Design Patterns

Software techniques

BME AUT
Department of
Automation and
Applied Informatics

Content

- Design Patterns
  - Definition
  - Overview
  - Classification
  - What is the advantage of using design patterns?
  - Detailing the most important patterns

- You should be familiar with some of them from Java
  - We'll now take an application-centric view

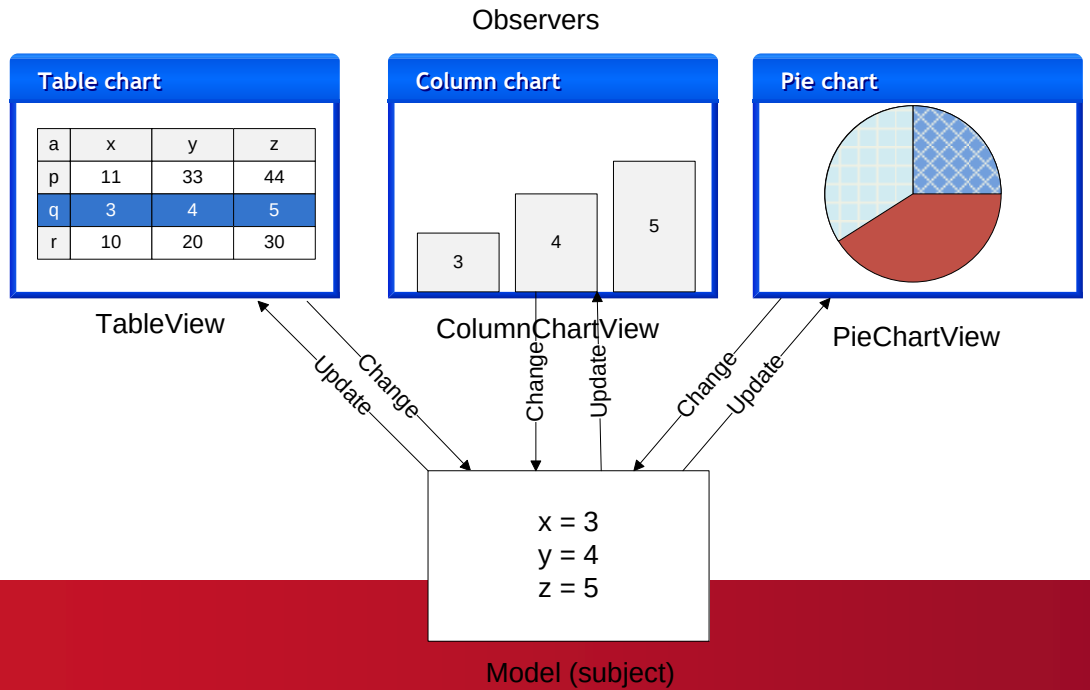
Definition

- Design patterns describe and detail the solution of frequent problems. They can be used to efficiently solve frequent designing problems.

- Example: MVC architecture

Example- Model-View-Controller (MVC) architecture

- Creating user interfaces with Smalltalk-80
- There are three roles/participants
  - Model – application object
  - View – presentation
  - Controller – handling user interactions
- The relationship between the Model and the Views



Describing design patterns

- There are four basic elements to describe a design pattern
  - Pattern name
    - The name of the pattern. Identifies the pattern.
    - ***It is important because of the communication***: developers just have to name the pattern and the role of the object in the pattern to identify its responsibility.
  - Problem
    - The ***problem*** and its ***context***, a usually a concrete example helps understanding
  - Solution
    - Describes the elements (classes) of the solution, the roles, responsibilities and the co-operation of the elements. This is **not** a concrete implementation but an abstract solution.
  - Consequences
    - The consequences of using the pattern. Lists some practical experiences and helps to choose the correct pattern.

Classifying the patterns based on their scope

- The level of the software system where the pattern is used (architecture -> subsystem-> … -> coding in a given programming language)
- Three groups

Classifying the patterns based on their scope

- I. Architectural patterns – discussed on the previous lecture
  - **Introduces the basic structure for the whole system**. The basic structure of the software system. Defines the subsystems, their responsibilities and the connections between them.
  - E.g.:
    - Layers
    - Model-View-Controller architecture
      - Document-based windowing systems
    - Document-view architecture
      - Document-based windowing systems
    - Pipes
      - Processing streams
    - …

Classifying patterns based on their scope

- II. Design patterns
  - This is what we usually mean by design patterns (many times architectural patterns and idioms are not explicitly called design patterns).
  - They are used inside a subsystem or component
    - Scope-wise they are situated in the middle
    - They have no effect on the basic structure (architecture) of the application
    - They are programming language independent
  - Source: E. Gamma R. Helm R. Johnson J. Vlissides: Design Patterns

Classifying the patterns based on their scope

- III. Idioms
  - *Low level design patterns. They are usually tied to a programming language.*
  - Solving problems that frequently occur in a given programming language.
  - E.g. in C++: Smart Pointer
    - Environment
      - An object is used from multiple classes
    - Problem
      - A reference to the shared object must be present in all classes that use it. Question: whose responsibility is it to free up the memory when the object is not used any more (i.e. how to avoid memory leaks)?
    - Solution
      - Define a smart pointer class. It can be used just like an ordinary pointer but it also provides some additional features mainly reference counting. The concrete object can just be accessed through the smart pointer. The smart pointer counts the active references and it will free up the memory when the counter is decreased to zero.
    - Implementation
      - In C++, using templates. Better yet, use the implementation from STL ☺

Classifying the patterns based on their scope

- Idioms– smart pointer example(*don't memorize*)

a, without the delete we will have a memory leak

b, the auto_ptr smart pointer class is responsible for the reference counting and in case of need it will call the delete

```
int main(int argc, char* argv[])
{
    MyClass* p = new MyClass;
    p->DoSomething();
    delete p;
}
```

```
// FRAGMENT of smart pointer template class
// (reference counting is missing)
template <class T>
class auto_ptr {
    T* ptr; // pointer to the object
  public:
    explicit auto_ptr(T* p = 0) : ptr(p) {}
    ~auto_ptr() { delete ptr; }
    T& operator*() { return *ptr; }
    T* operator->() { return ptr; }
    // ...
};

int main(int argc, char* argv[])
{
    auto_ptr<MyClass> p(new MyClass);
    p->DoSomething();
}
```

Classifying the patterns based on their scope

- From now on we will just deal with the middle layer (design patterns)
- We will not be string with our categorization. Many design pattern can be used as an architectural patterns or as idioms.

Categories

- We can classify design patterns based on different criteria
- Criteria
  - Purpose
    - Creational – concerns the creation of the objects
    - Structural – concerns the composition of the objects
    - Behavioral – concerns the cooperation of the objects
  - Scope
    - Class – emphasizes classes and inheritance
    - Object – emphasizes objects and associations between objects

Design pattern categories (don't memorize) based on the aforementioned book

		Creational	Structural	Behavioral
	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Describing design patterns

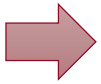
- **Design patterns do not have a specific description language**
  - OMT or UML is used
    - Class diagram
    - Sequence diagram
  - Textual description

Pattern catalog

- The format of pattern description in the catalog (don't need to know)
  - **Name**: the name of the pattern
  - **Intent**: the purpose of the pattern, what it is used/good for
  - **Also known as:** alternative names
  - **Motivation**: practical example, it shows how the pattern solves the concrete problem
  - **Applicability**: context where the pattern can be used
  - **Structure:** the class and object diagram for the pattern
  - **Participants:** short description of the classes in the pattern
  - **Collaboration**: describes the co-operation of the classes in the pattern
  - **Consequences**: evaluation, how efficient does the pattern solve the problem
  - **Implementation**: details about the implementation problems of the pattern, covers language-dependent problems, short code examples
  - **Sample code:** typical code example
  - **Known uses**: references for existing systems that use the pattern.
  - **Related patterns**: closely related patterns.

How do patterns help to solve problems?

- They help in the designing phase
- They help in the following problems:
  - **I. Finding and defining the needed object**
    - Defining the number and the size of the object
    - Defining the interfaces of the objects
    - Implementing the objects
  - **II. Design for reuse**
  - **III. Design for change**

Details 

How do patterns help to solve problems?

- It is difficult to find the necessary objects
  - When designing a system based on a natural language specification collecting the nouns and verbs may help to find the necessary objects and classes and their hierarchy.
  - But systems usually have got classes that do not exist in the real world but play important roles in the system:
    - Certain low level classes (e.g. an array class)
    - Control/manager classes
    - …
  - Design patterns help to find the non-trivial classes.

Design for reuse

- Reusability is not trivial. We have to pay attention to it during system design!
  - an efficient way to reduce the costs of the development
  - on the long term it may be worth developing a framework

- Framework definition
  - In a certain domain (e.g. windowing systems, document handling systems, mathematical systems, graphical systems, CAD systems, etc.):
    - Defines the architecture of the application, and provides some basic building-blocks (classes).
    - In OO environment: pre-defined co-operative classes with pre-defined responsibilities. How to use the built in classes?
      - We have to derive from them
      - We have to override the virtual/abstract methods of the base class

      (E.g. In Windows Forms the Form is the base class)
    - Goal: as little coding as possible when developing using the framework.
    - Gives a consistent structure to our applications.
    - At the architectural level there is no need to design, as the architecture is determined by the framework (design for reuse)

Design for change

- Design for change is not trivial. We have to pay attention to it during system design!
- Why?
  - It is difficult to design a system
  - At the beginning we quite often don't know the exact task
  - Software products are usually iteratively improved after the release
- When we design the system we have to pay attention to the fact that the system will be modified later (design for change)!
  - We can preserve the existing code even when we have to modify the application.
  - When we have to change some part of the application,
    - The modifications should be local
    - As little change should be made on the existing code as possible (e.g. just deriving from a base class is necessary)
    - Development is faster, there won't be so much new bugs!
  - Don't use it everywhere, as designing for change may also take a lot of time (costs a lot). Use it where changes are expectable.

Principle

- Goal: reduce the dependency between the subsystems or components
  - A change should just effect one/a few number of component(s)
    - Changeability
    - Reusability

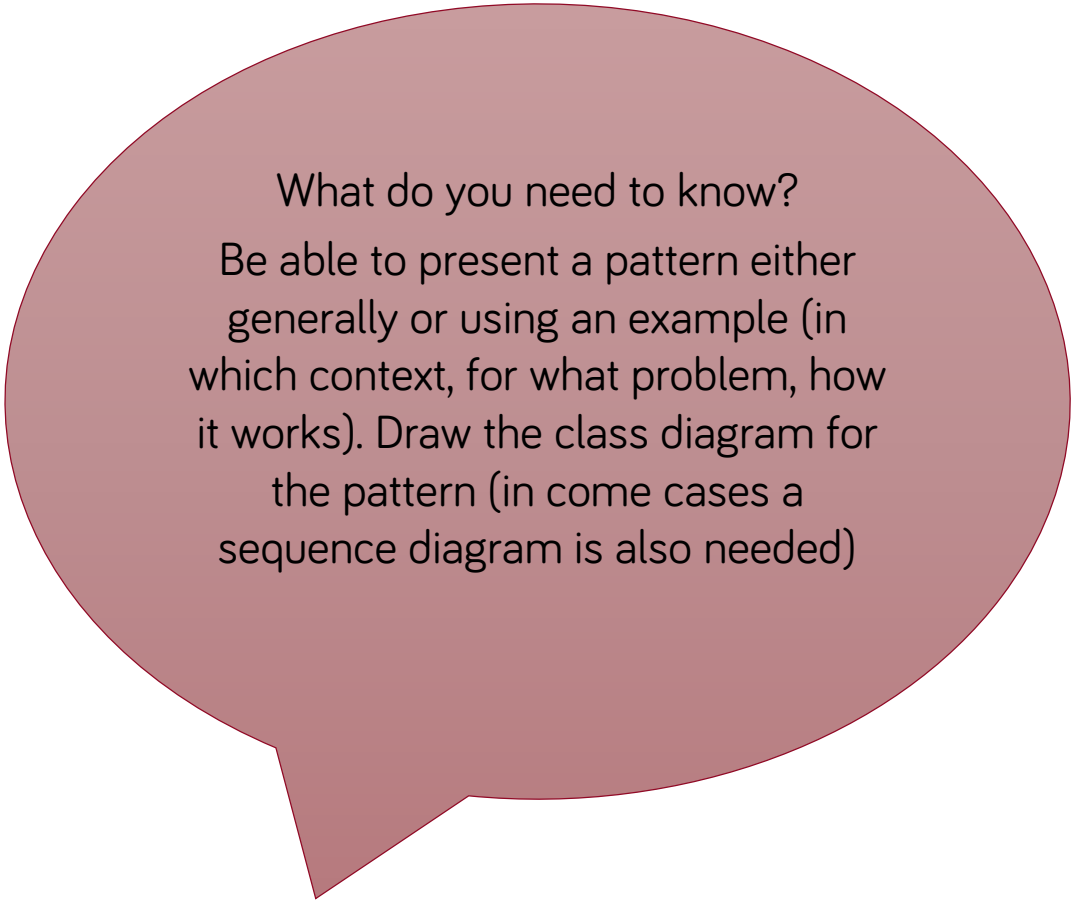
System of Patterns

- What does „system of patterns” mean?

  During the system design we use multiple design patterns. We usually use one architectural pattern. This will provide the basic structure. During the detailed design we may use multiple design patterns that are combined with each other.

Some important and frequent patterns

- Creational
  - Factory method
  - Abstract factory
  - Singleton
  - Prototype
- Structural
  - Adapter
  - Bridge
  - Proxy
  - Composite
  - Facade
- Behavioral
  - Template method
  - Iterator
  - Observer
  - Mediator
  - Strategy
  - Command



Creational patterns

Factory method

Abstract factory

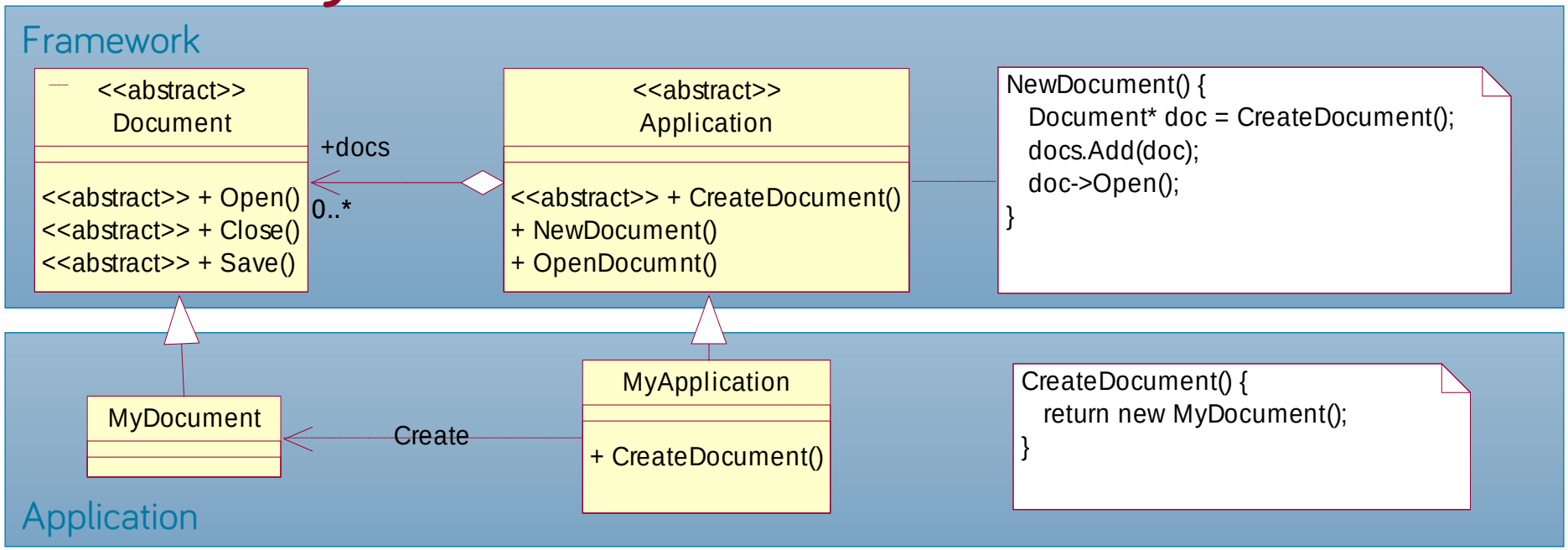
Singleton

Factory method

Factory Method

- Purpose
  - It defines an interface to create objects, but the particular instantiation is delegated to the derived class (the derived class will determine the exact type of the instantiated class). **The Factory Method facilitates the delegation of the instantiation to the derived class.**
- It is often called „virtual constructor”
- Example
  - Framework that supports multiple documents (e.g. Visual Studio)
  - Two key classes of the framework
    - *Application*
    - *Document*
  - Both of them are abstract classes, the programmer has to derive from them to implement a custom classes. (let’s call them *MyApplication* and *MyDocument*)
  - The *Application* class of the framework doesn’t exactly know what kind of *Document* class should be created (as the *MyDocument* class did not even exist when the framework was created), it only knows when it should be created.

