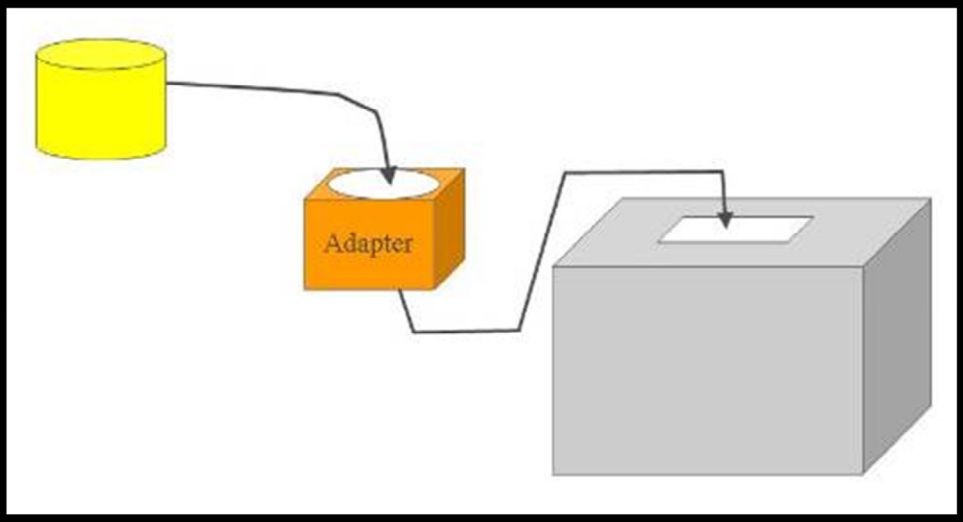
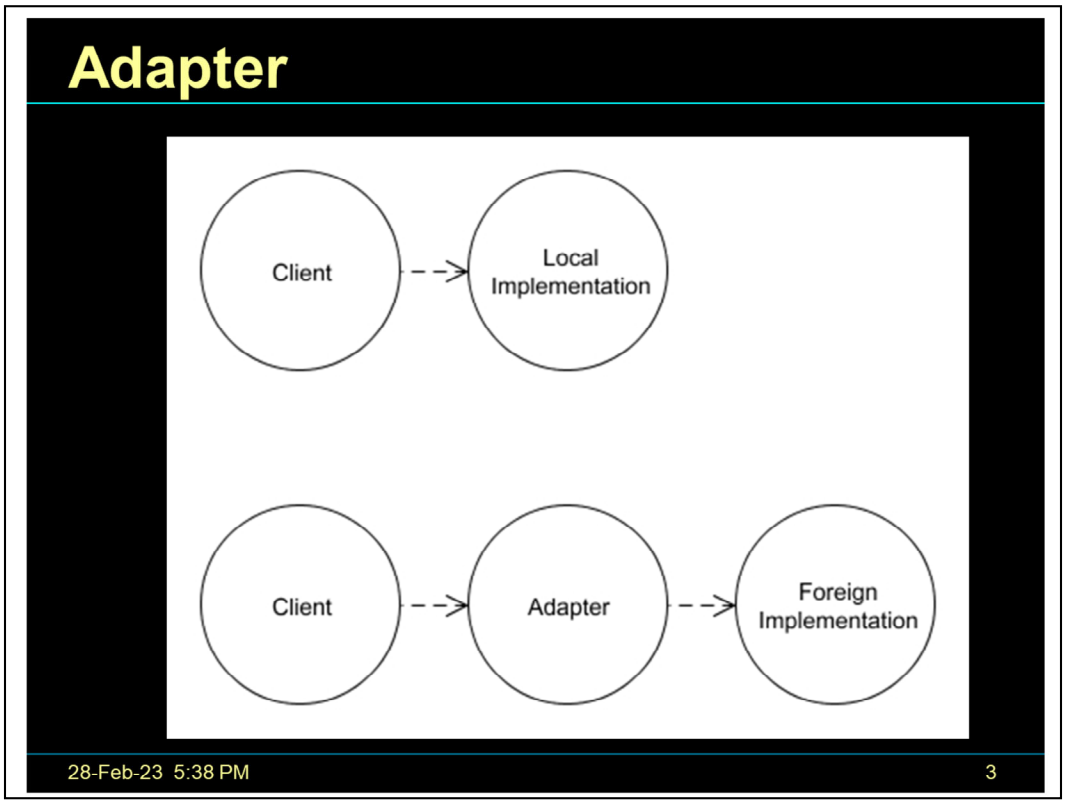
Adapter DP

