CSE 223: Programming -2
08-Structural Design Patterns (I)

Prof. Dr. Khaled Nagi

Department of Computer and Systems Engineering,

Faculty of Engineering, Alexandria University, Egypt.

Agenda

- *Describe common ways that different types of objects can be organized to work with each other*
- *Structural patterns help you compose groups of objects into larger structures, such as complex user interfaces or accounting data.*
- Adapter
- Iterator
- Bridge
- Façade
- Flyweight

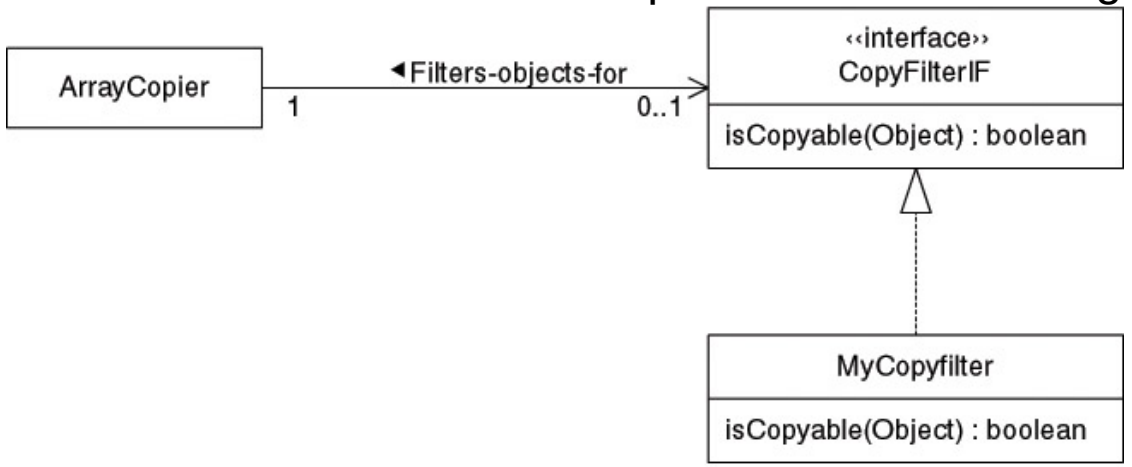
Part I

- Dynamic linkage
- Virtual proxy
- Decorator
- Cache management

Adapter

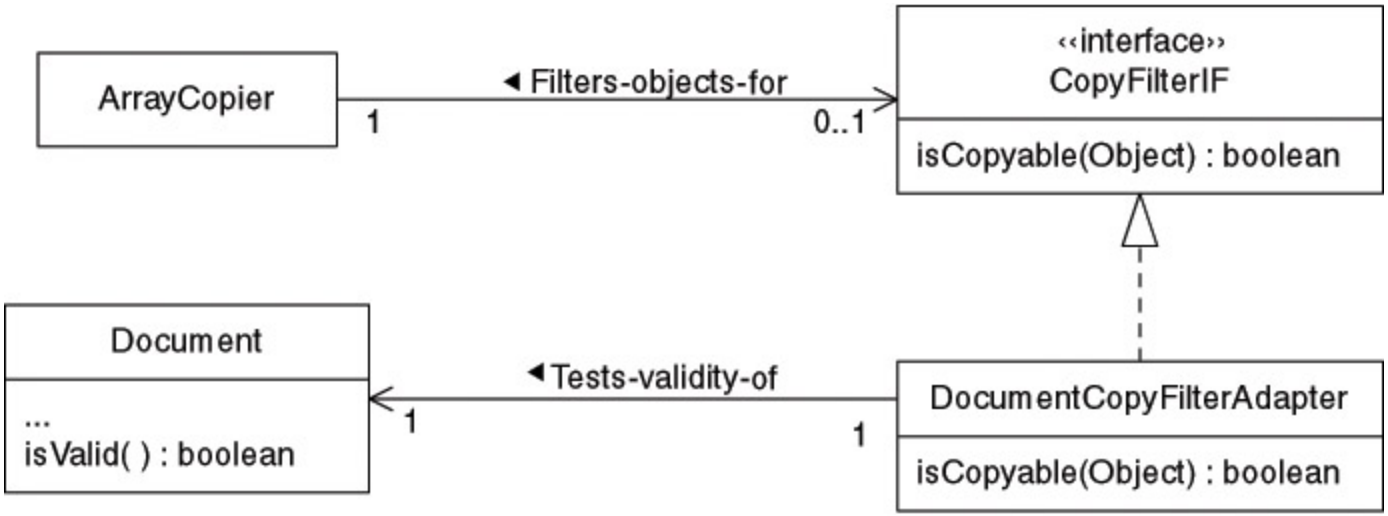
Adapter

- An adapter class implements an interface known to its clients and provides access to an instance of a class not known to its clients.
- An adapter object provides the functionality promised by an interface without having to assume what class is used to implement that interface.
- Example
  - Suppose that you are writing a method that copies an array of objects. The method filters out objects that do not meet certain criteria (the copied array may not contain all of the elements in the original array).
  - To promote reuse, you want the method to be independent of the filtering criteria being used.



