Software Engineering

Lesson Design Pattern 08
Adapter, Facade, Bridge
v1.2

Uwe Gühl

Fall 2007/ 2008

Contents

- Adapter
- Facade
- Bridge

Used sources:

- [GHJ04] Gamma, Helm, Johnson, Vlissides: Design Pattern, Addison Wesley, 2004
- [Hus08] Vince Huston: Design Pattern, www.vincehuston.org/dp/, 2008

Adapter

- Intent:
  - Alternative name: Wrapper
  - converts an interface of a class
  - offers for an interface of a specified class another interface, so that it can be used by a client to enable collaboration
  - ... is a Structural Pattern

Adapter

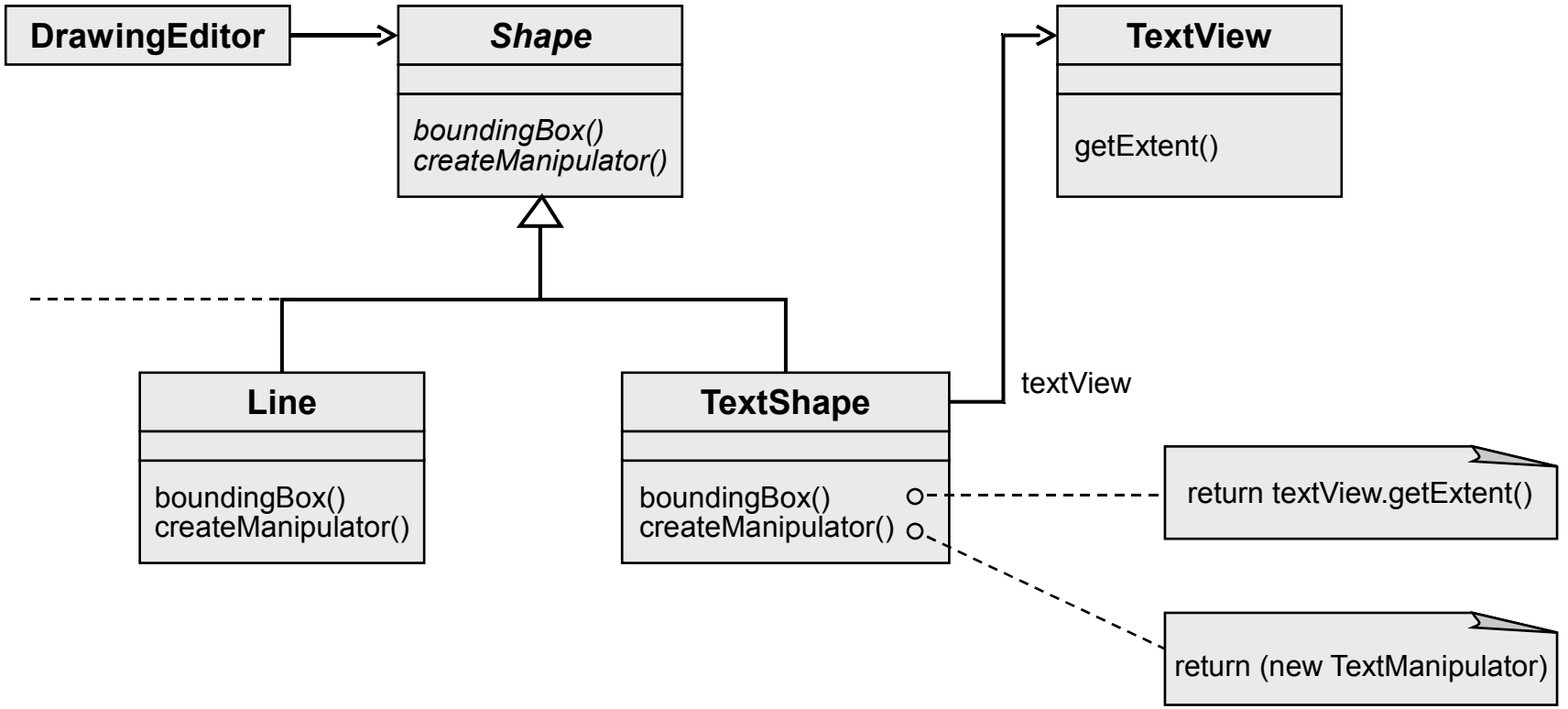
- Motivation
  - A program working with different objects and using their common interface, can be extended only by other objects implementing this common interface
  - An adapter makes it possible, that objects that don't implement a common interface could be used by a common interface – without changing the objects themselves

Adapter

- Ideas
  - The adapter implements the necessary interface and changes requests of a client to requests, which the object to be adapted could understand

