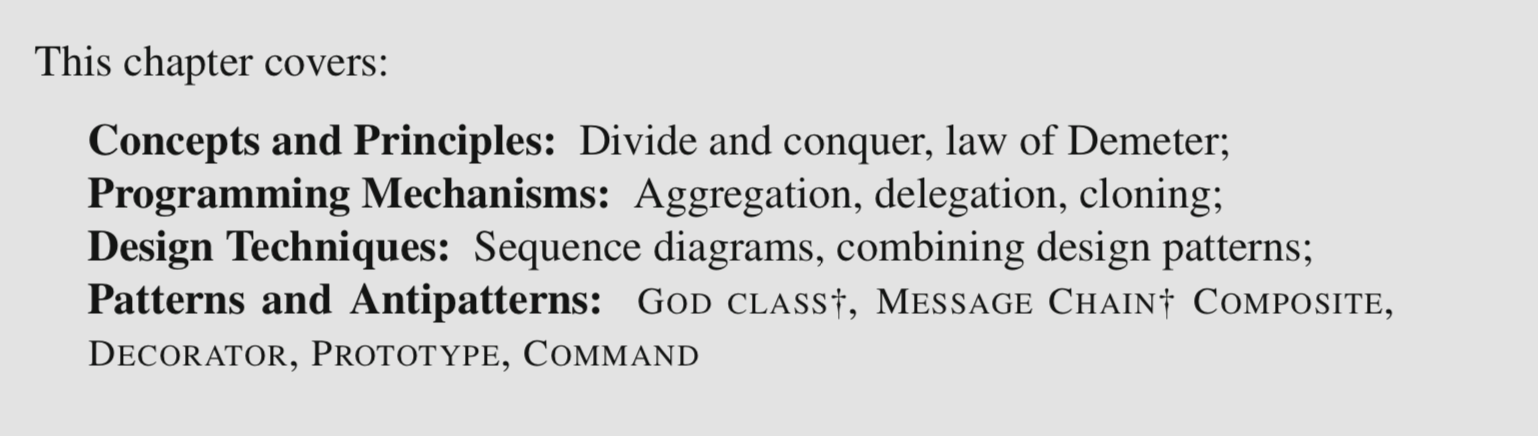
Chapter 6

Composition



6.1 Composition and Aggregation

A general strategy for managing complexity in software design is to define larger abstractions in terms of smaller ones. This is an example of the general “divide and conquer” problem-solving strategy. In practice one way to assemble different pieces of code, data, or computation, is through object composition. **For an object to be composed of other objects means that one object stores a reference to one or more other objects**. Composition is a way to provide a solution to two common software design situations.

* In a first situation, we have one abstraction whose intrinsic *representation* is that of a collection of other abstractions. One term often used to refer to the object that is composed of other objects is the aggregate, whereas the objects being aggregated are the elements. [essential parts of the aggregate object]

e.g. Deck is an aggregate of Card, String is an aggregate

* A second situation in which composition is helpful is to break down a class that would otherwise be too big and complex.

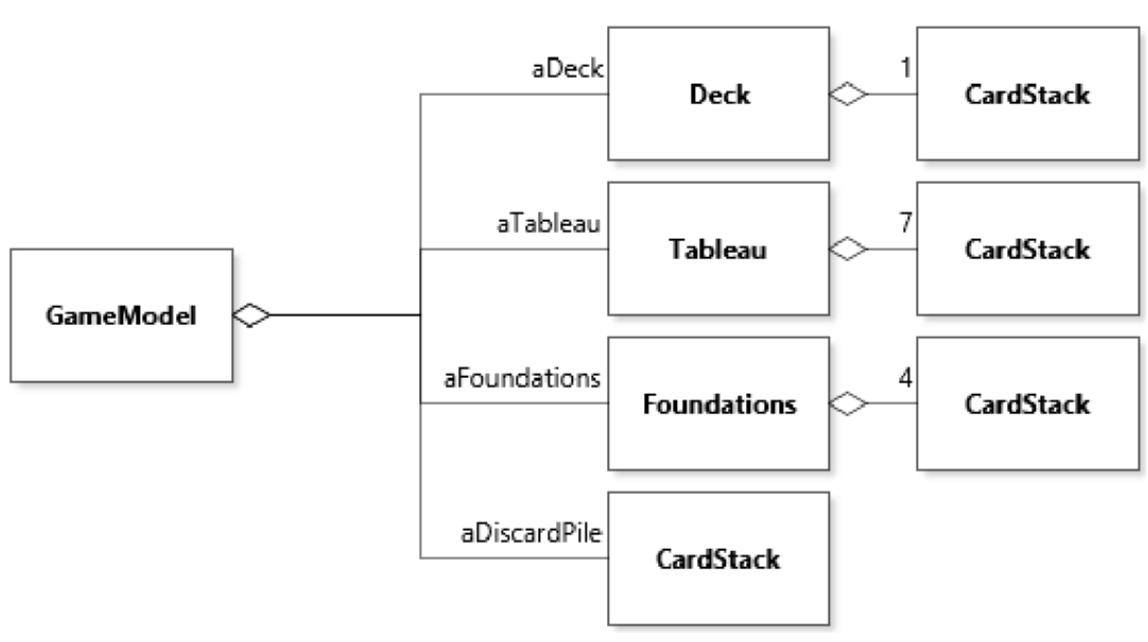
GOD CLASS†: an unmanageable class that knows everything and does everything.