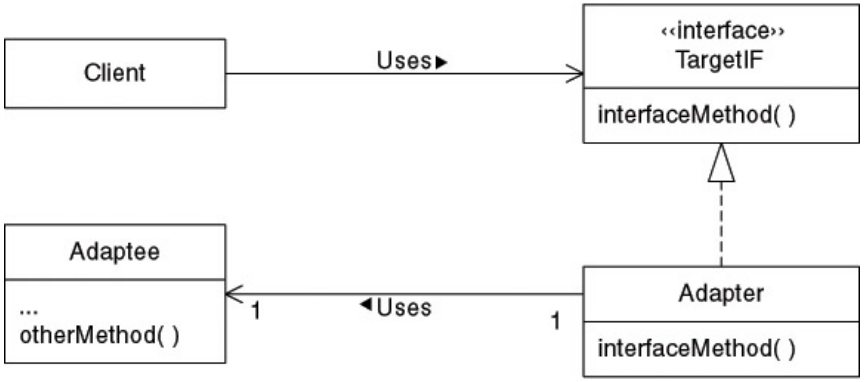
Adapter

- Problem
  - Sometimes the logic needed for filtering is in a method of the objects being filtered. If these objects do not implement the CopyFilterIF interface, then the ArrayCopier object cannot directly ask these objects whether they should be copied.
- Solution



Adapter

- Forces
  - You want to use a class that calls a method through an interface, but you want to use it with a class that does not implement the interface. Modifying the class to implement the interface is not an option either because:
    - You do not have the source code for the class.
    - The class is a general-purpose class and it would be inappropriate for it to implement an interface for a specialized purpose.



Adapter

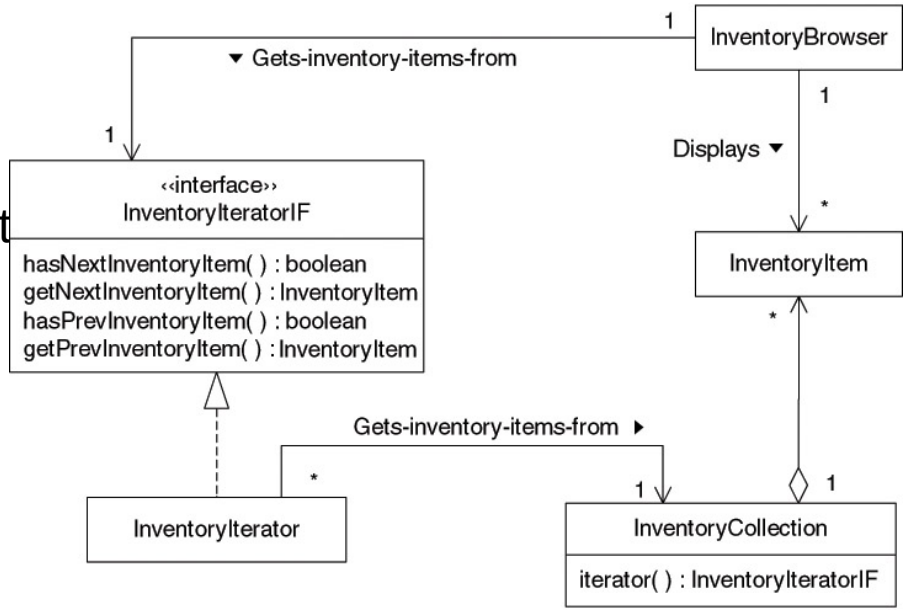
- Implementation Issues
  - Implementation of the adapter class is rather straightforward but how the Adapter objects will know which instance of the Adaptee class to call. There are two approaches:
  - Pass a reference to the Adaptee object as a parameter to the adapter object's constructor or one of its methods. This allows the Adapter object to be used with any instance or possibly multiple instances of the Adaptee class.
  - Make the adapter class an inner class of the Adaptee class.

```
class CustomerBillToAdapter implements AddressIF {
      private Customer myCustomer;
      public CustomerBillToAdapter(Customer customer) {
         myCustomer = customer;       } // constructor
     public String getAddress1() {
         return myCustomer.getBillToAddress1();
    }
     public void setAddress1(String address1) {
         myCustomer.setBillToAddress1(address1);
     } // setAddress1(String)
 …
 } // class CustomerBillToAdapter
```

