Design Patterns

Denis Ponizovkin

April 28, 2019

Design Pattern

Definition

In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design.

Roots

The idea was introduced by the architect Christopher Alexander and has been adapted for various other disciplines, most notably computer science.

> The elements of this language are entities called patterns. Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice. — Christopher Alexander

From architecture to software

GoF

Christopher Alexander is the architect who first studied patterns in buildings and communications and developed "pattern language" for generating them. His work has inspired us time and again. There are many ways in which our work is like Alexander's. Both are based on observing existing systems and looking for patterns in them. Both have templates for describing patterns. Both rely on natural language and lots of examples to described patterns rather than formal languages, and both give rationales for each pattern. — Design Patterns.

Why do I need to study patterns

- Reusability;
- The use of common terminology;
- Design patterns provide us with an abstract, high-level view of both the problem and the whole process of object-oriented development.
- design patterns allow a developer or a group of developers to find design solutions for complex problems without creating a cumbersome class inheritance hierarchy.

Other advantages

- Efficiency improvement of single developers and whole group of developers;
- The use of many design patterns also allows you to create more modifiable and flexible software;
- Properly studied design patterns greatly assist in a common understanding of the basic principles of object-oriented design.

Association

If two classes in a model need to communicate with each other, there must be link between them, and that can be represented by an association (connector). We can define a one-to-one, one-to-many, many-to-one and many-to-many relationship among objects.