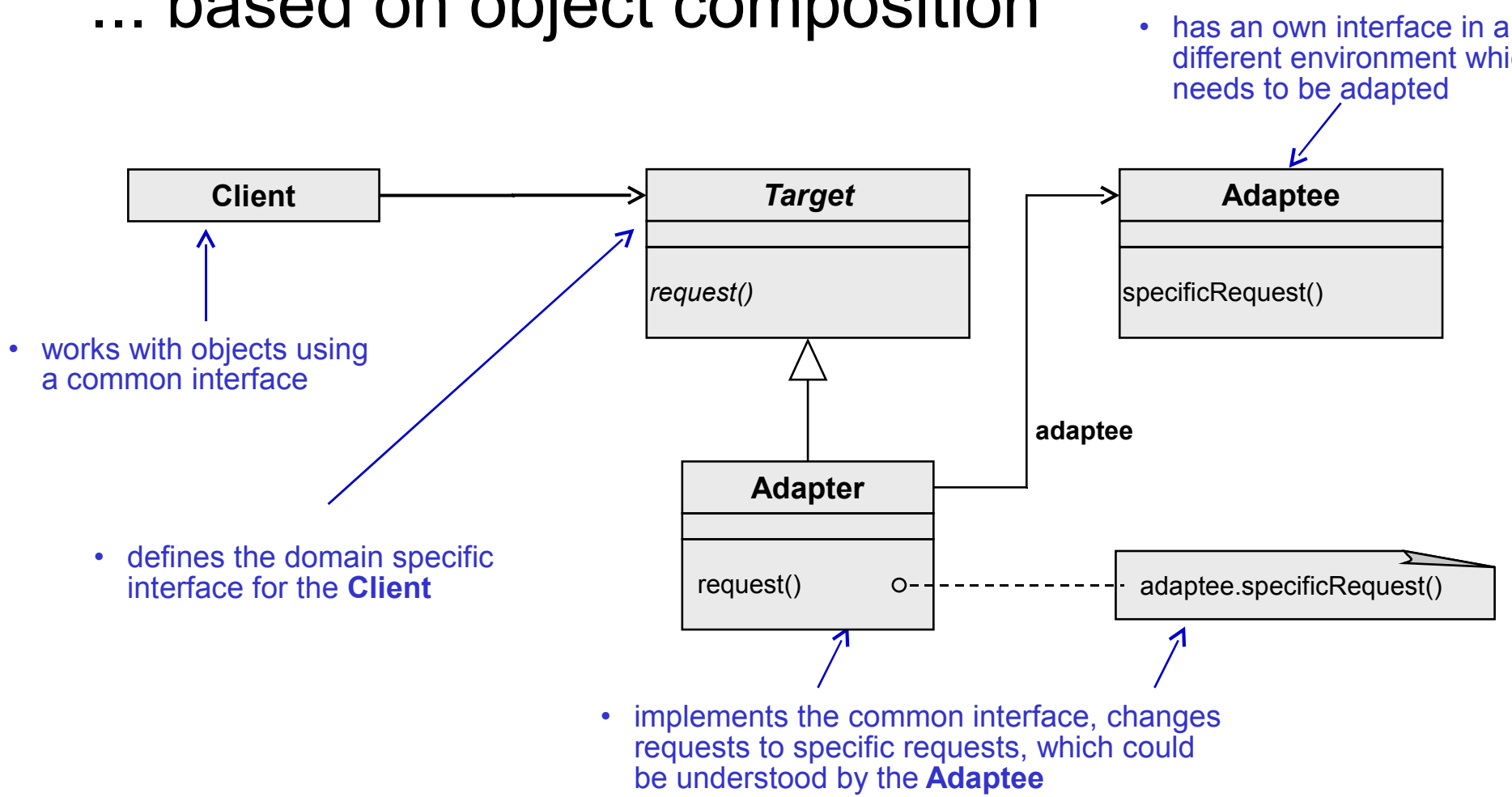
Adapter

- Example



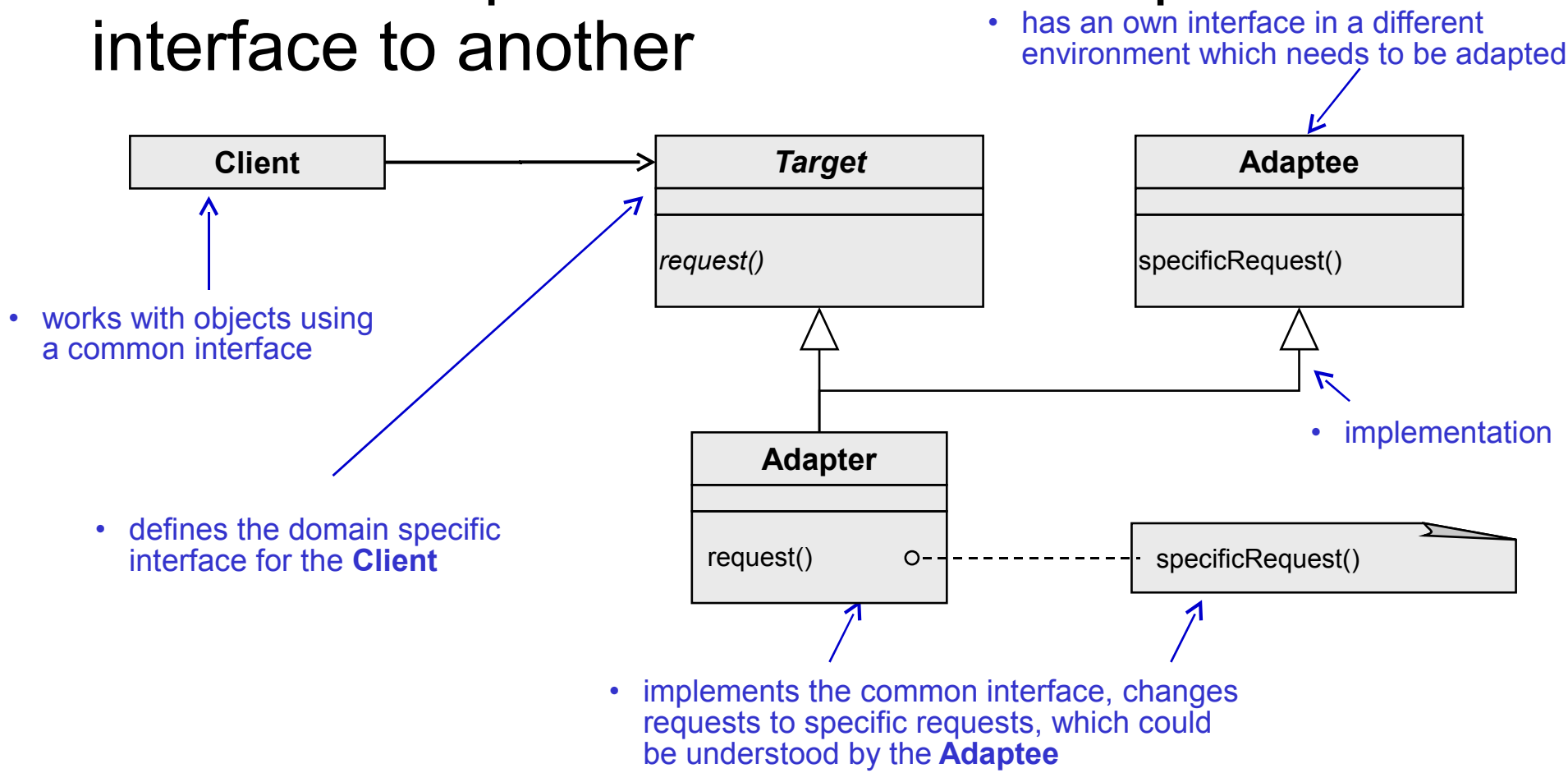
Adapter

- Structure – Object Adapter
  ... based on object composition



Adapter

- Structure – Class Adapter
  ... uses multiple inheritance to adapt an interface to another