To avoid god classes and similar design degradation, we can use composition to support a mechanism of *delegation*. The idea of delegation is that **the aggregate object delegates some services to the objects that serve a role of specialized service to the aggregate**. This looser form of composition is also known as aggregation.

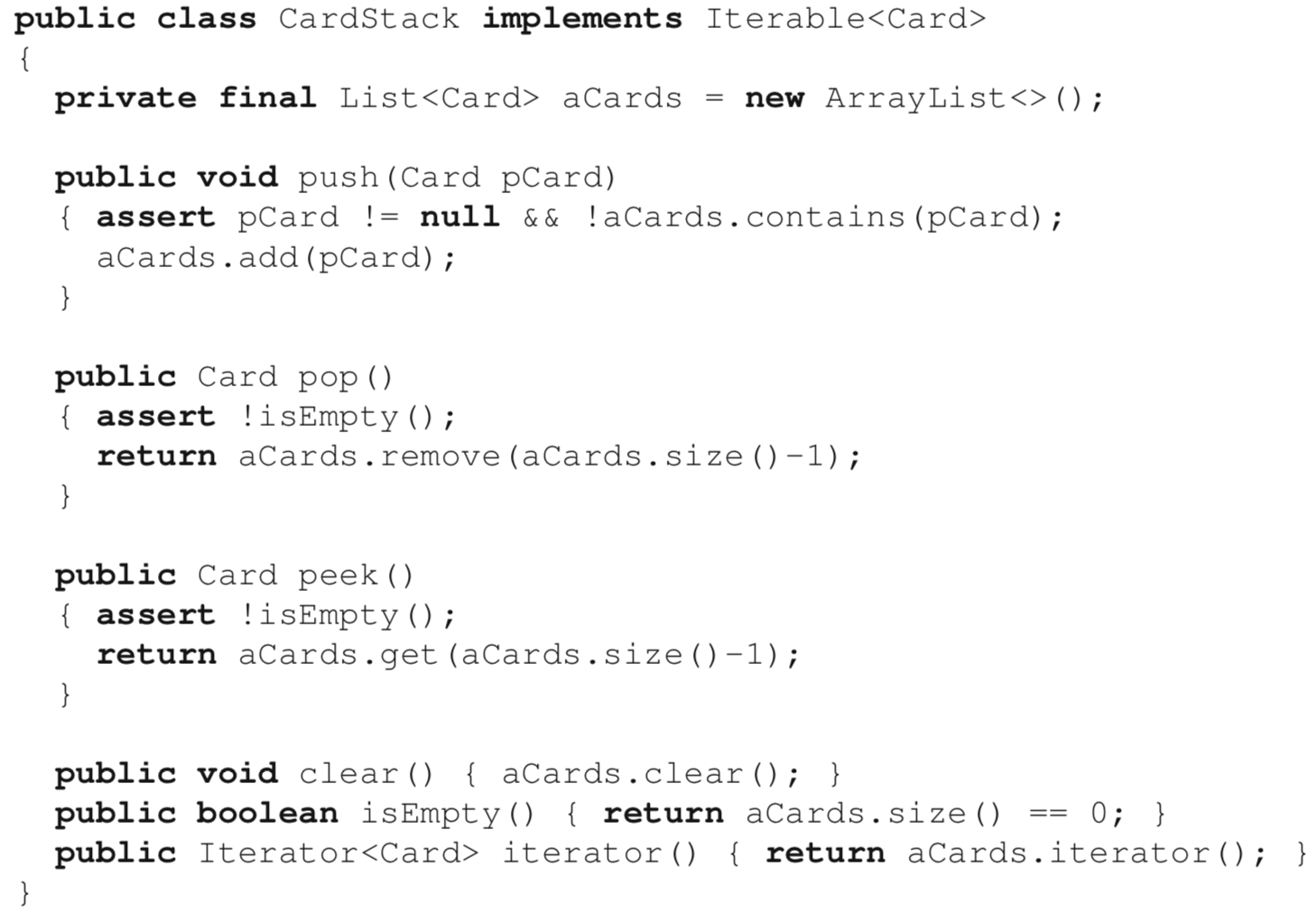
[service providers for the aggregate object]

One important property of composition is that it is transitive. An object that is composed of other objects can, itself, be one component or delegate of another parent object. Ultimately, many structures in object-oriented programs are *object graphs* that group simpler component and delegate objects into progressively more and more complex aggregates.

