Introduction to Software Engineering

ECSE-321

Unit 15 – Design Patterns (Part 1)

Collections

- Warehouse for software objects
  - Lists
  - Sets
  - Hash tables
  - Queues
- Need mechanisms for iterating over the objects in the collections
  - Browse a warehouse
  - Search a warehouse

Collections and Iteration

A **collection** is an object that holds or contains other objects.

Iteration over a collection or **collection iteration** is traversal and access of each element of a collection.

Iteration Mechanisms

- An **iteration mechanism** is a language feature or a set of operations that allow clients to access each element of a collection.
- For example:
  - Java and C/C++ have for-loop constructs that support iteration over “collections”

Iteration Mechanism Operations

- *Initialize*—Prepare the collection for traversal
- *Access current element*—Provide client access to the current element
- *Advance current element*—Move on to the next element in the collection
- *Completion test*—Determine whether traversal is complete

Other Iteration Mechanism Requirements

- *Information hiding*—The internal structure of the collection must not be exposed.
- *Multiple simultaneous iteration*—It must be possible to do more than one iteration at a time.
- *Collection interface simplicity*—The collection interface must not be cluttered up with iteration controls.
- *Flexibility*—Clients should have flexibility during collection traversal.

Iteration Mechanism Design Alternatives: Residence

Iteration mechanism residence:

- Programming language—As in Java or Visual Basic
  - Depends on the language
- Collection—A *built-in* iteration mechanism resides in the collection
- Iterator—An external entity housing the iteration mechanism
  - An **iterator** is an entity that provides serial access to each elements of an associated collection.

Iteration Mechanism Design Alternatives: Control

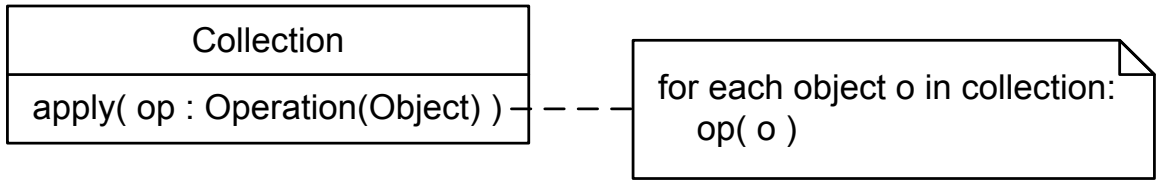
Iteration mechanism control:

- **External iteration control**—The iteration mechanism provides access to collection elements as directed by the client; the client calls the iteration control operations.
- **Internal iteration control**—The iteration mechanism accepts operations from clients that it applies to elements of the collection; the iteration mechanism calls the iteration control operations.

Iteration Mechanism Design Alternatives: Summary

		Residence	
		Collection	**Iterator**
Control	**External**	Collection with built-in external control	Iterator with external control
	Internal	Collection with built-in internal control	Iterator with internal control

Built-In Internal Control: Implementation




```
printObject( o : Object ) {
    print( o )
}
…
Collection c
…
c.apply( printObject )
```

Built-In Internal Control: Evaluation

- Hides collection internals 👍
- Does not complicate the collection interface 👍
- Multiple simultaneous iteration is not easy 👎
- Client has little control over iteration—no flexibility 👎

Built-In External Control: Implementation

For each kind of iteration desired

- Add the iteration control operations (or their equivalents) to the collection
- Other operations may be needed to provide flexibility

Built-In External Control: Evaluation

- Hides collection internals 👍
- Greatly complicates the collection interface 👎
- Multiple simultaneous iteration is not easy 👎
- Client has control over iteration—adequate flexibility 👍

Change During Iteration

- What should happen when a collection is changed during iteration?
- Requirements for a *coherent* iteration mechanism specification:
  - *Fault tolerance*—The program should not crash.
  - *Iteration termination*—Iteration should halt.
  - *Complete traversal*—Elements always present should not be missed during traversal.
  - *Single access*—No element should be accessed more than once.
- A **robust iteration mechanism** is one that conforms to some coherent specification

Iterator Pattern

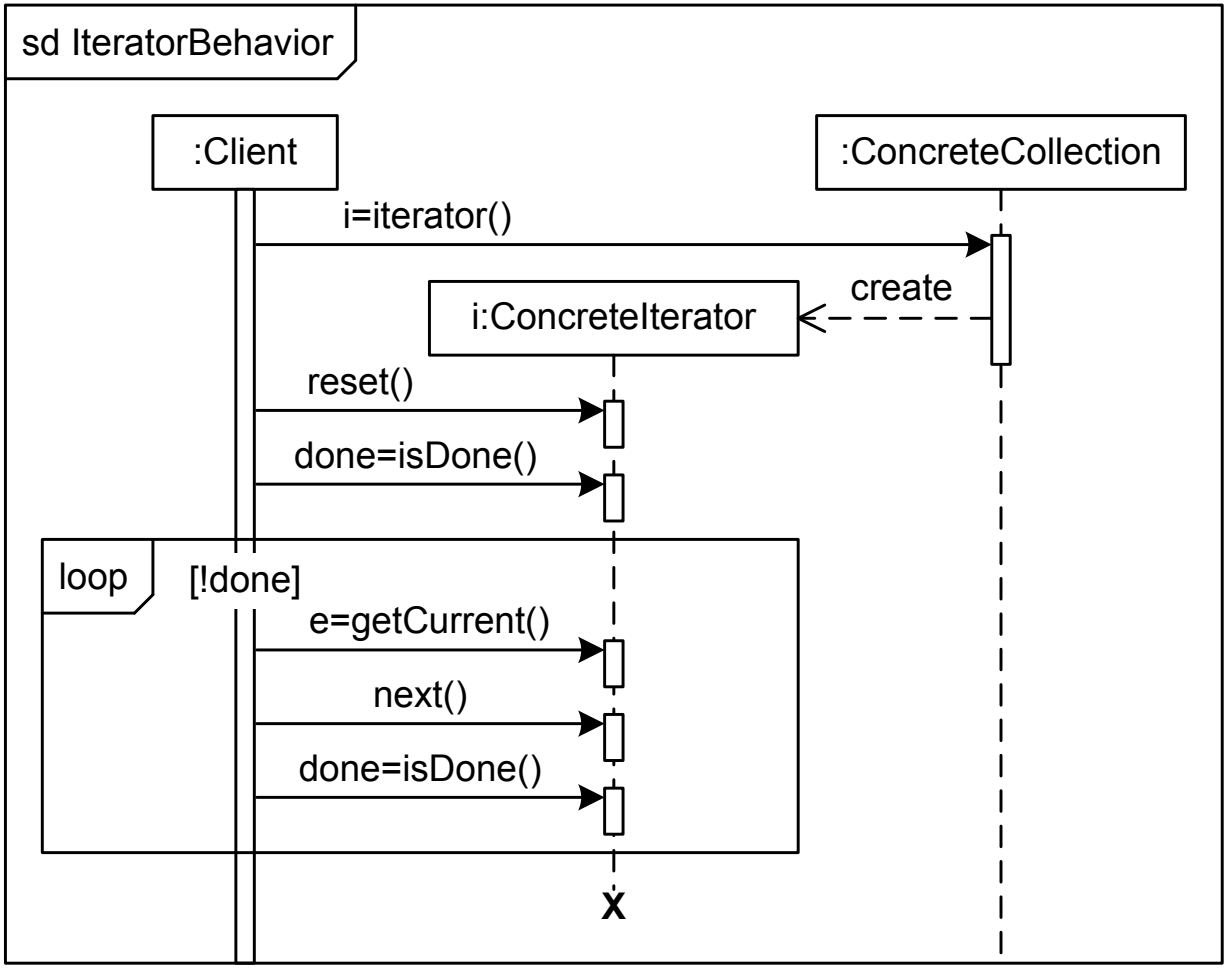
The **Iterator pattern** is an object-oriented design pattern for externally controlled iterators.

