

Lecture 4: Design Patterns

...

Intro to Design Patterns

What's a Design Pattern?

- Design patterns are typical solutions to commonly occurring problems in software design.
- Patterns are blueprints that you can customize to solve a recurring design problem in your code.
- The pattern is not a specific piece of code, but a general concept for solving a particular problem.
 - You can't just find a pattern and copy it into your program, the way you can with off-the-shelf functions or libraries.

What does a Design Pattern consist of?

Most patterns are described formally so people can reproduce them in many contexts.

- **Intent** of the pattern briefly describes both the problem and the solution.
- **Motivation** further explains the problem and the solution the pattern makes possible.
- **Structure** of classes shows each part of the pattern and how they are related.
- **Code example** makes it easier to grasp the idea behind the pattern.

Patterns: How High-Level are they?

- The most basic and low-level patterns are often called **idioms**.
 - They usually apply only to a single programming language.
 - Python: `new_list = [elem.strip() for elem in old_list]`
- Most design patterns we will learn are higher-level
 - Not language-specific
 - Refer to the structure of a set of classes, or the structure of the whole system.

Who invented patterns?

- No one, and everyone!
- Patterns are typical solutions to common problems in object-oriented design.
- When a solution gets repeated over and over in various projects, someone eventually gives it a name and describes the solution in detail.
- That's basically how a pattern gets discovered.

Why learn patterns?

- You might manage to work as a programmer for many years without knowing a single pattern.
 - you might be implementing some patterns without even knowing it!
- So why would you spend time learning them?
 - Design patterns are a toolkit of tried and tested solutions to common problems in software design.
 - Knowing patterns teaches you how to solve all sorts of problems using principles of object-oriented design.
- Design patterns define a common language that you and your teammates can use to communicate more efficiently.
 - You can say, “Oh, just use a Singleton for that,” and everyone will understand the idea behind your suggestion. No need to explain what a singleton is if you know the pattern and its name.

Design Patterns

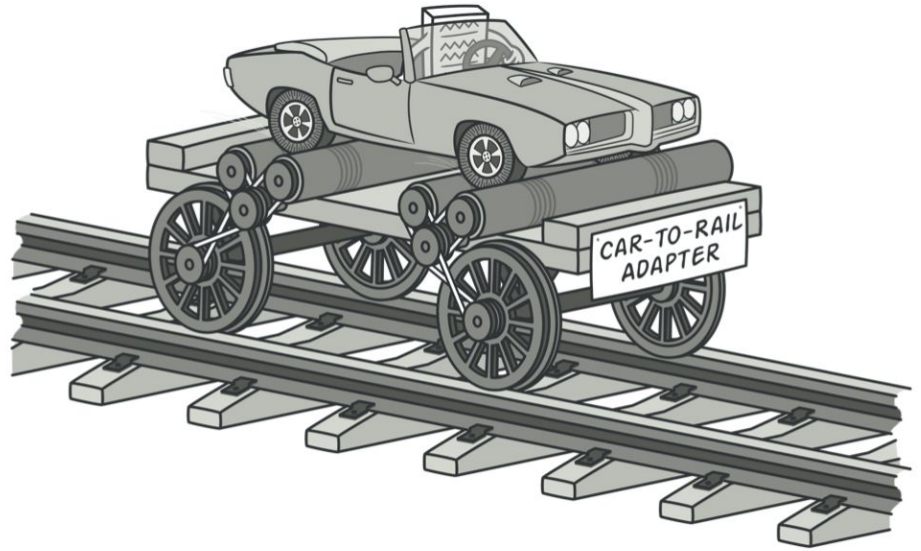
- Patterns can be categorized by their purpose
- We will cover three types:
 - **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.
 - **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
 - **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.

Our First Design Pattern: Adapter

Adapter Design Pattern

Also known as: Wrapper

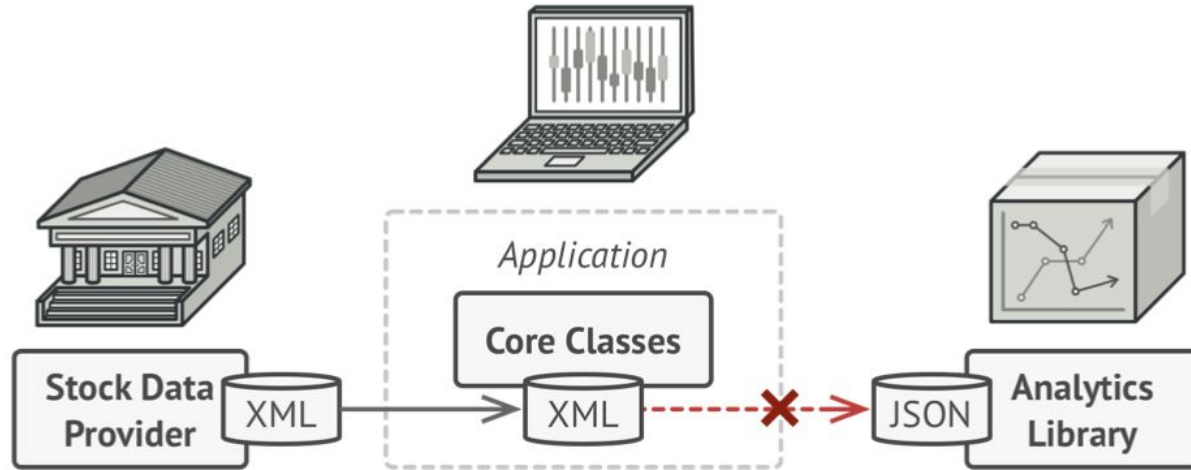
Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.