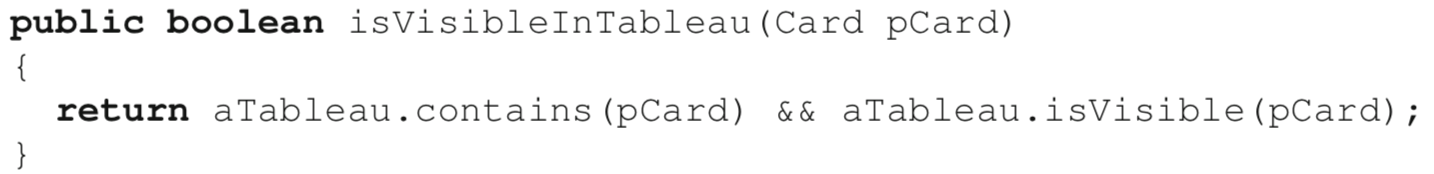
Composition relations are represented using the white diamond decoration. Note that the diamond is on the side of the aggregate. Normally, in a class diagram, model elements that represent a given class are not repeated. In this diagram I took the liberty of repeating CardStack for clarity. All CardStack elements, however, represent the same class.



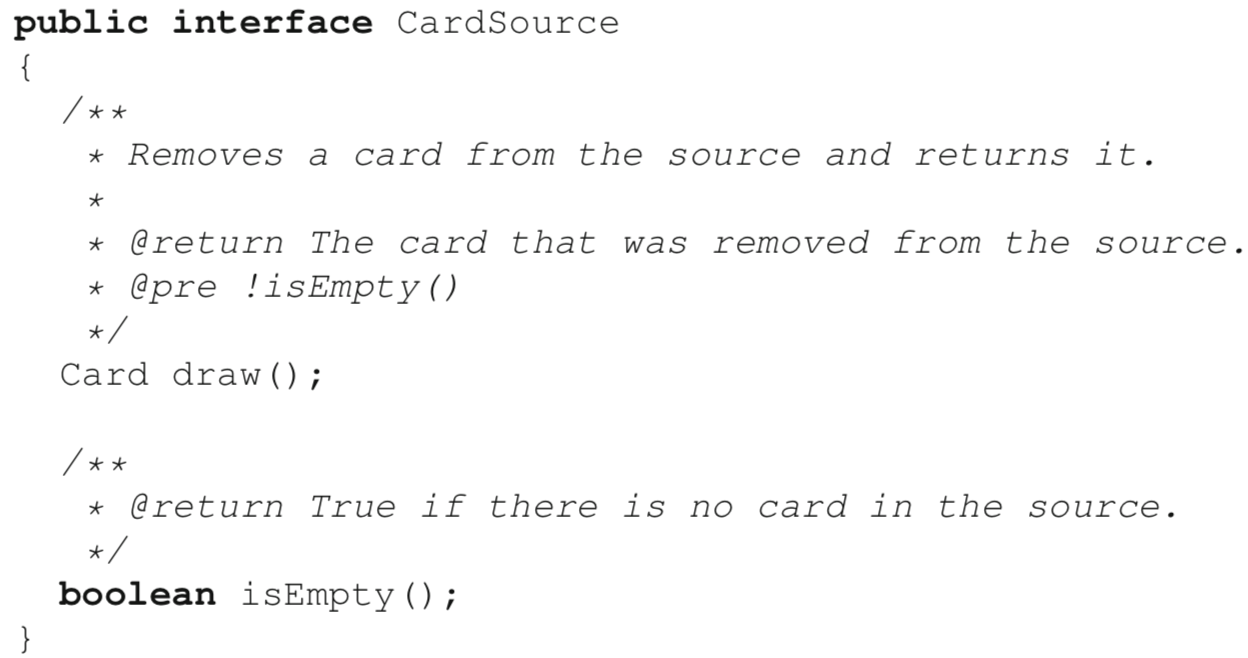
A first thing to observe is that in this version of the code, instead of having a Deck class aggregate Card objects using the List library type, I used composition to define a tighter type CardStack that provides a narrow interface dedicated to handling stacks of cards.



e.g. class GameModel needs a method isVisibleInTableau(Card) to determine whether a card is face up or down in the game tableau 🡪 delegated to class Tableau:



6.2 The Composition Design Pattern



[Class Definition]