An Analogy

Consider a warehouse full of items that a client must process one by one.

- Don’t allow clients into the warehouse (information hiding)
- Clerks are like iterators
- Clerks can fetch each item for clients (external control)
- Clerks can be instructed by the client and then process each element on their behalf (internal control)

Iterator Behavior



Too Many Patterns? Need for Classification

- Since mid-1990s, hundreds of design patterns have been published.
- How can designers keep them all in mind?
  - Many are not that important or have narrow application.
  - A pattern classification scheme can help designers remember many important patterns.

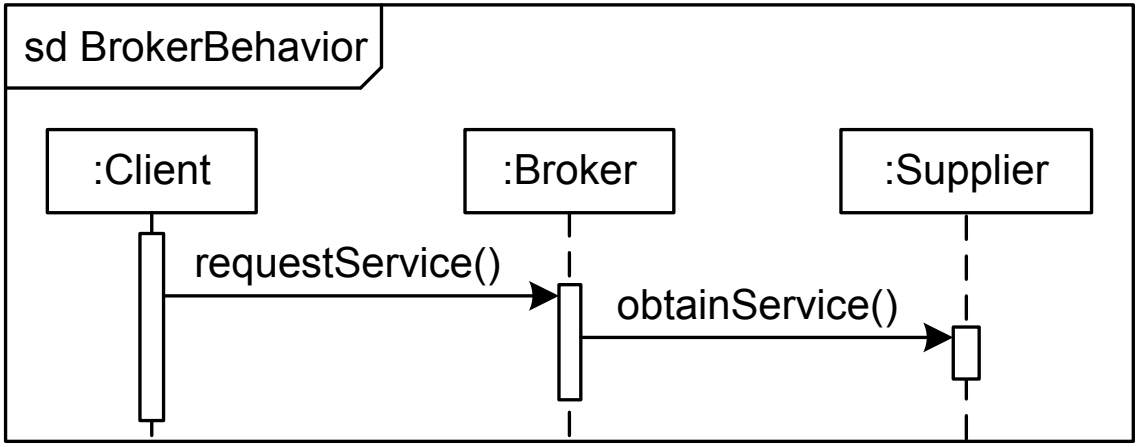
Pattern Categories

- **Broker patterns** have a client that needs a service from a supplier, and a *broker* that mediates the interaction between client and supplier.
- **Generator patterns** have a client who needs a new instance of a product, and a *generator* class that supplies the instance.
- **Reactor patterns** have a client that needs to respond to an event in a target. The client delegates this responsibility to a *reactor*.

Broker Pattern Structure

- The Client must access the Broker and the Broker must access the Supplier
- Most Broker patterns elaborate this basic structure