Iterator Pattern

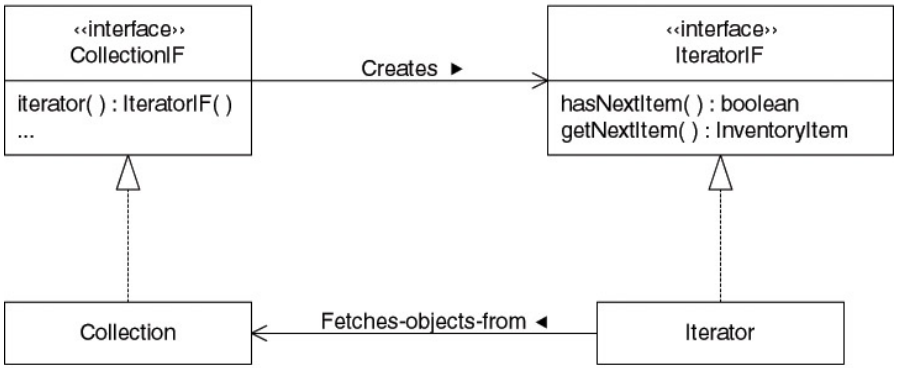
Iterator Pattern

- The Iterator pattern defines an interface that declares methods for sequentially accessing the objects in a collection.
- A class that accesses a collection only through such an interface is independent of:
  - the class that implements the interface
  - the class of the collection
- Example
  - Suppose you are writing classes to browse inventory in a warehouse.
  - There will be a user interface that allows a user to see:
    - the description
    - quantity on hand,
    - location



Iterator Pattern

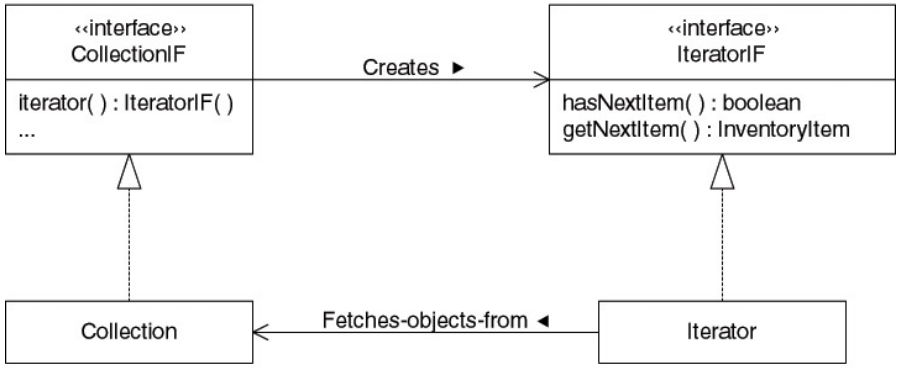
- Forces
  - A class needs access to the contents of a collection without becoming dependent on the class that is used to implement the collection.
  - A class needs a uniform way of accessing the contents of multiple collections.
- Implementation Issues
  - It is common for iterator interfaces to define additional methods when they are useful and the underlying collection classes support them.



  - In many cases, an iterator class's traversal algorithm requires access to a collection class's internal data structure. Thus, iterator classes are often implemented as a private inner class of a collection class.
  - Modifications to collection items while iterating

Iterator Pattern

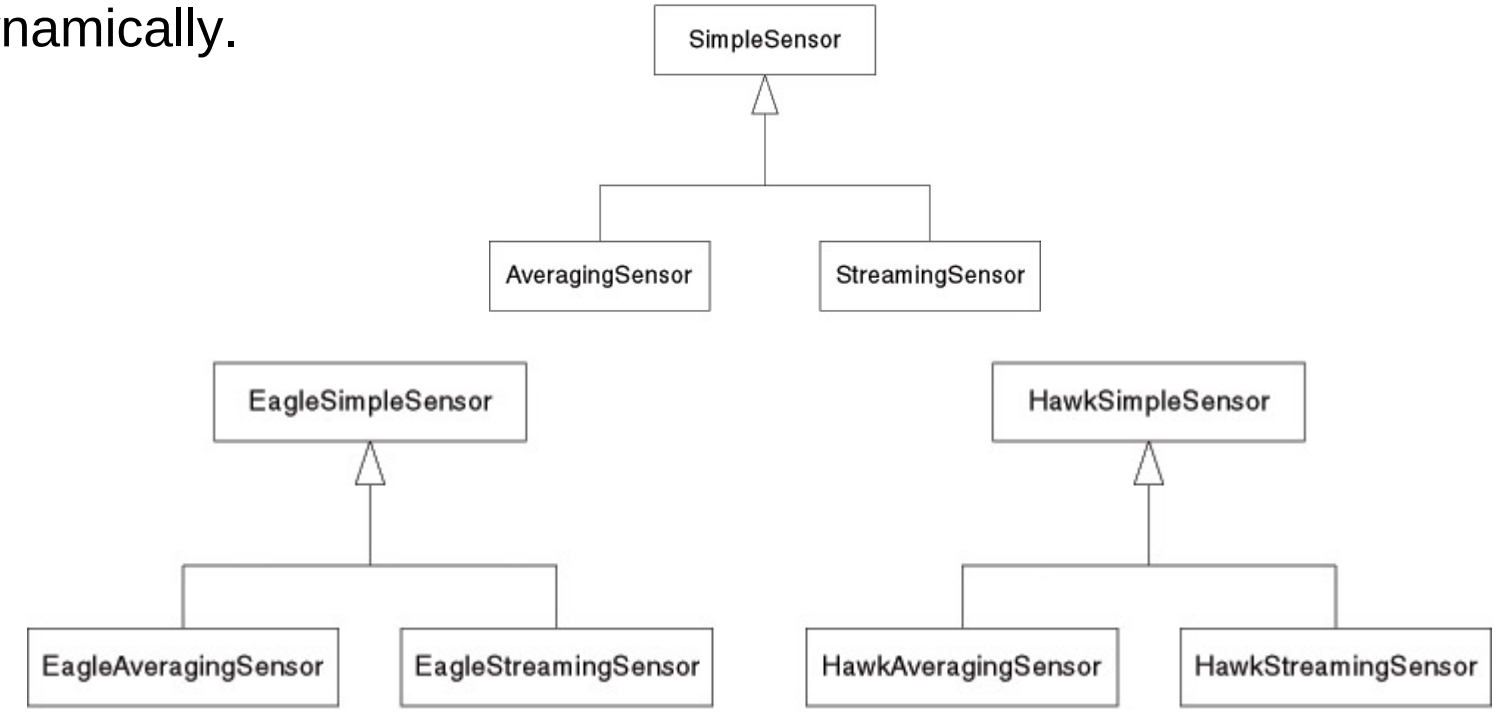
- Consequences
  - It is possible to access a collection of objects without knowing the source of the objects.
  - By using multiple iterator objects, it is simple to have and manage multiple traversals at the same time.
  - A collection class may provide different kinds of iterator objects to traverse the collection in different ways