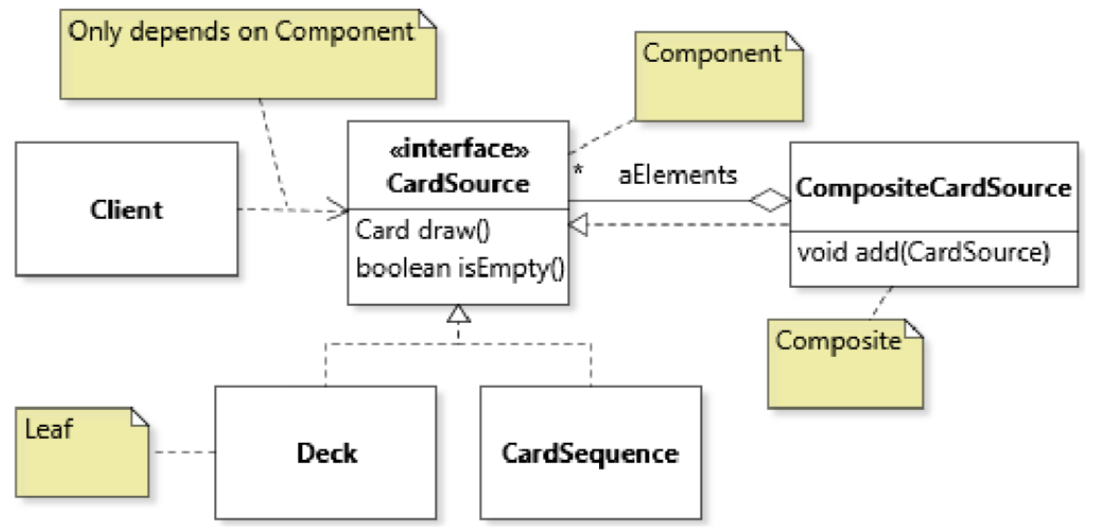


The main feature of this design decision is that the set of possible implementations of CardSource is *specified statically (in the source code), as opposed to dynamically (when the code runs)*. Three major limitations of this static structure are:

* The number of possible structures of interest can be very large. As illustrated by the fifth definition, DeckAndFourAces, supporting all possible configurations leads to a *combinatorial explosion* of class definitions.
* Each option requires a class definition, even if it is used very rarely. This clutters the code unnecessarily, because most implementations would probably look very similar.
* In running code, it is very difficult to accommodate the situation where a type of card source configuration is needed that was not anticipated before launching the application.

[Object Composition – open-ended configuration]

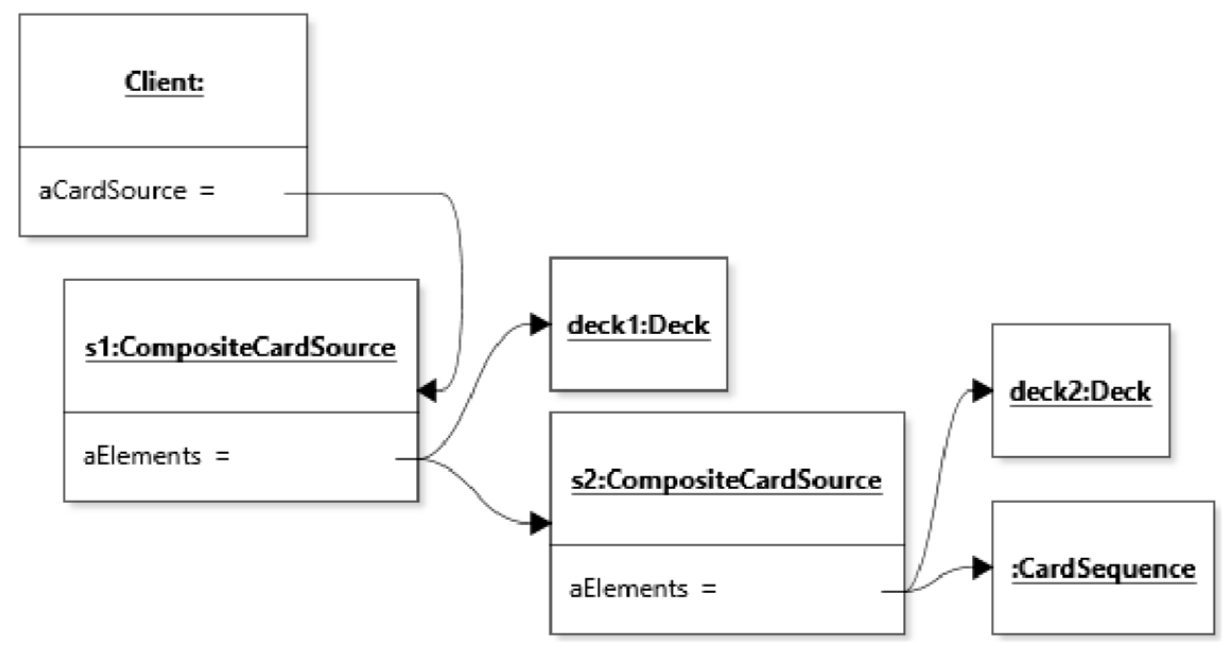
The fundamental idea to support this approach is to define a class that represents multiple CardSources while still behaving like a single one.