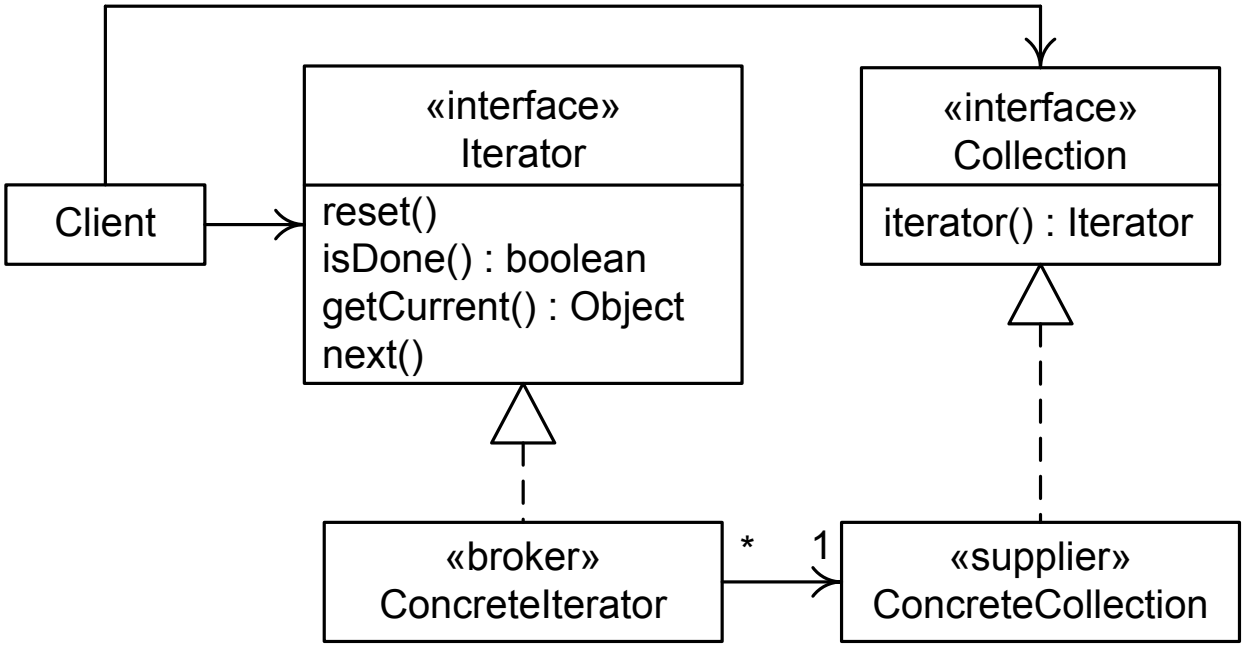
Broker Pattern Behavior



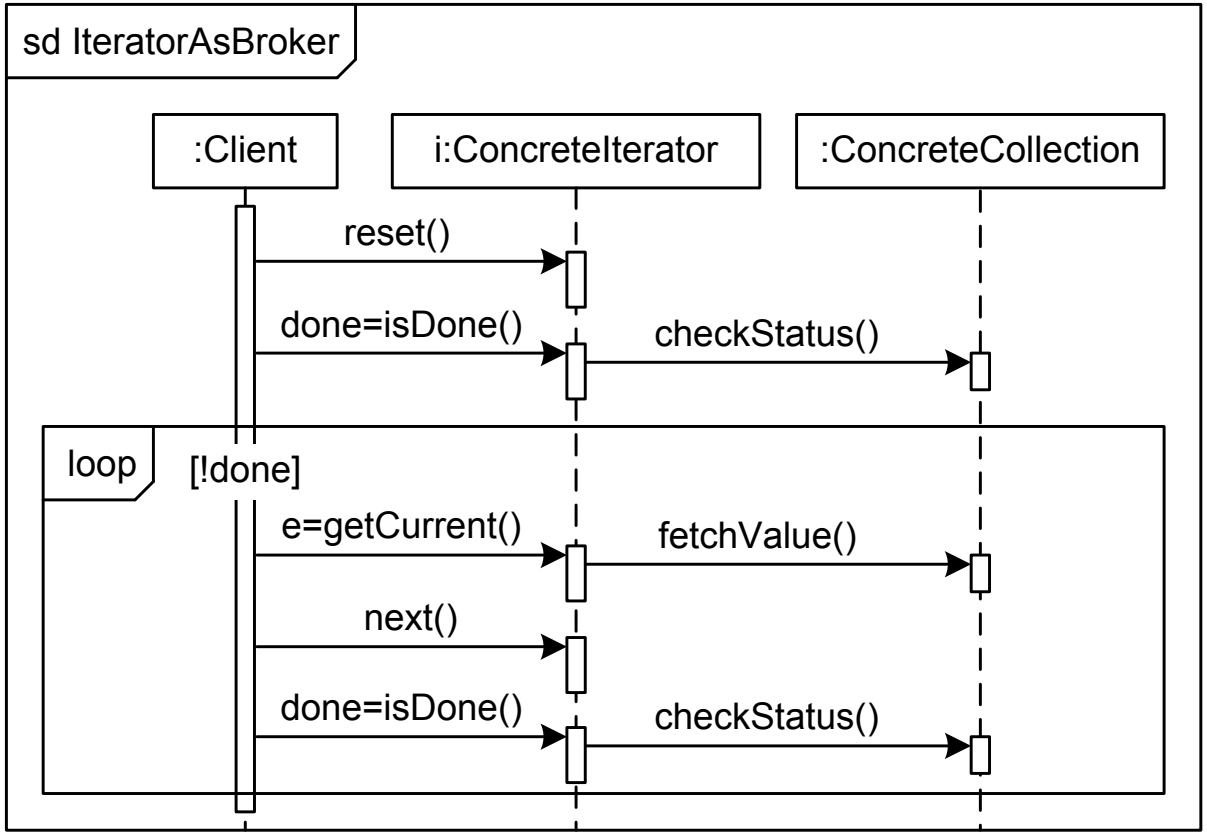
Broker Pattern Advantages

- *Simplify the Supplier*—A Broker can augment the Supplier's services.
- *Decompose the Supplier*—A complex Supplier can offload some of its responsibilities to a Broker.
- *Facilitate Client/Supplier Interaction*—A Broker may present a different interface, handle interaction details, etc.