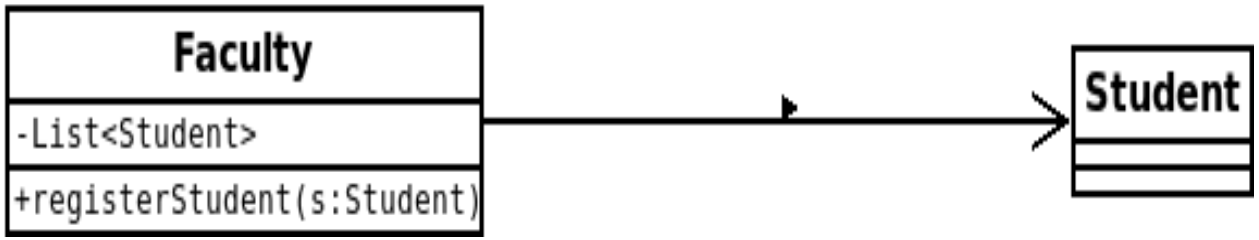


Figure: Unidirectional assocation example

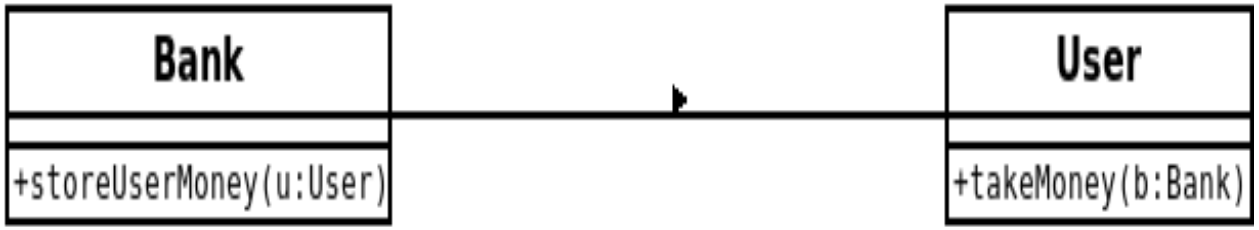
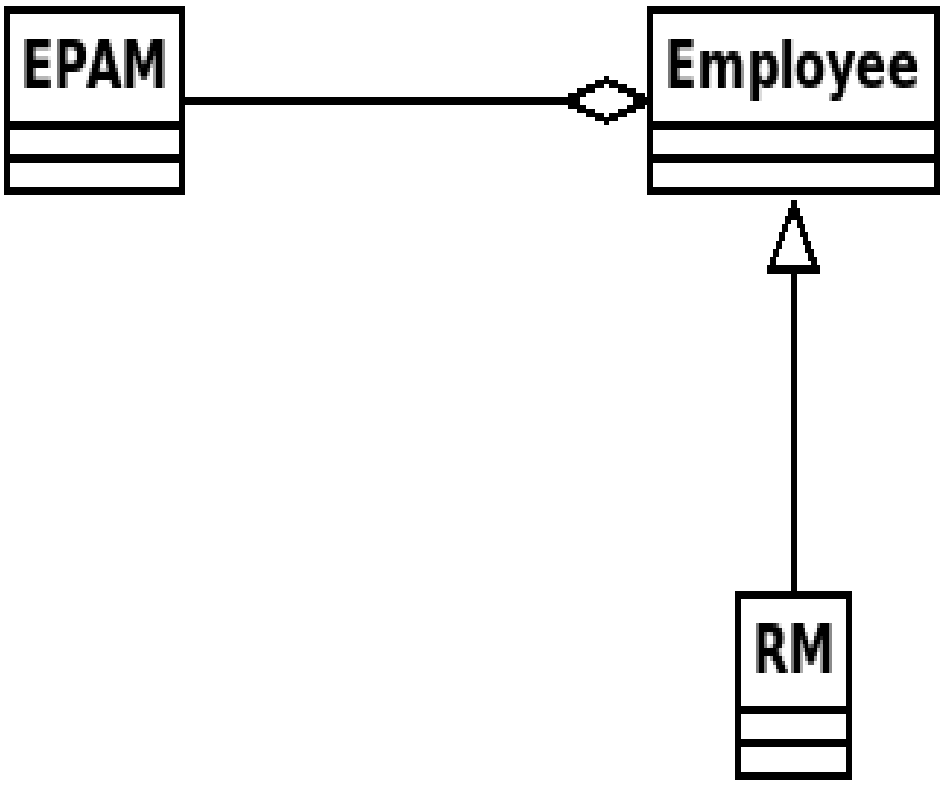


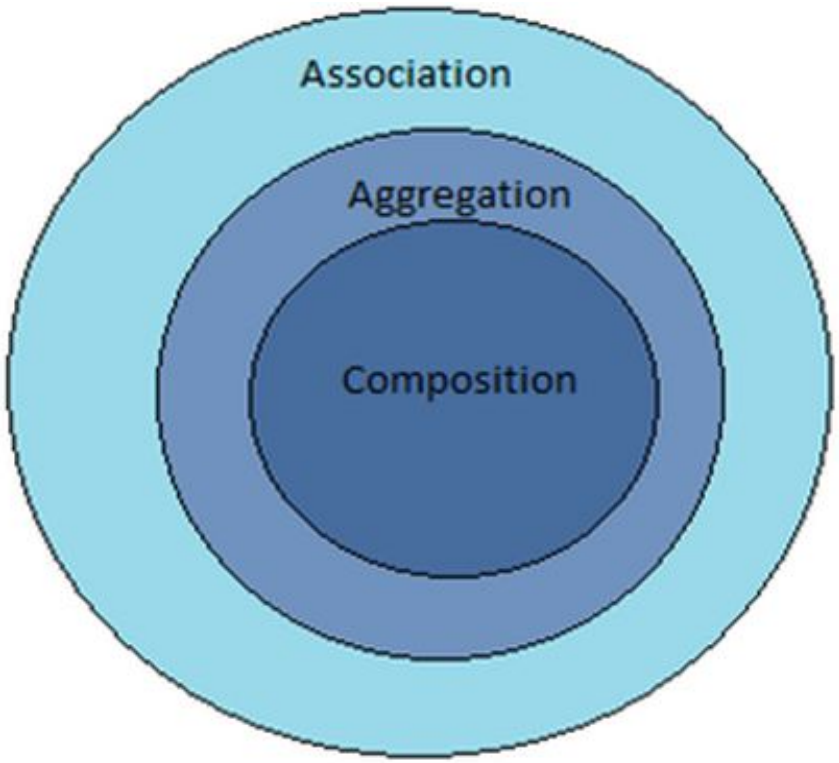
Figure: Bidirectional assocation example

Aggregation



Composition

Association, aggregation, composition



Definition

Structural patterns are concerned with how classes and objects are composed to form larger structures. Structural class patterns use inheritance to compose interfaces or implementations. Structural object patterns describe ways to compose objects to realize new functionality.