Mismatch Cure - Adapter

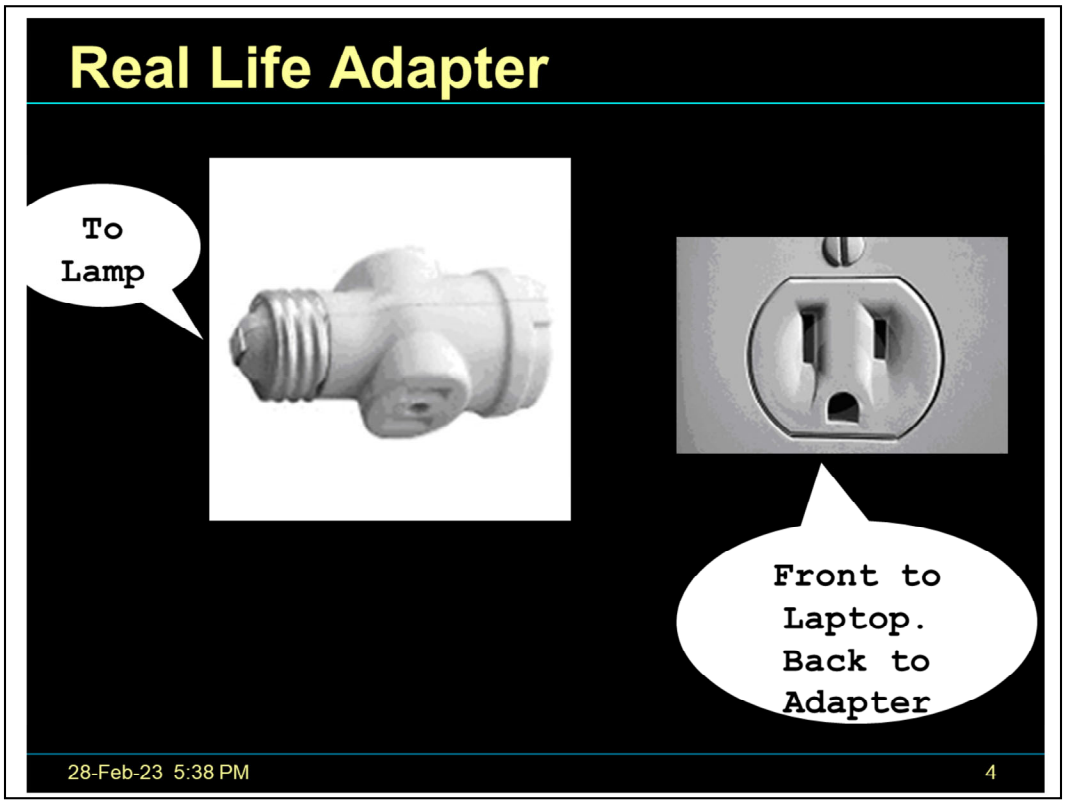




We use the Adapter DP when

We want to use an existing class and its interface does not match with the one we need.

We need to use several existing subclasses but it is impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of the parent class.



I travel a lot, and so my primary computing device is a laptop. I notice that even though laptop batteries have improved dramatically (lithium-ion or nickel-mental-hydride as opposed to nickel-cadmium), still they seem to quickly lose their ability to hold a charge for very long.

As a result, when I am at a coffee shop, airport terminal gate, or hotel lobby, I am always "on the hunt" for a wall outlet to plug my power brick into. Unfortunately, I am rarely the only person on this hunt, and typically someone else will have found the one outlet, and is "hogging" it.

My solution is to look for a lamp. A lamp has what I need -- 110 volts at 60 cycles, paid for by somebody else :). It does not present it in the form that I need it, however (two little slots that my power brick plug will fit into), but in a form that is appropriate for a light bulb (a threaded socket and a copper tab in the bottom).

