

Facade and Adapter

Comp-303 : Programming Techniques Lecture 19

Alexandre Denault
Computer Science
McGill University
Winter 2004

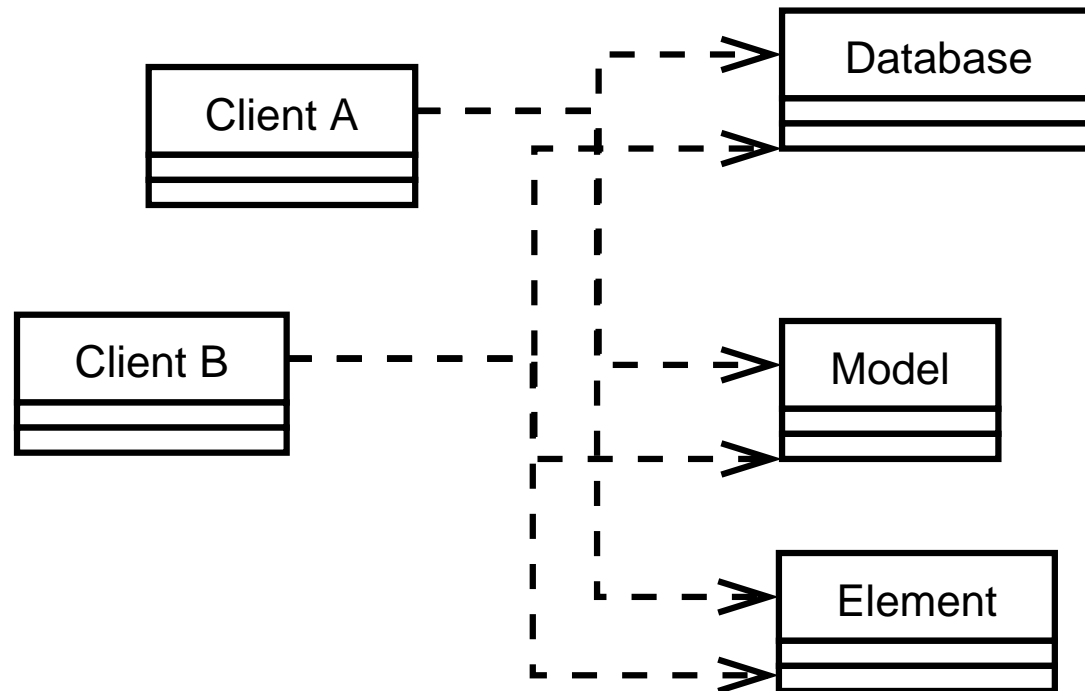
Last lecture . . .

- Abstract Factories allow you to use families of objects in a generic way.
- Factory methods allow you to choose the appropriate subclass at runtime.

Typical Situation

- I'm building an inventory application with a database.
- The *Client* class uses the *Database*, *Models* and *Elements* classes.
- First, a *Client* object must open a connection to the *Database*.
- From the *Database*, it gets a *Model*.
- It then uses a *Model* to get an *Element*.
- Finally, it retrieves the much needed information from *Element*.

Too many relations!