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

Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems. One way to achieve this goal is to introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem.

Example

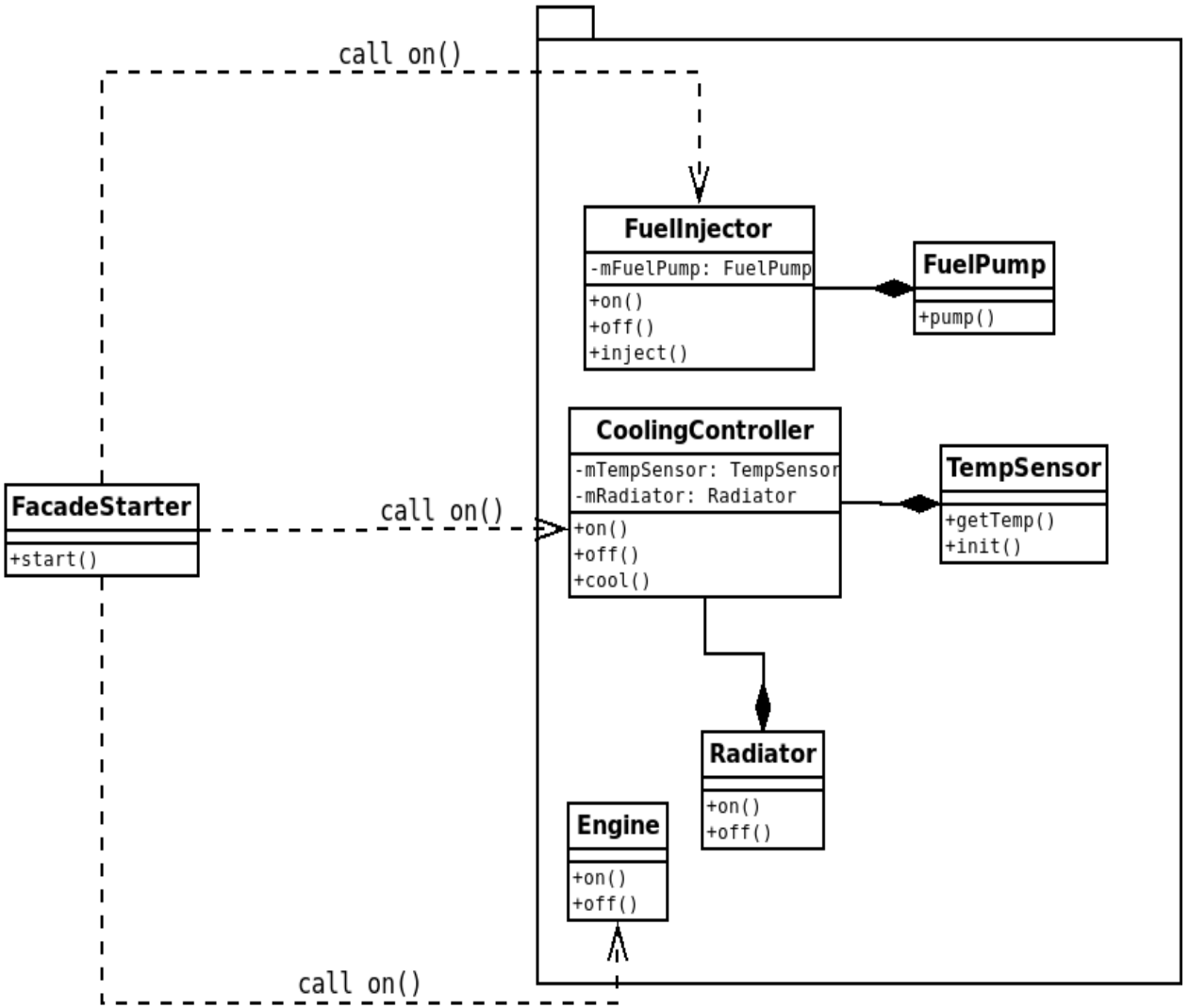


Figure: Facade for starting a car system

Applicability

- 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.