The composite element has two important features:

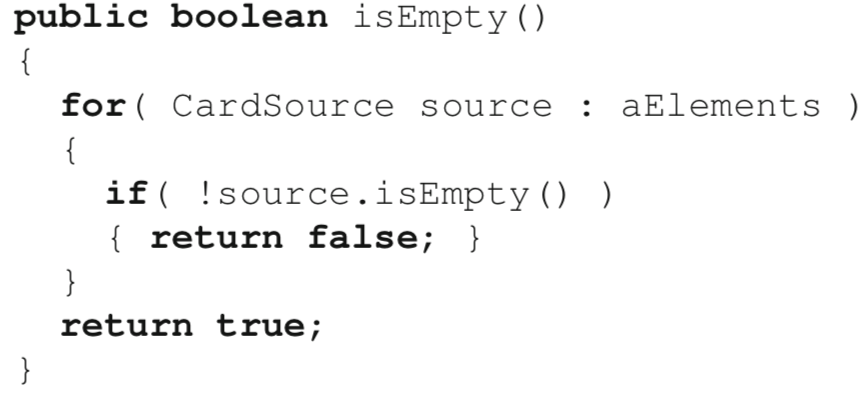
* It aggregates a number of different objects of the component type (CardSource in our case). Using the component interface type is important, as *it allows the composite to compose any other kind of elements, including other composites*. In our application, a composite CardSource can aggregate any kind of CardSource: instances of Deck, CardSequence, or anything else that implements CardSource.
* It implements the component interface. This is basically what allows composite objects to be treated by the rest of the code in exactly the same way as leaf elements.

The diagram also captures the important insight that for the COMPOSITE to be effective, client code should depend primarily on the component type, and not manipulate concrete types directly.



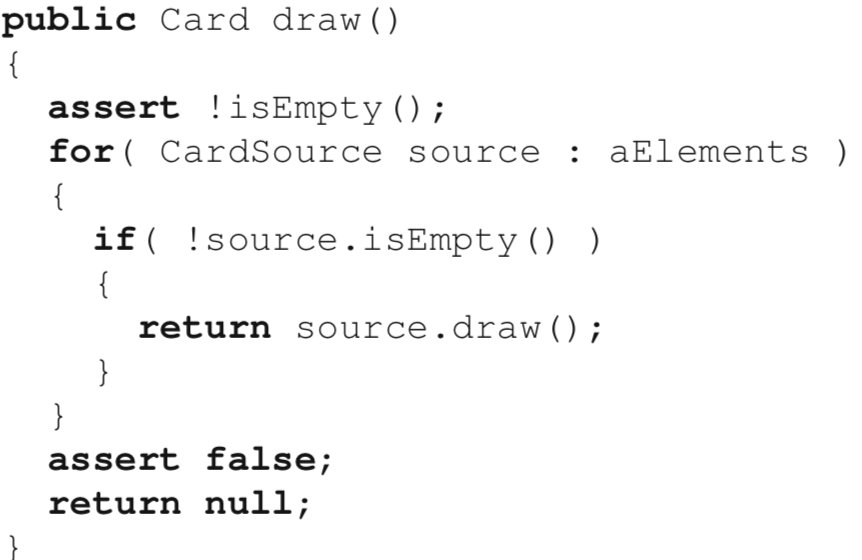
When applying the COMPOSITE as part of a design, the implementation of the methods of the component interface will generally involve an iteration through all the aggregated elements.

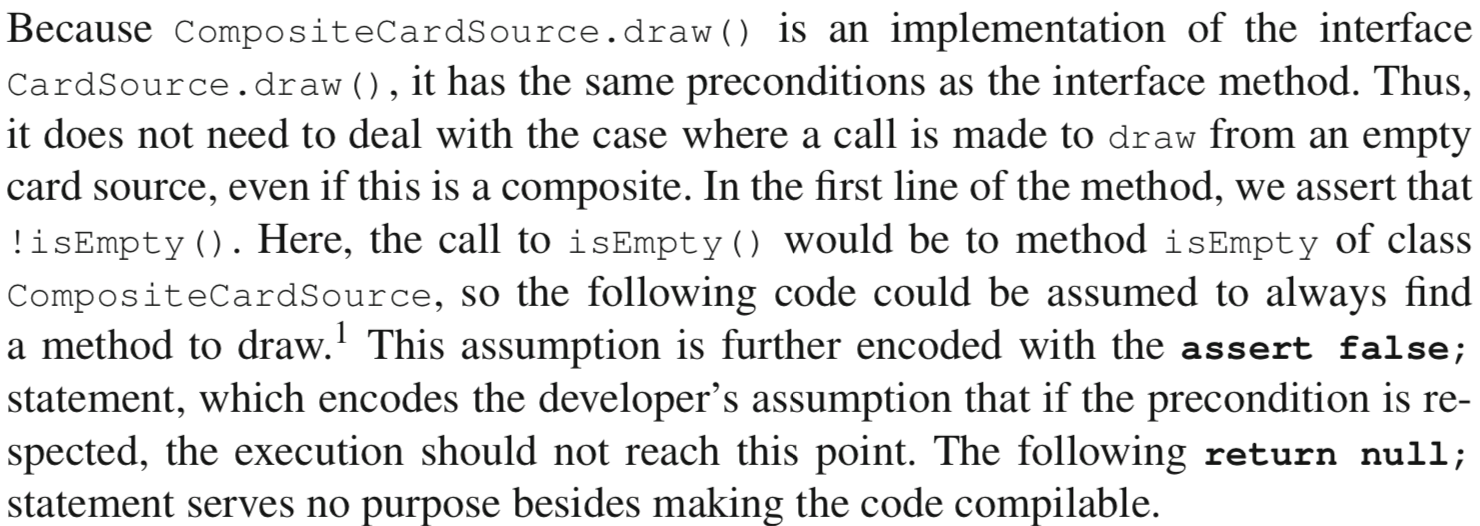
e.g. CompositeCardSource.isEmpty()



e.g. draw()

Instead of delegating the method call to all elements, we only need to iterate until we can find one card to draw.