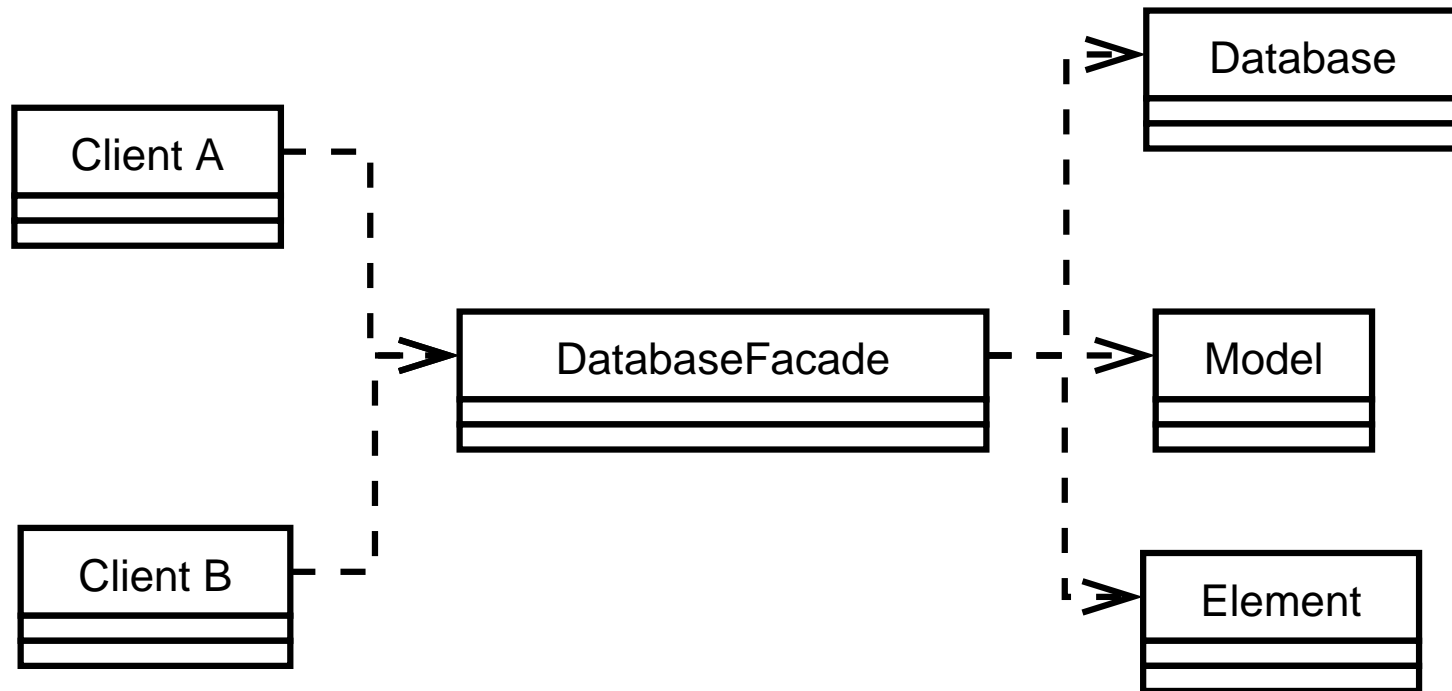
Facade

- *Pattern Name* : Facade
- *Classification* : Structural
- *Intent* : Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Motivation

- In our example, *Client* doesn't really need to know about *Database*, *Model* or *Element*.
- It only needs to retrieve information from *Element*.
- Thus, we can build a Facade.

Motivation (cont.)

