

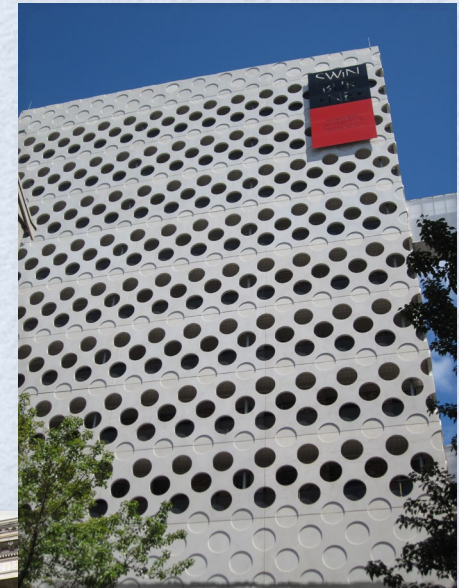
# Design Pattern



# What is a Design Pattern?

Christopher Alexander says:

“... [A] pattern describes a problem which 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.”





# Essential Design Pattern Elements

A pattern has four essential elements:

- The **pattern name** that we use to describe a design problem,
- The **problem** that describes when to apply the pattern,
- The **solution** that describes the elements that make up the design, and
- The **consequences** that are the results and trade-offs of applying the pattern.



# Design Patterns Are Not About Design

- Design patterns are not about designs such as linked lists and hash tables that can be encoded in classes and reused as is.
- Design patterns are not complex, domain-specific designs for an entire application or subsystem.
- Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.



# Creational Patterns

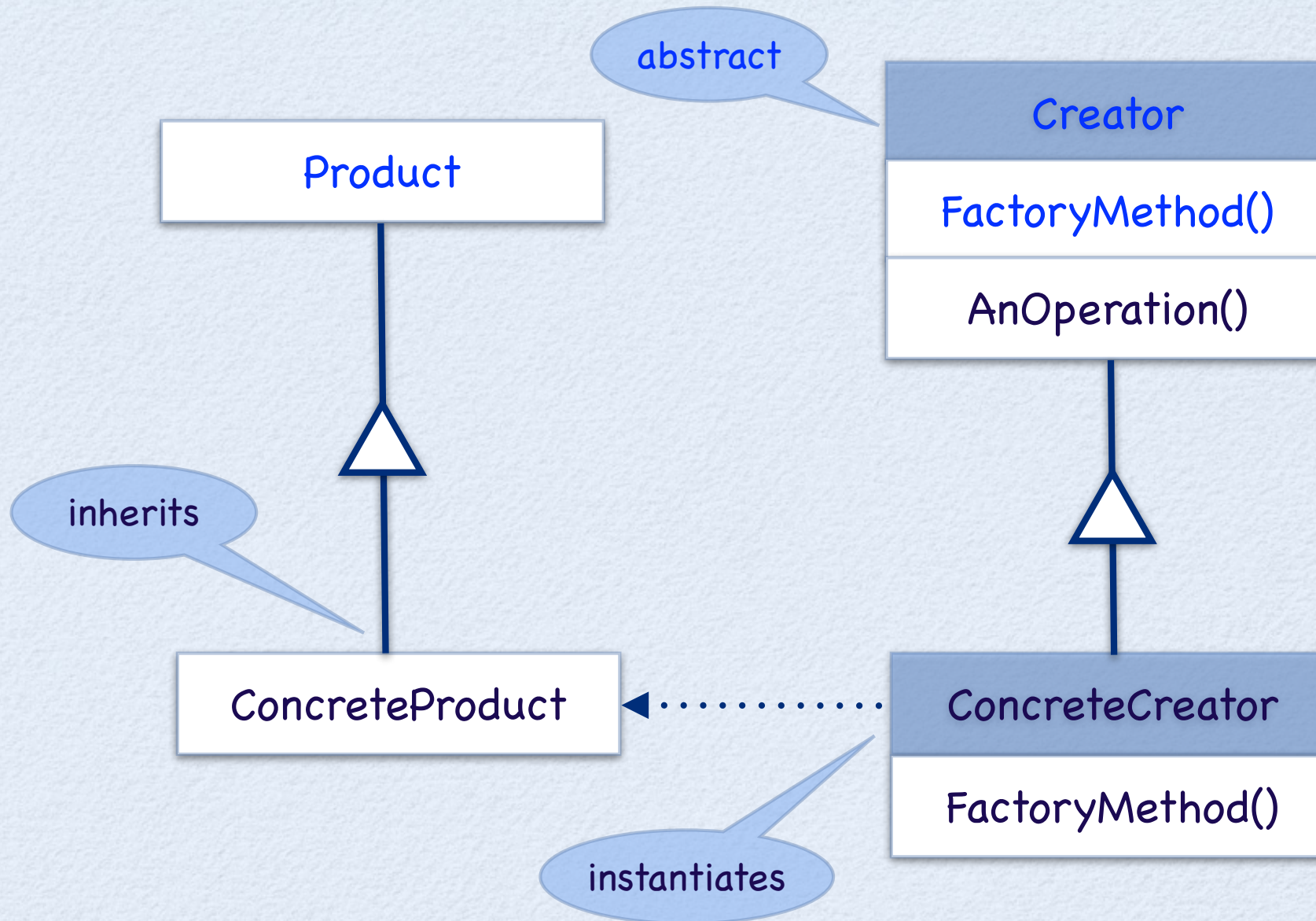
- Creational patterns abstract the instantiation process. They help to make a system independent of how its objects are created, composed, and represented.
- Main forms:
  - Creational patterns for classes use inheritance to vary the class that is instantiated.
  - Creational patterns for objects delegate instantiation to another object.