Factory Method

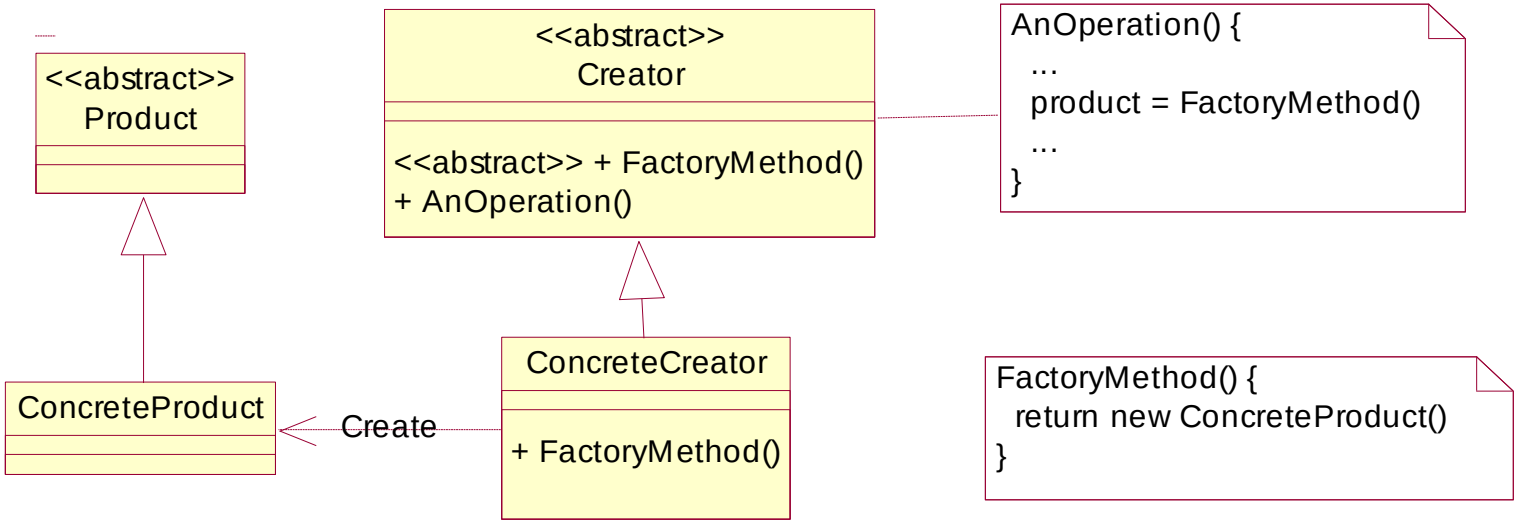


- When the user selects the „New” menu item, the framework calls the Application::NewDocument() method.
- In this method „*new MyDocument()*” couldn't have been written as the MyDocument class did not event exist at that time.
- Solution: to instantiate the document the *Application::NewDocument()* calls the *Application::CreateDocument()* abstract function, that should be overridden in the derived class by the developer, to return a *MyDocument* –typed document instance.

Factory Method

- Solution
  - Delegate the instantiation to the derived class (creating the document instance)

- Structure



Factory Method

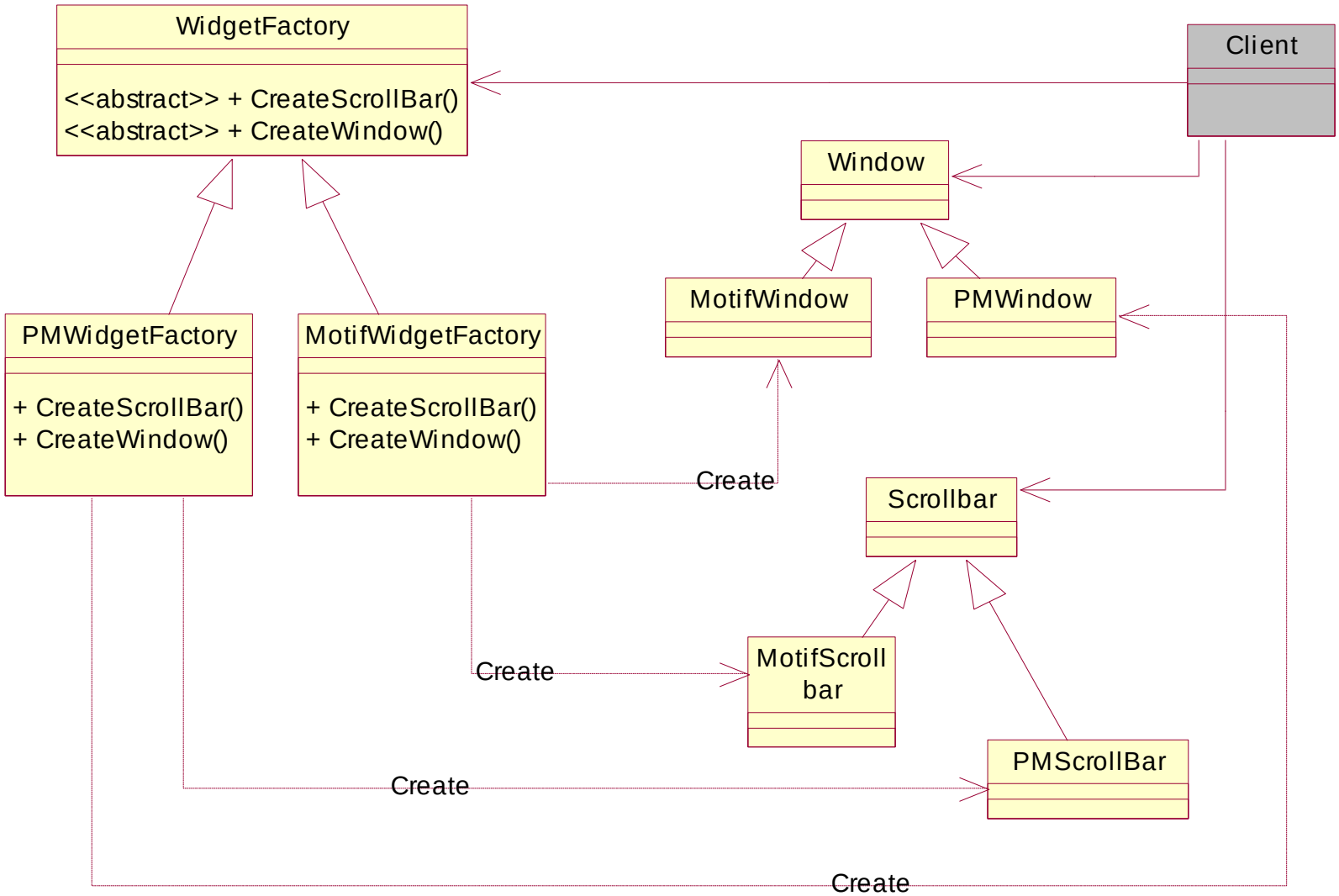
- Use it when
  - A class does not know in advance the exact type that it should instantiate
  - A class wants its derived classes to determine the exact types that are instantiated.

Abstract Factory

Abstract factory - Example

- Example
  - Windowing system, GUI controls (form, button, drop-down list, etc.)
  - The application should support user interfaces with multiple different “look-and-feel”s
    - E.g. Motif, Presentation Manager
  - How to do it?
  - Create a different version for each GUI element and each “look-and-feel”
  - Don’t hardwire the applied user interface elements into the application (neither their creation nor their usage).
    - Encapsulate the instantiation, delegate it to an other class
    - In the application we should only refer to the certain classes through interfaces (the implementation will be replaceable)

Abstract factory

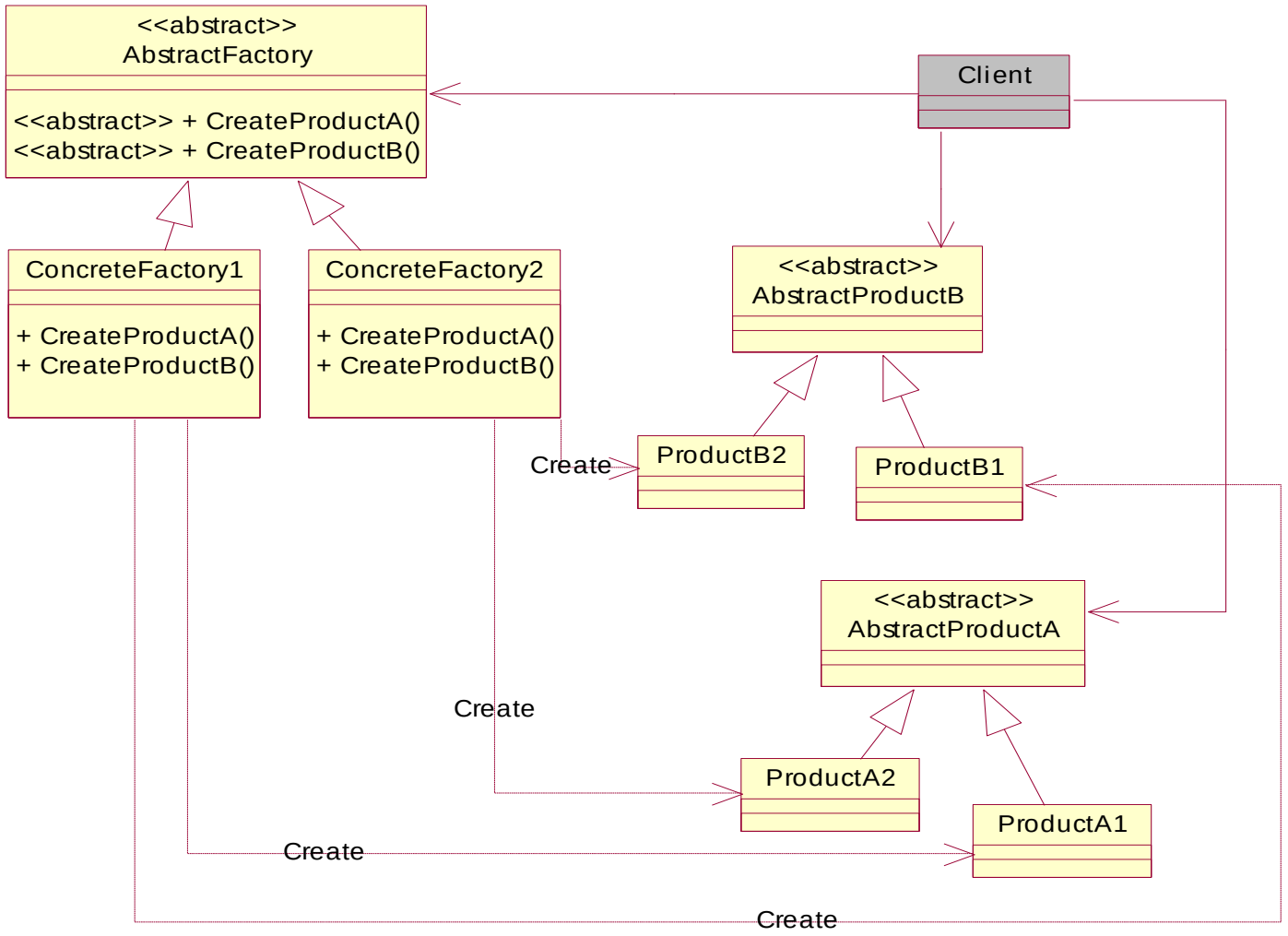


Abstract factory - Solution

- Create an abstract WidgetFactory class that supports the creation of all the user interface elements.
- All the methods of the WidgetFactory class return a user interface element instance (e.g. the returned type should be Window instead of MotifWindow)
- All the custom presentation managers have got their corresponding WidgetFactory derived class (MotifWidgetFactory, PMWidgetFactory), that methods return user interface elements being typical for the given presentation (MotifWindow, PMWindow).
- The client uses a concrete WidgetFactory implementation (e.g. PMWidgetFactory) through the general WidgetFactory interface to create a particular user interface element. These particular user interface elements are only referenced through their abstract base classes (e.g. Window instead of PMWindow)

  (In the client code there will be no concrete Factory for user interface element classes. They may only occur during the configuration when the MotifManager or the PresentationManager class is instantiated. Thus the client is totally independent from the chosen particular presentation type.)

Abstract factory - Structure



Practical example

- Problem: Design a provider independent data access layer for multi-layer applications (e.g. in ADO.NET)
  - ICommand interface
    - SqlCommand
    - OracleCommand
    - OleDbCommand
  - IConnection interface
    - SqlConnection
    - OracleConnection
    - OleDbConnection
  - ...
- The solution (with abstract factory):
  - IDbFactory interface (CreateCommand, CreateConnection, etc. operations).
    - SqlDbFactory implementation
    - OracleDbFactory implementation
    - OleDbDbFactory implementation

This is a built-in solution in .NET 2.0!

Abstract factory

- Use it when
  - The system should be independent from the concrete elements that it creates
  - The system should support multiple product families
  - The system should co-operate with tightly-combined „product” families and it should be forced in the system (e.g. a Motif scrollbar couldn’t be used with a Presentation Manager Window)

- **Advantages**
  - Concrete classes are separated
  - Product families are easily replaceable
  - Facilitates consistency between product families

- **Disadvantages**
  - It is difficult to introduce a new product. The whole architecture of the Abstract Factory should be modified, as it is the interface that records the creational operations. (We can get around this sometimes using inheritance).

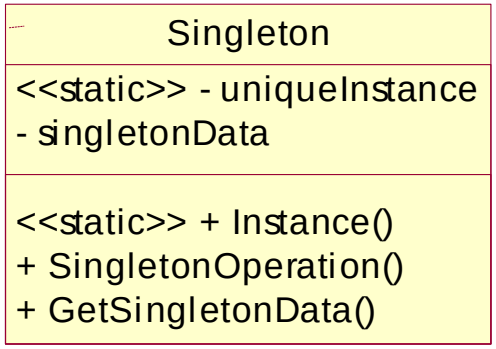
Singleton

Singleton

- Purpose
  - Guarantees the creation of only one instance of a class and provides a global access to that.
- This is a frequently used pattern
  - A window manager object
  - A file system manager object
  - …
- Solution
  - Singleton
    - It should be the responsibility of the class to provide the single instance.
    - Provide global access to the single instance
  - It requires programming language support

Singleton

In this case it is not the structure but the code that is important.




```
Instance() {
  return uniqueInstance;
}
```

The single instance can be created and queried by the help of the Instance class-operation (static!).

- The same object is returned for each call
- We can write it anywhere in the code:
  - In C++: Singleton::Instance()
  - In Java: Singleton.GetInstance()
  - In C# a property should be used: Singleton.Instance
    - → global access to the instance

Singleton C++

C++ implementation

The constructor of the Singleton class is protected! It guarantees that only the static Instance can create an instance of it.

```
class Singleton
{
    public:
        static Singleton* Instance();
    protected:
        Singleton() {; }
    private:
        static Singleton* _instance;
};

Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance()
{
    if (_instance == 0)
    {
        _instance = new Singleton;
    }
    return _instance;
}

int main(int argc, char* argv[])
{
    Singleton* s1 = Singleton::Instance();
}
```

Singleton C#/Java

C#/Java implementation

The constructor of the Singleton class is protected! It guarantees that only the static Instance can create an instance of it.

```
public class Singleton
{
    private static Singleton instance = null;
    public static Singleton Instance
    {
        get
        {
            if (instance == null)
                instance = new Singleton();
            return instance;
        }
    }

    protected Singleton() { }
    public void Print() {…}
}
```

Singleton

Don’t need to know

The previous solutions are not thread-safe

In the previous examples the initialization of the static member variables takes place when they are first used.

