[CS 225] Advanced C/C++

Lecture 11: Type erasure pattern

Sławomir "Swavek" Włodkowski
(Summer 2020)

Agenda

- Data types
  - What is it?
  - Strong typing / weak typing
  - Static typing / dynamic typing
- Data types vs. polymorphism
- Type erasure pattern (TEP)
- Adapter design pattern

Data types

Data type is a *category* of an object, a reference, a function, a template or an expression that implies:

- Memory representation of bit sequences, in particular:
  - The **size,**
  - The **alignment,**
  - The **layout**,
- Permitted and forbidden **operations**,
- Allowed and excluded **values**.

Strong typing / weak typing

A language is *colloquially* referred to as **strongly-typed** when it includes strict data type checking rules.

Typically, it means that you cannot work with an object as if it is of a different type than declared.

C is considered weakly-typed (`void*`, `malloc`), while C++ promotes strongly-typed programming (`new`), but by modern standards both are considered weakly-typed.

Static typing / dynamic typing

A language is *classified* as **statically-typed** when a data type describes a variable/a constant and it does not change. In dynamic typing, types describe individual objects.

Static typing allows for static type-safety verification done in compile-time to avoid run-time problems. Dynamic typing allows for programming focused on objects, resulting in simpler languages, with fewer or no type declarations at all.

Both C and C++ are statically typed.

Data types

Generally, data types are a good thing… but…

Data types vs. static polymorphism

In C++, static polymorphism for types implies use overloads or templates.

In templates, any type that matches **operations** and supports involved **values** can be used as a template argument (*duck-typing*).

Static polymorphism maintains static type checking,
at the cost of more obscure error messages.

Data types vs. static polymorphism

Static polymorphism causes problems for the type system:

- How to avoid code where a use of a template forces a programmer to use more templates?
- How to not allow for certain data types?
  Solutions: SFINAE (soon) and *C++20 (concepts, constraints)*

Data types vs. dynamic polymorphism

In C++, dynamic polymorphism for types implies use of pointers to common base classes (often interface classes). Any type that offers the “is-a” relationship matches **operations** on the interface and supports involved **values** accepted at the interface.

Dynamic polymorphism maintains static type checking, but limits supported operations to the capability of a base class.

Data types vs. dynamic polymorphism

Dynamic polymorphism causes problems for the type system:

- What about types that do not share a base class?
- How to get rid of using pointers to a base class?
  Pointers are a responsibility and contribute to many bugs…

Type erasure pattern (TEP)

Motivation:

Offer a data type that is not polluting code with templates and abstracts any other data type that matches the interface but does not have to share a base class.

Type erasure pattern (TEP)

Motivation:

Offer a data type that is not polluting code with templates (*a non-template type!*) and abstracts (*no raw pointers!*) any other data type that matches the interface (*static typing!*) but does not have to share a base class (*unrelated types!*).

Type erasure pattern (TEP)

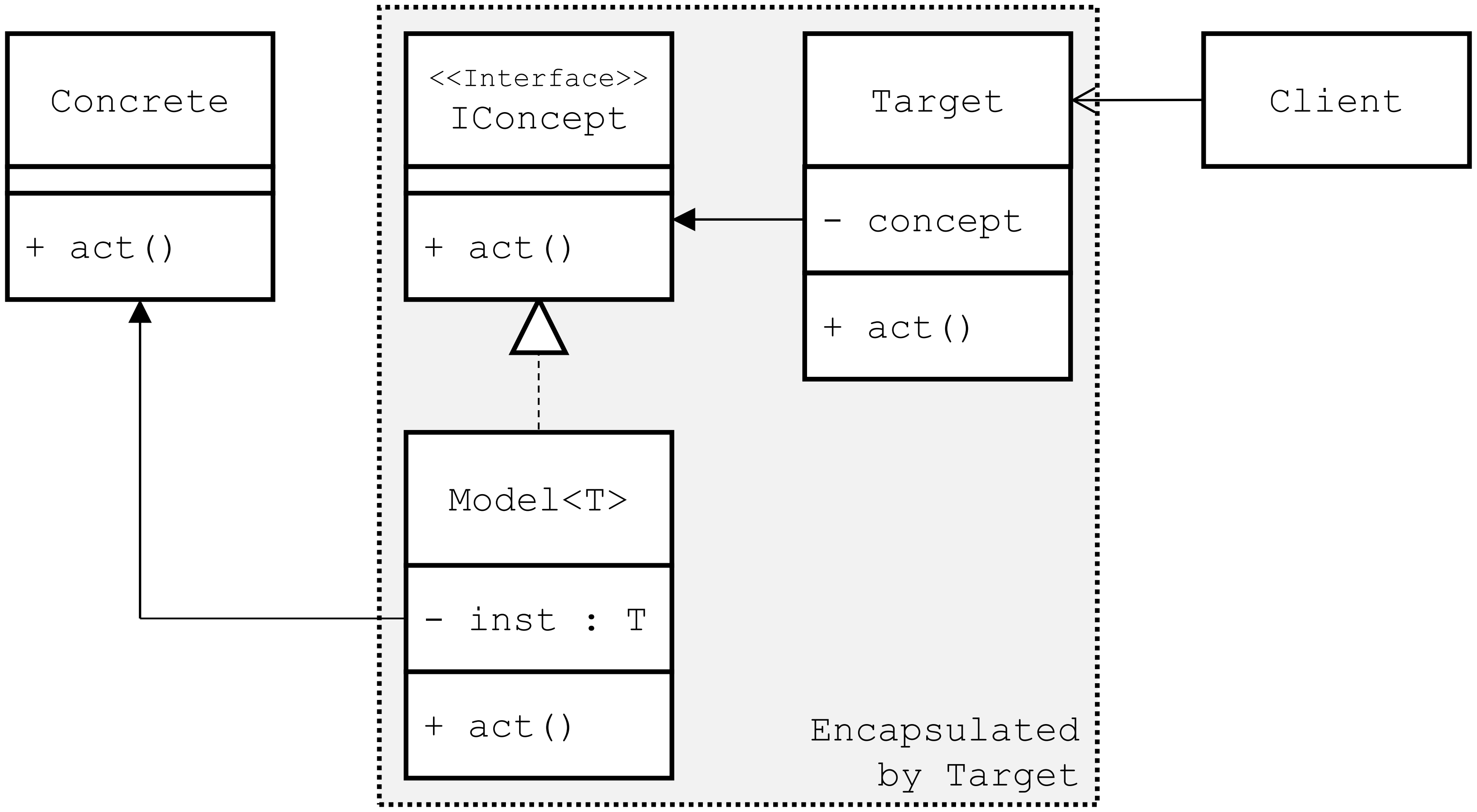
<u>Example:</u>

```
Type1 x;           // Type1 and Type2 do not
Type2 y;           // share a base class.

f(x);              // f(Tep arg) is not overloaded,
f(y);              // and is not a template.

Tep array[2] = {x, y};
```