Bridge Pattern

Bridge Pattern

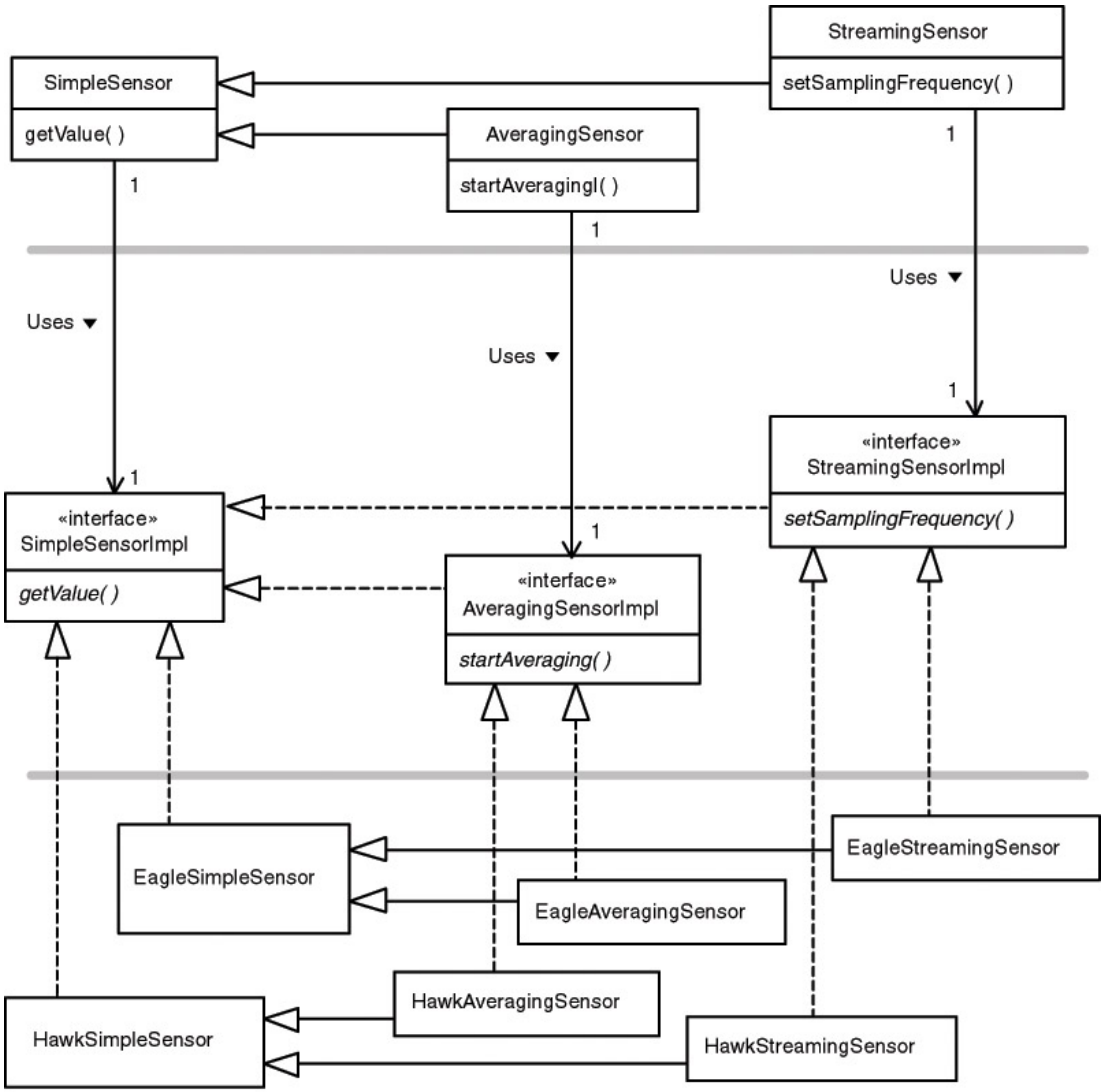
- The Bridge pattern is useful when there is a hierarchy of abstractions and a corresponding hierarchy of implementations.
- Rather than combining the abstractions and implementations into many distinct classes, the Bridge pattern implements the abstractions and implementations as independent classes that can be combined dynamically.