The C# compiler generates thread-safe code for this solution.

```
public class Singleton
{
    private readonly static Singleton instance =
        new Singleton();

    public static Singleton Instance
    {
        get { return instance; }
    }

    protected Singleton() { }
    public void Print() {…}
}
```

Overview of creational design patterns

- The purpose of the creational design patterns: instantiate types to support flexibility, extendibility and reusability.
  - Factory method: delegates the instantiation to the derived class
  - Abstract factory: factory hierarchy to instantiate elements of different product families
  - Singleton: only one instance exists, provides global access to the single instance
  - …
- As an alternative to the factory-like creational patterns, most modern languages support some kind of a reflection technique
  - The type that we want to instantiate can be specified as a string.
  - Disadvantage of reflection:
    - much slower
    - not strongly typed anymore, bigger chance to make a mistake

Structural patterns

Adapter

Composite

Facade

Proxy

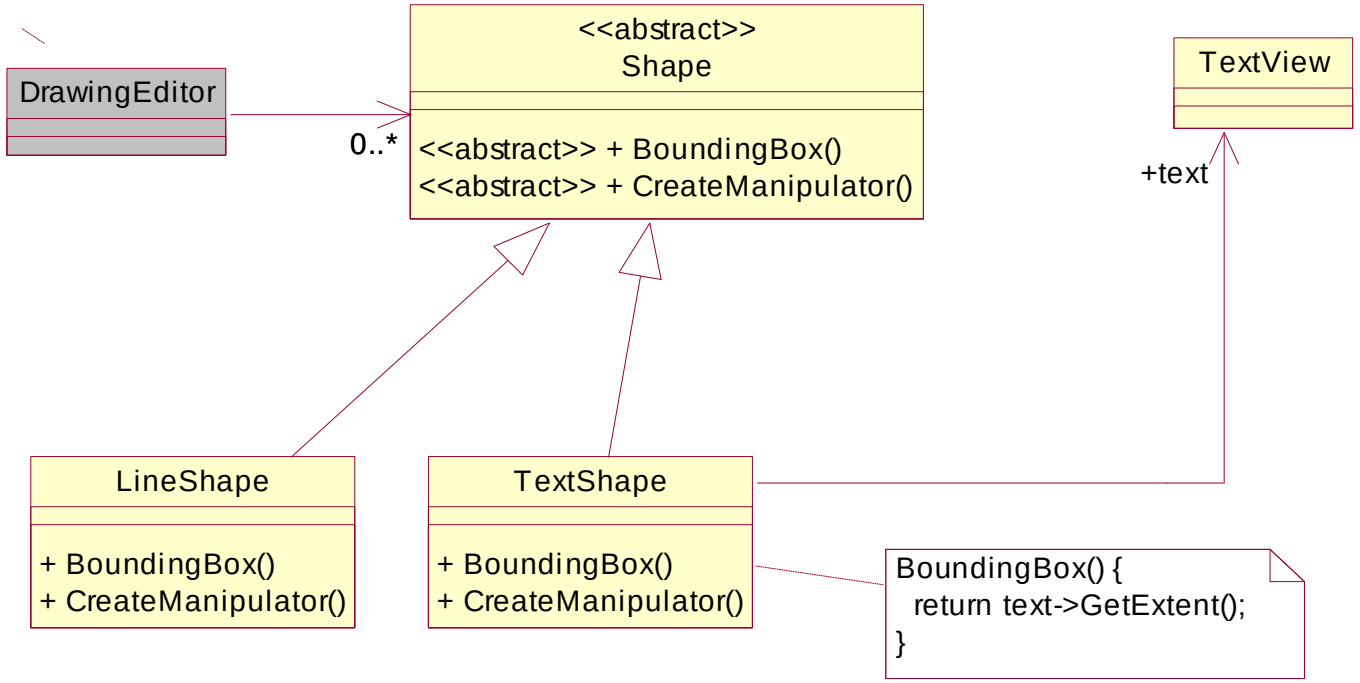
Adapter

(Wrapper)

Adapter

- Purpose
  - Converts the interface of a class to an other one that is expected by the client. Facilitates the co-operation of classes even if they have got incompatible interfaces.
- Example
  - Graphical Editor
    - Graphical shapes (line, polygon, **text**) – they derive from the Shape class (line – LineShape, polygon – PoligonShape, text – TextShape, etc.)
    - The implementation of the TextShape class is difficult, but suppose we have got a TextView class that knows everything that is needed for the TextShape
    - Problem: The TextView class can not directly be used as it hasn't got the required interface, e.g. it does not derive from the Shape class (it does not support the Shape interface, so it can not be handled as a Shape)
    - Solution: use the Adapter design pattern

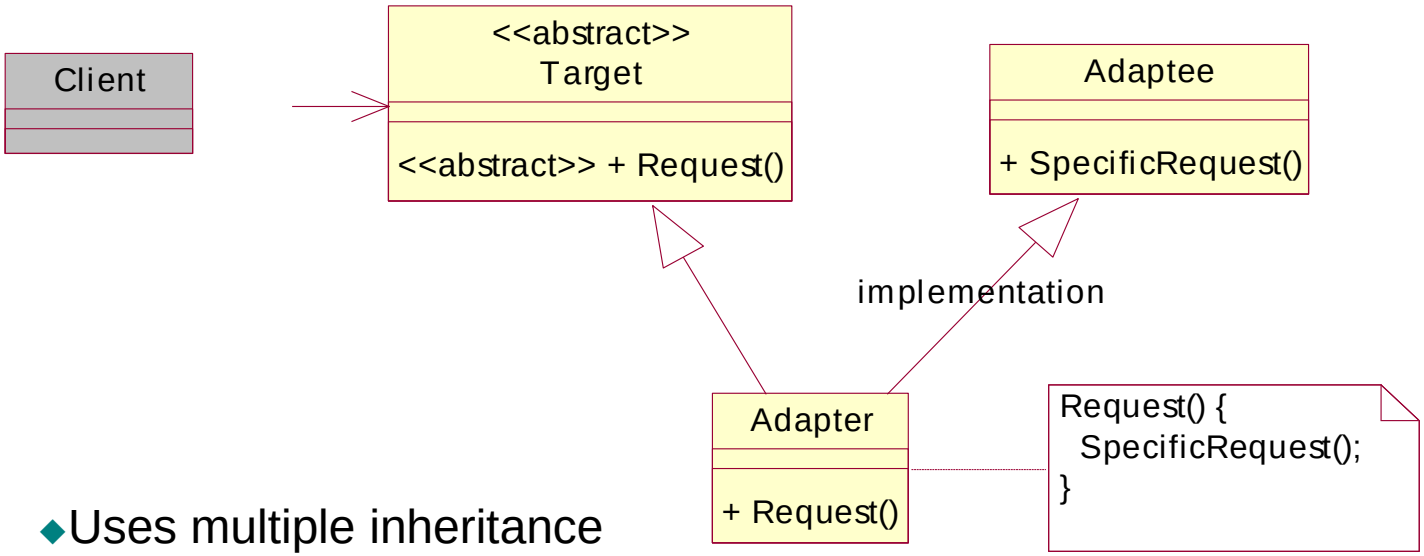
Adapter



- **Two types**
  - Class adapter
  - Object adapter
- **The above example is an „object” adapter**

Adapter

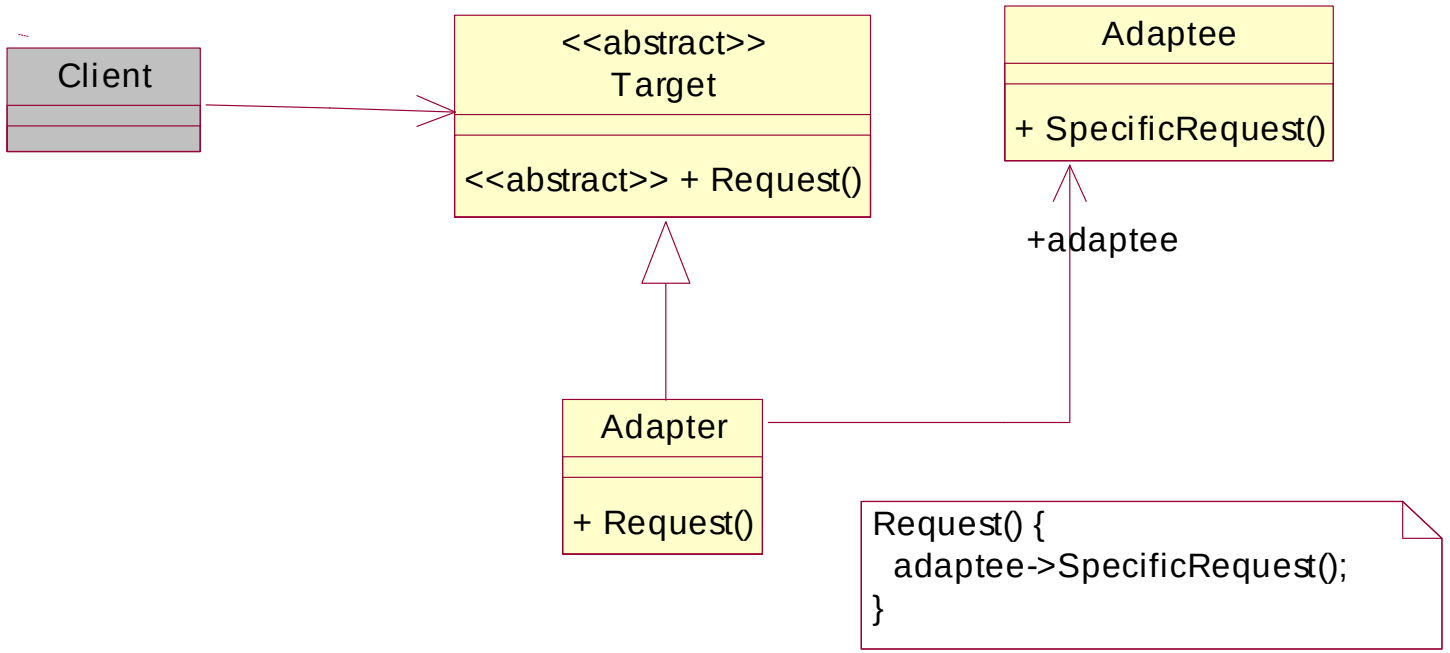
- Structure- class adapter (first version)



- Uses multiple inheritance
- *Participants*
  - *Adaptee (TextView):* interface that is adapted (this is the one that needs to be converted)
  - *Adapter (TextShape) :* this is the converter, converts interface Adaptee to interface Target
  - *Target (Shape):* the interface that the client expects

Adapter

- **Structure – object adapter (second version)**



- Uses object composition, uses delegation
- Participants: just like in the *Class Adapter*
  - The Adapter contains a reference of the Adaptee
  - The Adapter delegates the operations to the Adaptee
  - The Adapter can contain multiple Adaptees.

Adapter

- Implementation
  - In C++ in case of using a class adapters
    - Private inheritance from the Adaptee class (implementation inheritance)
    - Public inheritance from the Target class (interface inheritance)

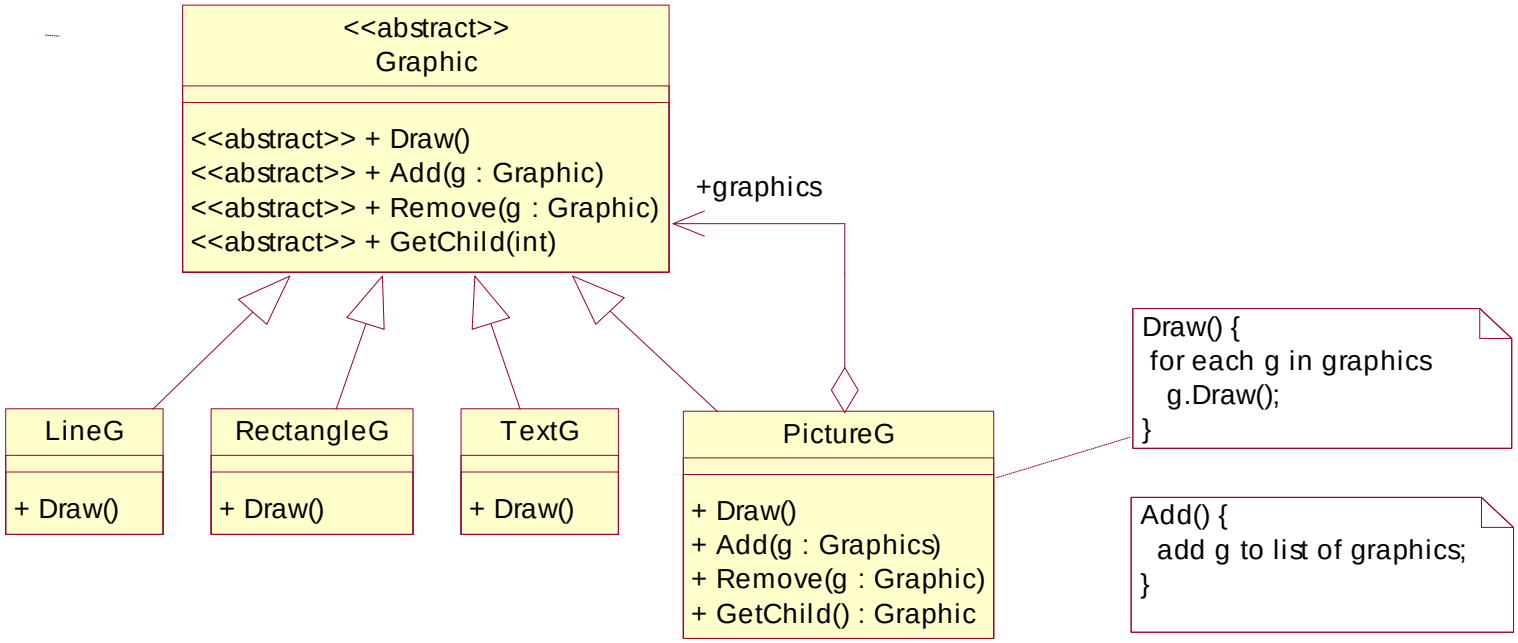
Adapter

- Use it when
  - We want to use a class that doesn't implement the required interface
  - We want to create a reusable class but we don't know the required interface for that in advance (pluggable adapters)

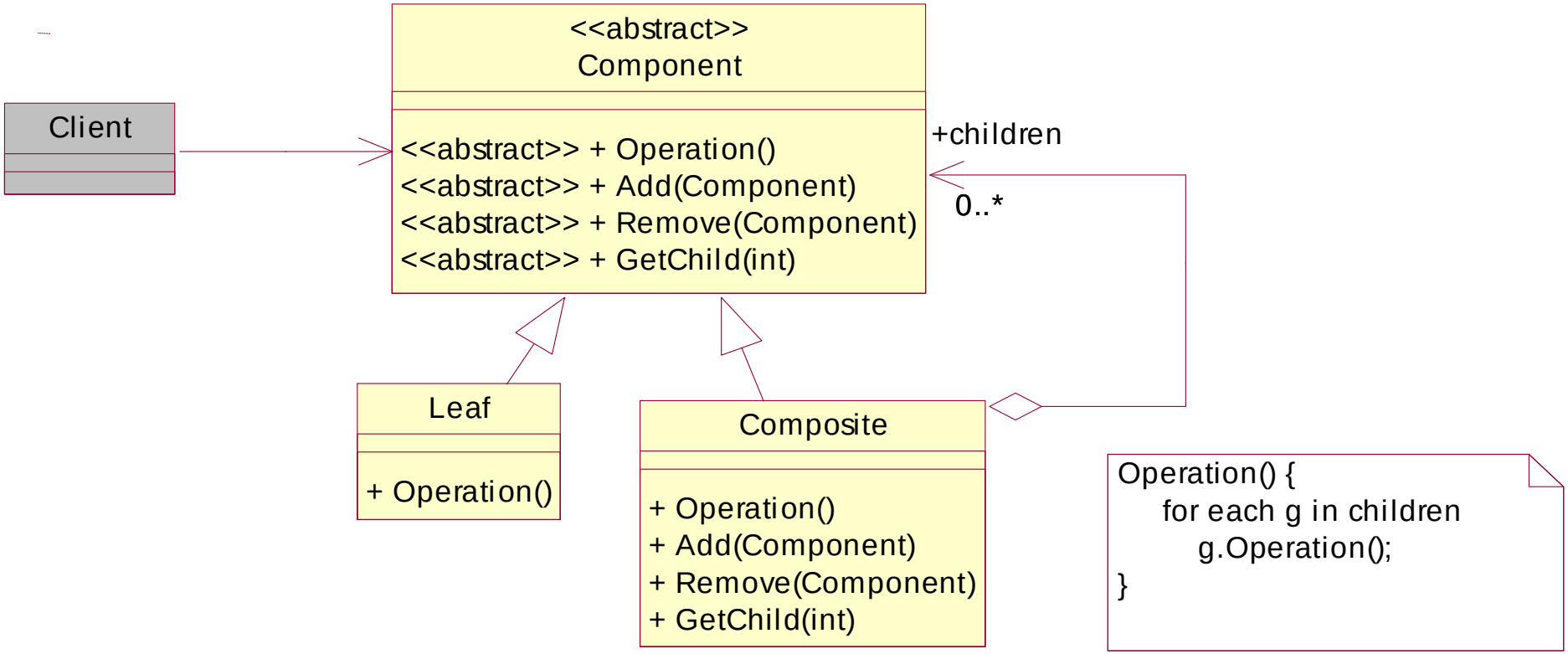
Composite

Composite

- Purpose
  - Arrange objects that are in whole-part relationships into a tree structure
  - It hides from the client whether an object is a composite or a simple one. Thus, all the objects can be handled uniformly.
- Example
  - A graphical application that facilitates the creation/usage of compound graphic objects.