Type erasure pattern (TEP)



Adapter design pattern

Pattern name

Adapter
Wrapper

Problem

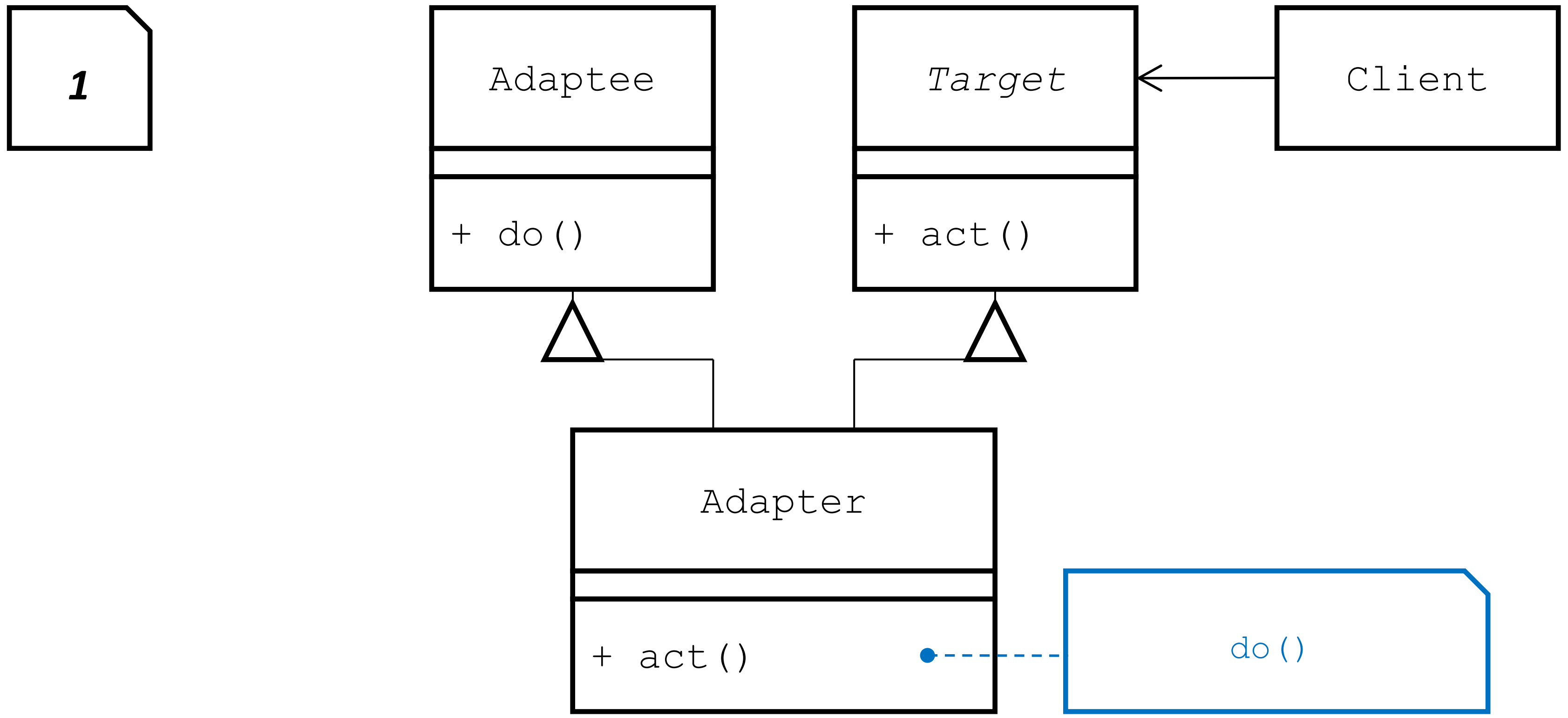
Convert the actual interface of a class into another interface client expects. Adapter lets classes work together that could not otherwise because of matching functionality, but incompatible interfaces.