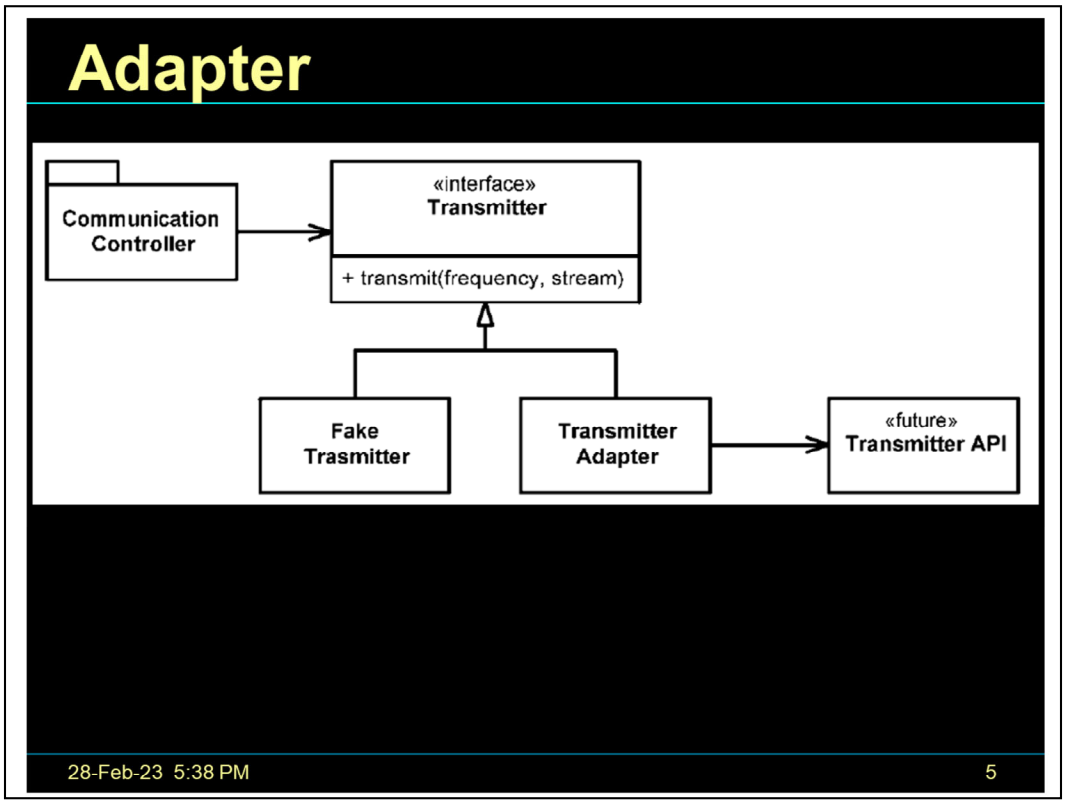
So, I carry one of these in my laptop bag: Left one

I (quietly) unscrew the light bulb from the lamp, screw in my adapter, and plug my laptop into it.

My laptop was designed to a specific interface. This one: Right one.