Composite - structure



Composite

- Use it when
  - You want to express the whole-part relationship between objects
  - You want to hide from the client whether an object is an elementary or a composite one. You want to handle them in a uniform way

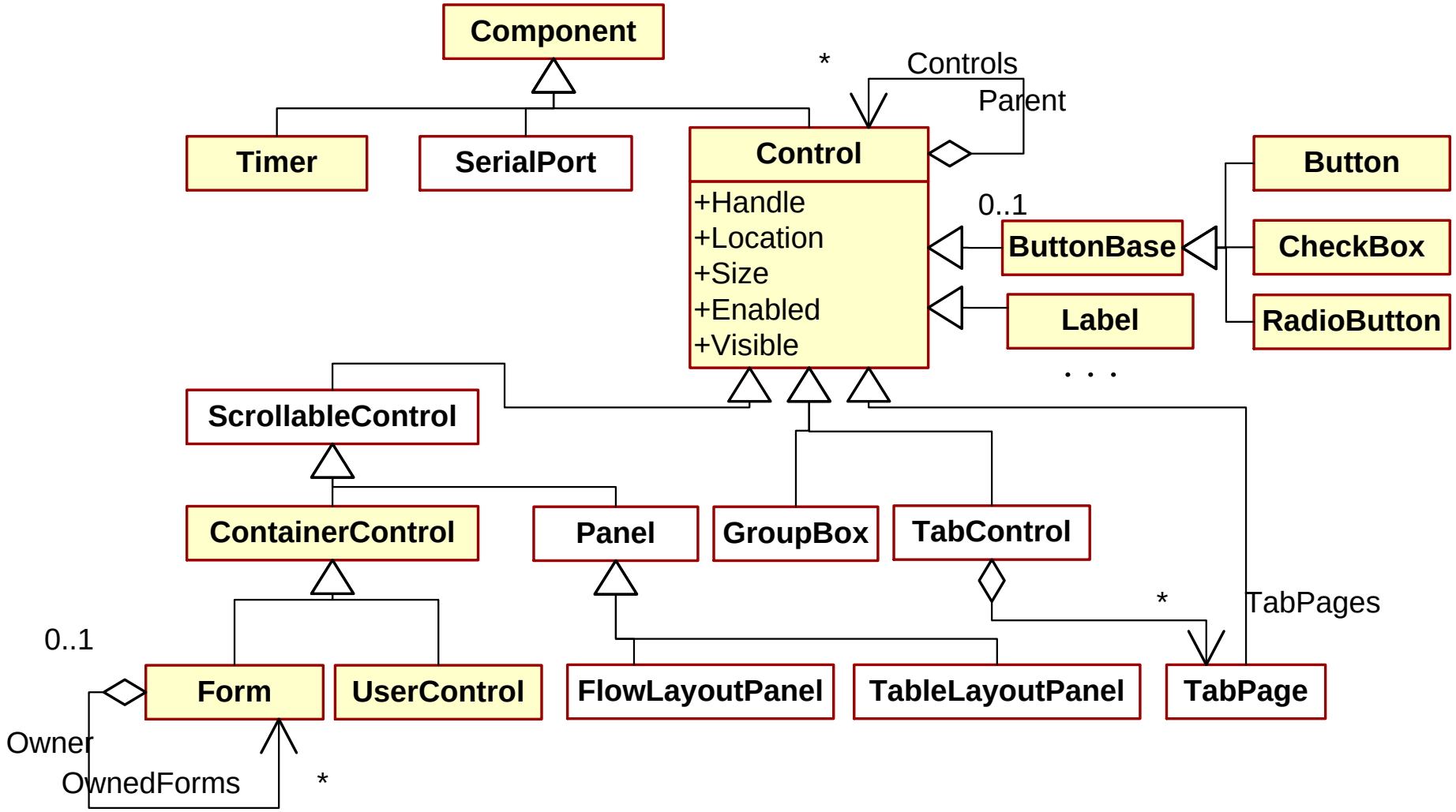
Composite

- Question: In which class should we implement the logic for the composite operations (Add, Remove, …), these operations can not be used in the leaf objects (LineG, TextG, etc.)
  - In the **Component** class (just like in the example):
    - Advantage: the objects are uniform, as both have got Add, Remove, GetChild operations
    - Disadvantage: not safe, but we may throw an exception if an operation shouldn't be called on an object (e.g. Add on a leaf object)
  - In the ***Composite*** class:
    - Disadvantage: the opposite of the previous solution
    - E.g. the ***Component*** base class may have an **IsComposite***: bool* virtual method that is overridden in the derived classes. For the leaf classes it should return false otherwise it should return true.

Composite

- Question: Where does this pattern appear in Windows Forms applications?

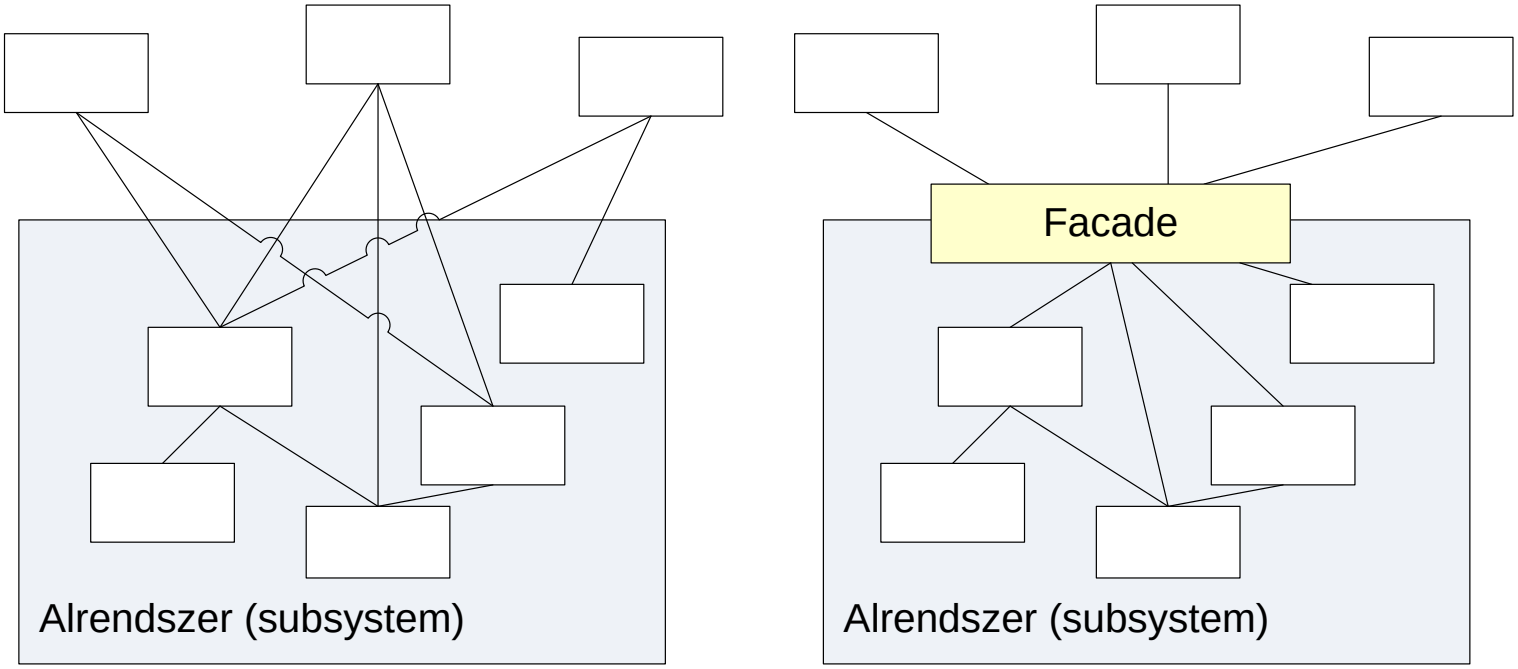
Component/Control hierarchy



Facade

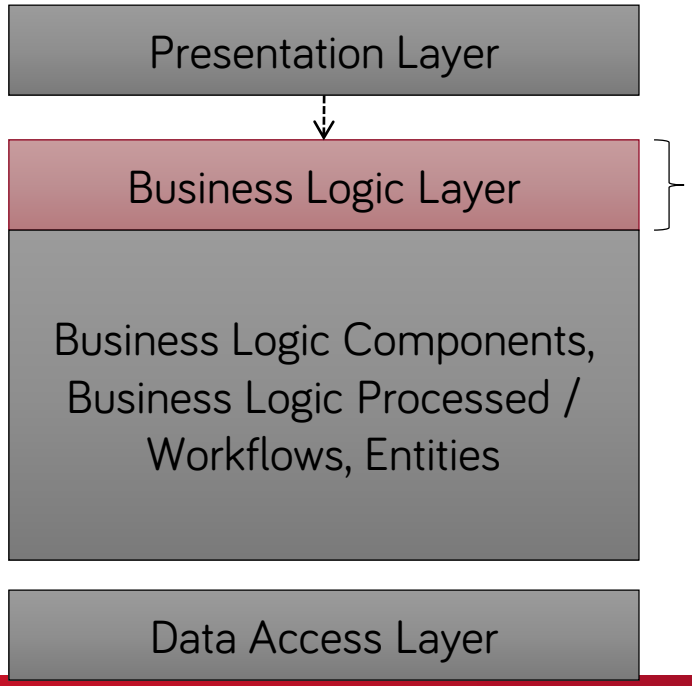
Facade

- Purpose
  - Introduces a uniform interface instead of the many interfaces of a subsystem.
  - It defines a higher level interface that makes the use of the subsystem more simple.
  - Example



Facade

- Example 1
  - Compiler
    - Command line compiler: cpp.exe, this has got many parameters
    - We can not access to the internal of the compiler (parser, tokenizer, etc.)
- Example 2
  - Multi-layer (usually enterprise) applications



The Presentation Layer just accesses this thin Business Logic Layer

Facade

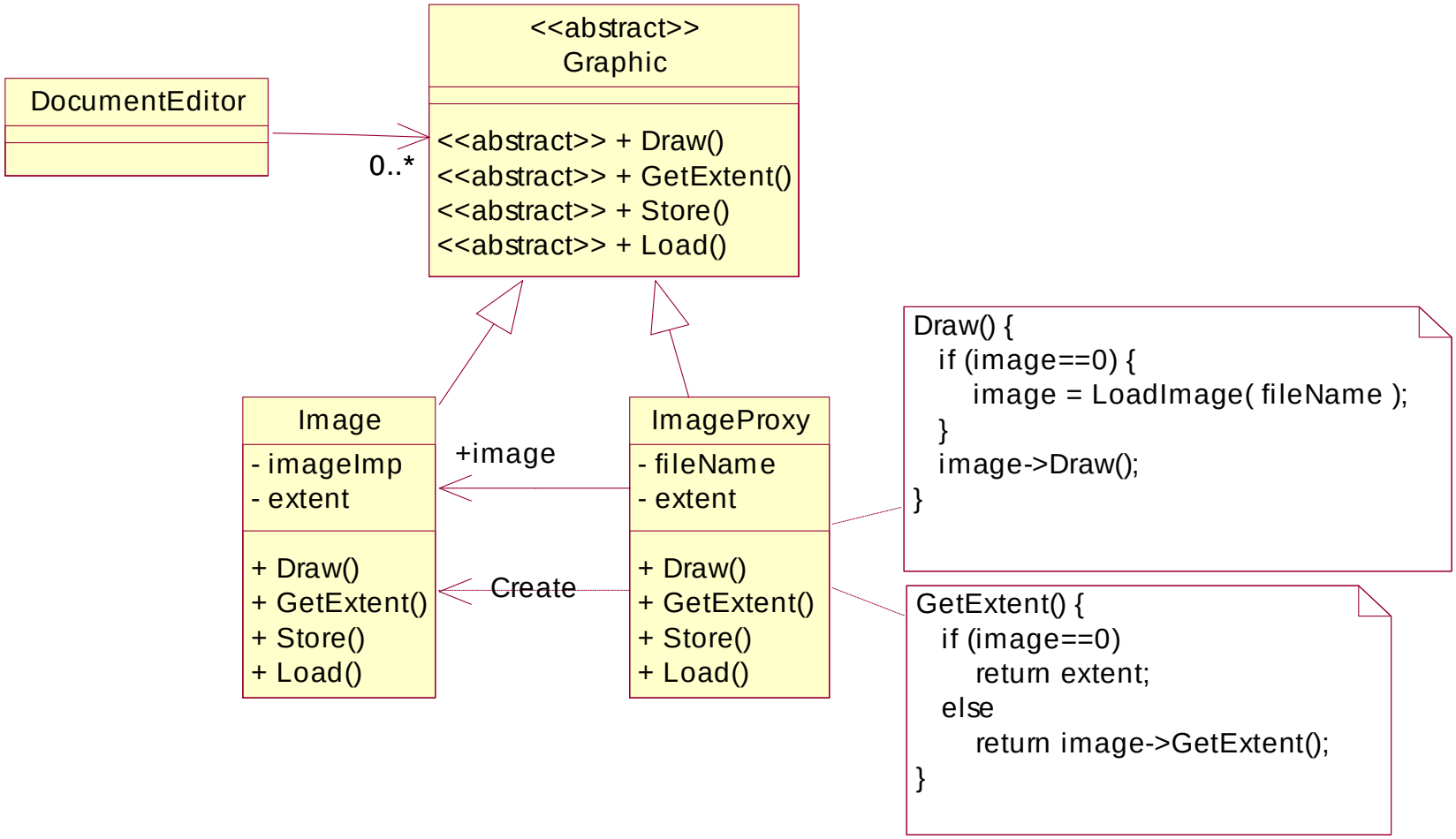
- Use it when
  - We need a simple interface above a complex system
  - There are many dependencies between the components of the client and the subsystem. Introducing a Facade reduces the dependency between them and it supports portability, too.
  - In case of layers
- Remark
  - Question: should we let the client access the internal components of the subsystem?
    - Introducing the Facade does not force the client to use it, the client can still use the internal components

Proxy

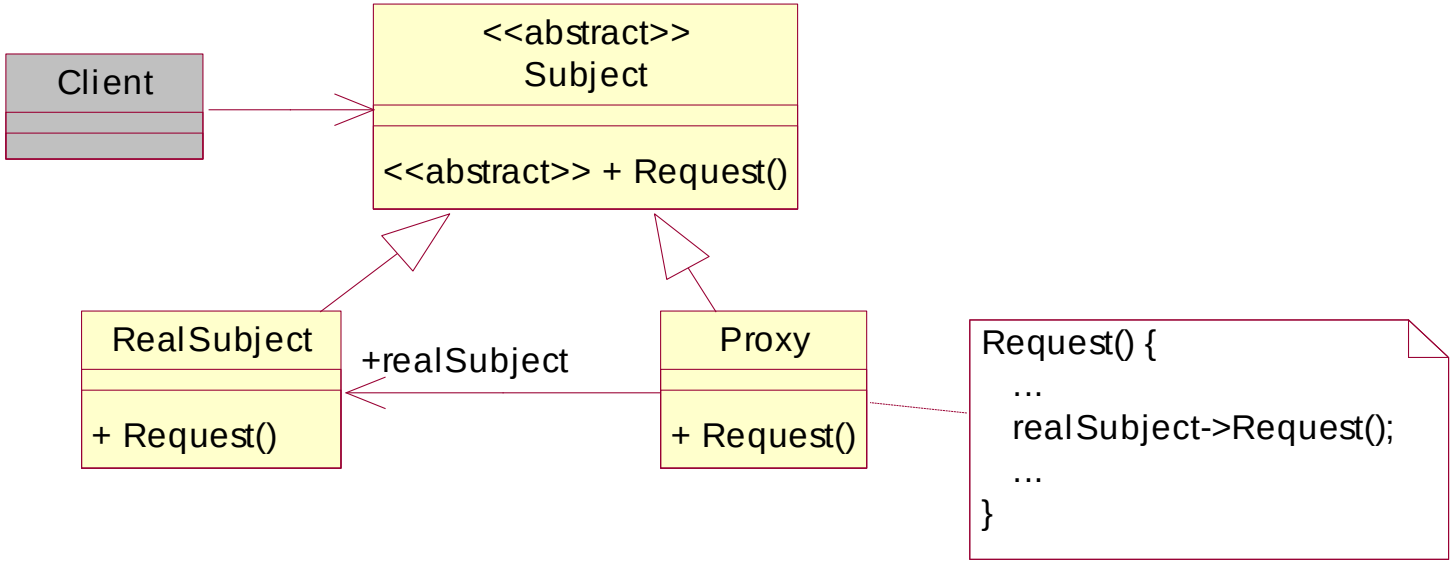
Proxy

- Purpose
  - Instead of the real object, a substitute is used that controls access to the real one
- Examples
  - Text-editor
    - Many large images
    - They don't need to always be displayed, only when we scroll there, that is when they are visible.
  - Solution: Proxy
    - Substitute the image with an object (proxy) that displays the image if that is loaded otherwise it first loads that and than displays that.

Proxy


<<abstract>>
Graphic
<<abstract>> + Draw()
<<abstract>> + GetExtent()
<<abstract>> + Store()
<<abstract>> + Load()
DocumentEditor
0..*
Image
- imageImp
- extent
+ Draw()
+ GetExtent()
+ Store()
+ Load()
+image
Create
ImageProxy
- fileName
- extent
+ Draw()
+ GetExtent()
+ Store()
+ Load()
Draw() {
if (image==0) {
image = LoadImage(fileName);
}
image->Draw();
}
GetExtent() {
if (image==0)
return extent;
else
return image->GetExtent();
}

Proxy



- ***Subject***: common interface for the Subject and the Proxy (this is the core of the pattern!)
- ***RealSubject***: this is the real object that is hidden by the proxy
- ***Proxy***: the substitute object. Contains a reference for the real object. It controls the access to the real object. It may also be responsible for creating and deleting the object.