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

Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires.

Example. The task

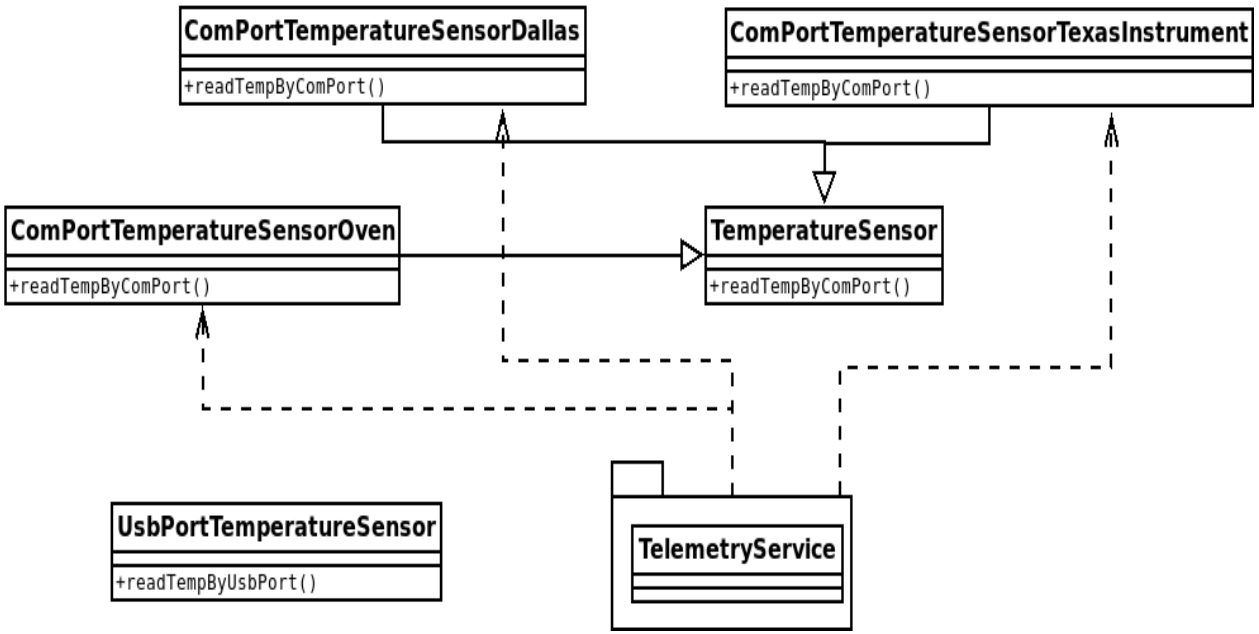


Figure: Adapter usage case

Example. The task resolution

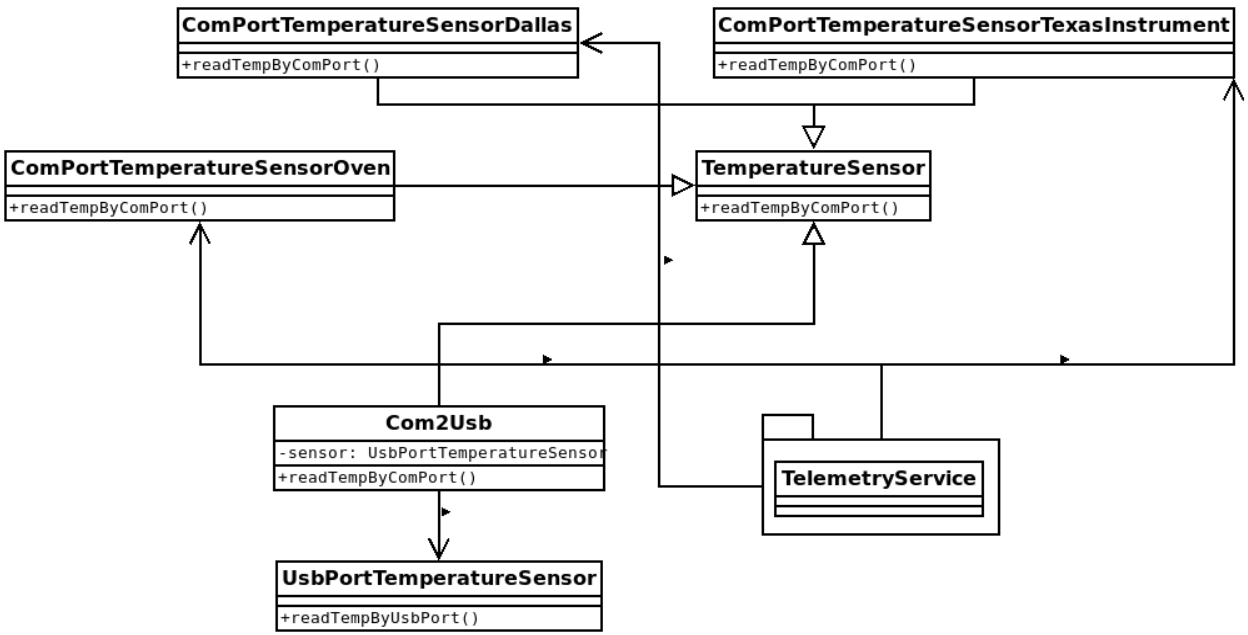


Figure: Adapter usage case

Example. The task resolution

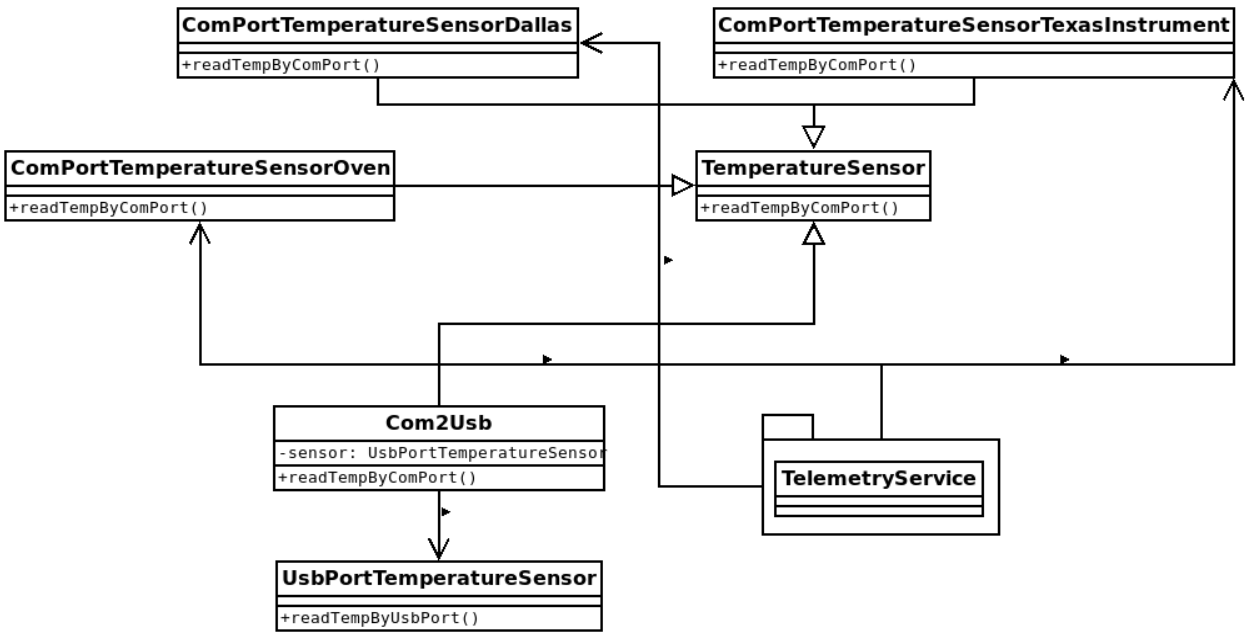


Figure: Adapter usage case

Realisation

Two types of adapters

- object — realisation via composition;
- class — realisation via inheritance.

Applicability

- 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.
- (object adapter only) you need to use several existing subclasses, but it's unpractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Intent

Decouple an abstraction from its implementation so that the two can vary independently