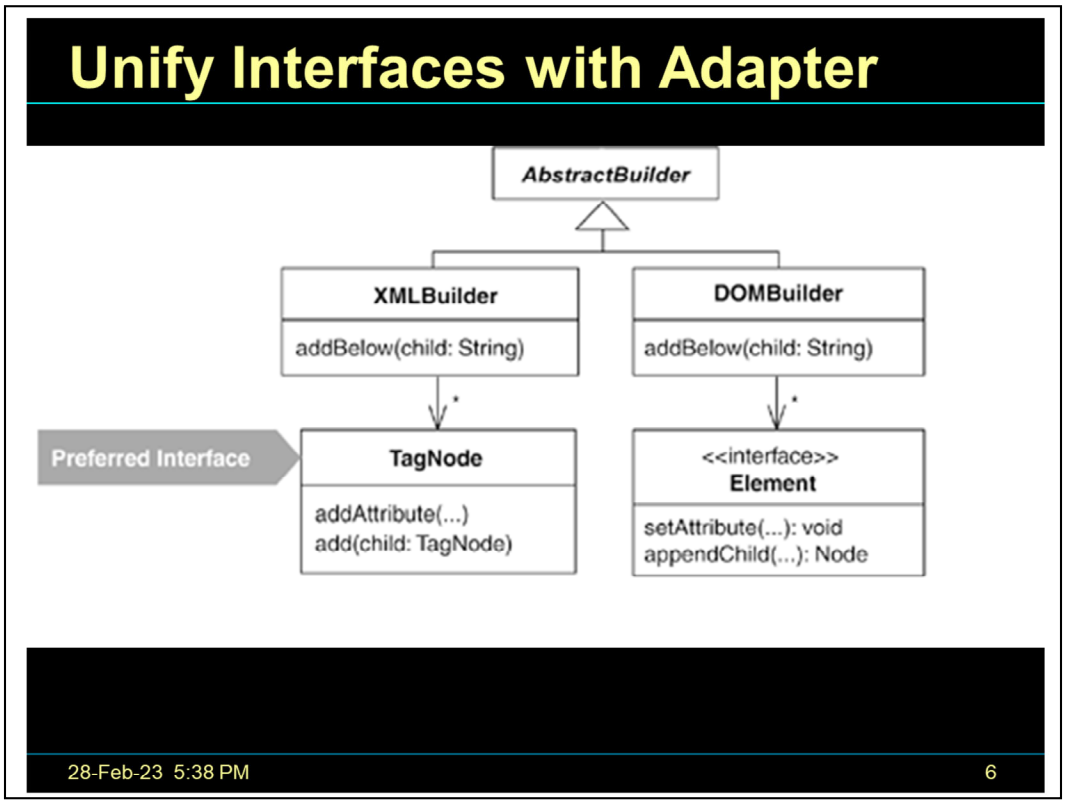
All wall sockets like this one (all across the United States) are exchangeable for one another, from my laptop's point of view. In a sense, they are a polymorphic set. The lamp socket is not exchangeable because its interface is not acceptable, even though the "stuff it has to offer" is just what I need.

The adapter allows my laptop to consume the electricity the way it was designed to, and makes the lamp socket interchangeable with the wall sockets of the world.

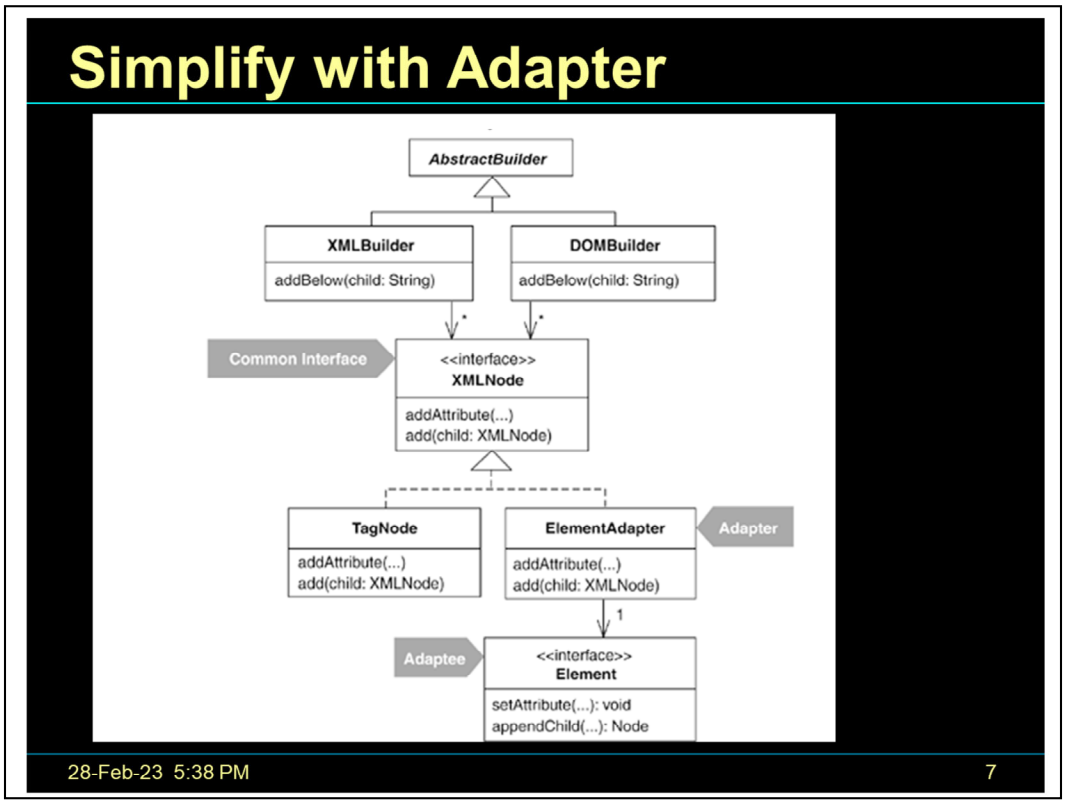


We are writing code for Communication controller. The code for Transmitter will be written later on.

