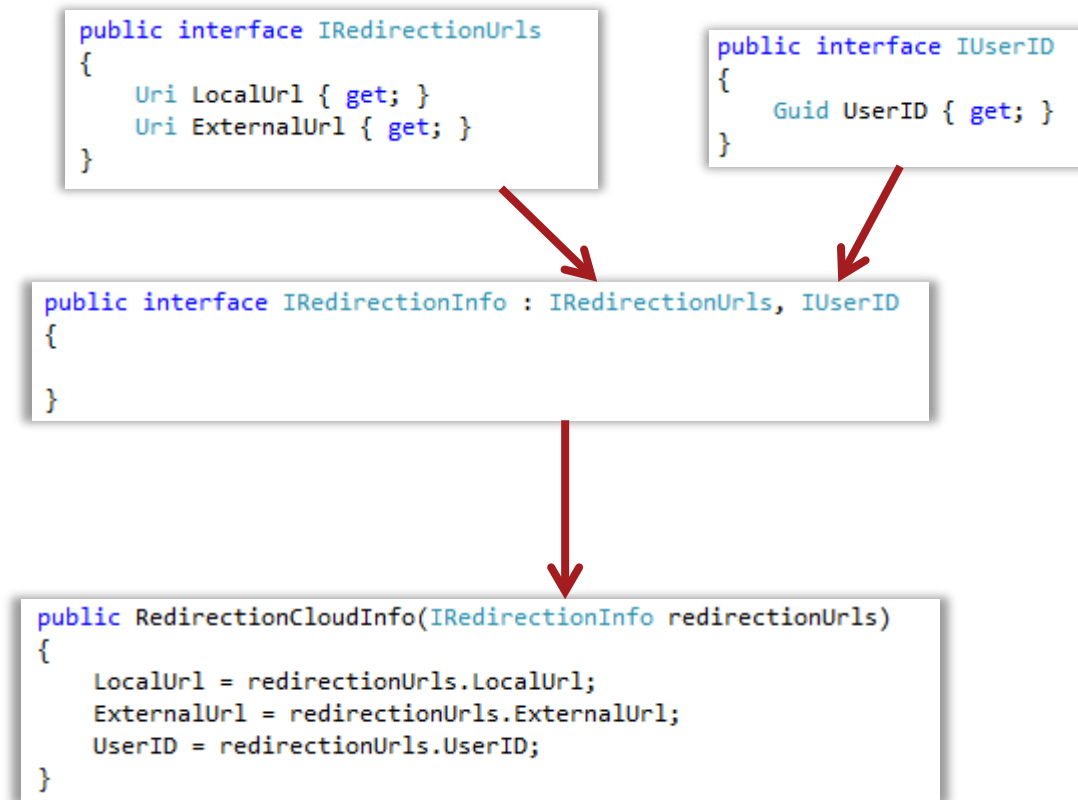
    /// <summary>
    /// Pin 15 SCL (I2C Serial Clock for DDC)
    /// </summary>
    Stream SerialClock { get; }

    /// <summary>
    /// Pin 16 SDA (I2C Serial Data Line for DDC)
    /// </summary>
    Stream SerialData { get; }

    /// <summary>
    /// Pin 17 DDC/CEC/HEC Ground
    /// </summary>
    Stream Ground { get; }

    /// <summary>
    /// Pin 19 Hot Plug Detect (All versions) and HEC Data+ (Optional, HDMI 1.4+ with Ethernet)
    /// </summary>
    Stream HotPlug { get; }
}
```

Clients should not depend upon interfaces that they do not use



Each interface should specifically describe only what is needed and nothing more.

D – DIP - Dependency inversion principle

- High-level modules should not depend on low-level modules: both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

Dependency Inversion Principle (DIP)



"High level modules should not depend upon low level modules. both should depend upon abstractions."

"Abstractions should not depend upon details. Details should depend upon abstractions."

Screwdriver bits do not care what brand or type of Slotted Screwdriver they are used with.

- **Loose Coupling**
- **Dependency Injection / Inversion of Control**

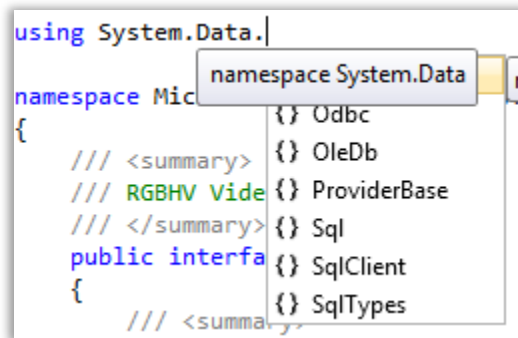
Dependency Injection / Inversion of Control