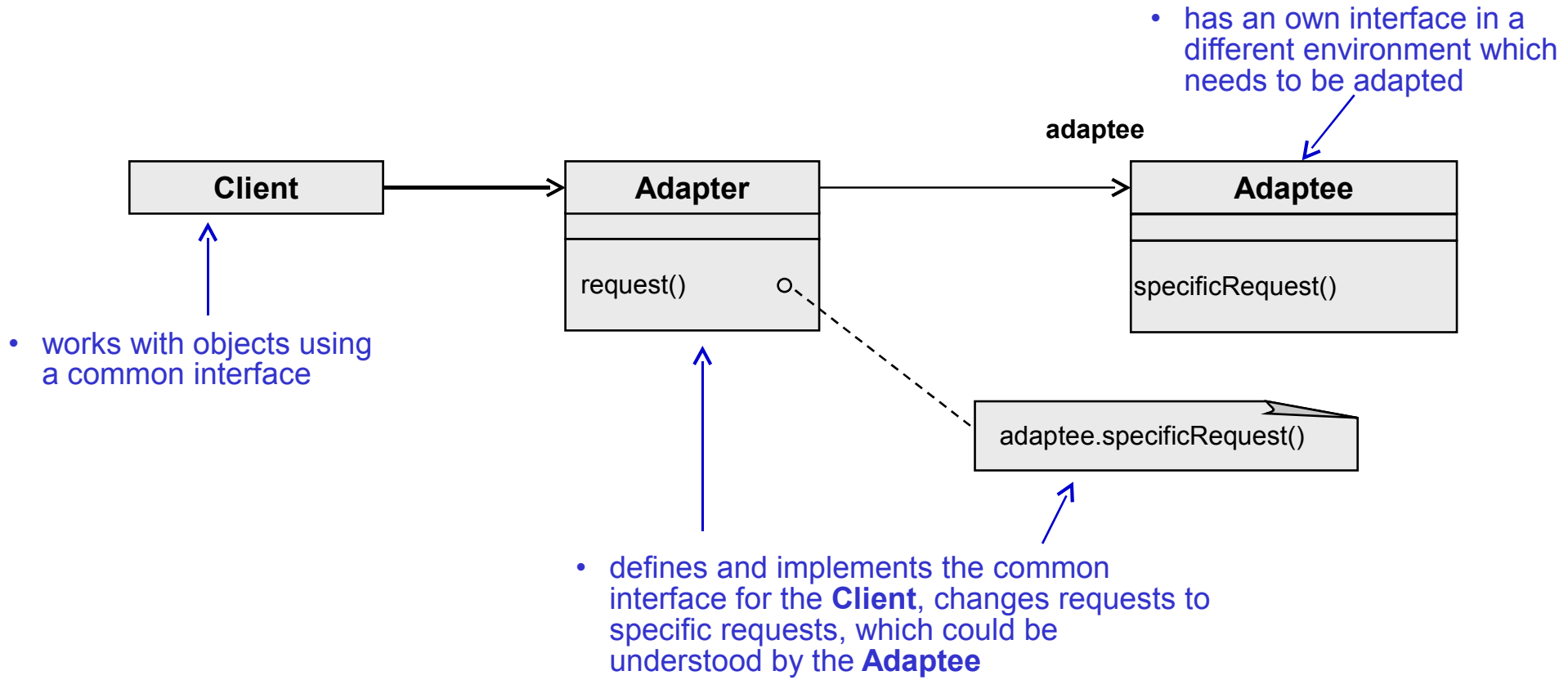


Adapter

- Structure –
  Difference between Object Adapter and Class Adapter
  - A class adapter inherits the implementation of the adaptee and delegates requests to the corresponding implementation
  - An object adapter has a relation to an object of the adaptee and delegates the requests to this object

Adapter

- Structure – simplified



Adapter

- Collaboration
  - Clients call operations on an Adapter instance
  - The Adapter call adaptee operations fulfilling the request

Adapter

- Applicability
  Use the Adapter Pattern if
  - an existing class should be used, but its interface does not match the one which is needed
  - a reusable class should be created that cooperates with other classes without compatible interfaces
  - several existing subclasses should be used without changing their interfaces by subclassing – an object adapter can adapt the interface of its parent class

Adapter

- Consequences – Class adapter

+ An adapter as a subclass of an adaptee can overwrite its behavior
+ Introduces only an object, no additional connection to the adaptee necessary
– A class adapter does not work, if we would like to adapt classes *and* subclasses

Adapter

- Consequences – Object adapter

+ An individual adapter could collaborate with multiple adaptees, also with corresponding subclasses

– Overwriting of behaviour of an adaptee is more difficult – a subclassing of the adaptee would be necessary and a reference

Adapter

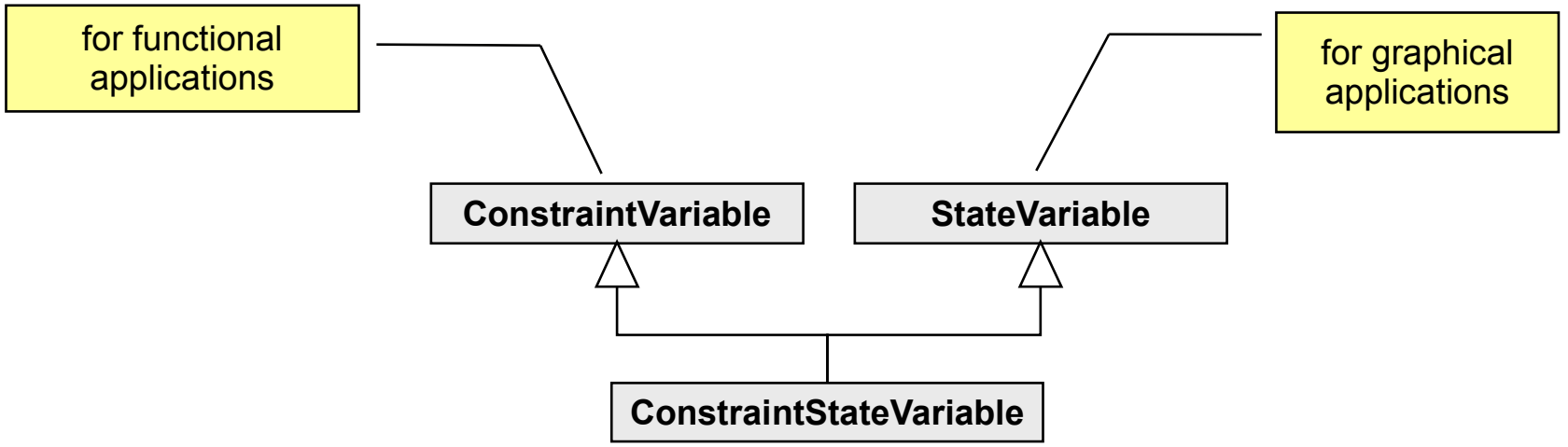
- Consequences

- Adjusting the adapter
  How similar is the target interface to adaptee's?
  Possible ranges:
  - Simple interface conversion
    e. g. changing names of operations
  - Supporting an entirely different set of operations
- Pluggable adapters
  to describe classes with built-in interface adaption

Adapter

- Consequences

+ Two way adapters
  In multiple inheritance different adapters for different purposes / different clients could be offered