Fake transmitter is a mock object.



Clients interact with two classes, one of which has a preferred interface.



Benefits and Liabilities

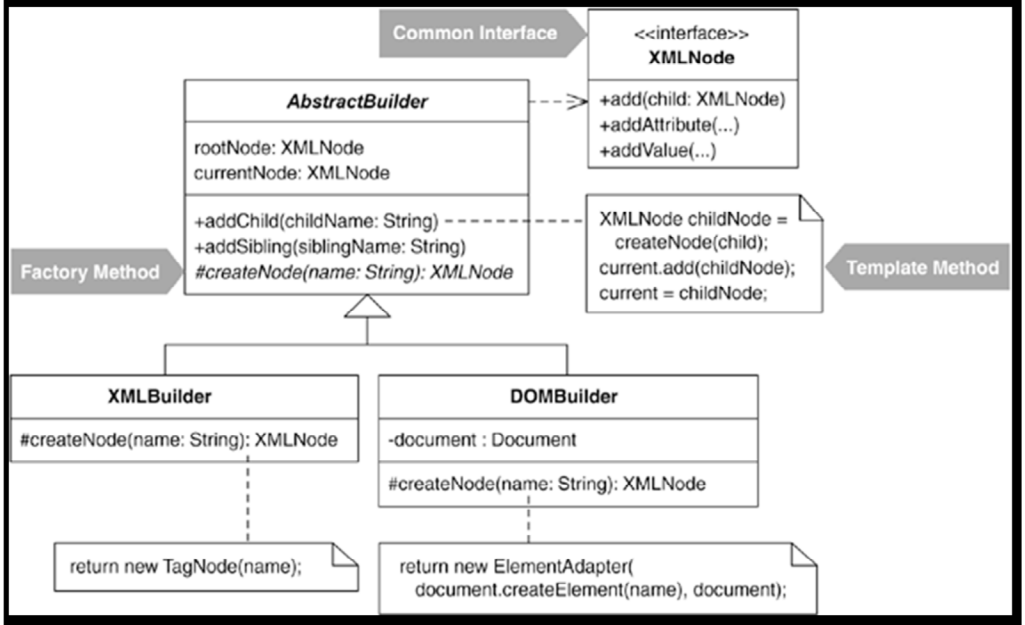
+ Removes or reduces duplicated code by enabling clients to communicate with alternative classes via the same interface.