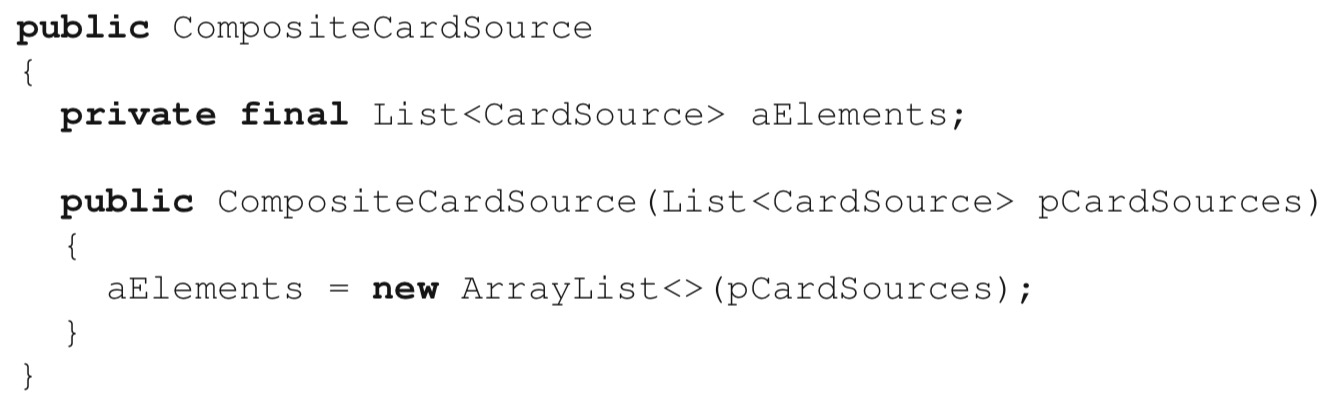
[how to add to the composite the instances of the component that it composes]

* add method (as part of the composite’s interface) (see UML)

In turn, this strategy leads to a second design question, which is whether to include the add method in the component or not. The more common solution is to not include it in the component, but there may be some situations where it makes more sense to include it on the interface of the component so that the component and all its children have the same interface.

* constructor

For example, we could pass a list of card sources as input:



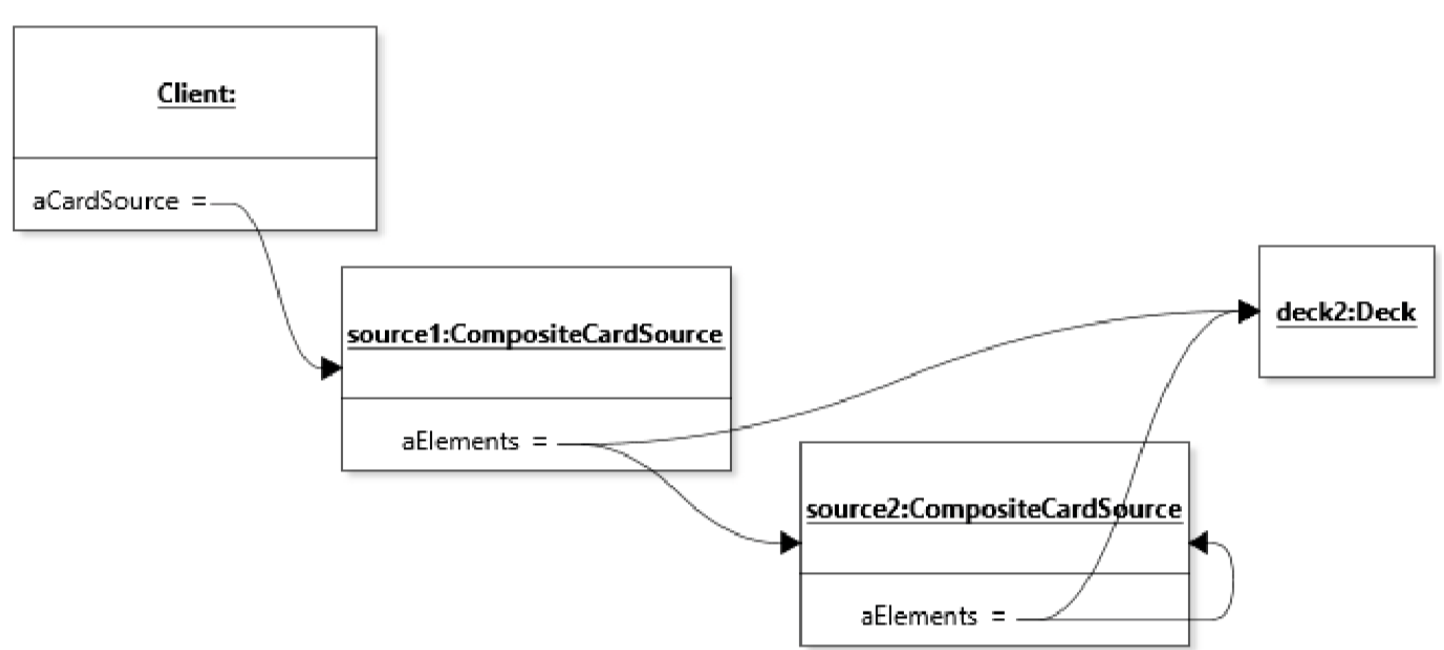
Here we use of the copy constructor, to avoid leaking a reference to the private collection structure