Adapter

- Implementation
  - Implementing adapters in C++:
    - Class adapter – via *multiple inheritance*
      Adapter would be a subtype of Target and not of Adaptee because it would inherit
      - publicly from Target
      - privately from Adaptee
    - Object adapter – via *composition*
      no specific characteristics
  - Class adapter in Java
    Technique to use: Implementing the interface and extending another class

Adapter

- Implementation
  - Pluggable adapter
    - First step: Define the smallest subset of operations for the interface
    - Second – decide about implementation approach

      a) using abstract operations
      Subclasses must implement the abstract Adaptee interface

      b) Using delegate objects
      forwarding requests to a delegate object

      c) Parameterized adapter (Smalltalk)
      Here you use one or more “blocks” supporting adaption without subclassing

Adapter

- Known Uses (see [GHJ+95])
  - ET++Draw,
  - InterViews 2.6,
  - NEXT AppKit,
  - Smalltalk 80 ValueModel Hierarchy
  - Event handling in Java-AWT (ActionListener)

Adapter

- Related Patterns
  - Object adapter and Bridge use the same structure to solve different problems
    - Bridge separates interfaces from its implementation so that they can vary easily and independently
    - An Adapter has the idea to change an interface of an existing object

Adapter

- Related Patterns
  - Decorator enhances another object without changing the interface – it is more transparent to an application than an adapter. Decorator could be used recursively what is not possible with adapters
  - Proxy defines a surrogate for another object and does not change its interface
  - Different pointing – the client object interacts with
    - the top-level interface – in case of Adapter and Proxy
    - the “decorator” bottom classes – in case of Decorator

Facade

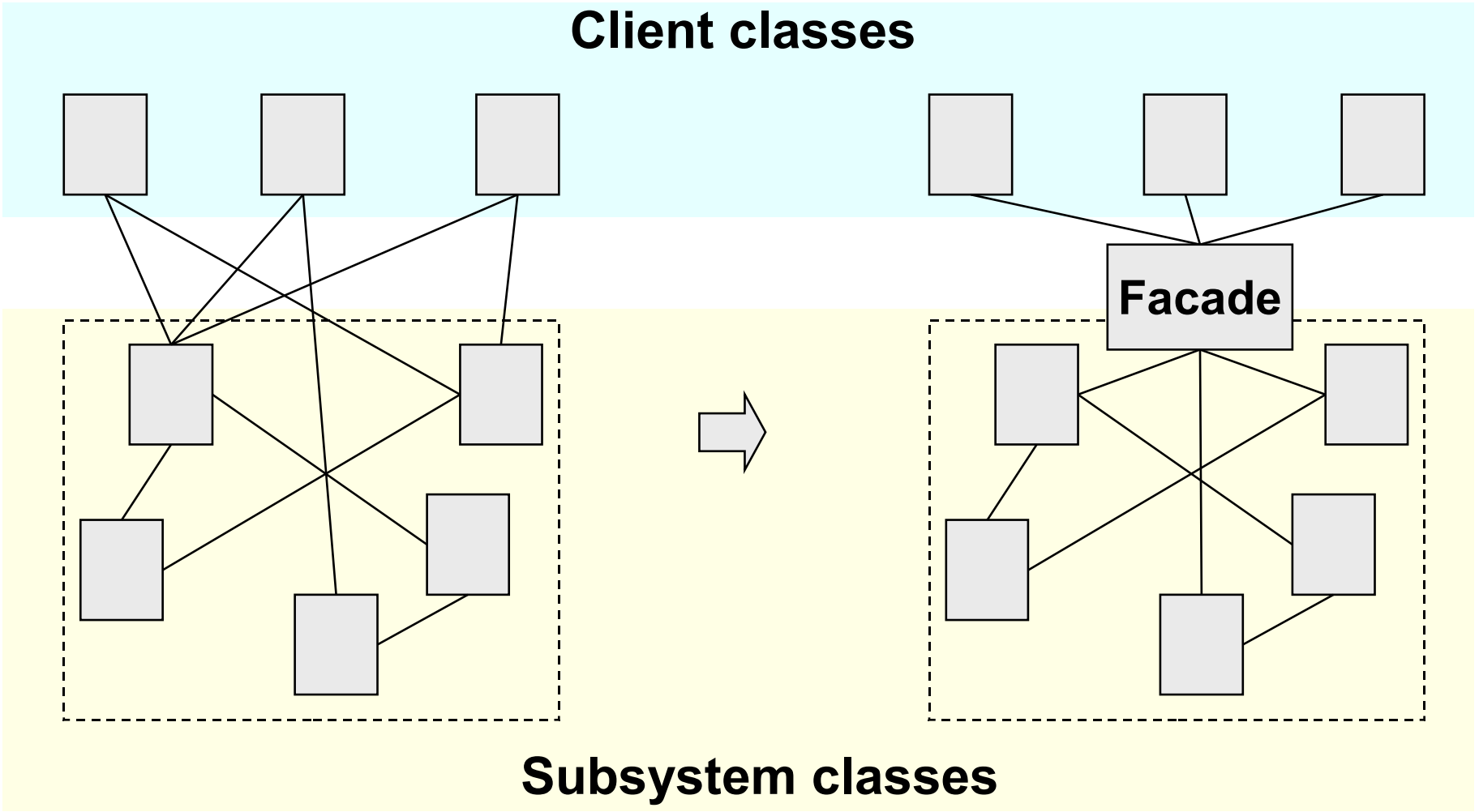
- Intent
  - A Facade provides a unified interface for a component, that means it sums up many interfaces of a subsystem to one interface
  - ... is a Structural Pattern

Facade

- Motivation
  - Structuring a system into subsystems or components reduces complexity
  - General design goal:
    - Reduction of communication and dependencies between subsystems or components
  - Offer of a simplified interface → the Facade

Facade

- Motivation



Facade

- Idea
  - A class offers a uniform interface for all classes of a subsystem / component and delegates requests to corresponding classes
  - This class is named Facade