Bridge Pattern

- Problem
  - The problem with this solution is not just that it does not reuse classes for simple, averaging, and streaming sensors.
  - Because it exposes differences between manufacturers to other classes, it forces other classes to recognize differences between manufacturers and therefore be less reusable.
  - The challenge here is to represent a hierarchy of abstractions in a way that keeps the abstractions independent of their implementations.

Bridge Pattern



Bridge Pattern

- Forces
  - When you combine hierarchies of abstractions and hierarchies of their implementations into a single class hierarchy, classes that use those classes become tied to a specific implementation of the abstraction.
  - Changing the implementation used for an abstraction should not require changes to the classes that use the abstraction.
  - You would like to reuse logic common to different implementations of an abstraction. The usual way to make logic reusable is to encapsulate it in a separate class.
  - You would like to be able to create a new implementation of an abstraction without having to re-implement the common logic of the abstraction.
  - You would like to be able to extend the common logic of an abstraction by writing one new class rather than writing a new class for each combination of the base abstraction and its implementation.
  - When appropriate, multiple abstractions should be able to share the same implementation.

Bridge Pattern

- Implementation Issues
  - The most basic decision to make is whether:
    - abstraction objects will create their own implementation objects or
    - delegate the creation of their implementation objects to another object.
- Consequences
  - The Bridge pattern keeps classes that represent an abstraction independent of the classes that supply an implementation for the abstraction.
  - The abstraction and its implementations are organized into separate class hierarchies.
  - You can extend each class hierarchy without directly impacting another class hierarchy.
  - Classes that are clients of the abstraction classes do not have any knowledge of the implementation classes, so an abstraction object can change its implementation without any impact on its clients.