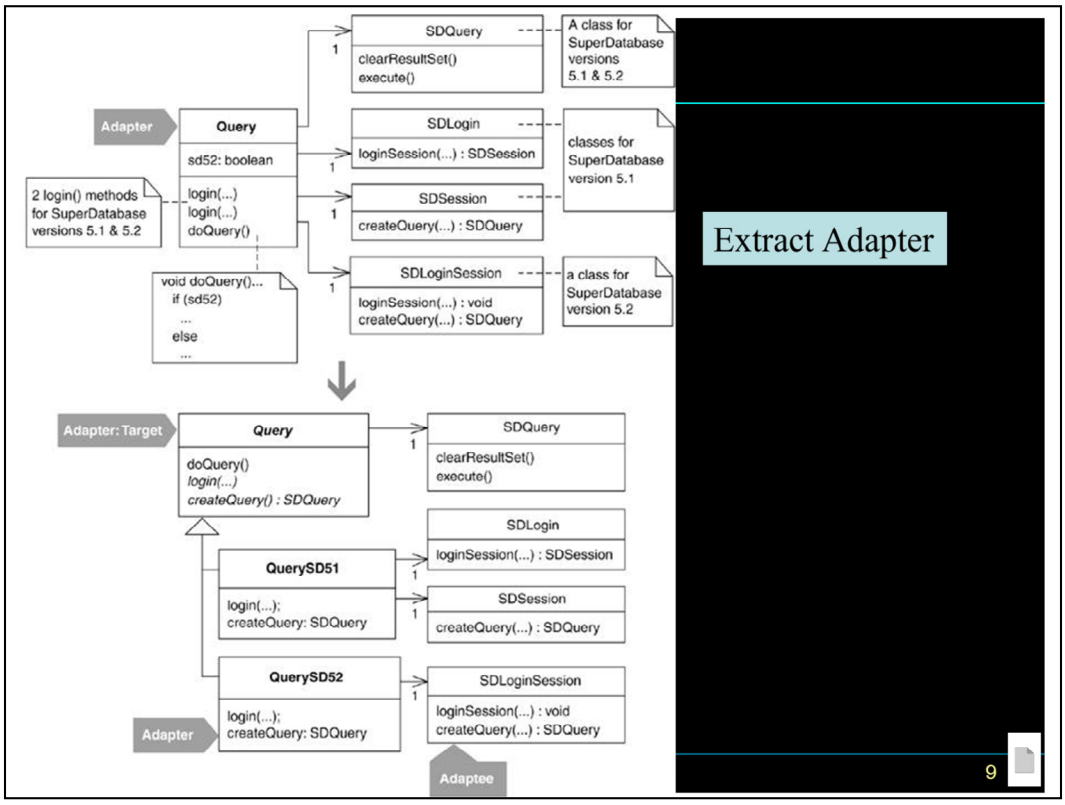
+ Simplifies client code by making it possible to communicate with objects via a common interface.

+ Unifies how clients interact with alternative classes.

– Complicates a design when you can change the interface of a class rather than adapting it.

Pull up similar code





One class adapts multiple versions of a component, library, API, or other entity.
Extract an Adapter for a single version of the component, library, API, or other entity.

While software must often support multiple versions of a component, library, or API, code that handles these versions doesn't have to be a confusing mess. Yet I routinely encounter code that attempts to handle multiple versions of something by overloading classes with version-specific state variables, constructors, and methods. Accompanying such code are comments like "This is for version X—please delete this code when we move to version Y!" Sure, like that's ever going to happen. Most programmers won't delete the version X code for fear that something they don't know about still relies on it. So the comments don't get deleted, and many versions supported by the code remain in the code.