Broker Example: Iterator Form



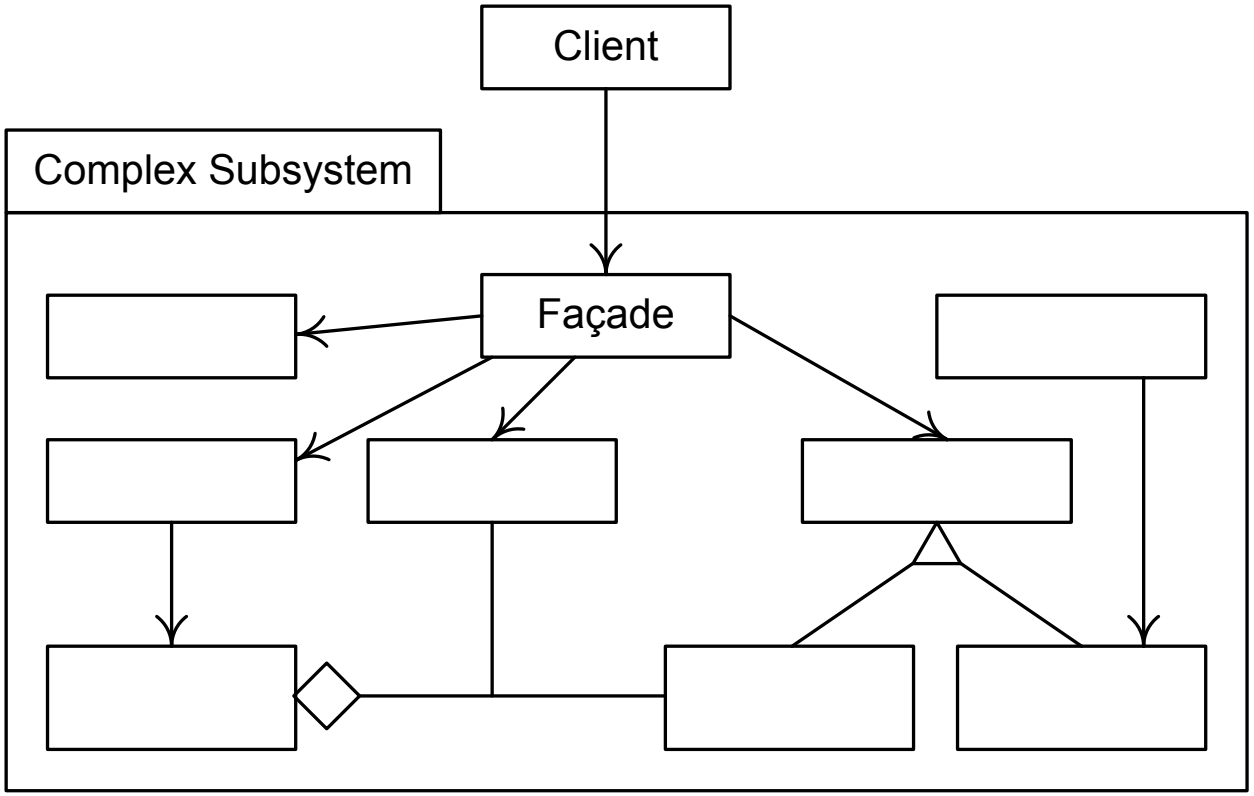
Broker Example: Iterator Behavior



The Façade Pattern

- The Façade pattern eases interaction between a client and a sub-system of suppliers by providing a simpler interface to the sub-system.
- The broker class is a façade that provides simplified sub-system services to clients.
- Analogy: a travel agent
- Examples:
  - Compiler
  - Memory management system

Façade Pattern Structure



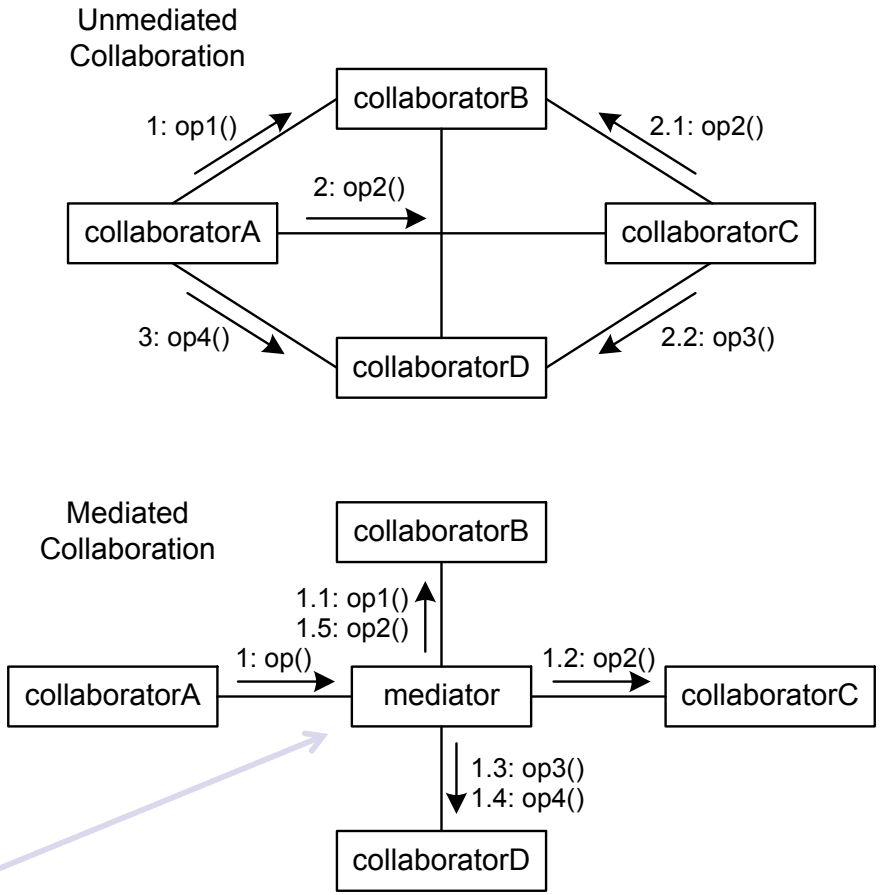
When to Use a Façade

- Use the Façade pattern when there is a need to provide a simplified interface to a complex sub-system.
- Façades can also help decouple systems.
  - If the façade mediates all interaction with a client, then the sub-system can be changed without affecting the client.
- A façade may work like an *adapter* by providing a new interface to a sub-system (adapters are discussed later).

The Mediator Pattern

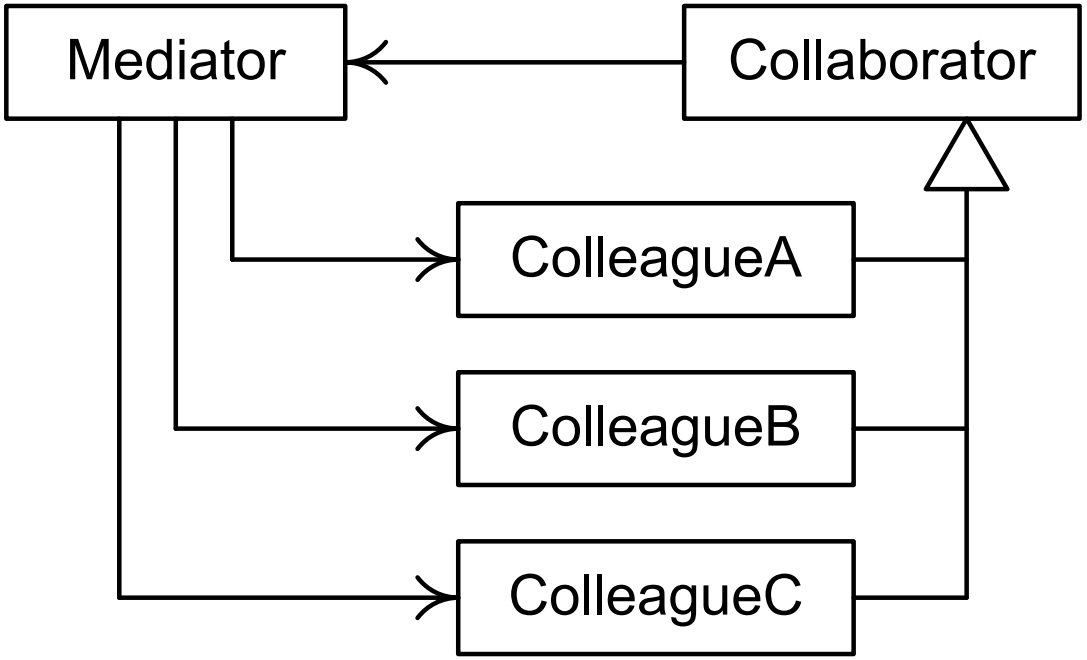
- The Mediator pattern reduces coupling and simplifies code when several objects must negotiate a complex interaction.
- Classes interact only with a mediator class rather than with each other.
- Classes are coupled only to the mediator where interaction control code resides.
- Mediator is like a multi-way Façade pattern.
- Analogy: a meeting scheduler