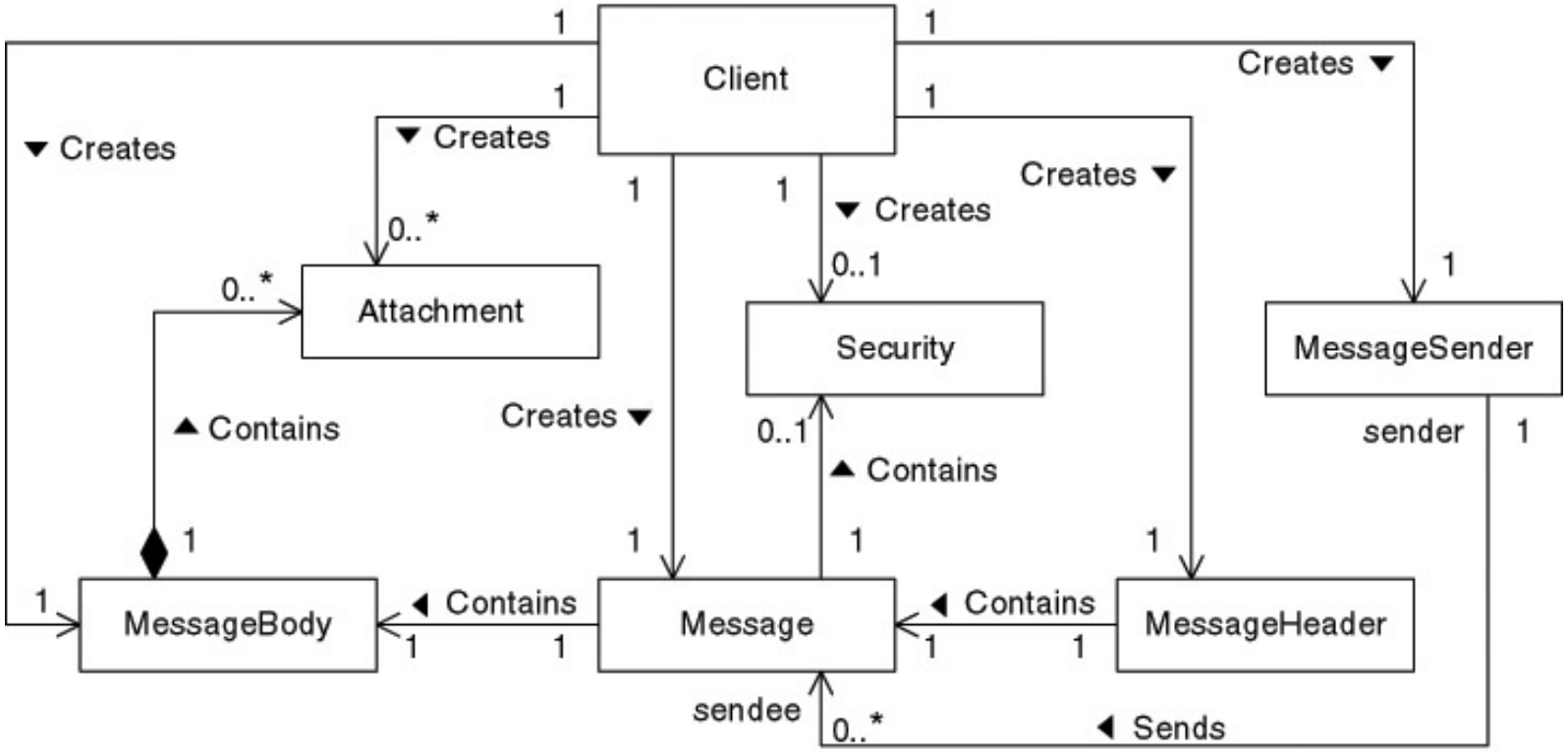
Façade Pattern

Façade Pattern

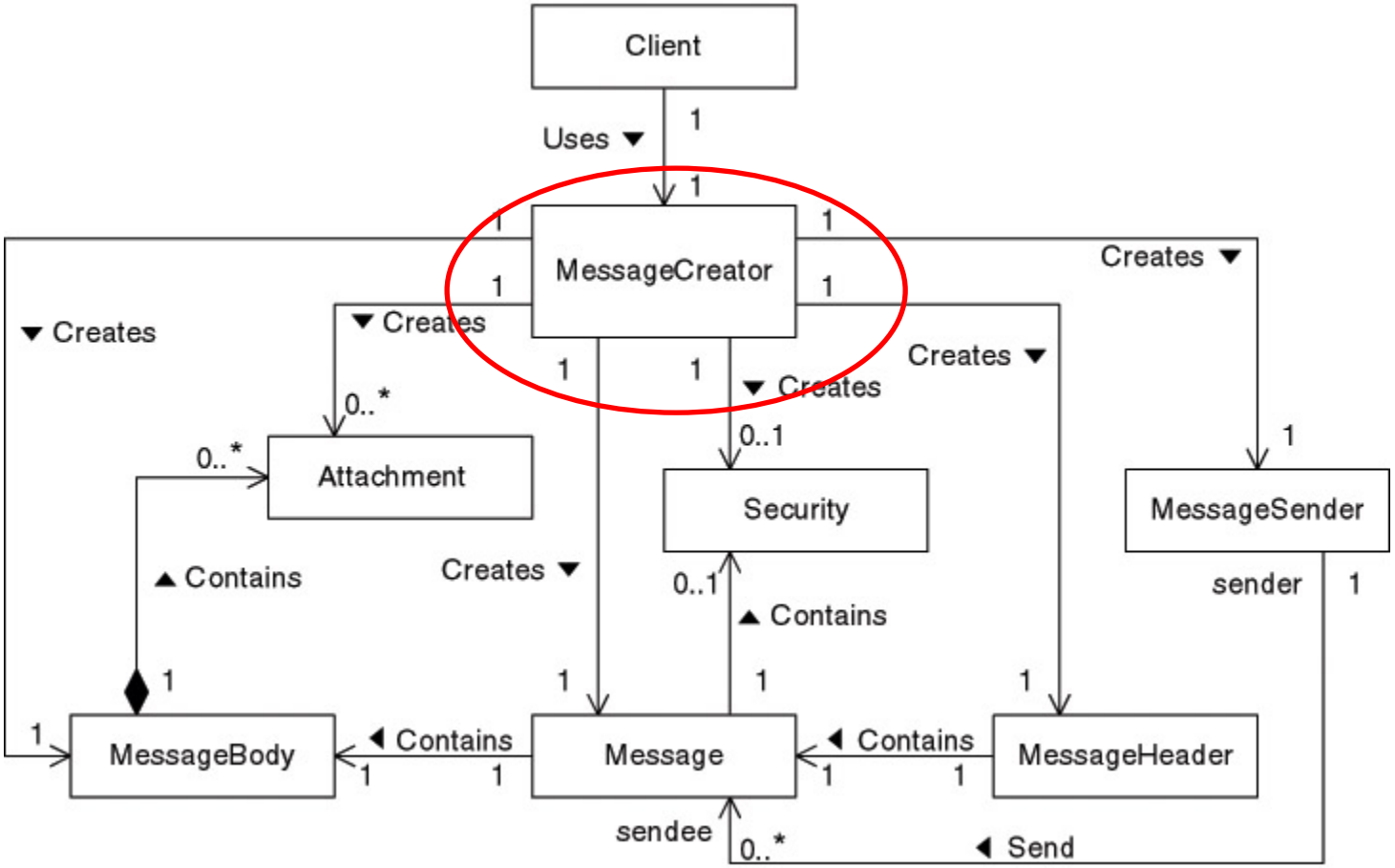
- The Façade pattern simplifies access to a related set of objects by providing one object that all objects outside the set use to communicate with the set.
- Example: *creation* and *sending* of email messages. Classes may include the following:
  - A **MessageBody** class whose instances will contain message bodies
  - An **Attachment** class whose instances will contain message attachments that can be attached to a **MessageBody** object
  - A **MessageHeader** class whose instances will contain header information (to, from, subject, etc.) for an email message
  - A **Message** class whose instances will tie together a **MessageHeader** object and a **MessageBody** object
  - A **Security** class whose instances can be used to add a digital signature to a message
  - A **MessageSender** class whose instances are responsible for sending Message objects

Façade Pattern

- A client must know of at least six classes, the relationships between them, and the order in which it must create instances of the classes.