Adapter: Problem

- Imagine that you're creating a stock market monitoring app.
- The app:
 - 1) downloads the stock data from multiple sources in XML format
 - 2) displays nice-looking charts and diagrams for the user.
- At some point, you decide to improve the app by integrating a smart 3rd-party analytics library.
- But there's a catch: the analytics library only works with data in JSON format.

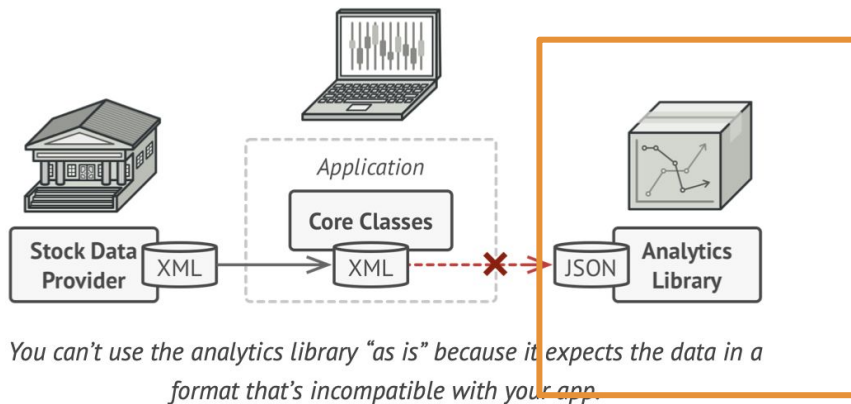
Adapter: Problem



You can't use the analytics library "as is" because it expects the data in a format that's incompatible with your app.

Adapter: Solution?

- You could change the external analytics library to work with XML.
- However, this might break some existing code that relies on the library.
- And worse, you might not have access to the library's source code in the first place, making this approach impossible.



Adapter: Solution

- You can create an **adapter**.
- This is a special object that **converts the interface of one object so that another object can understand it**.
- An adapter **wraps** one of the objects to hide the complexity of conversion happening behind the scenes.
- The wrapped object isn't aware of the adapter.
- Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate.

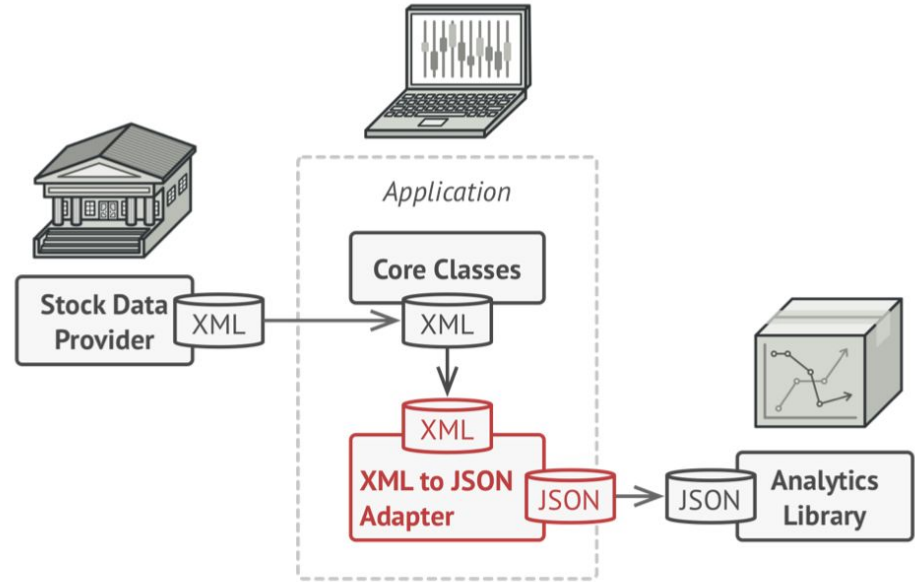
Adapter: Solution

1. The adapter gets an interface, compatible with one of the existing objects.
2. Using this interface, the existing object can safely call the adapter's methods.
3. Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.

Sometimes it's even possible to create a two-way adapter that can convert the calls in both directions.

Adapters - Solution

To solve the dilemma of incompatible formats, you can create XML-to-JSON adapters for every class of the analytics library that your code works with directly. Then you adjust your code to communicate with the library only via these adapters.



Adapter: Pros

- Single Responsibility Principle.
 - You can separate the interface or data conversion code from the primary business logic
- Open/Closed Principle. You can introduce new types of adapters into the program without breaking the existing client code

Adapter: Cons

- The overall complexity of the code increases because you need to introduce a set of new interfaces and classes.
- Sometimes it's simpler just to change the service class so that it matches the rest of your code.

Adapter: In-Class Exercise

- Wrap an object that operates in meters and kilometers with an adapter that converts all of the data to imperial units such as feet and miles.
- Make class Speedometer:
 - Returns in miles per hour
 - You have a Car_Display that can only support KM/hr
- Starter Code in Class Github!
 - adapter.py
 - Submit via Canva (should be short/easy)