The main reason for adopting the “add method” strategy is if we need to modify the state of the composite at run time. However, this comes at a cost in terms of design structure and code understandability, because we need to deal with a more complex life-cycle for the composite object and have to manage the difference between the interface of the component (which does not have the add method) and the one of the composite (which does). If run-time modification of the composite is not necessary, then it is likely a better option to initialize the composite once and leave it as is. In the context of the CardSource example, it would not result in an immutable composite (we still draw cards), but in other contexts immutability may be an additional advantage.

Some practical aspects related to using the pattern are independent from the structure of the pattern itself. These include:

* The location of the creation of the composite in client code;
* The logic required to preserve the integrity of the object graph induced by this design.

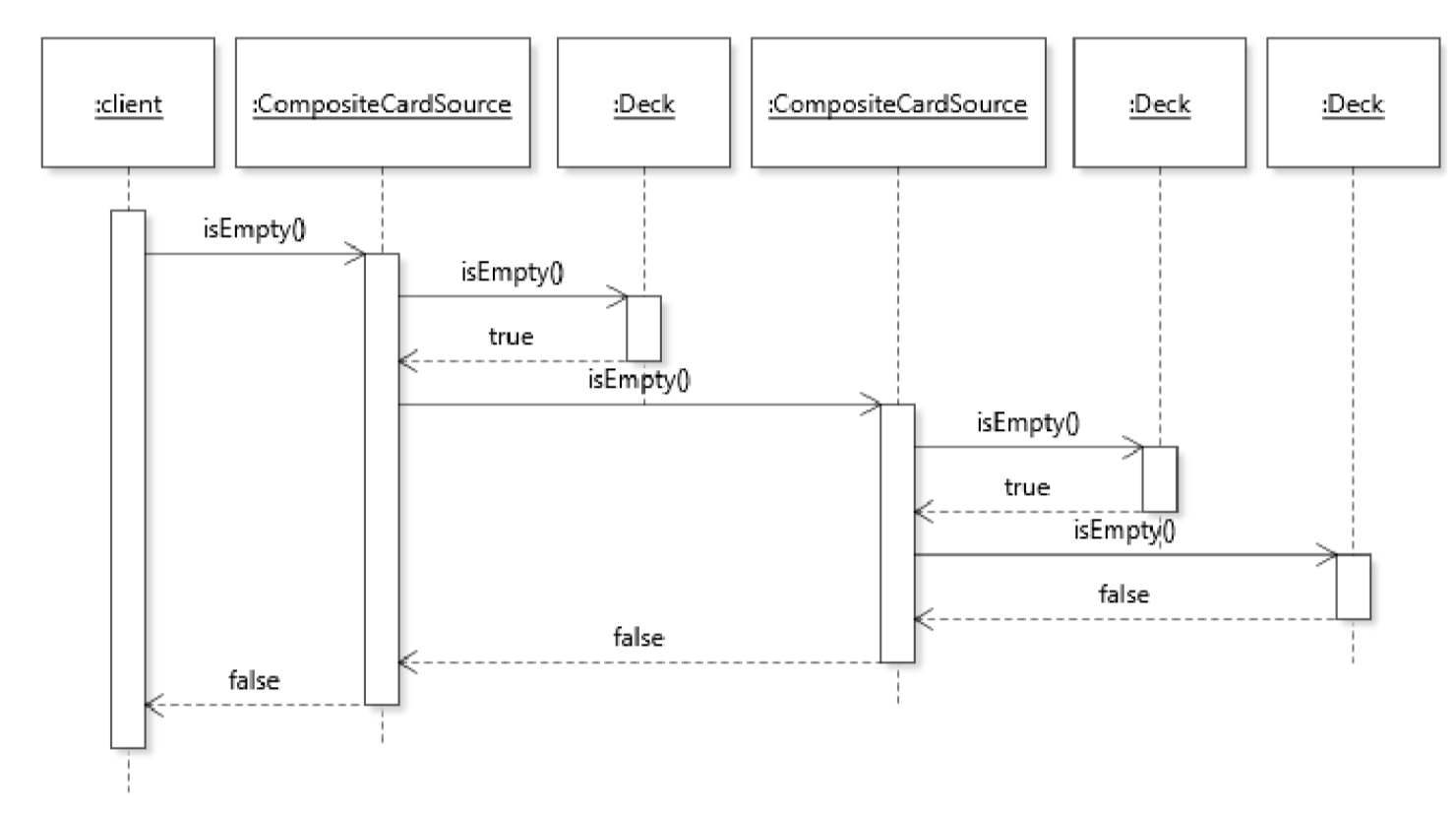
Because these concerns are context-dependent, their solution will depend on the specific design problem at hand. However, it is important to be aware that simply creating a well-designed composite class is not sufficient to have a correct application of the COMPOSITE. For example, with the design of UML, it could be possible to write code that results in the object graph below. However, this outcome is very likely undesirable, because the shared deck instance between source1 and source2 and the self-reference in source2 would lead to unmanageable behavior.