# Example: Factory Method

- Intent:
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Collaborations:
  - Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate **ConcreteProduct**.

# Structure of Factory Method





# Classical Example

- A classical example of factory method is that of **iterators**.
- An iterator provides access to elements of a collection. A concrete iterator methods **isolate** the **caller** from knowing which class to instantiate.



# Structural Patterns

- Structural patterns are concerned with how classes and object 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. The added flexibility of object composition comes from the ability to change the composition at runtime, which is impossible with static class composition.

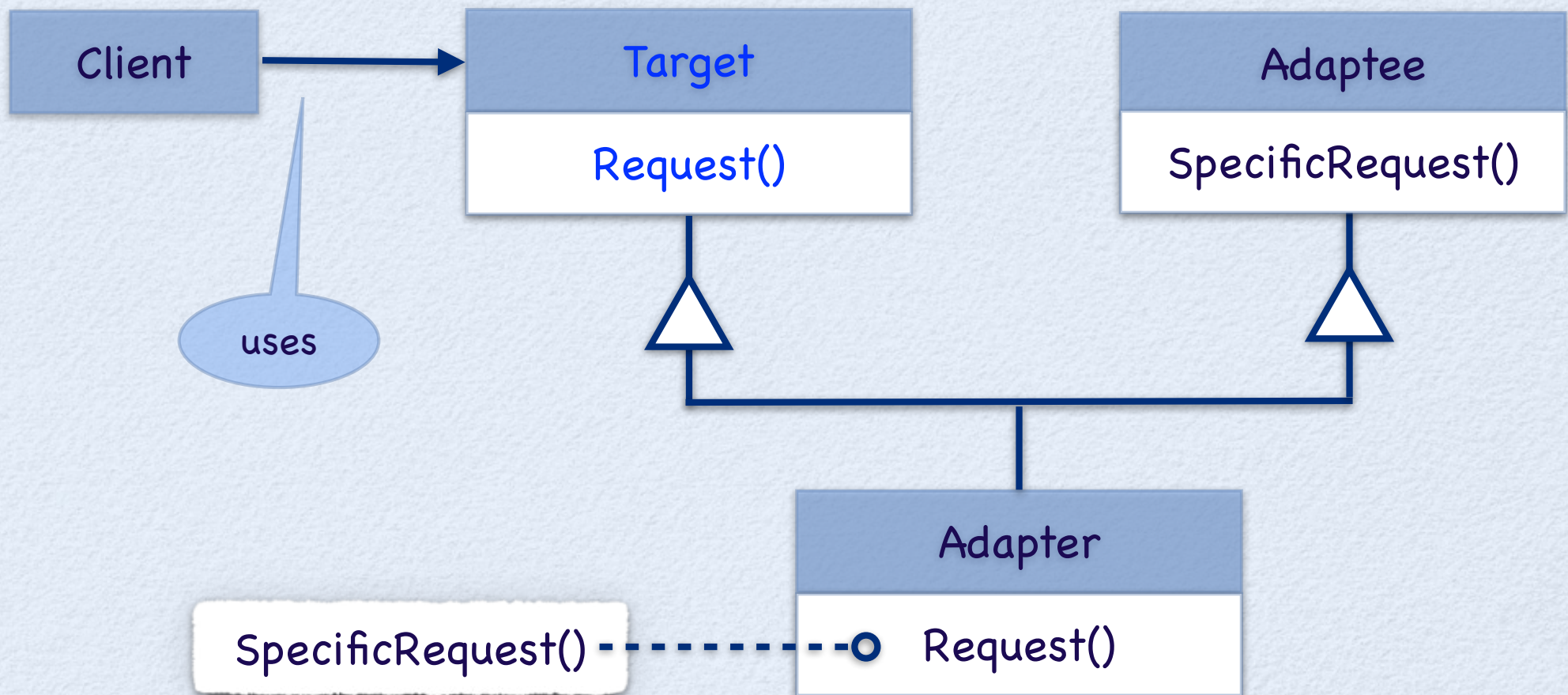


# Example: Adapter

- Intent:
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.
- Collaborations:
  - Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.