```
public class AutomobileController
{
    public EventHandler OnCarReachsLocation;
    public Automobile Car
    {
        get;
        private set;
    }
    public AutomobileController(Automobile car)
    {
        Car = car;
    }
    public void GoToLocation(Double latitude, Double longitude)
    {
        Car.StartEngine();
        Car.StartMoving();
        //MovingLogic(latitude, longitude);
        Car.StopMoving();
        Car.StopEngine();
        if (OnCarReachsLocation != null)
            OnCarReachsLocation.Invoke(this, new EventArgs());
    }
}
```

```
public class VehicleController
{
    public EventHandler OnVehicleReachsLocation;
    public IVehicle Vehicle
    {
        get;
        private set;
    }
    public VehicleController(IVehicle vehicle)
    {
        Vehicle = vehicle;
    }
    public void GoToLocation(Double latitude, Double longitude)
    {
        Vehicle.StartEngine();
        Vehicle.StartMoving();
        //MovingLogic(latitude, longitude);
        Vehicle.StopMoving();
        Vehicle.StopEngine();
        if (OnVehicleReachsLocation != null)
            OnVehicleReachsLocation.Invoke(this, new EventArgs());
    }
}
```

```
public interface IVehicle
{
    void StartEngine();
    void StartMoving();
    void StopEngine();
    void StopMoving();
    string ToString();
}
```

High & low level modules should depend upon abstractions



No reference or dependency to a specific Data access technology should ever exist outside of the Data Access Assembly this includes ADO.net, LinqToSql, EF4, Nhibernate...

Only Interfaces and POCO (Plain Old CLR Objects) should be referenced between assemblies and primarily they should reference an interface.

SOLID Bad Smells

- If-Statement, which distinguishes types -> **OCP**
- Interface methods, which do not have to be implemented (in most cases) -> **ISP**
- Modelling of IS-A relations, which do not fit -> **LSP**
- Interface Pollution -> **ISP**
- FAT Interfaces -> **ISP**
- Method names like: UpdateAndSaveCustomer(), VerifyAndSaveData() -> **SRP**
- Class names like: CustomerManager -> **SRP**

A SOLID Summary

- **SRP:** There should never be more than one reason for a class to change.
- **OCP:** Software entities should be open for extension but closed for modification
- **LSP:** Functions that use references to base classes must be able to use objects of derived classes without knowing it.
- **ISP:** Clients should not be forced to depend upon interfaces that they do not use .
- **DIP:**
 - High level modules should not depend upon low level modules. Both should depend upon abstractions.
 - Abstractions should not depend upon details. Details should depend upon abstractions

Summary – Design patterns

	Structural patterns	Behavioral patterns	Creational patterns
Class-related patterns	Class Adapter pattern		
Object-related patterns	Object Adapter pattern Decorator pattern Composite pattern Facade pattern	Strategy pattern Mediator pattern Observer pattern Iterator pattern	Abstract factory pattern Singleton pattern

- All design patterns have been derived from experience.
 - Design patterns provide appropriate solutions for recurring problems.
 - Some design patterns are supported in standard libraries.
 - Design patterns can be flexibly implemented in various ways.
 - Design patterns provide a common vocabulary for developers.
 - Design patterns can be combined with each other
- Example: **Observer iterates** over the observed objects.

Summary – Design patterns

Critical comments:

- Design patterns are *ideas* for solutions but no finished solutions.
- Design patterns must be adapted to the specific problem.
- The use of design patterns requires experience in the design of object-oriented software.
- Design patterns usually comprise only a few classes, many design patterns are obvious object-oriented solutions.
- Design patterns are difficult to recognize in source code.
- It is even more difficult to recognize combinations of design patterns in the source text.
- A senseless use of design patterns does not make software any better.



Next lecture

7.5.2018

**Implementing
Design Patterns**