Proxy

- Remote Proxy
  - A local representative of a remote object. The communication with the remote object is transparent for the client. It doesn't even notice that it is communicating with a different process / an other computer.
- Virtual Proxy
  - On-demand creation of resource intensive objects (e.g. images)
- Protection Proxy
  - The access control to the real object is based on permissions.
- Smart Pointer
  - Encapsulating a pointer so that additional features can be provided (e.g. reference counting, locking)

Behavioral patterns

Template method

Command

Memento

Observer

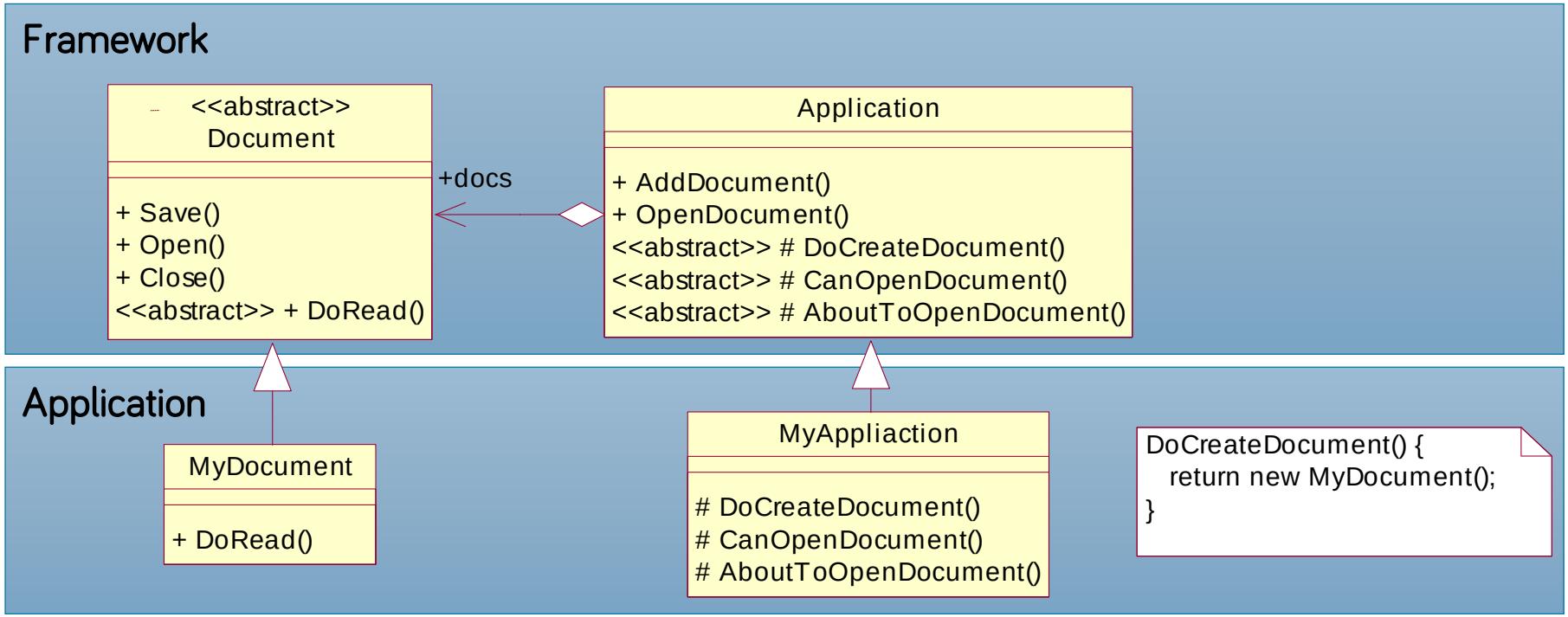
Iterator

State

Template method

Template method

- Purpose
  - Defines the skeleton of an algorithm in the base class and delegates the implementation of some steps to the derived class.
- Example: Opening a document in a framework
  - Let us have two classes in the framework: ***Application*** and ***Document***. The programmer has to derive from them to implement the application-specific custom behavior.



Template method

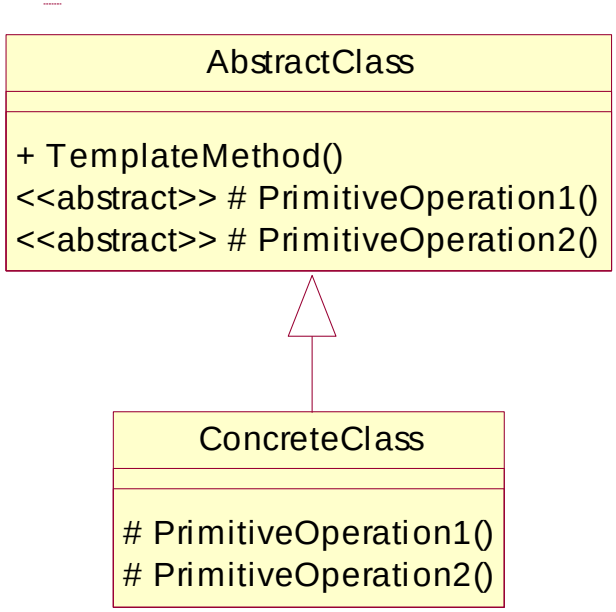
- The OpenDocument method is a so called template method in the example
  - Defines the order of the operations
  - Calls some abstract methods that need to be implemented in the derived class.

```
// Az Application class is a part of the framework
void Application::OpenDocument (const char* name) {
   if (!CanOpenDocument(name)) {
       // cannot handle this document
       return;
    }
    // DoCreateDocument must be overwridden in the derived
     // MyApplication class. This will contain
     // the fully-fledged implementation
    Document* doc = DoCreateDocument();

    if (doc) {
            _docs->AddDocument(doc);
            AboutToOpenDocument(doc);
            doc->Open();
            doc->DoRead();
    }
}
```

Template method

- Structure




```
TemplateMethod() {
  ...
  PrimitivOperation1();
  ...
  PrimitivOperation2();
  ...
```

Template method

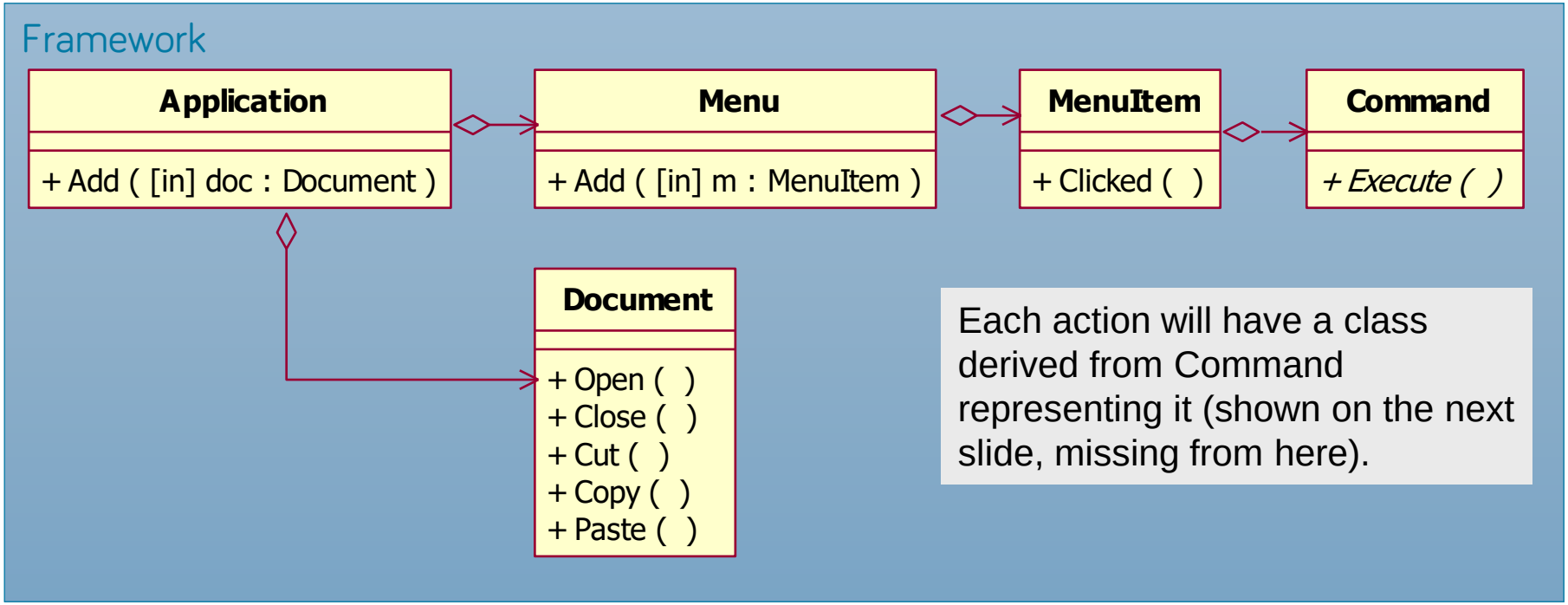
- Consequences
  - The common parts of an algorithm can be placed into the base class while the derived class contains the custom implementation.
  - We can avoid code duplication. The base class defines and calls some abstract methods that need to be overridden in the derived classes. The overridden abstract methods of the derived classes contain the custom implementations.
  - We can define so called hook methods: these are the extension points in the code.
- Notes
  - Typical in the case of frameworks
  - In .NET presents a simpler solution: delegates. This pattern is usually used in C++ and Java.
  - It has got nothing to do with C++ templates (those are generic types)

Command

Command

- Purpose
  - Encapsulating an operation into an object. It facilitates the reuse, undo, queue and easy modification of the operations. The operations are usually responses to client interactions.
- Alternative name
  - Action
- The concept and the implementation is quite language-dependent (C++, .NET, etc.)
- Example: user commands
  - Menu, button, toolbar button
  - Participants e.g.: Application, Document, Menu, Submenu, Menu item, etc.
  - Problem: the writers of the framework couldn't have implemented the handling of the menu item selection (what should happen when a menu item is selected) →
  - How to react to the menu item click event?
    - Callback function – not object-oriented (structural) solution
    - Adapter-based solution – Java
    - Delegate-based solution- .NET
    - Command pattern

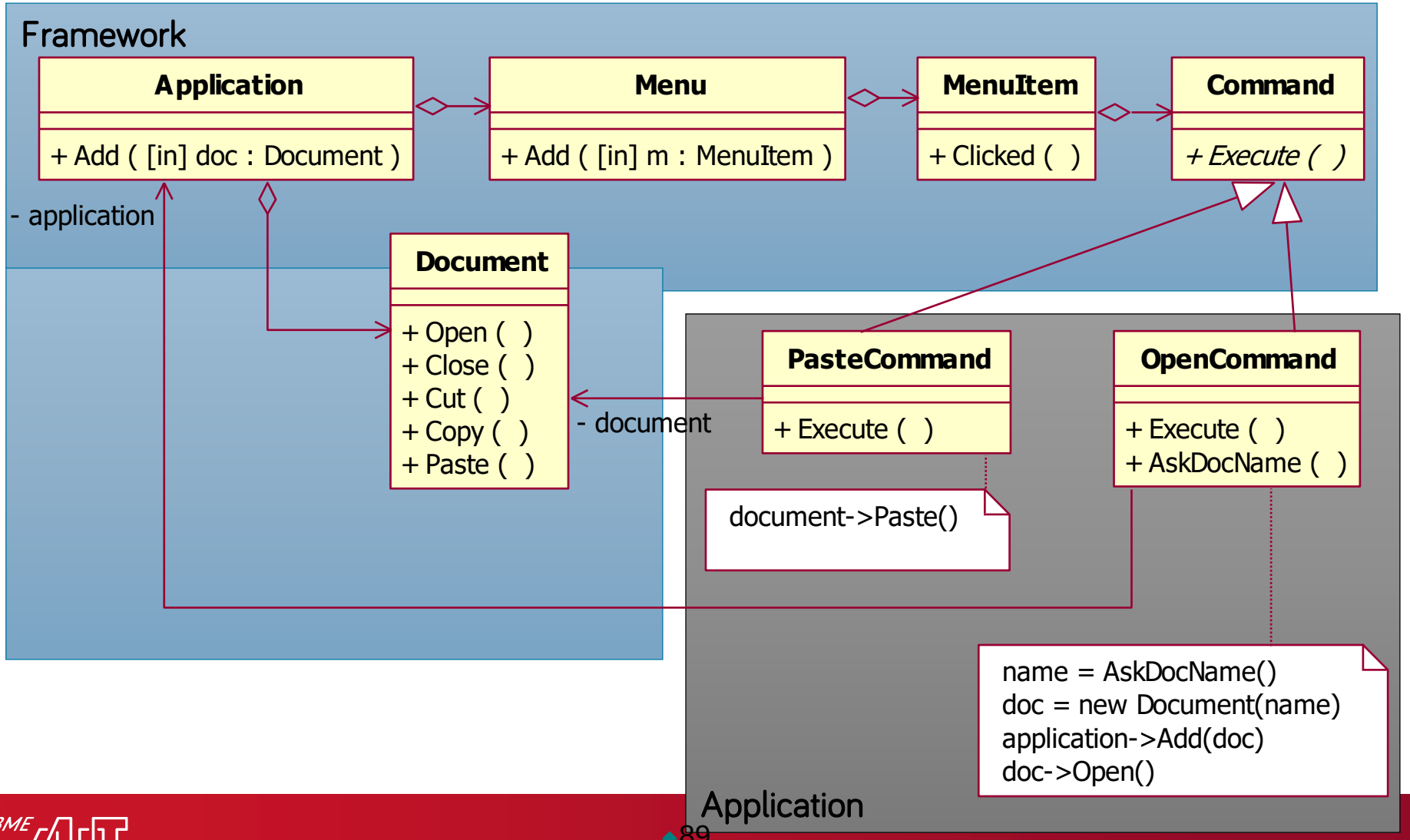
Command



- Encapsulate user actions (and the corresponding operations) into classes that derive from the ***Command*** base class and assign them to the menu items!