Using a Mediator

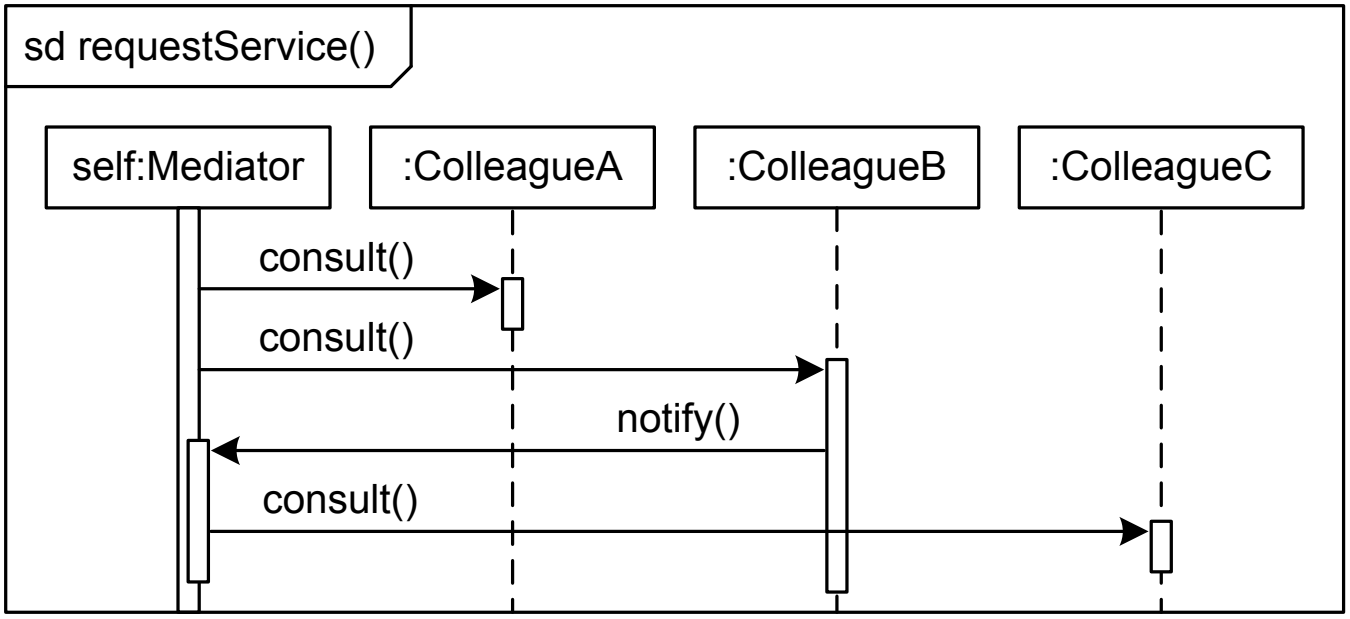


Mediator is like a communication hub – single point of contact and synchronization

Mediator Pattern Structure



Mediator Behavior



When to Use a Mediator

- Use the Mediator pattern when a complex interaction between collaborators must be encapsulated to
  - Decouple collaborators,
  - Centralize control of an interaction, and
  - Simplify the collaborators.
- Using a mediator may compromise performance.

The Adapter/Wrapper Patterns

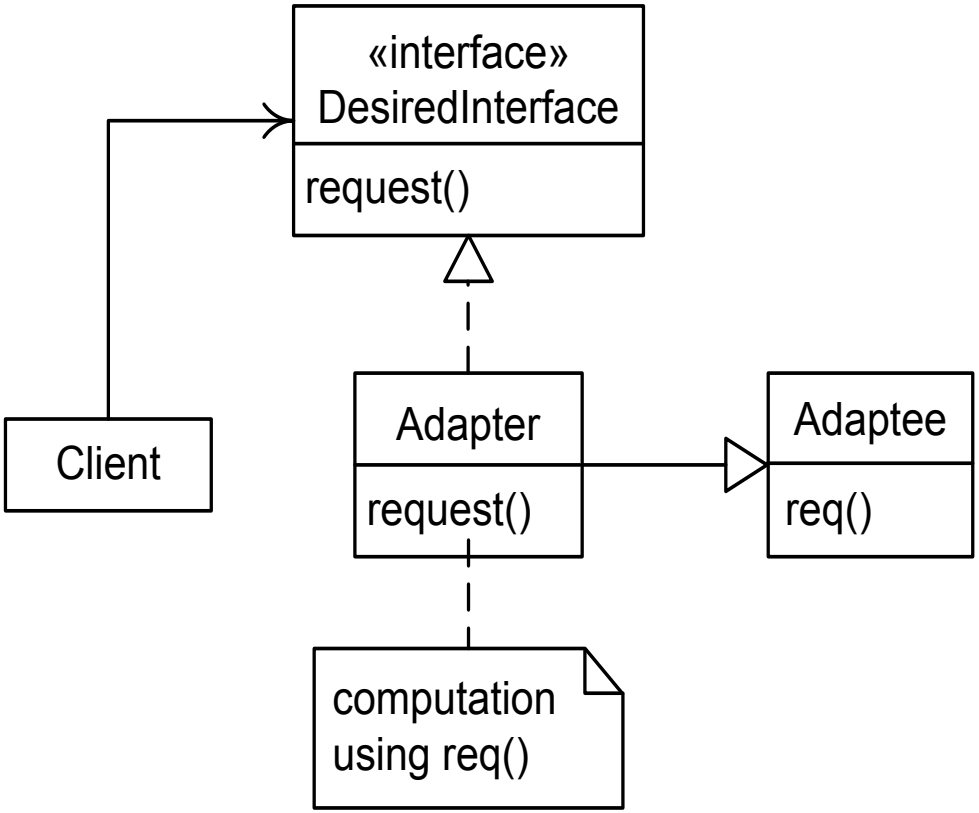
- Often a component has reusable function but not a usable interface.
- An **adapter** or **wrapper** is a component that provides a new interface to an existing component.
- Analogy: electrical or plumbing adapters