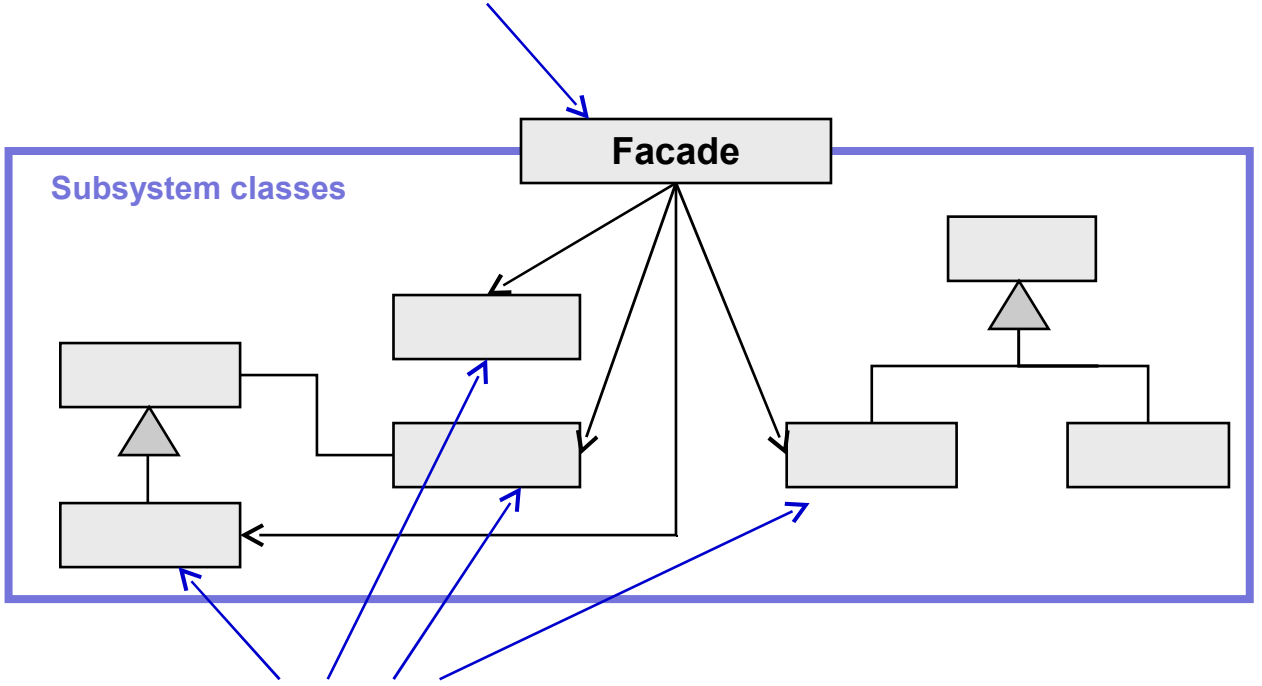
Facade

- Applicability
  - For a complex subsystem a simple interface should be offered, which is sufficient for most clients
  - Many dependencies between clients and application classes
  - Many tiers of a system are planned
    → a facade could be entry point of each tier

Facade

- Structure

- knows, which subsystem classes are responsible for which request
- delegates requests to the corresponding subsystem objects or classes



- implement the functionality of the subsystem
- process the requests delegated by the **Facade** object
- don't know the **Facade** object.

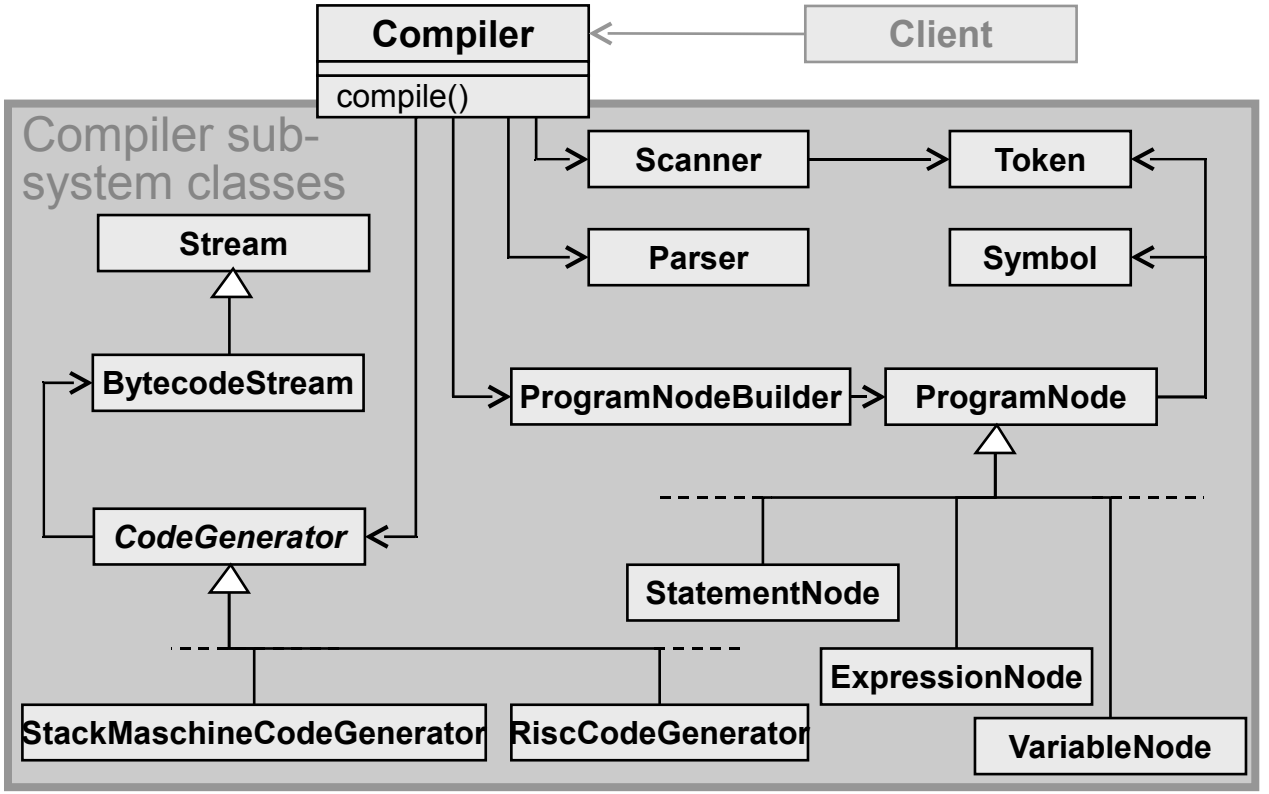
Facade

- Collaboration
  - Clients communicate with the subsystem by sending requests to the Facade.
    Facade forwards the requests to the corresponding objects, maybe additionally has to translate its interface to the subsystem interfaces
  - Clients using the facade don't access subsystem / component objects directly

Facade

- Example

```
public BytecodeStream compile(
    CharArrayReader input)
{
  BytecodeStream output;
  Scanner scanner =
    new Scanner(input);
  ProgramNodeBuilder builder;
  Parser parser;
  parser.parse(scanner, builder);
  RISCCodeGenerator generator =
    new RISCCodeGenerator(output);
  ProgramNode parseTree =
    builder.getRootNode();
  parseTree.traverse(generator);
  return output;
}
```