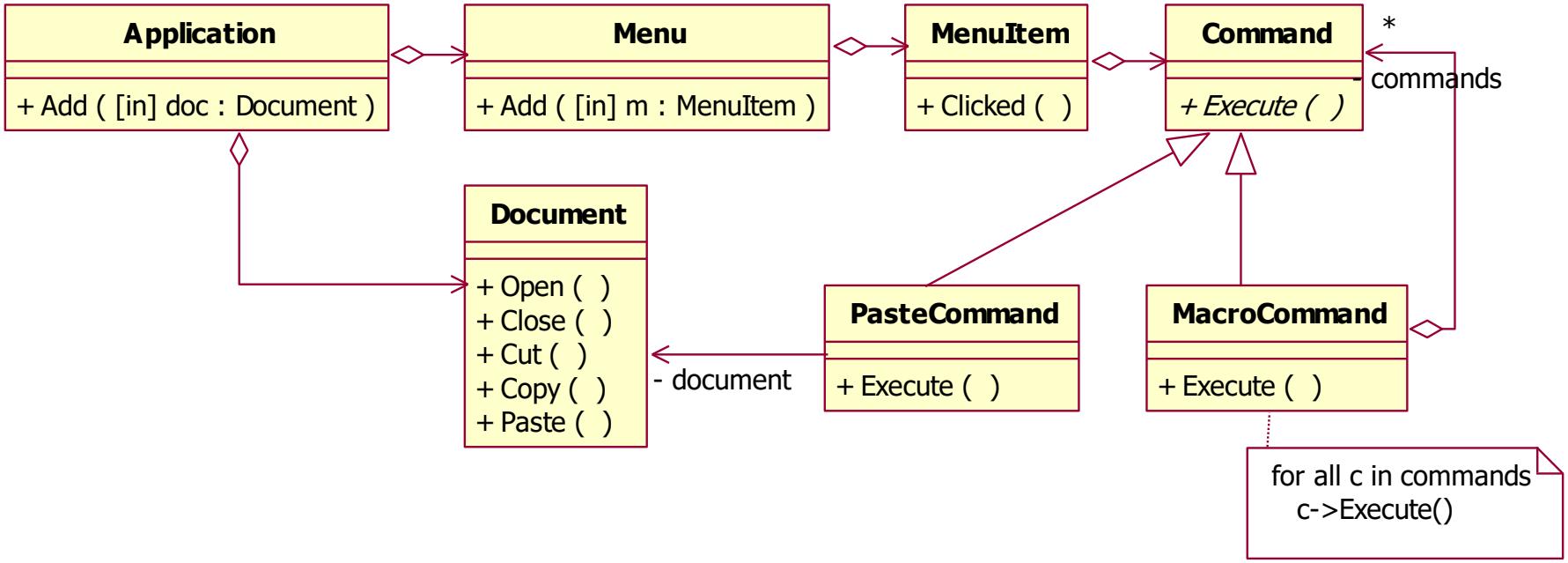
Command

Example of custom implementation: Paste, Open



Command

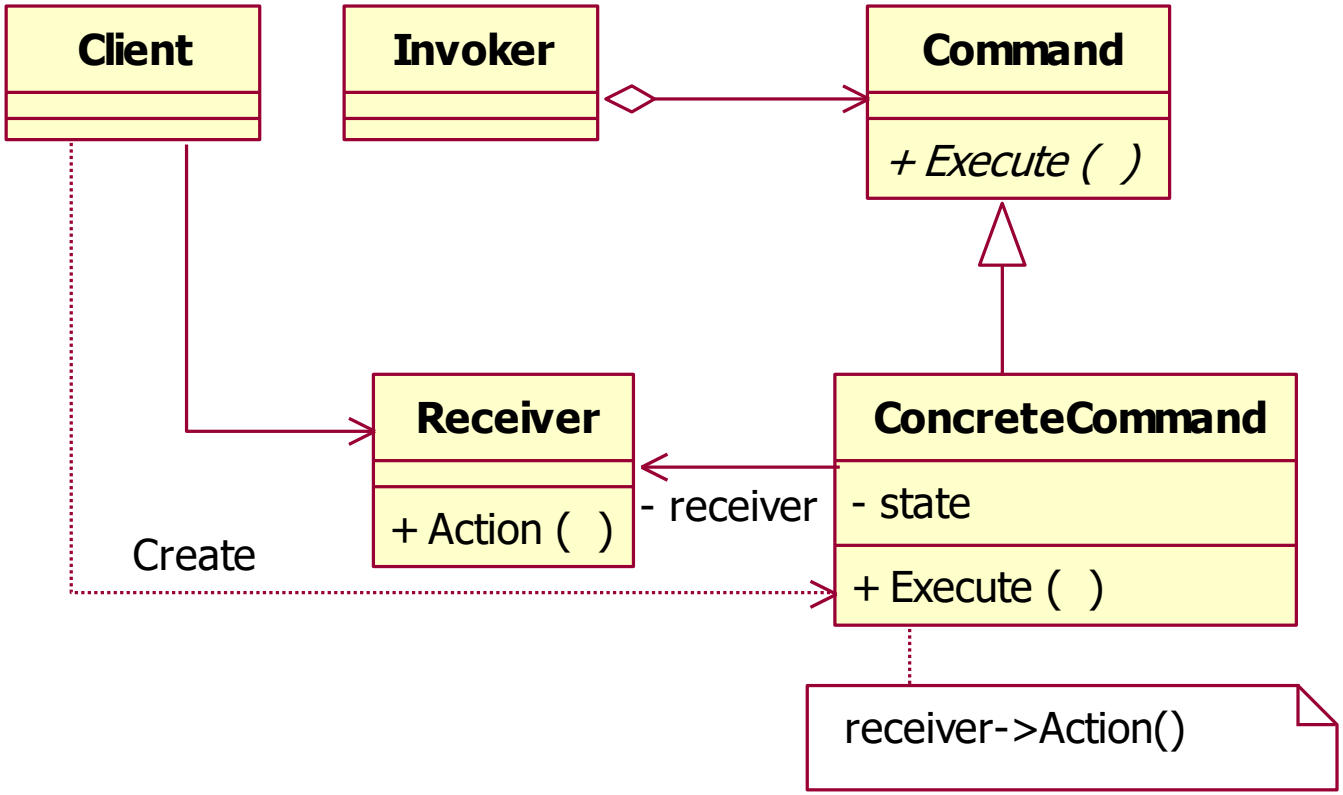
Macro: sequence of commands



Command

- Use it when
  - If in a structured programming language you would use a callback function (to achieve partial functionality), in OO use a command instead.
  - You would like to queue the operations (commands) that can even be executed in separate threads/processes. The parameters of the operations can be stored in the command objects.
    - A special case is when you would like to execute the commands one-by-one in a thread. E.g. in the GUI thread, but in this case the Control.Invoke() is the good solution
  - Supporting undo of the command – the state before the command can be saved

Command - Structure

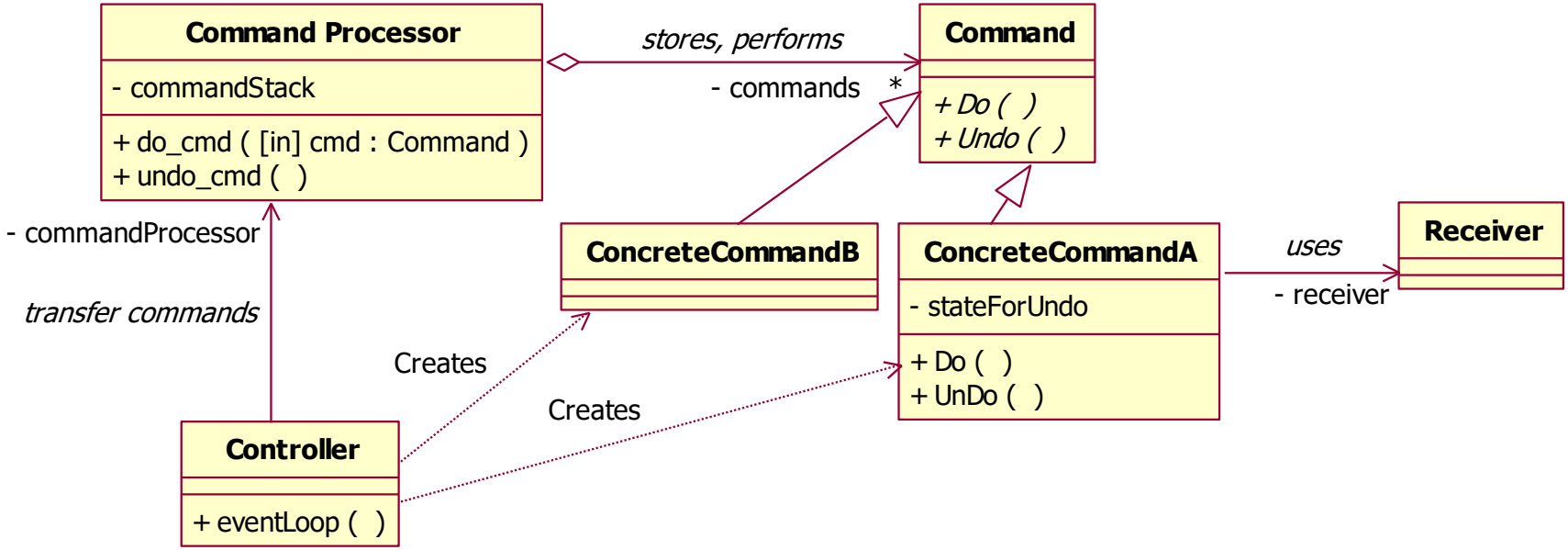


Command

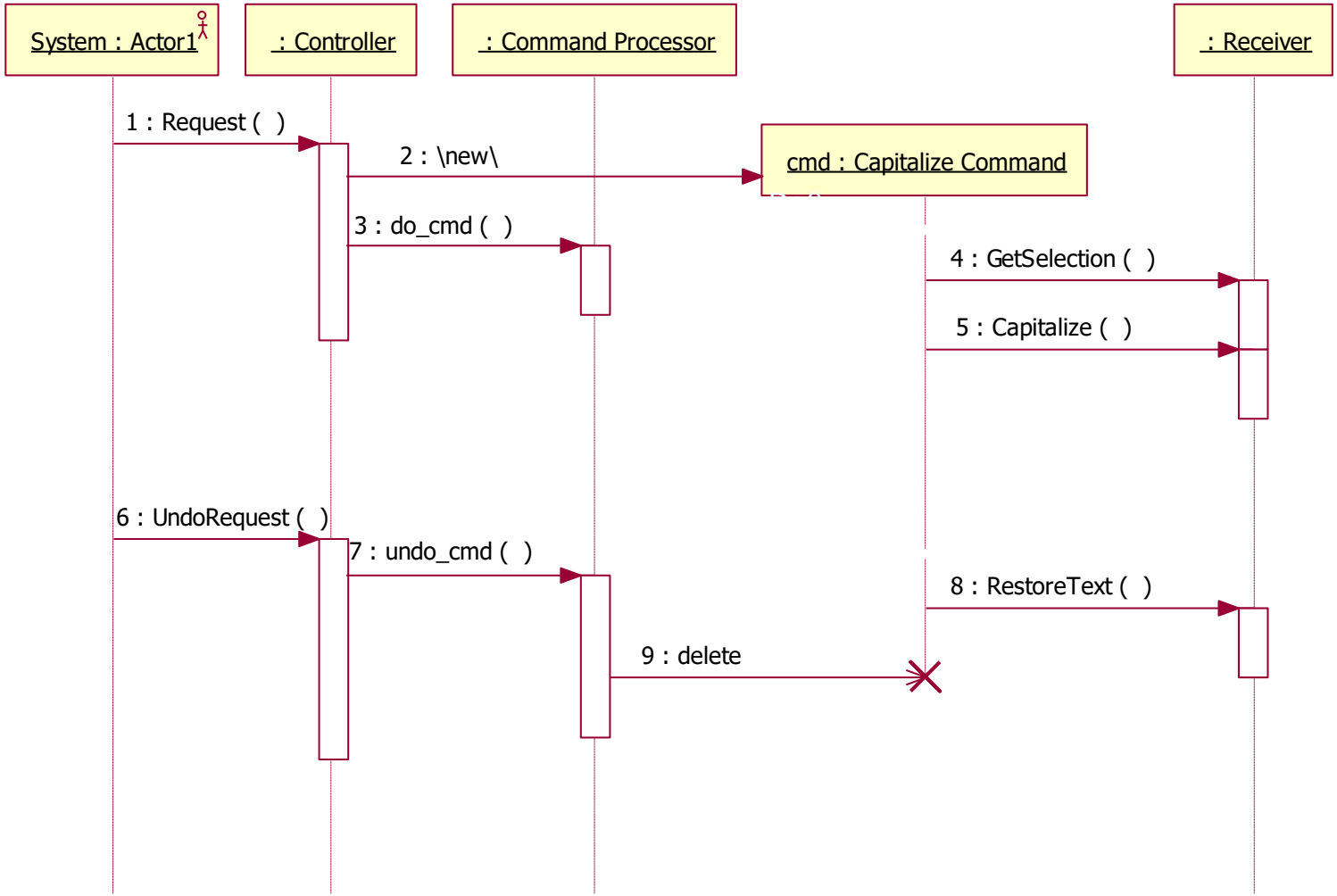
- The handling of the command is separated from the invoker of the command
- Makes an extendable system. New commands can be added by deriving from the Command base class
- Adding new commands does not effect the existing ones.
- Complex commands are also supported
- A command can be assigned to multiple user interface elements (menu item, toolbar button, etc.)

Command

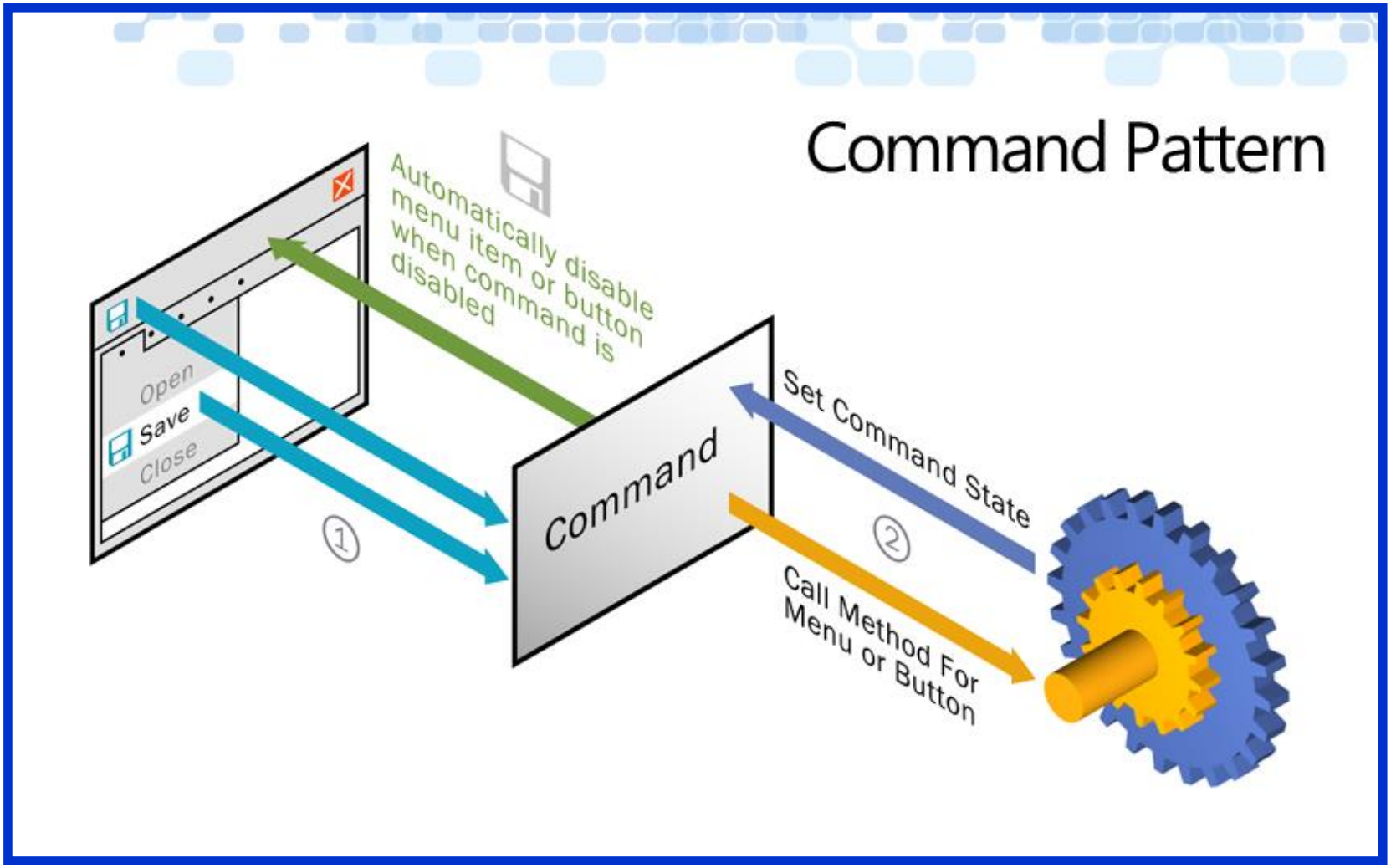
- Command Processor
  - A variation of the Command pattern
  - Supports a basic undo out of the box
  - The *Command Processor* is the key class
    - Stores the Command objects
    - Activates (executes) the Command objects and provides some other services



Command Processor



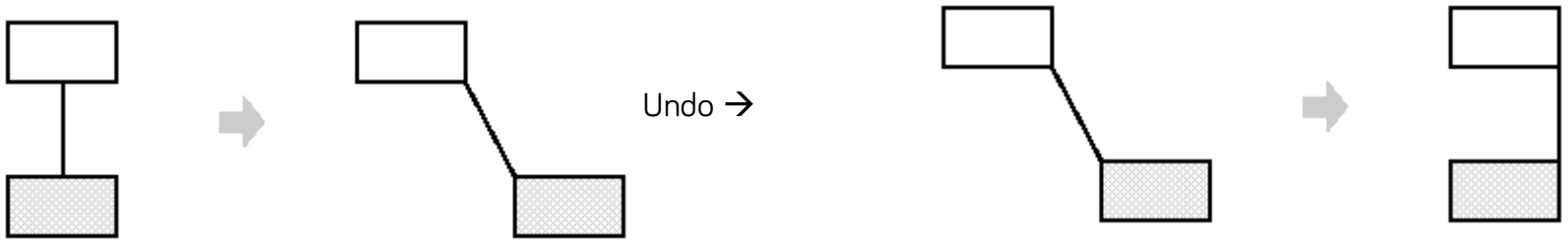
CommandBinding in .NET



Memento

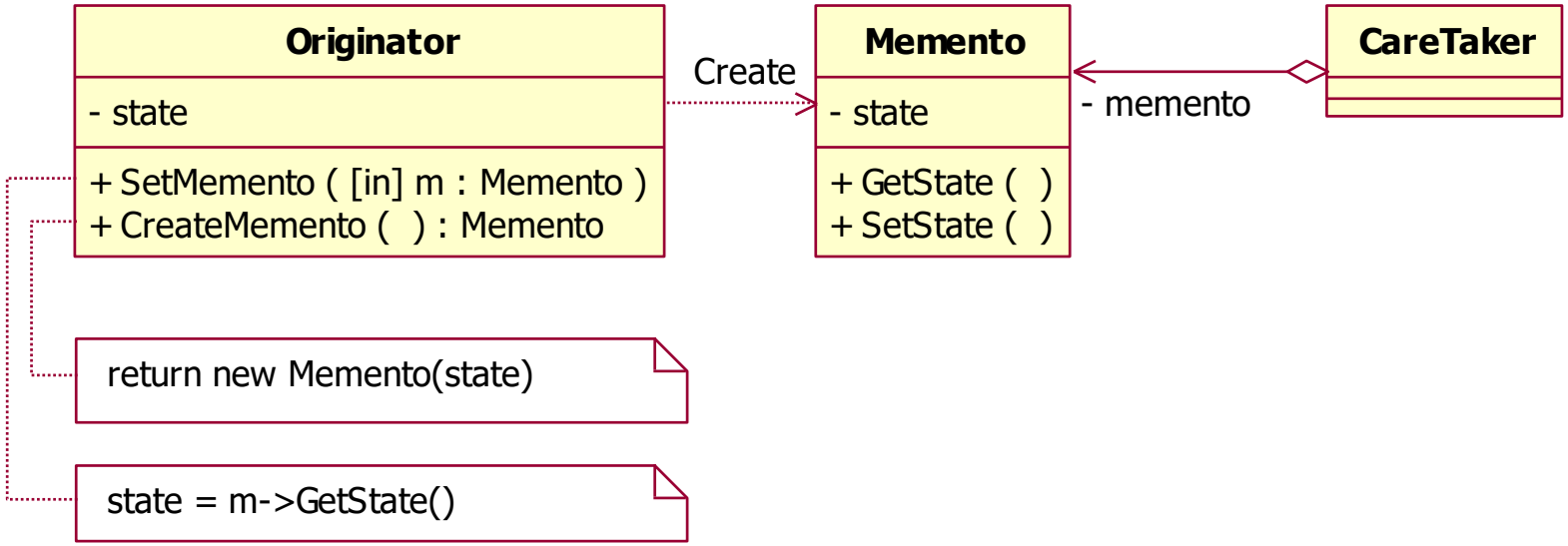
Memento

- Purpose
  - Sharing and storing the internal state of an object so that it can be restored later.
- Example: Undo operation for a Document
- Solution:
  - Invertible operations in the Command pattern
    - It is hard to be solved in many cases
    - It is impossible in many cases
      - The public interface of the object doesn't have an operation for the undo.