Class and Object Adapters

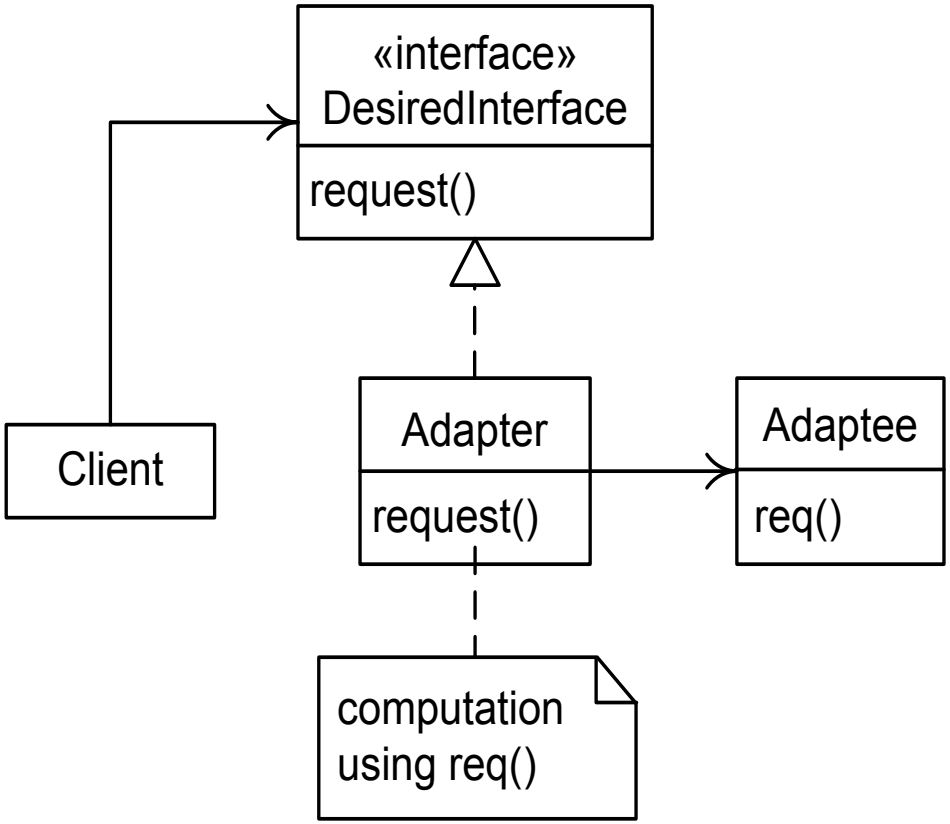
An *adaptee* may be given a new interface by an *adapter* in two ways:

- *Inheritance*—The adapter may sub-class the adaptee; this is the **Class Adapter** pattern
- *Delegation*—The adapter may hold a reference to the adaptee and delegate work to it; this is the **Object Adapter** pattern