Facade

- Consequences

+ The Facade pattern offers a simple programming interface: Clients could use subsystems or components easier, as less objects and classes must be known

Facade

- Consequences

+ Facade allows loose coupling between subsystems and its clients
  - Components of a subsystem could be changed without consequences on the client
  - Systems could be divided in layers
  - Complex dependencies could be reduced and allow independent development of subsystems
  - Lower compiling dependencies increase the portability

Facade

- Consequences

+ Despite the use of the Facade Pattern the access to the complex classes is still established – the direct use of the classes of a subsystem is still possible
+ It's possible to do changes in the subsystems without changing client code. This reduces dependencies in compiling

Facade

- Implementation
  - Reduction of the client subsystem coupling
    - Possible with different concrete subclasses of an abstract Facade class
    - Subclasses of this Facade are standing for different implementations of a subsystem
    - Clients communicate with the subsystem using the interface of the abstract class and don't notice the concrete implementation

Facade

- Implementation
  - Public versus private subsystem classes
    - A subsystem could offer private and public interfaces
    - The Facade would be part of the public interface – it describes all classes clients could access.
    - In a “name space” (package in Java) private interfaces could be supported, accessible only inside a “name space”

Facade

- Known Uses
  - Object-Works (OW) Smalltalk Compiler System
  - ET++,
  - Choices Operating System
  - Java Data Base Connectivity (JDBC) in java.sql package
  - java.net.URL
  - java.util.concurrent.Executors

Facade

- Related Patterns
  - Abstract Factory can be used with facade to offer an interface for creating subsystem objects
  - Mediator has similar ideas as it abstracts functionality of existing classes – but Mediator focuses on abstraction of communication and centralizes functionality
  - If only one Facade object is required it could be a Singleton

Facade

- Discussion
  - Is Facade really a Design Pattern?
  - There is not a typical class diagram structure of it!
  - But of course it's needed, because in using Design Patterns, you end up with flexibility, and lots of classes
  - The Facade pattern is meant to make things more manageable.

inspired by Heinz M. Kabutz, http://www.javaspecialists.eu/archive/Issue112.html

Bridge

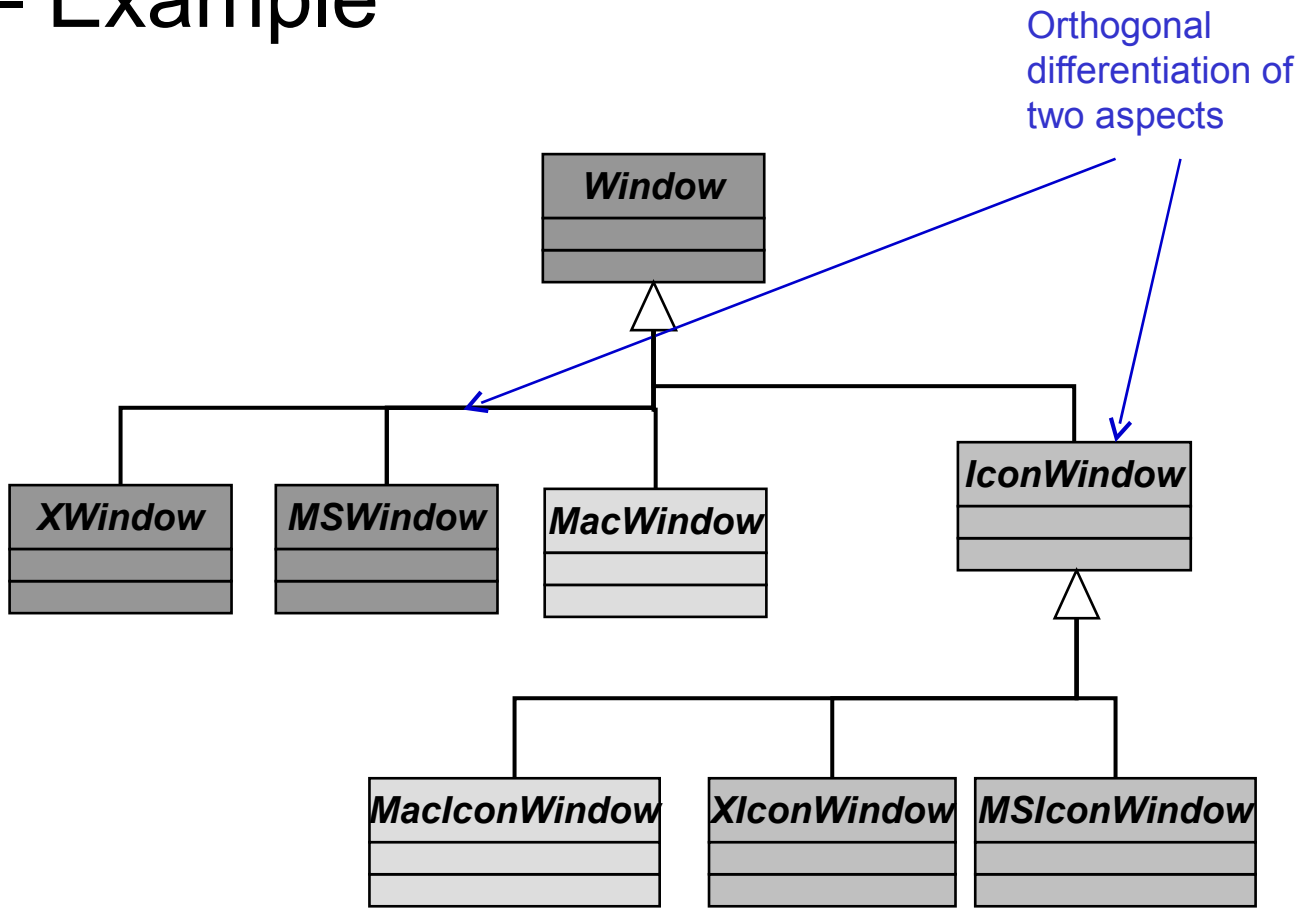
- Intent:
  - Decouple an abstraction from it's implementation so that the two can vary independently.
  - Publish interface in an inheritance hierarchy, and bury implementation in its own inheritance hierarchy
  - ... also known as „Handle / Body“
  - ... is a Structural Pattern

Bridge

- Motivation
  - An Abstraction could have many implementations (Example: Graphical User Interfaces)
  - Inheritance has some disadvantages
    - less flexible
      - if implementation has to be changed during runtime
      - if there is a big number of abstract classes and corresponding implementations
    - dependencies in the hierarchy make modifications and extensions difficult