Façade Pattern - Solution



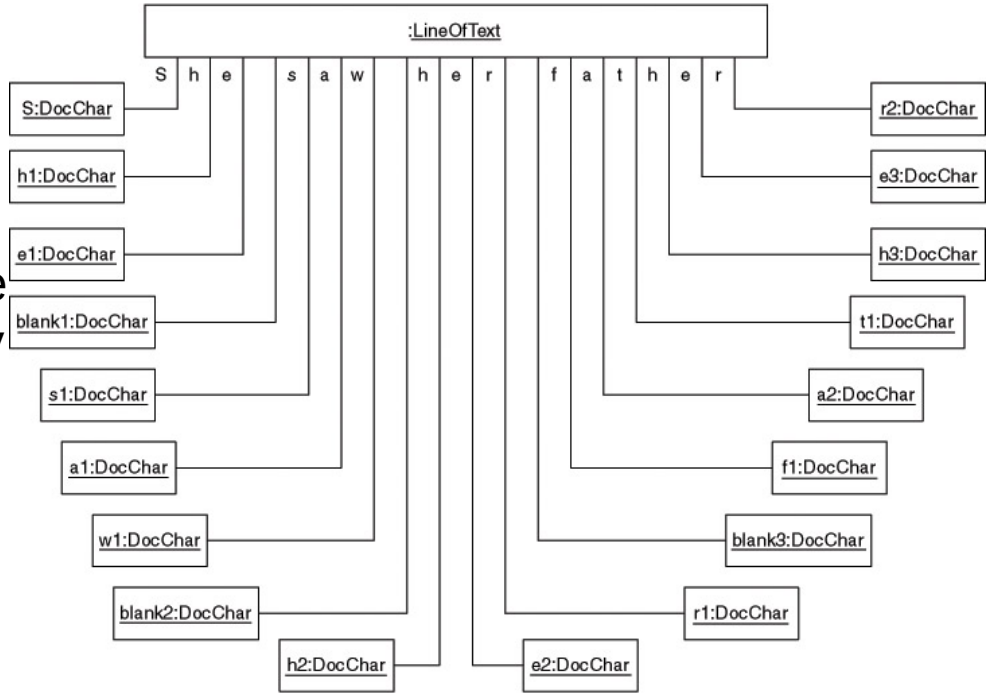
Façade Pattern

- Forces
  - There are many dependencies between classes that implement an abstraction and their client classes. The dependencies add noticeable complexity to clients.
  - You want to simplify the client classes, because simpler classes result in fewer bugs. Simpler clients also mean that less work is required to reuse the classes that implement the abstraction.
  - You are designing classes to function in cleanly separated layers. You want to minimize the number of classes that are visible from one layer to the next.
  - Less documentation: properly document the Façade.
  - More freedom in hiding the internals.
  - Elimination of the coupling between a client class and the classes that implement an abstraction