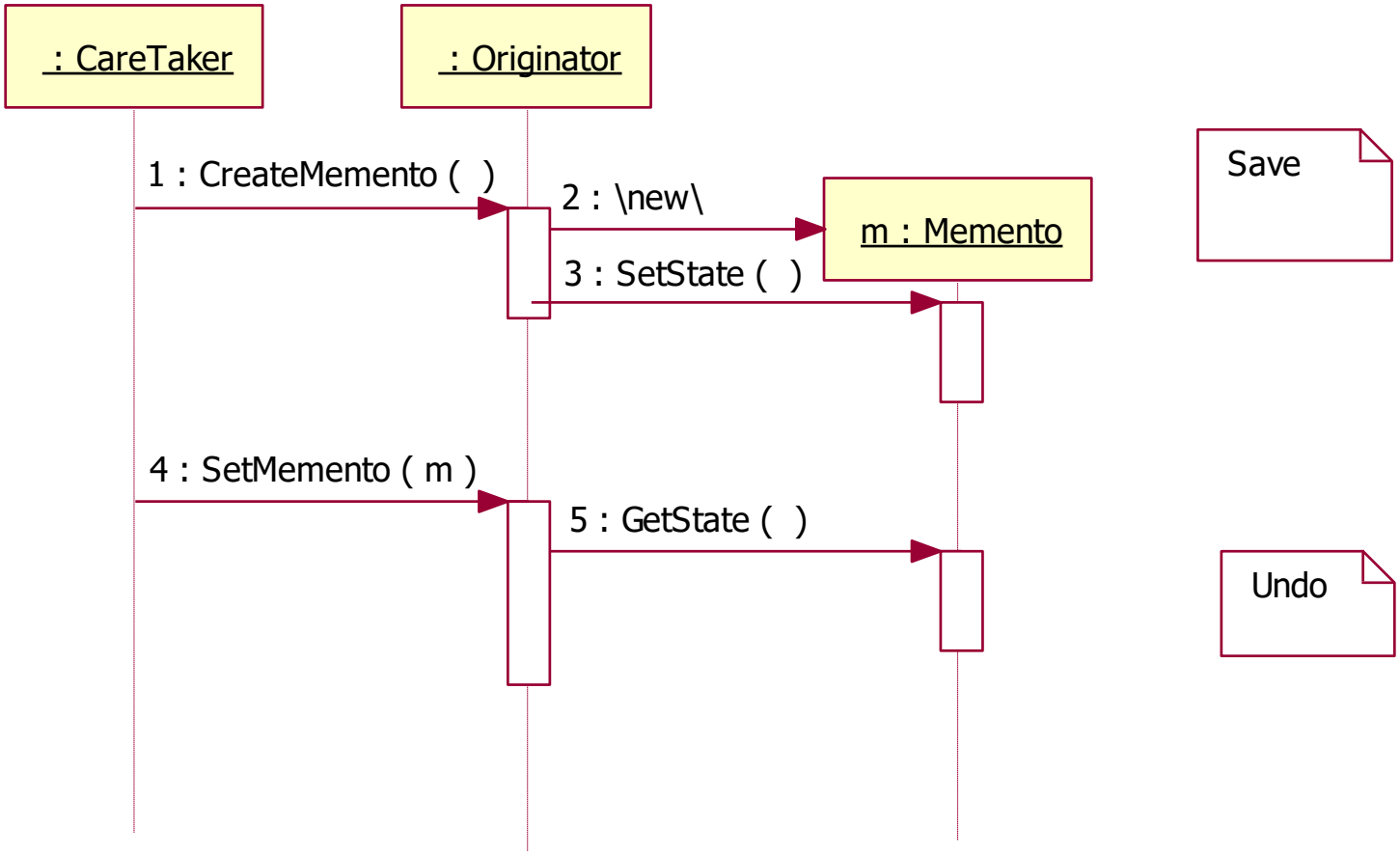
  - In these cases there is just one solution: we have to save the original state of the object.

Memento



- *Originator*: this is the object of which state we want to restore.
  - The *CreateMemento*() creates a Memento that holds the actual state of the object
  - The *SetMemento*() restores the state of the object based on a previously saved Memento.
- *Memento*: stores the state of the Originator. Theoretically only the Originator should be able to read its content. E.g.: using the „friend" declaration in C++
- *CareTaker*: Contains and maintains the memento objects.

Memento


: CareTaker
: Originator
1 : CreateMemento ( )
2 : \new\
m : Memento
3 : SetState ( )
4 : SetMemento (m)
5 : GetState ( )
Save
Undo

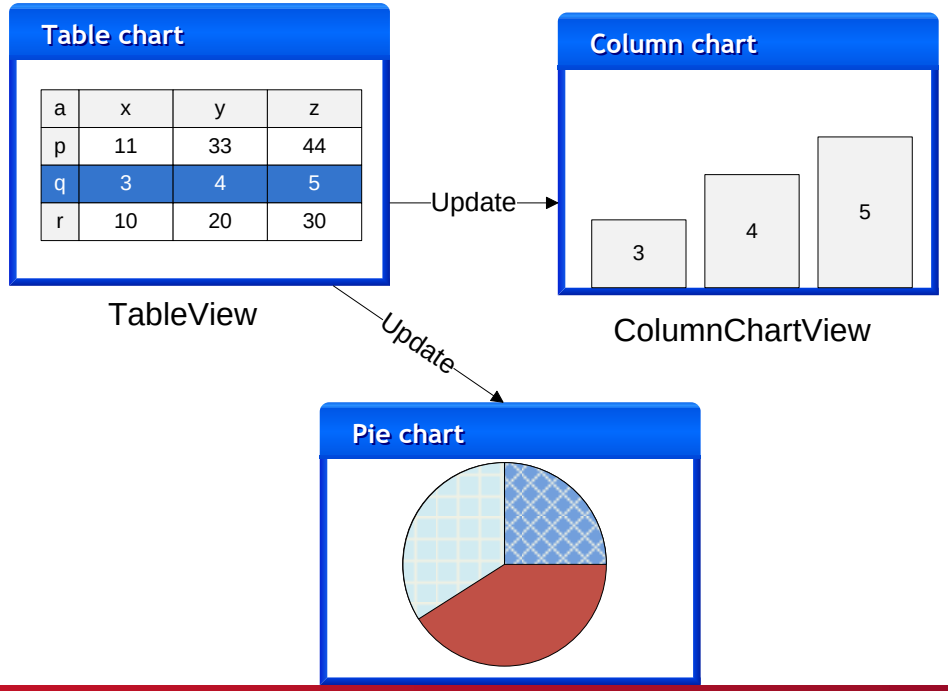
Memento

- Use it when
  - the state of an object needs to be restored and a direct interface that support this would break OO encapsulation.
    - Advantages:
      - It preserves the concept of encapsulation
      - Makes the Originator simpler
    - Disadvantages
      - Using mementos is often resource-intensive
      - It is hard to determine in advance the amout of memory required by the Caretaker

Observer

Observer

- Purpose
  - How can objects notify each other about the change of their state without having dependencies to each other.
- Example: MVC or Document-View architecture
  - The user changes the data through one of the views, how can the rest of them be notified about the change?



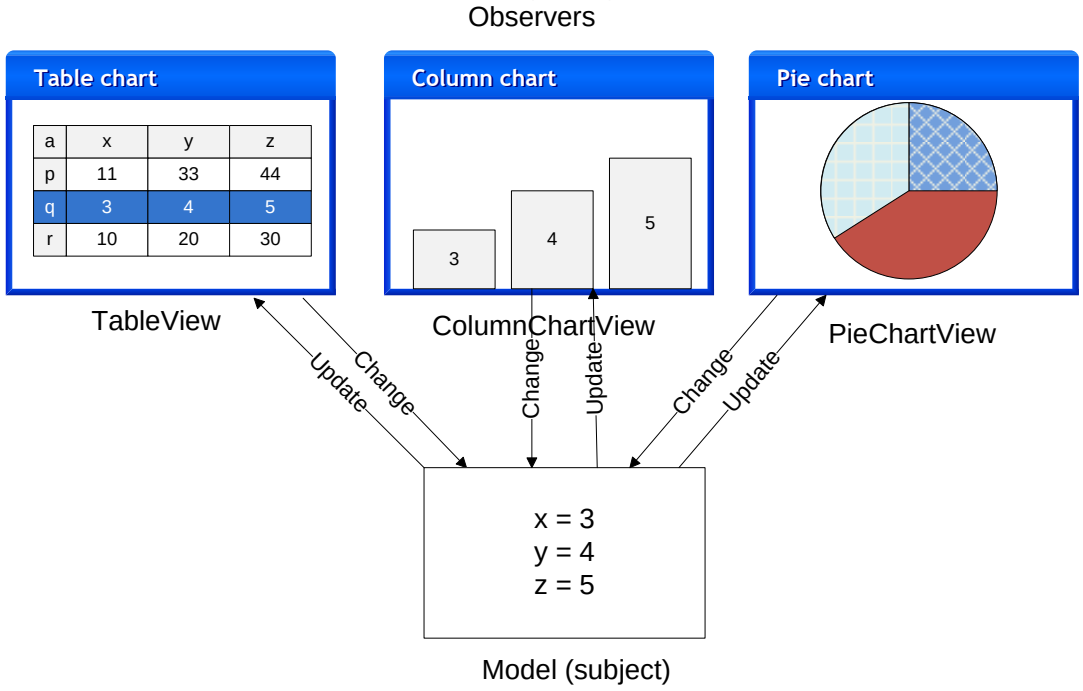
Observer

- The disadvantages of the direct method calls between the views
  - Dependency on the certain class. E.g. the ***TableView*** is dependent on the ***ColumnChartView*** and the ***PieChartView*** classes.
  - If we want to add a new view the existing ones have to be modified.
  - The model (business logic) is not reusable as it is bound to the presentation. It would be straightforward if the business logic wouldn't contain references to particular views as in this case it could be reused
  - It is difficult to maintain, reuse or develop as there is strong dependency between the classes.

Observer

Solution

- MVC or the Document-View architecture is a solution for the previous problem
- Separate the data from the presentation. Let us put the data into a model class and the different presentations to view classes.
- Multiple views (observers) can be registered to a model.
- If one of the views changes the model, the model will notify all the registered views about the change.
- As a response for the notification the view queries the actual state of the model and refreshes itself.
- The model just references the views through a general interface (observer)

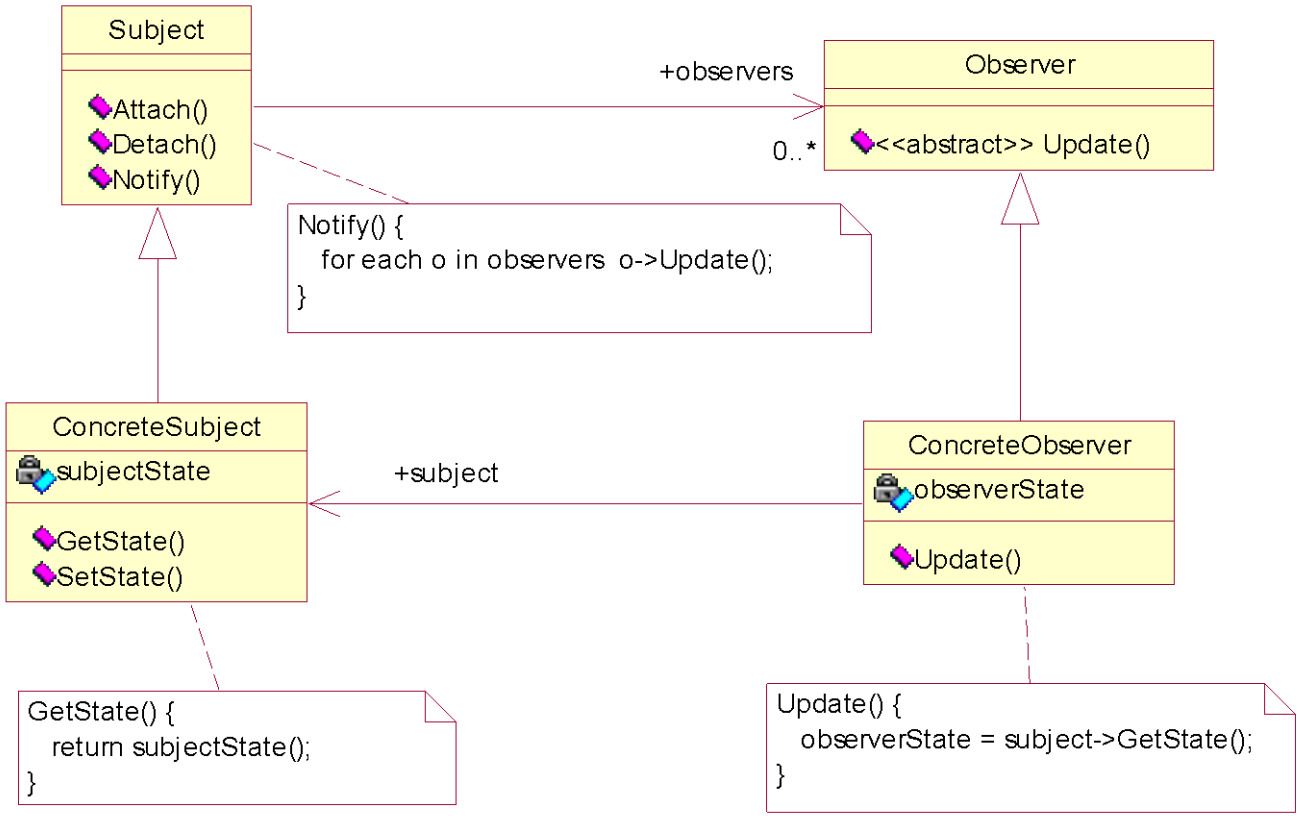


Observer

- **Advantages**
  - The model just uses ***IView*** references in the view list, so the model is independent from the concrete views. The model is reusable!
  - This is a simple mechanism to keep all the views consistent.
  - The system can easily be extended with new view classes. Neither the model nor the existing view classes need to be modified.