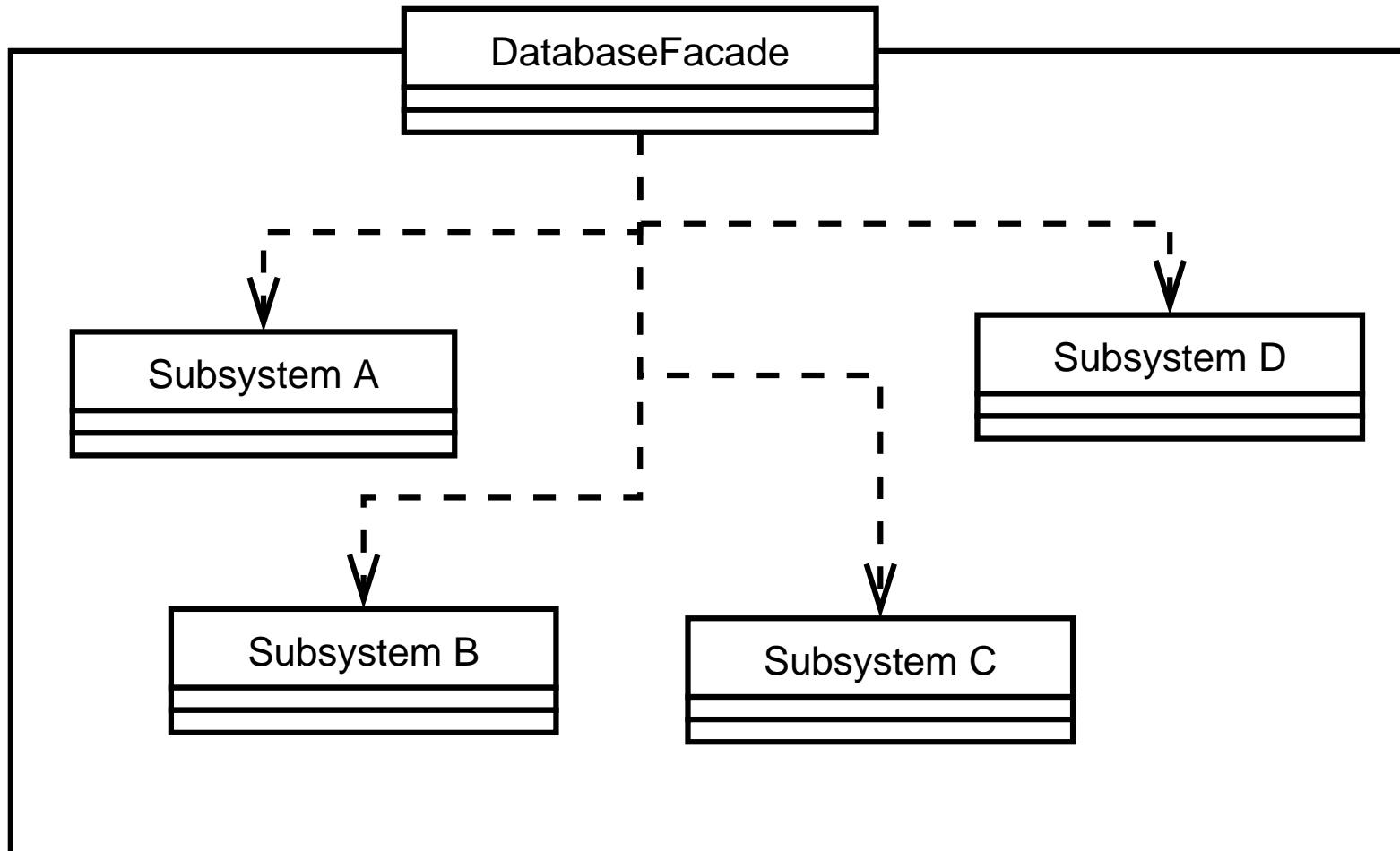


Applicability

The Facade pattern should be used when ...

- ... you want to provide a simple interface to a complex subsystem.
- ... there are many dependencies between clients and the implementation classes of an abstraction.
- ... you want to layer your subsystems.

Structure



Participants

- Facade :
 - knows which subsystem classes are responsible for a request.
 - delegates client requests to appropriate subsystem objects.
- Subsystem :
 - implement subsystem functionality.
 - handle work assigned by the Facade object.
 - have no knowledge of the Facade (no reference to it).

Collaborations

- Clients communicate with the subsystem by sending the requests to Facade.
- Facade forwards the request to the appropriate subsystem.
- Though Facade doesn't do any the of the work, it might need to translate the request.
- Clients that use Facade have no knowledge of the subsystem.

Consequences

- It shields client from the many objects of the subsystem component. Thus, the subsystem is easier to use.
- It promotes weak coupling between the client and the subsystem.
- It doesn't prevent the client from using the subsystem.

Implementation

- Facade by itself is pretty simple to implement.
- You must decide if the subsystem should be made of private or public classes. This depends on if you want the client to be able to use the subsystem directly.
- You can make the Facade an abstract class. Thus, you can subclass it and provide different Facades to different subsystem.

Known Uses

- If a Facade is well built, the user shouldn't even know it's a Facade.
- Some subsystem in Java are good candidates for Facades (JDBC).

Related Patterns

- Abstract Factory
- Singleton
- Mediator

Something different

- The Facade is useful, but it doesn't help you when you need polymorphic behavior.
- For example, let's think of a drawing program like PaintBrush.
- Implementing objects to draw basic primitives like *Line* and *Circle* is not too complicated.
- We can make those subclasses of an abstract class name *Shape*.
- However, drawing text is a little more complicated.
- Many external libraries are available to draw text.
- However, how do we preserve our object hierarchy with *Shape*?