Adapter design pattern

Solution

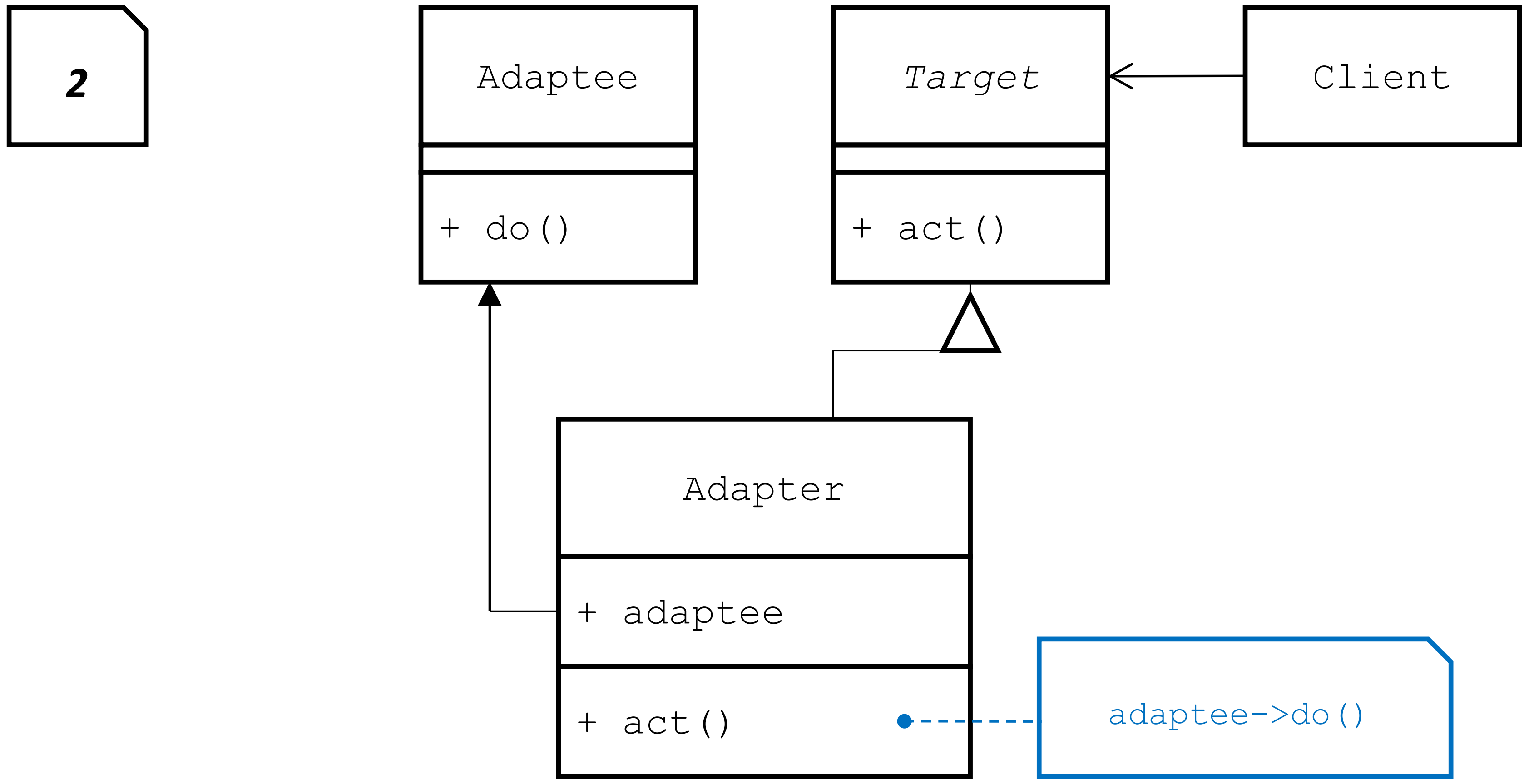
Create a class that matches the expected interface and wraps (by composition or inheritance) an object we want to adapt. Adapter can also provide more responsibilities beyond the adapted object to match the interface.

Adapter design pattern



UML diagram for an adapter design pattern with an inheritance relationship.

Adapter design pattern



UML diagram for an adapter design pattern with a composition relationship.