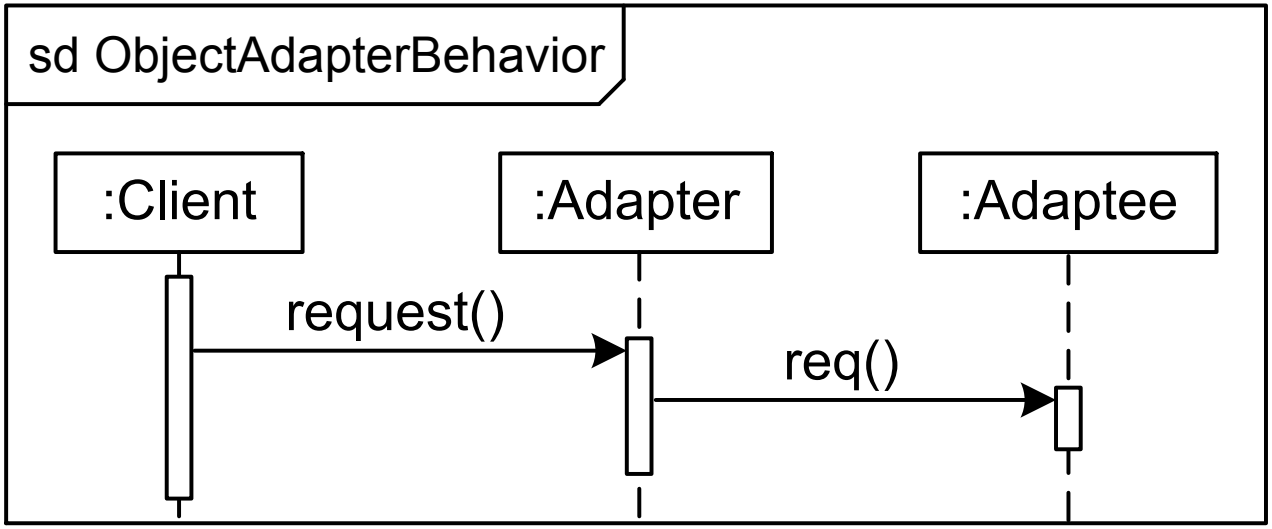
Class Adapter Structure



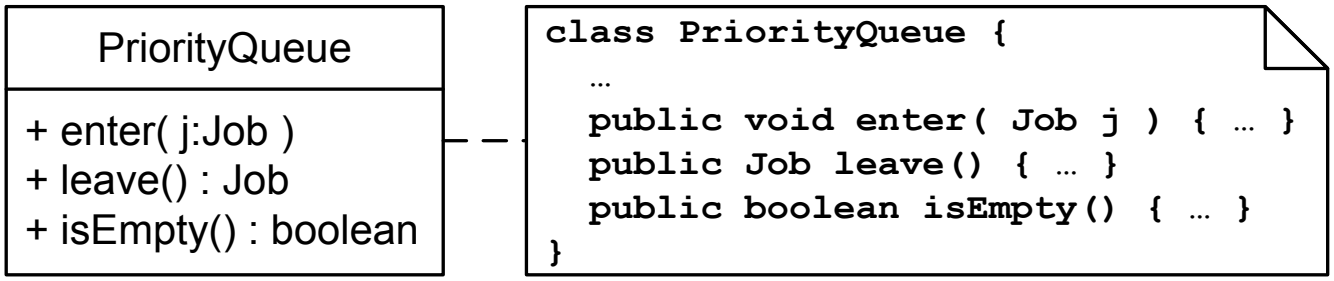
Object Adapter Structure



Object Adapter Behavior

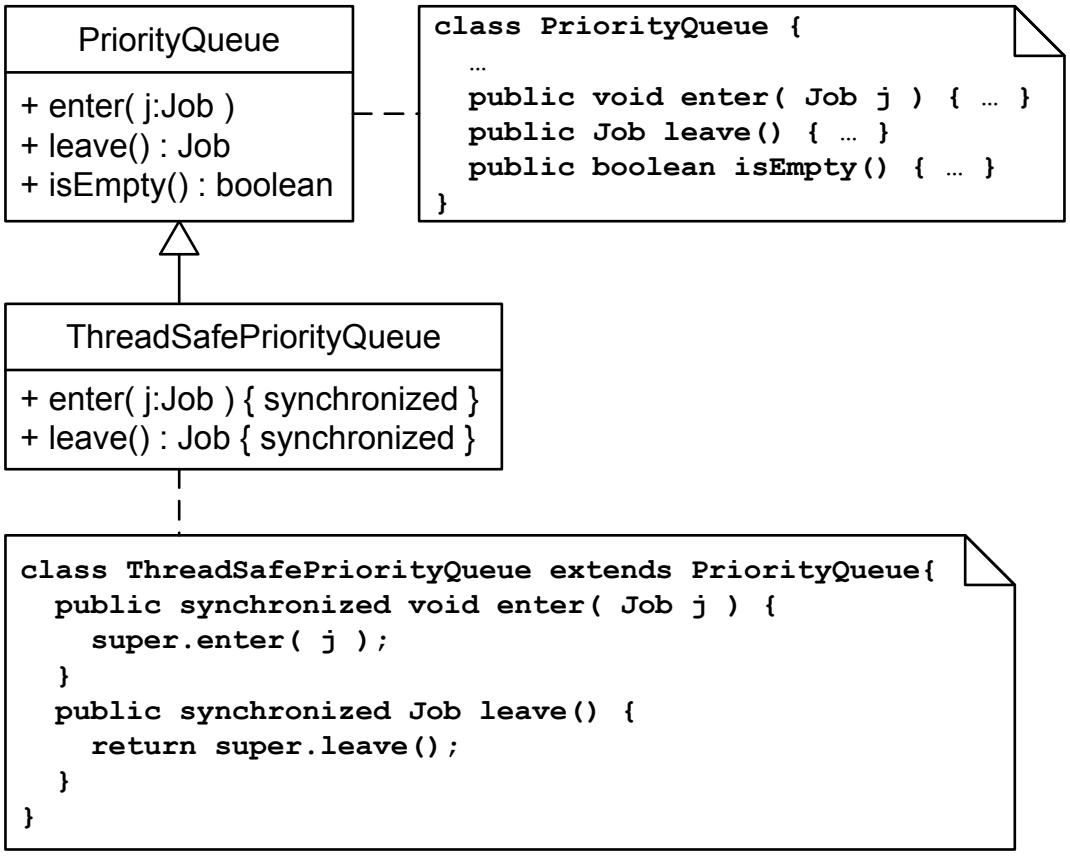


Example: A Thread-Safe Priority Queue—Problem

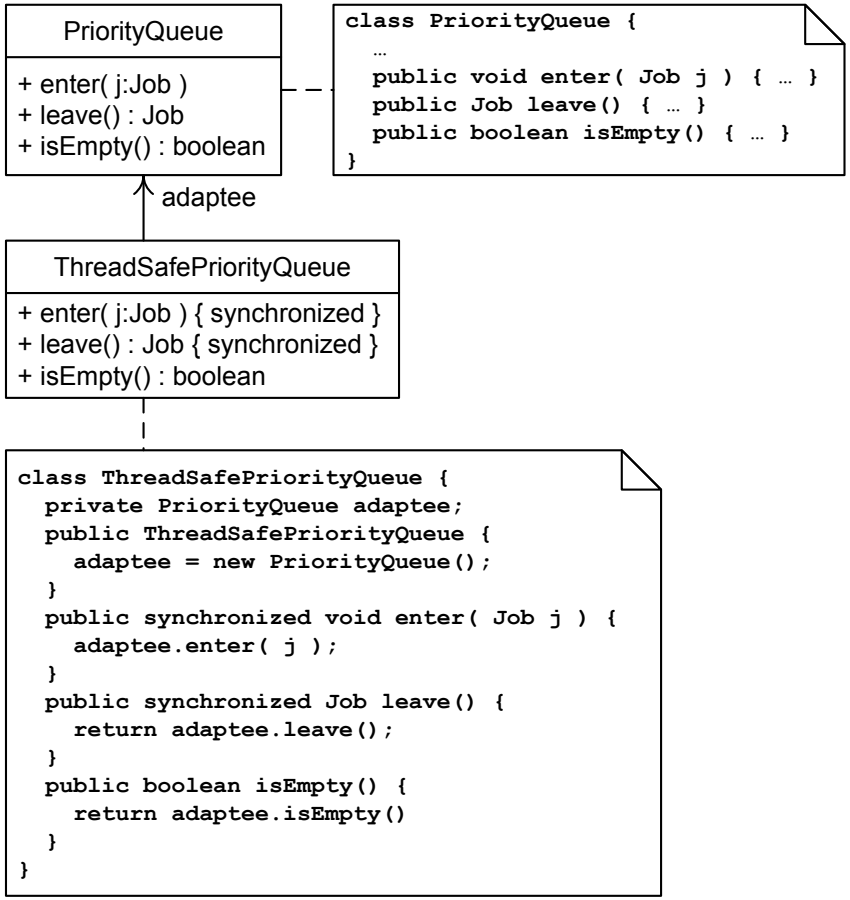


PriorityQueue works properly but is not thread-safe—how can we reuse this class in a thread-safe way?