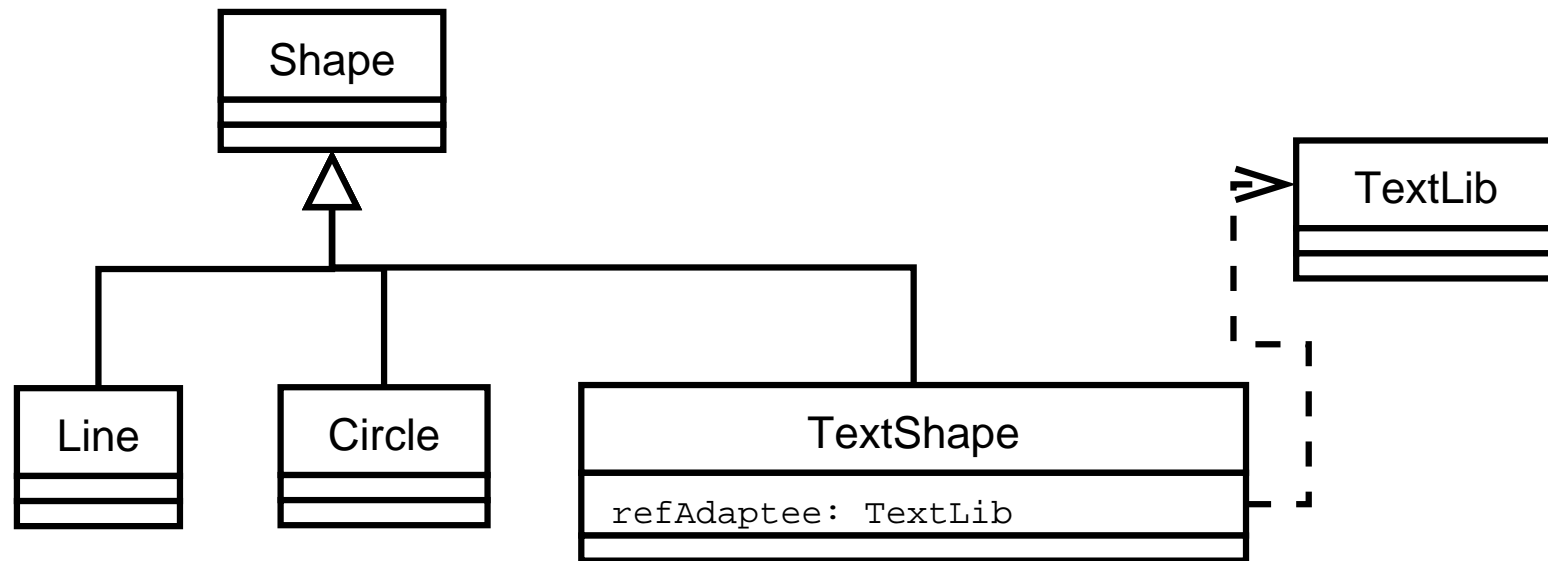
Adapter

- *Pattern Name* : Adapter
- *Classification* : Structural
- *Also known as* : Wrapper
- *Intent* : Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Motivation

- We can create a class named *TextShape* which extends shape.
- This classes will use the external libraries to draw text.
- It can also provide additional functionalities not found in the external libraries .

Motivation (cont.)