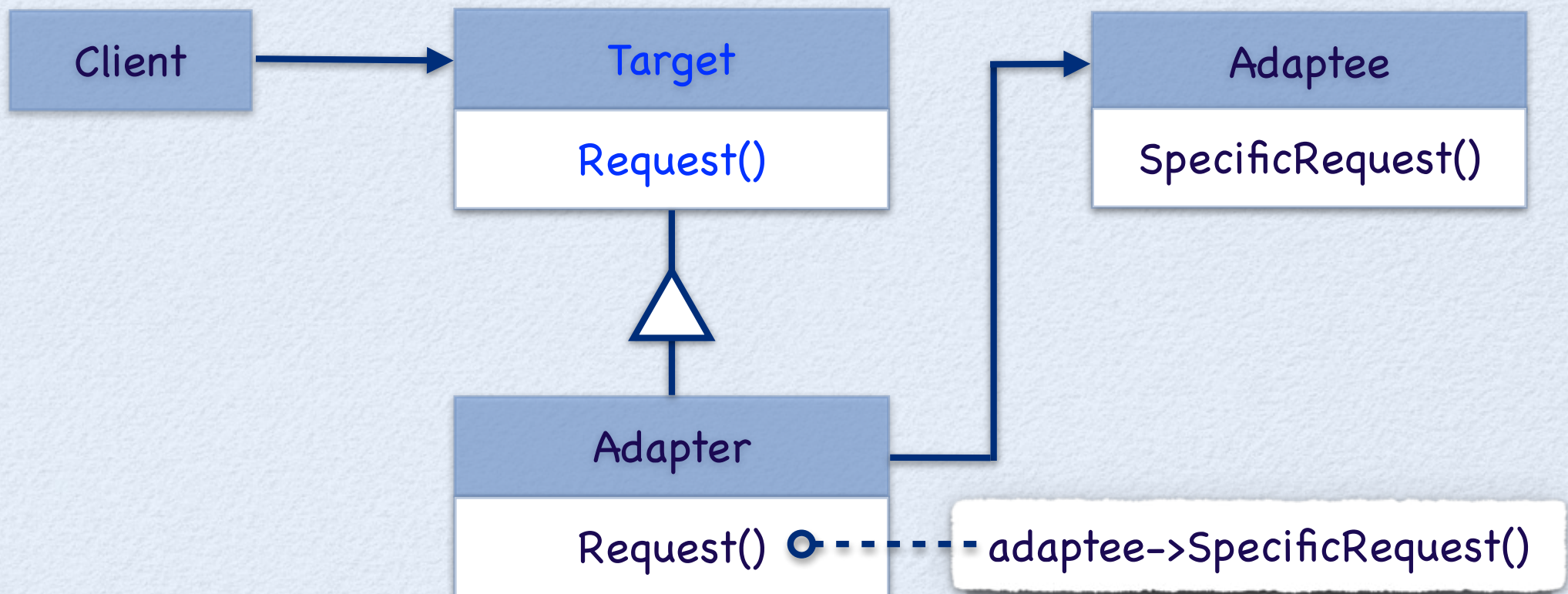


# Structure of a Class Adapter





# Structure of an Object Adapter





# Behavioral Patterns

- Behavioral patterns are concerned with algorithms and the assignment of responsibilities between classes and objects:
  - Behavioral **class** patterns use inheritance to distribute behavior between classes.
  - Behavioral **object** patterns use object composition rather than inheritance. Some describe how a group of peer objects cooperate to perform a task that no single object can carry out by itself.
- The classic example of a behavioral pattern is **Model-View-Controller** (MVC), where all views of the model are notified whenever the model's state changes.



# Example: Iterator

- Intent:
  - Provide a way to access the elements of n aggregate object sequentially without exposing its underlying representation.
- Collaborations:
  - A `ConcreteIterator` keeps track of the current object in the aggregate and can compute the succeeding object in the traversal.



# Structure of Iterator