Bridge

- Motivation – Example



Bridge

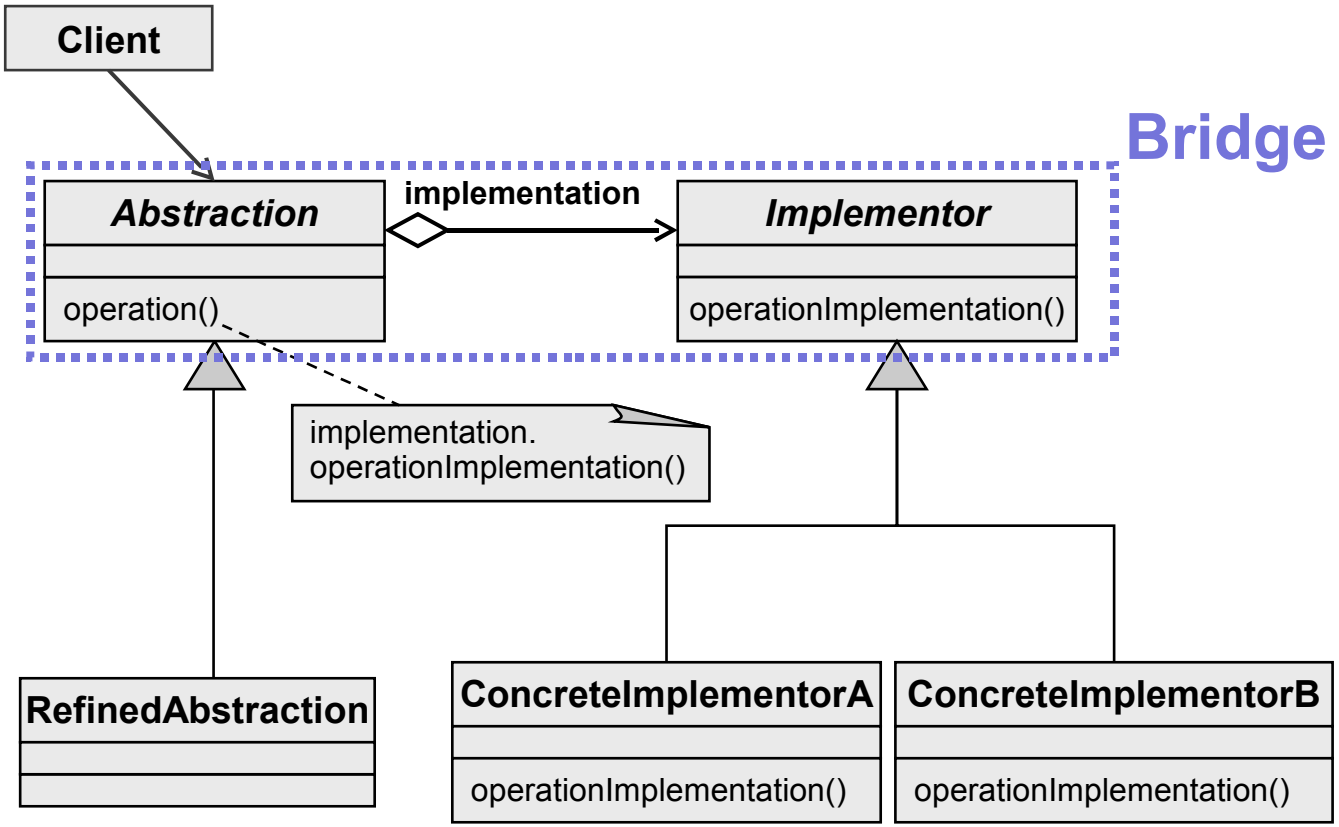
- Idea
  - The abstract classes (meaning the interface) and the implementation classes get concentrated in separate class hierarchies
  - The abstract classes delegate the tasks to the corresponding implementation classes.

Bridge

- Applicability
  - Abstraction should not be coupled with an implementation to close (change at runtime)
  - Abstraction and implementation should be extendable with subclasses
  - If you extend the system and the number of classes is growing above average – this is a hint that objects should be separated
  - Several objects should share an implementation, but the client should not notice