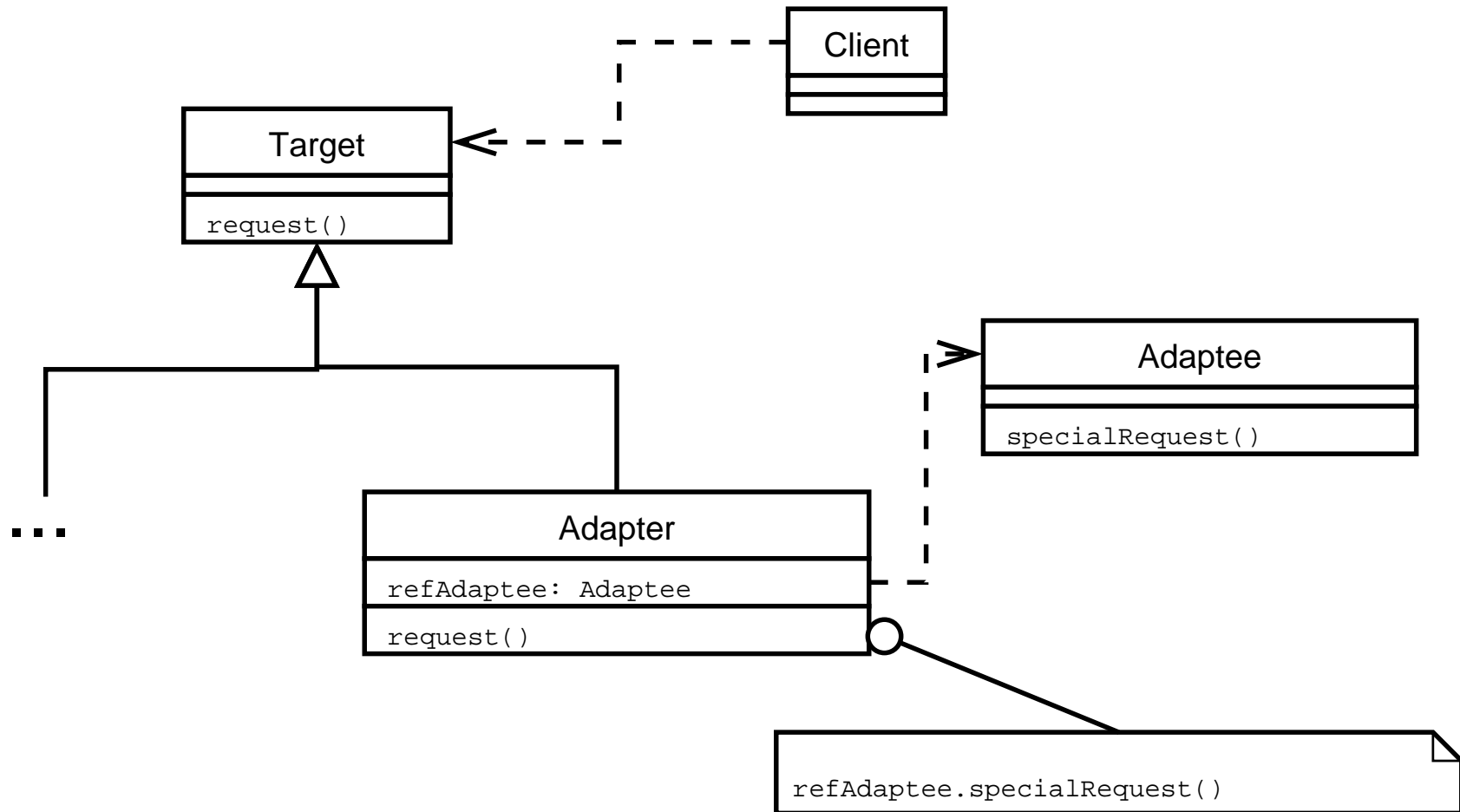


Applicability

The Adapter pattern should be used when ...

- ...you want to use an existing class, and its interface does not match the one you need.
- ...you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
- ...you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one.

Structure



Participants

- Target : defines the domain-specific interface that Client uses.
- Client : collaborates with objects conforming to the Target interface.
- Adaptee : defines an existing interface that needs adapting.
- Adapter : adapts the interfaces of Adaptee to the Target interface.

Collaborations

- Clients sends a request to the Adapter.
- In turn, the Adapter sends that request (maybe after transforming it) to the Adaptee.