Flyweight

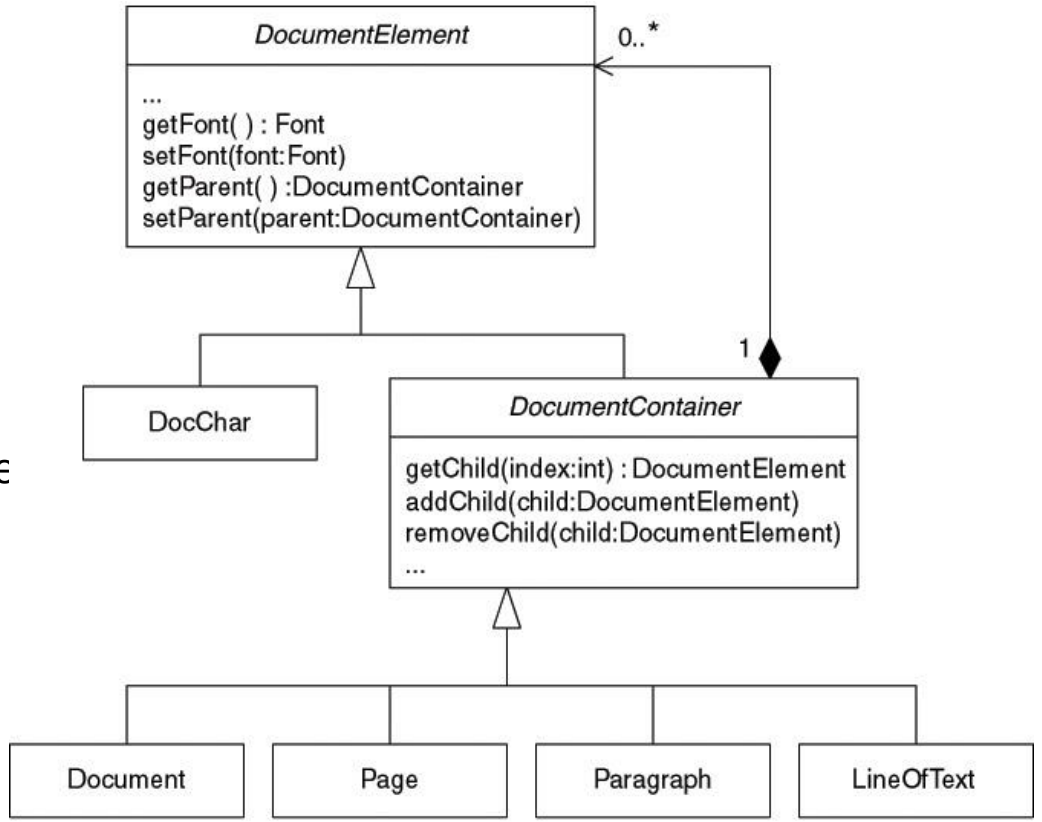
Flyweight

- If instances of a class that contain the same information can be used interchangeably, the Flyweight pattern allows a program to avoid the expense of multiple instances that contain the same information by sharing one instance.
- Problem
  - As you can see, the characters “h,” “e,” “ ,” “a,” and “r” are used multiple times.
  - In an entire document, all of the characters typically occur many times.
  - It is possible to reorganize the objects so that one DocChar object is used to represent all occurrences of the same character.

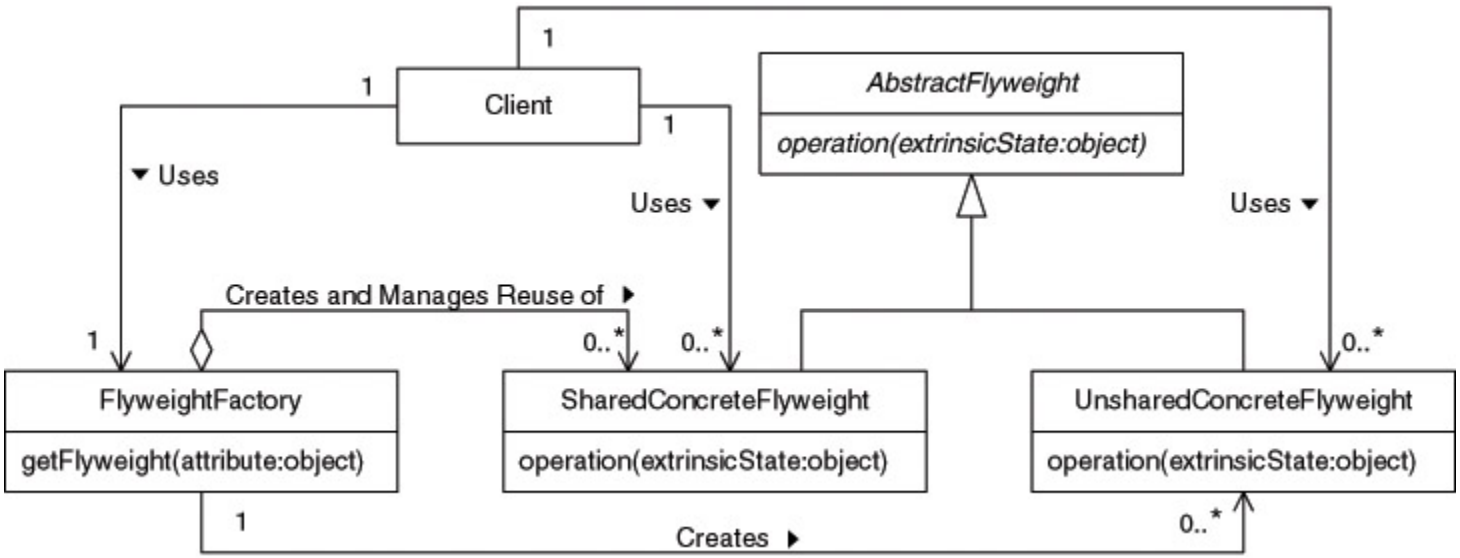


Flyweight

- Solution
- Forces
  - You have an application that uses a large number of similar objects.
  - You want to reduce the memory overhead of having a large number of similar objects.
  - The program does not rely on the object identity of any of the objects that you want it to share.
  - The more things that can be represented with the same object, the greater the memory savings.



Flyweight



- Implementation Issues and Consequences
  - There is a trade-off to make between the number of attributes you make extrinsic and the number of flyweight objects needed at runtime. The more attributes you make extrinsic, the fewer flyweight objects will be needed. The more attributes you make intrinsic, the less time it will take objects to access their attributes.
  - Using shared flyweight objects can drastically reduce the number of objects in memory but at the cost of increasing runtime and loosing object identities