Example: A Thread-Safe Priority Queue—Class Adapter



Example: A Thread-Safe Priority Queue—Object Adapter



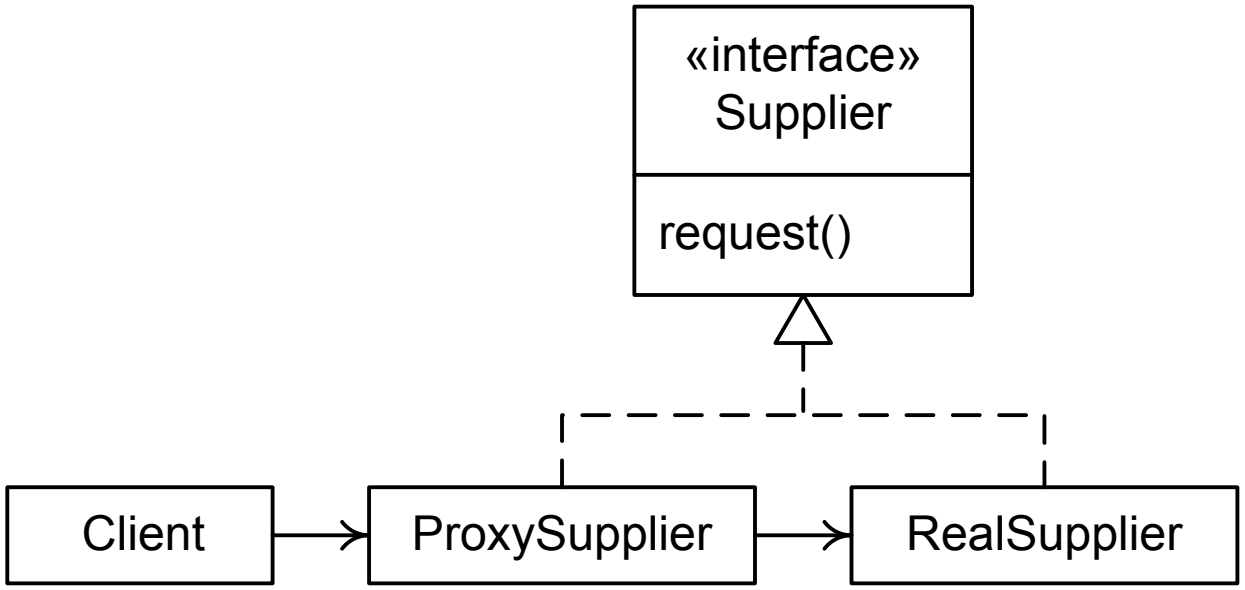
When to Use Adapters

- The current context of use expects a certain interface.
- A simplified interface is needed.
- Operations with slightly different functionality are needed.

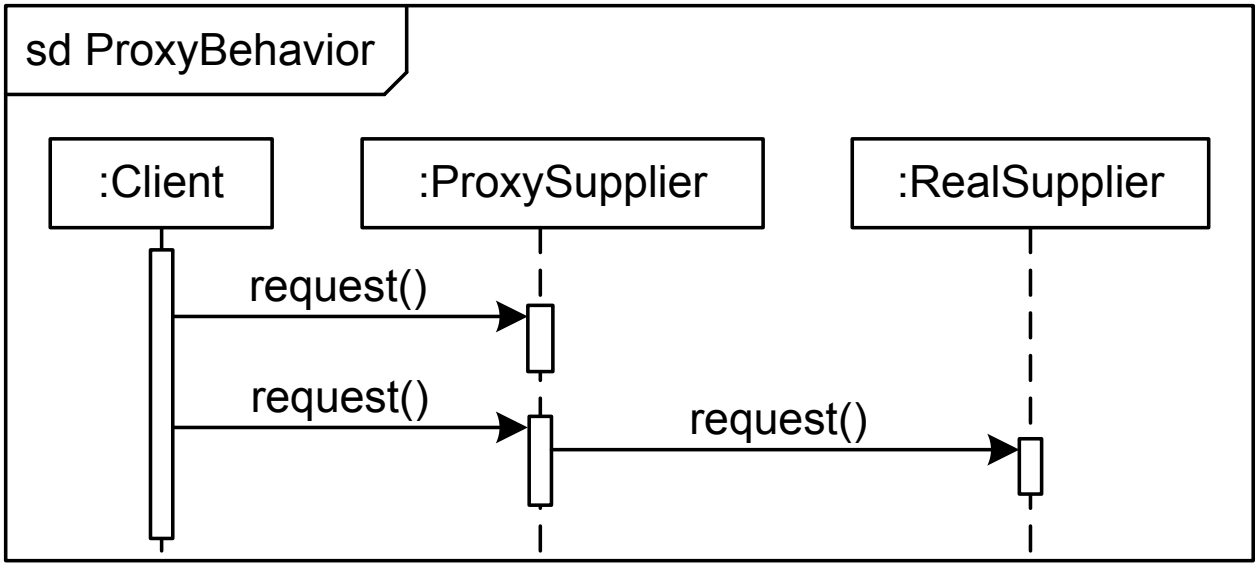
The Proxy Pattern

- Stand-ins for object may be needed because the real object
  - Is not locally available;
  - Is expensive to create; or
  - Needs protected access for security or safety.
- The stand-in must
  - Have the same interface as the real object;
  - Handle as many messages as it can;
  - Delegate messages to the real object when necessary.
- Analogy: a stand-in or proxy

Proxy Pattern Structure



Proxy Pattern Behavior



When to Use Proxies

- Use the Proxy pattern whenever the services provided by a supplier need to be mediated or managed in some way without disturbing the supplier interface.
- Kinds of proxies:
  - *Virtual proxy*—Delay the creation or loading of large or computationally expensive objects
  - *Remote proxy*—Hide the fact that an object is not local
  - *Protection proxy*—Ensure that only authorized clients access a supplier in legitimate ways