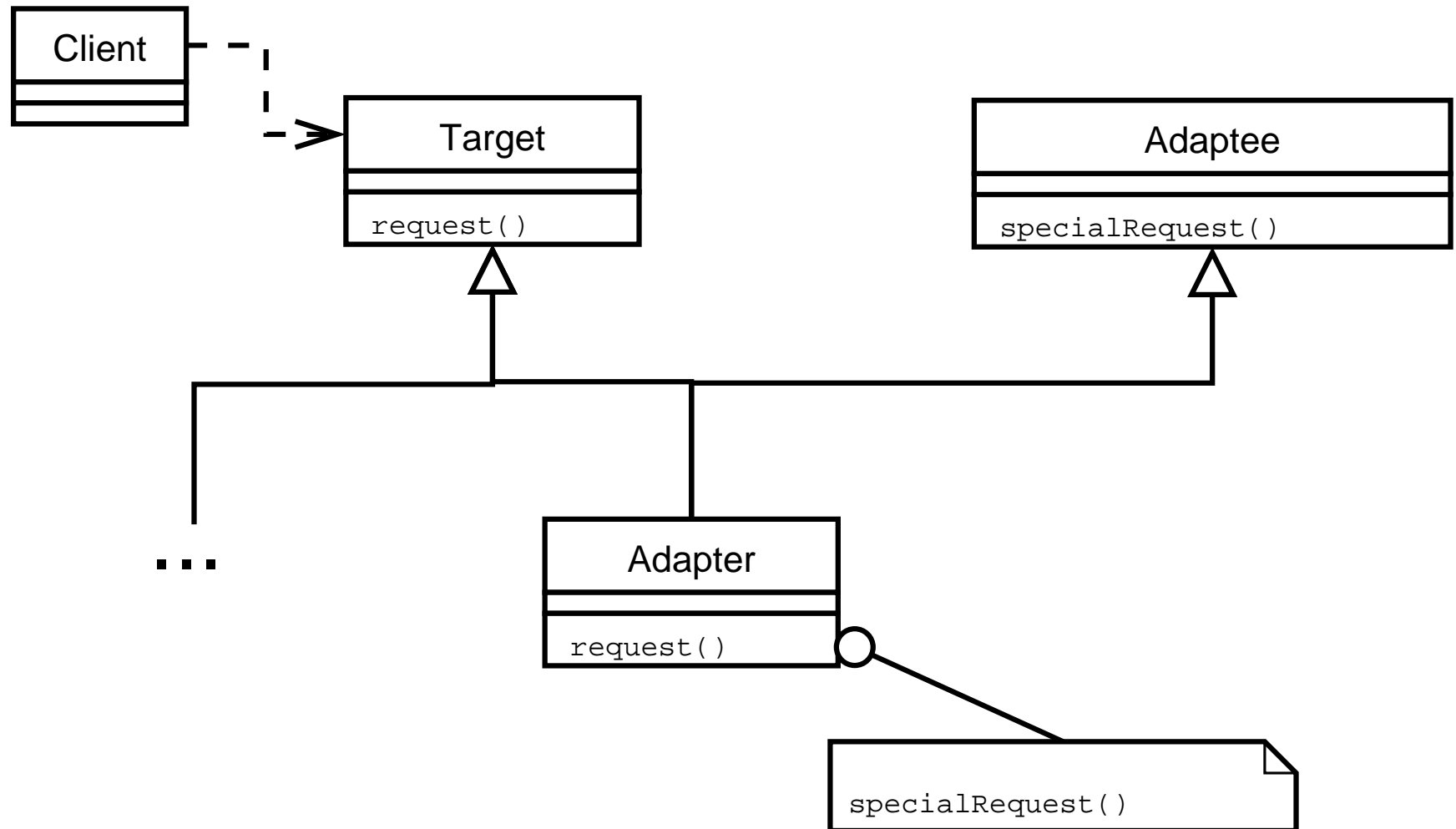
Consequences

- Allows you to use the Adaptee (with the help of the Adapter) as if it were part of your hierarchy.

Implementation

- The implementation of an adaptor is fairly straightforward.
- A single Adapter can work with many Adaptees.
- Another variation of the Adapter pattern is possible when you have multiple inheritance.

Implementation (cont.)



Known Uses

- The Java API already has a great number of Adapters built-in.
- One the most commonly used it the *WindowAdapter* class.
- Like Facade, when an Adapter is well built, the user shouldn't even know he's using an Adapter.

Related Patterns

- Bridge
- Decorator
- Proxy

Facade vs Adapter

- Both design patterns are very similar.
- They are often used as wrappers for more complex functions.
- However, Adapter is used when our wrapper must respect a particular interface and must support a polymorphic behavior.
- On the other hand, Facade is used when we want a easier/simpler interface to work with.

Summary

- Facade allows us to hide a complex subsystem, thus reduce coupling and make that subsystem easier to use.
- Adapter allows us to change the interface of a class, thus allowing us to that class in our hieratic (polymorphic behavior).

Tool of the day: SableVM

- SableVM is a portable bytecode interpreter written in C, and implementing the Java virtual machine specification, second edition.
- Its goals are to be reasonably small, fast, and efficient, as well as providing a well-designed and robust platform for conducting research.
- SableVM is licensed under the GNU Lesser General Public License (LGPL).
- It also makes use of a modified version of GNU Classpath.
- The initial development of SableVM was done as part of the Ph.D. research project of Etienne Gagnon.
- More information on SableVM is available at:
<http://www.sablevm.org/>

References

- These slides are inspired (i.e. copied) from these three books.
 - Design Patterns, Elements of Reusable Object-Oriented Software; Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides; Addison Wesley; 1995
 - Java Design Patterns, a Tutorial; James W. Cooper Addison Wesley; 2000
 - Design Patterns Explained, A new Perspective on Object Oriented Design; Alan Shalloway, James R. Trott; Addison Wesley; 2002