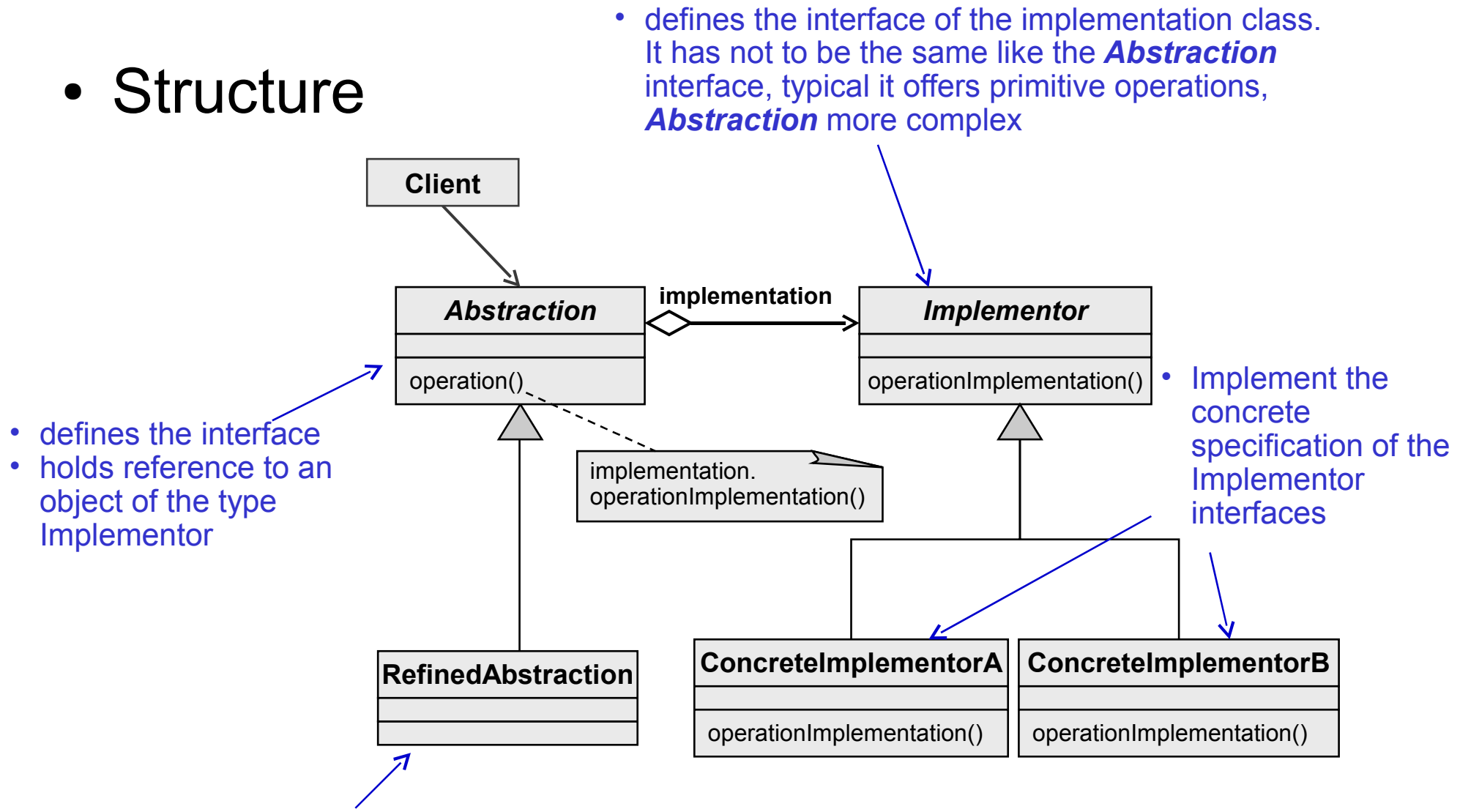
Bridge

- Structure



Bridge

- Structure



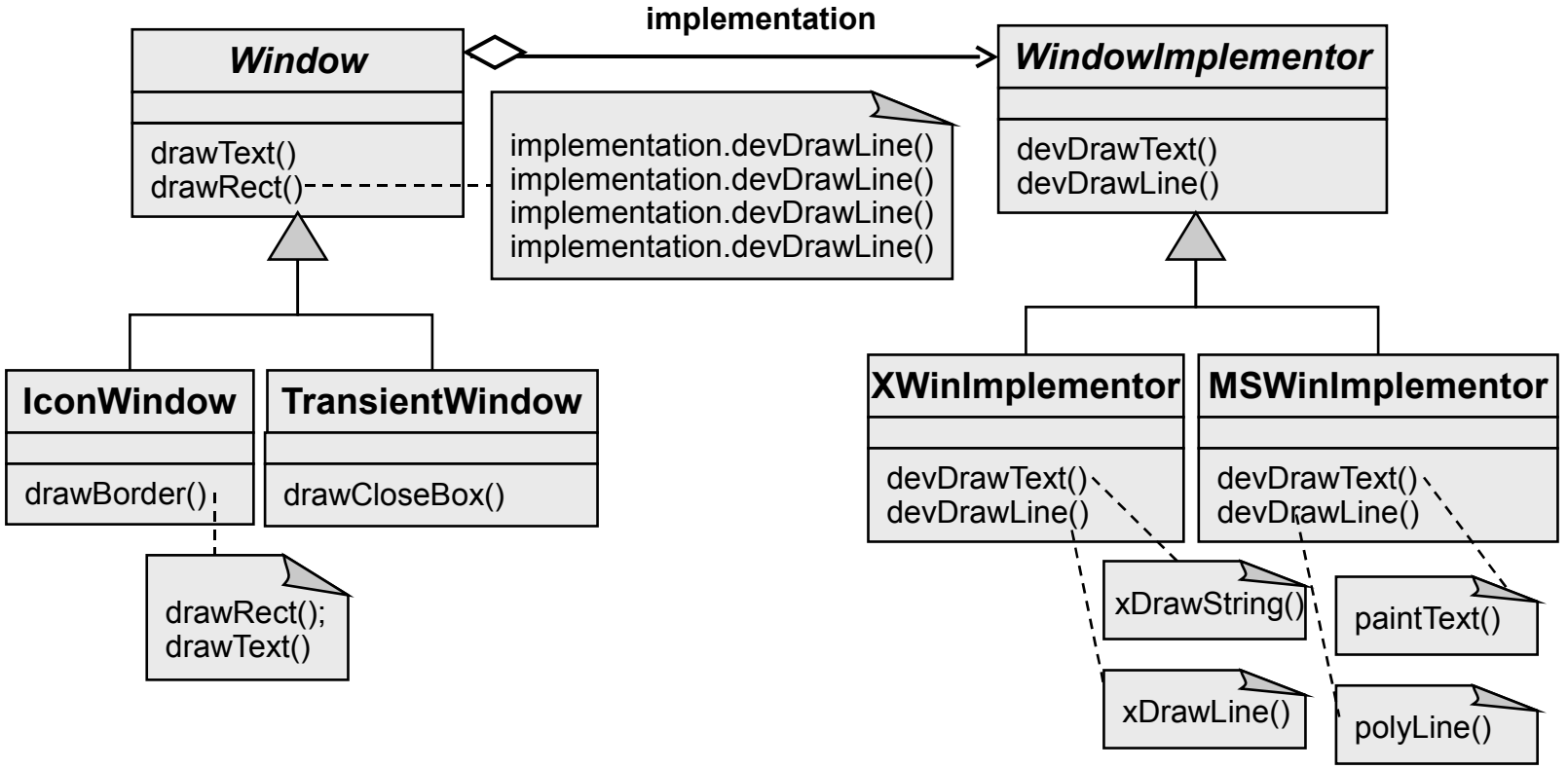
- defines the interface of the implementation class. It has not to be the same like the ***Abstraction*** interface, typical it offers primitive operations, ***Abstraction*** more complex
- defines the interface
- holds reference to an object of the type Implementor
- Implement the concrete specification of the Implementor interfaces
- extends behavior of ***Abstraction***

Bridge

- Collaboration
  - Abstraction forwards client requests to its Implementor object

Bridge

- Example



The **Abstraction classes** implement the methods of different window types in using the methods, made available by the **Implementation classes**

Bridge

- Consequences

+ Decoupling of interface from implementation
  - Interface to the client is stable
  - Implementation could be exchanged at runtime
+ Extensibility
  - The class hierarchies of Abstraction and Implementor could be extended independently
+ Implementation details could be hided more easily from the client

Bridge

- Implementation
  - Only one concrete Implementor
    An abstract Implementor is not necessary if there is only one implementation
  - Creation of the right Implementor object
    Who decides? And how?
    - Parameter
    - Default values
    - Delegation

Bridge

- Known Uses
  - ET++ (Windows)
  - libg++ (Sets)
  - NeXT AppKit (Images)
  - VisualAgeSmalltalk (Collections)
    At runtime the collection could be changed
  - Java AWT (Peer Interface)
    Widgets are created dependently of the operating system, that is different view on different systems

Bridge

- Related Patterns
  - Abstract Factory
    An Abstract Factory can create and configure a particular Bridge
  - Object adapter and Bridge use the same structure to solve different problems
    - Bridge separates interfaces from its implementation so that they can vary easily and independently
    - An Adapter has the idea to change an interface of an existing object