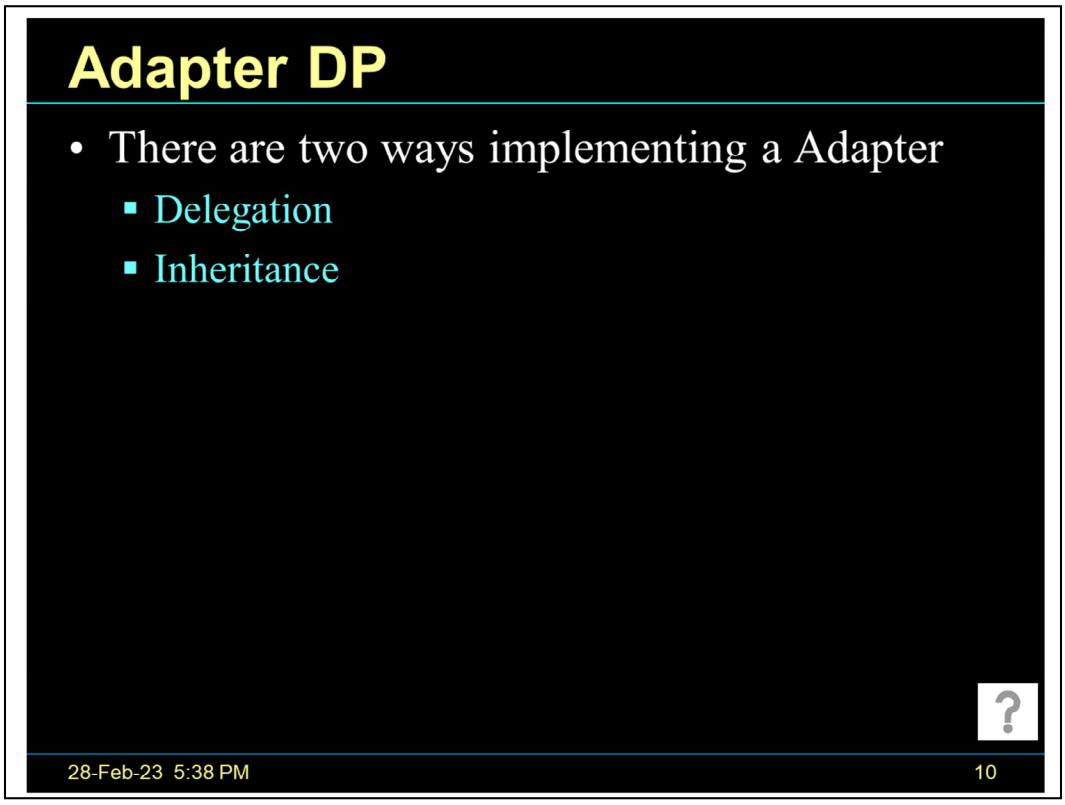
Benefits and Liabilities

\+ Isolates differences in versions of a component, library, or API.

\+ Makes classes responsible for adapting only one version of something.

\+ Provides insulation from frequently changing code.

– Can shield a client from important behavior that isn't available on the Adapter.



By object composition

We include the original class inside the new one and create the methods to translate calls within the new class.

This is called Object Adapter.

By inheritance

We derive a new class from the nonconforming one and add the methods we need to make the new derived class match the desired interface.

This is called Class Adapter.

Object Adapter works with subclasses while Class Adapter does not.

The object adapter can add functionality to all the adaptees at once.

Adapters in Java

- In a broad sense, there are already a number of adapters built into the Java language.
- Java adapters serve to simplify the complicated event interface. e.g. WindowAdapter, ComponentAdapter, ContainerAdapter, FocusAdapter, KeyAdapter, MouseAdapter and MouseMotionAdapter.
- Adapters in C++
  - queue, priority_queue and stack use other sequential containers like vector, list and deque internally.
  - back_insertor uses iterator internally.

The Read-only Interface Pattern

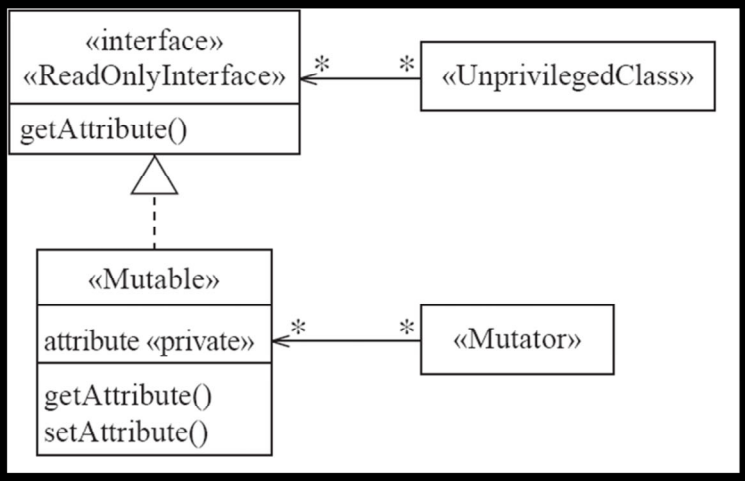
- ***Context***:
  - You sometimes want certain privileged classes to be able to modify attributes of objects that are otherwise immutable
- ***Problem***:
  - How do you create a situation where some classes see a class as read-only whereas others are able to make modifications?

Problem

- *Forces*:
  - Restricting access by using the **`public`**, **`protected`** and **`private`** keywords is not adequately selective.
  - Making access **`public`** makes it public for both reading and writing

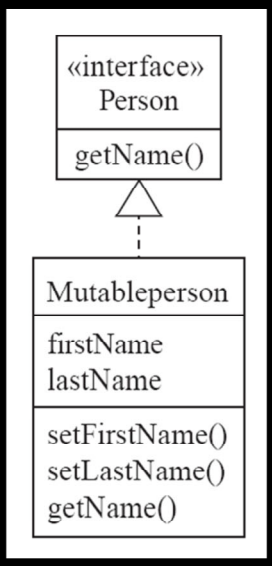
Read-only Interface

- ***Solution:***



Read-only Interface

- Example:



Read-only Interface

- Antipatterns:
  - Make the read-only class a *subclass* of the «Mutable» class
  - Override all methods that modify properties
    - such that they throw an exception