- **Generally, apart from the model-view case**
  - The above model facilitates the notification of some observer objects when the state of the observed object is changed without having any specific internal information about the observers.
  - This is the point of the *observer* pattern

Observer

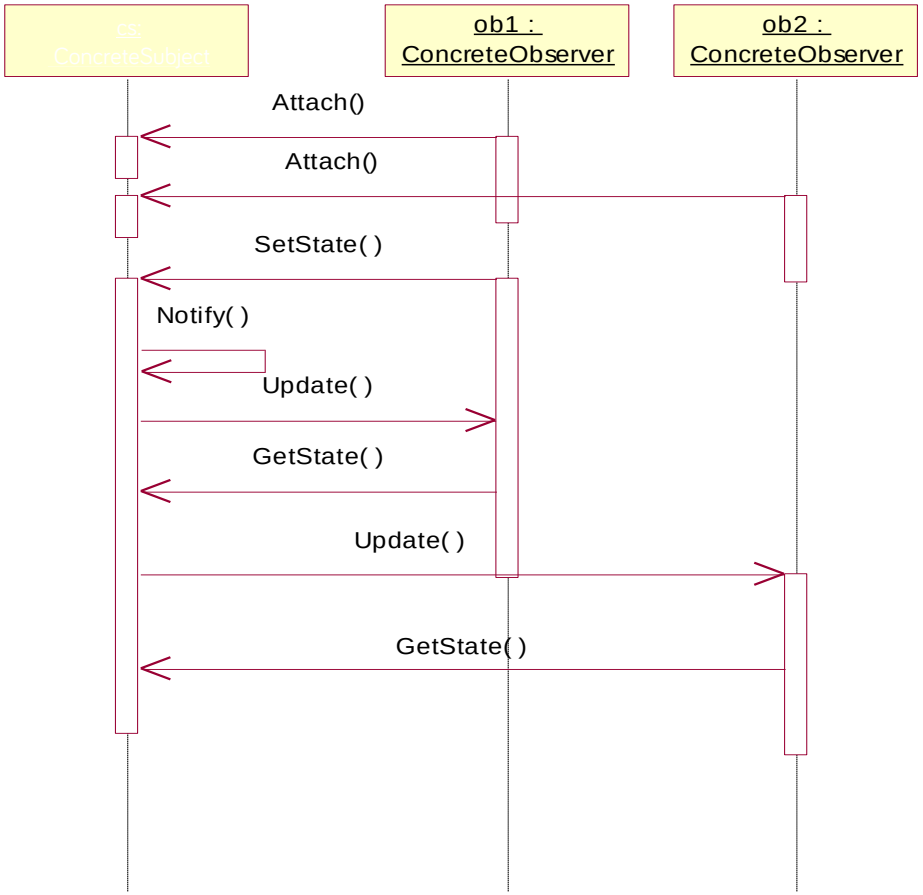

Subject
Attach()
Detach()
Notify()
+observers
0..*
Observer
<<abstract>> Update()
Notify() {
for each o in observers o->Update();
}
ConcreteSubject
subjectState
GetState()
SetState()
+subject
ConcreteObserver
observerState
Update()
GetState() {
return subjectState();
}
Update() {
observerState = subject->GetState();
}

Observer

- Participants
  - Subject
    - Stores the registered Observers
    - Defines an interface for registering, unregistering and notifying views.
  - Observer
    - Defines an interface for objects that want to be notified about the change of the Subject. (Update operation)
  - ConcreteSubject
    - Contains state information that are important for the observers.
    - Notifies all the registered observers when its state changes.
  - ConcreteObserver
    - Stores a reference for the observed ConcreteSubject object.
    - Stores a state information of the ConcreteSubject that needs to be kept in a consistent state with the real state.
    - Implements the Observer interface (Update operation). This is the method that the Subject calls when the state of the ConcreteSubject is changed. The state of the view is updated based on the actual state of the ConcreteSubject.

Observer

- **Dynamic view (registration and notification)**



Observer

- **Advantages**
  - Loose coupling between the Subject and the Observer
  - Supports broadcasting
- **Disadvantages**
  - Unnecessary updates
  - For an Update call all the state information is refreshed (though in the Update the Subject can specify the changed field(s)).
  - In the previous example the view that has initiated the Notify will also be updated (that is unnecessary)

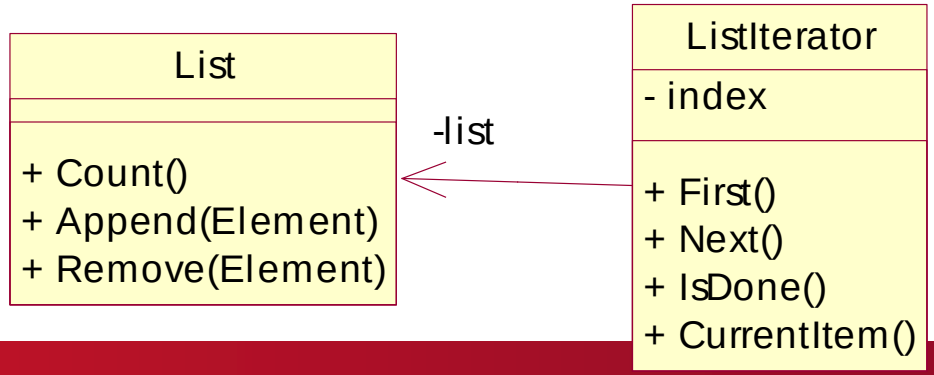
Observer

- Use it when
  - the change of the state of an object causes the change of the state of some other objects and the set of these objects is not known in advance.
- Note: the Observer is one of the most commonly used patterns

Iterator

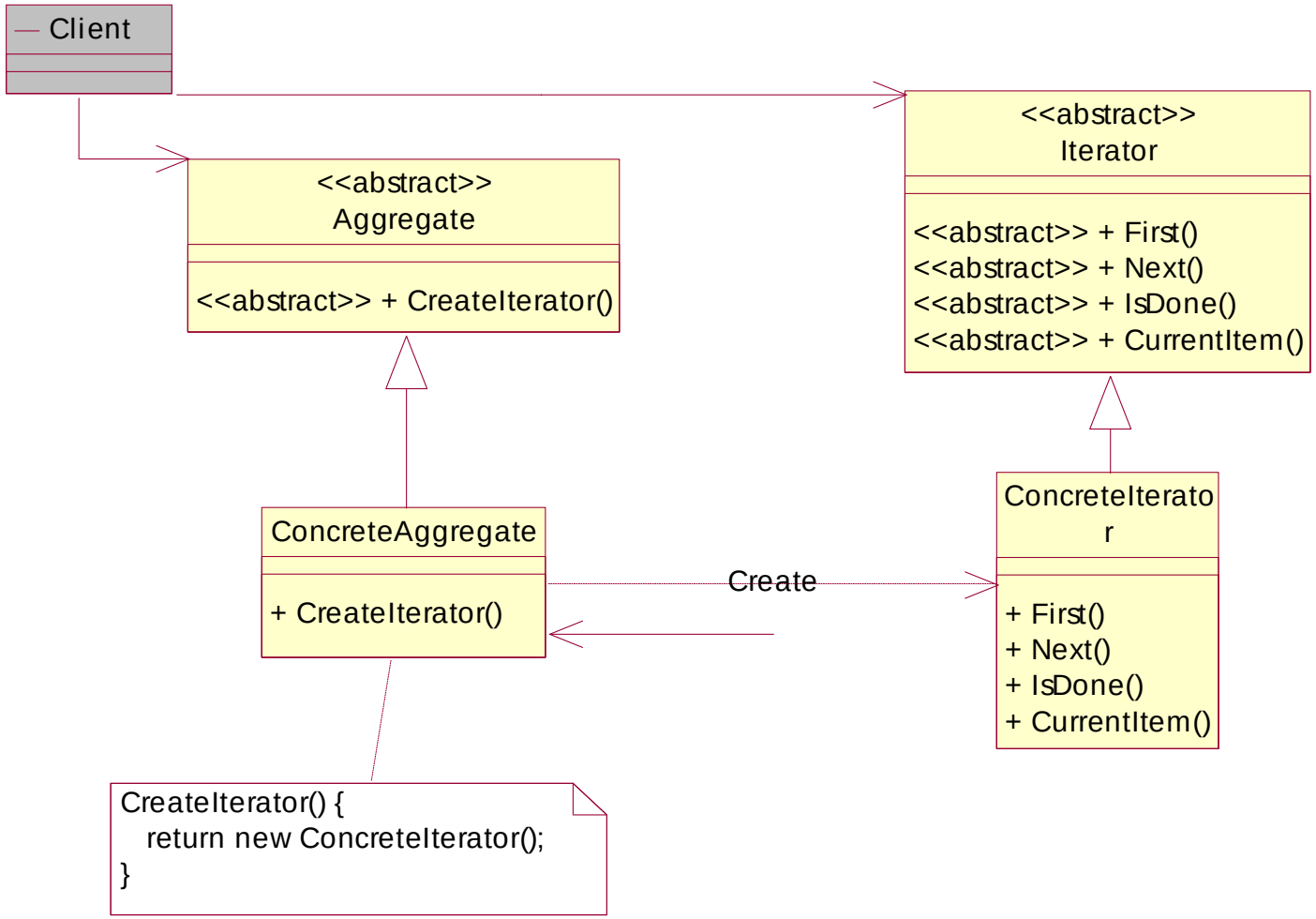
Iterator

- Purpose
  - Provides an interator for a complex (e.g. list) object without revealing the internal structure of the object.
- Example: List
  - We would like to iterate through the list without having any information about the internal structure of the list.
- Solution
  - Remove the operations of the iteration from the interface of the list and put it into a separate Iterator object.



Iterator

- Structure



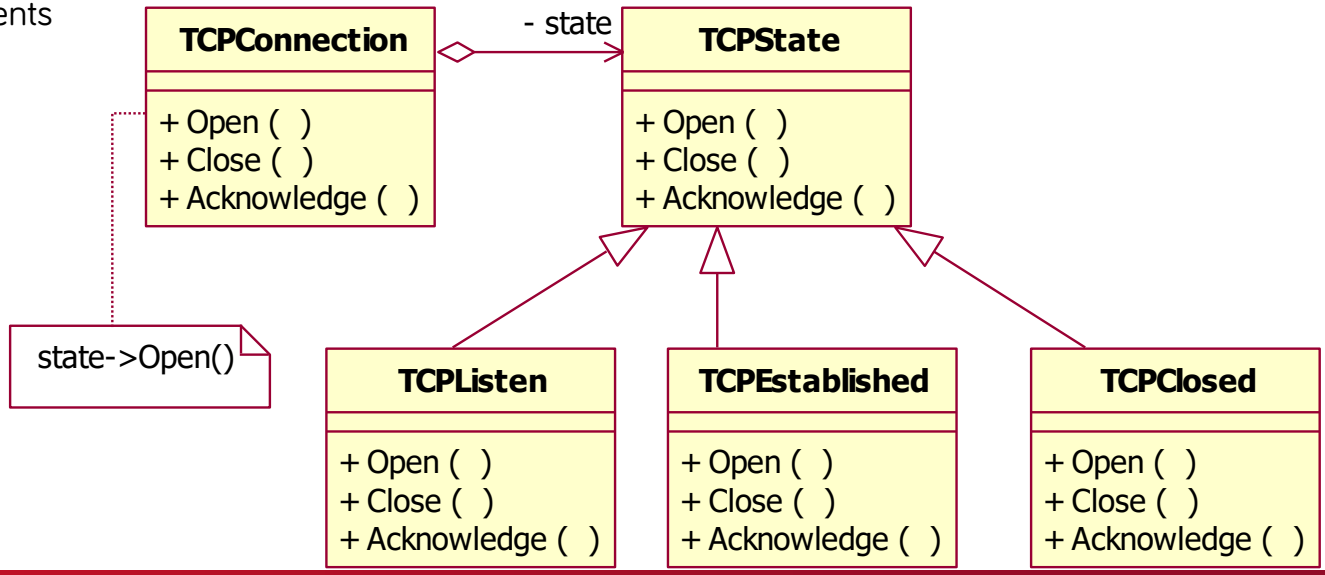
Iterator

- Use it when
  - We want to access the contained items of an object without revealing the internal structure of the containing object.
  - We want to allow access the items of the object in many ways.
  - We want to allow parallel accesses to the same list at the same time.
  - We want to access the items of multiple objects (e.g. lists) but we want to use a common (the same) interface.
- Implementation
  - In C++ it is worth using templates
  - Example: the STL contains many container and iterator classes.

State

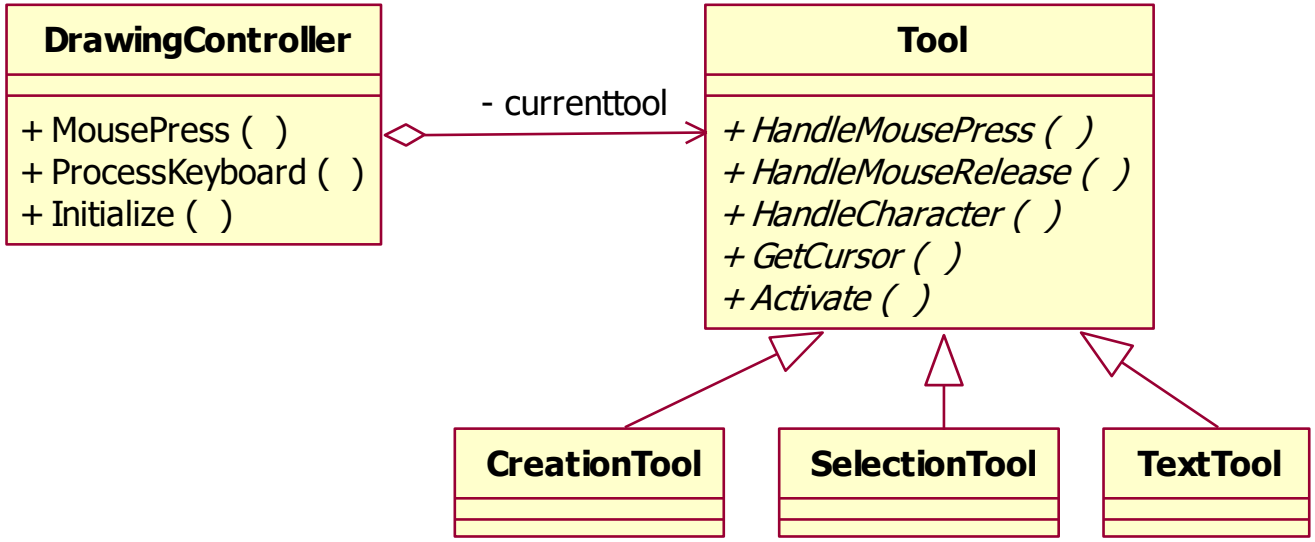
State

- Purpose
  - Facilitates the change of the behavior of an object when the state of the object is changed. The behavior of the object is state-dependent.
- Example 1
  - The TCPConnection class represents a network connection
  - There are three states: Listening, Established, Closed
  - The requests are served based on the actual state.
  - Solution
    - Using if statements
    - State pattern



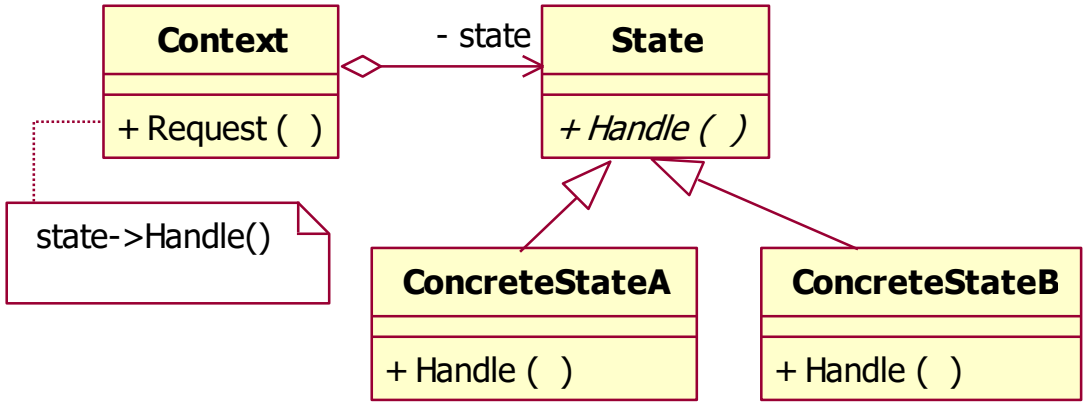
State

- Example 2
  - Handling the tools of a graphical application. (There is always one selected tool.)



State

- General solution



- Note
  - It is a usual solution that the Context passes itself to the operations of the State as parameter

```
// C++ example
void TCPConnection::Close ()
{
    state->Close(this);
}
```

  - The pattern does not specify who is responsible for the state management
    - Context
    - The State-derived classes. This is a distributed solution. Can be useful, but in this case the State-derived classes have to know each other.

State

- **Use it when**
  - The behavior of the object is dependent on its state, and the behavior has to be changed on the run-time as the state changes.
  - The operations would have huge conditional statements based on the state of the object.

- **Advantages:**
  - Encapsulates the state-dependent behavior of the object. It is easy to introduce a new state.
  - Nice code (no huge switch-case statements)
  - The State objects can be shared

- **Disadvantages:**
  - There will be more classes (use the pattern only if it is really needed)

References

- [1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: DESIGN PATTERNS, Elements of Reusable Object-oriented Software
- [2] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, Michael Stal: A SYSTEM OF PATTERNS, Pattern-Oriented Software Architecture
- Java design patterns:
  - http://www.patterndepot.com/put/8/JavaPatterns.htm
  - Pdf-ben letölthető!
- http://www.javacamp.org/designPattern/