6.3 Sequence Diagrams

Sequence diagrams: *Model certain design decisions related to object call sequences.* Just like object diagrams and state diagrams, sequence diagrams model the dynamic perspective on a software system. Like object diagrams and as opposed to state diagrams, sequence diagrams represent a specific execution of the code. They are the closest representation to what one would see when stepping through the execution of the code in a debugger, for example.

a call to isEmpty() on an instance of CompositeCardSource.



Each rectangle at the top of the diagram represents an object. An object in a sequence diagram is also referred to as implicit parameter, because it is the object upon which a method is called. Consistently with other UML diagrams that represent the system at run time, the object names are underlined and follow the convention name:type as necessary. Here I did not specify a type for the client because it does not matter, and did not specify a name for any of the other objects because it does not matter either.

*The dashed vertical line emanating from an object represents the object’s life line*. The life line represents the time (running from top to bottom) when the object exists, that is, between its creation and the time it is ready to be garbage-collected. When objects are placed at the top of the diagram, they are assumed to exist at the beginning of the scenario being modeled. The diagram thus shows an interaction between a client object and an instance of CompositeCardSource and all its component objects, all of which were created before the modeled interaction began. How these objects were created is an example of details left unspecified by a particular diagram.

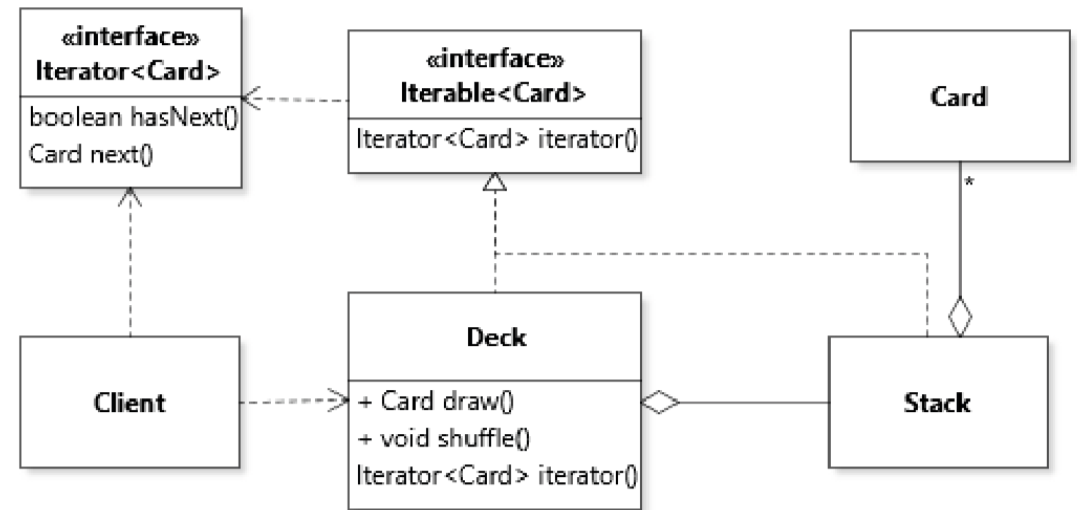
When representing the type of an object in a sequence diagram, there is some flexibility in terms of what type to represent in the object’s type hierarchy. We can use the concrete type of the object or one of its supertypes. As usual when modeling, we use what is the most informative. Here the CompositeCardSource and Deck objects are represented using their concrete type because the only other option is CardSource, which makes the information in the diagram less self-explanatory.

*Messages between objects typically correspond to method calls*. Messages are represented using a **directed arrow** from the caller object to the called object. By “called object” I mean “the object that is the implicit parameter of the method call”. Messages are typically labeled with the method that is called, optionally with some label representing arguments, when useful. When creating a sequence diagram that represents an execution of Java code, it is likely to be a modeling error if a message incoming on an object does not correspond to a method of the object’s interface. *Constructor calls are modeled as special messages with the label <<create>>.*

*Messages between objects induce an activation box, which is the thicker white box overlaid on the life line*. The activation box represents the time when a method of the corresponding object is on the execution stack (but not necessarily at the top of the execution stack).

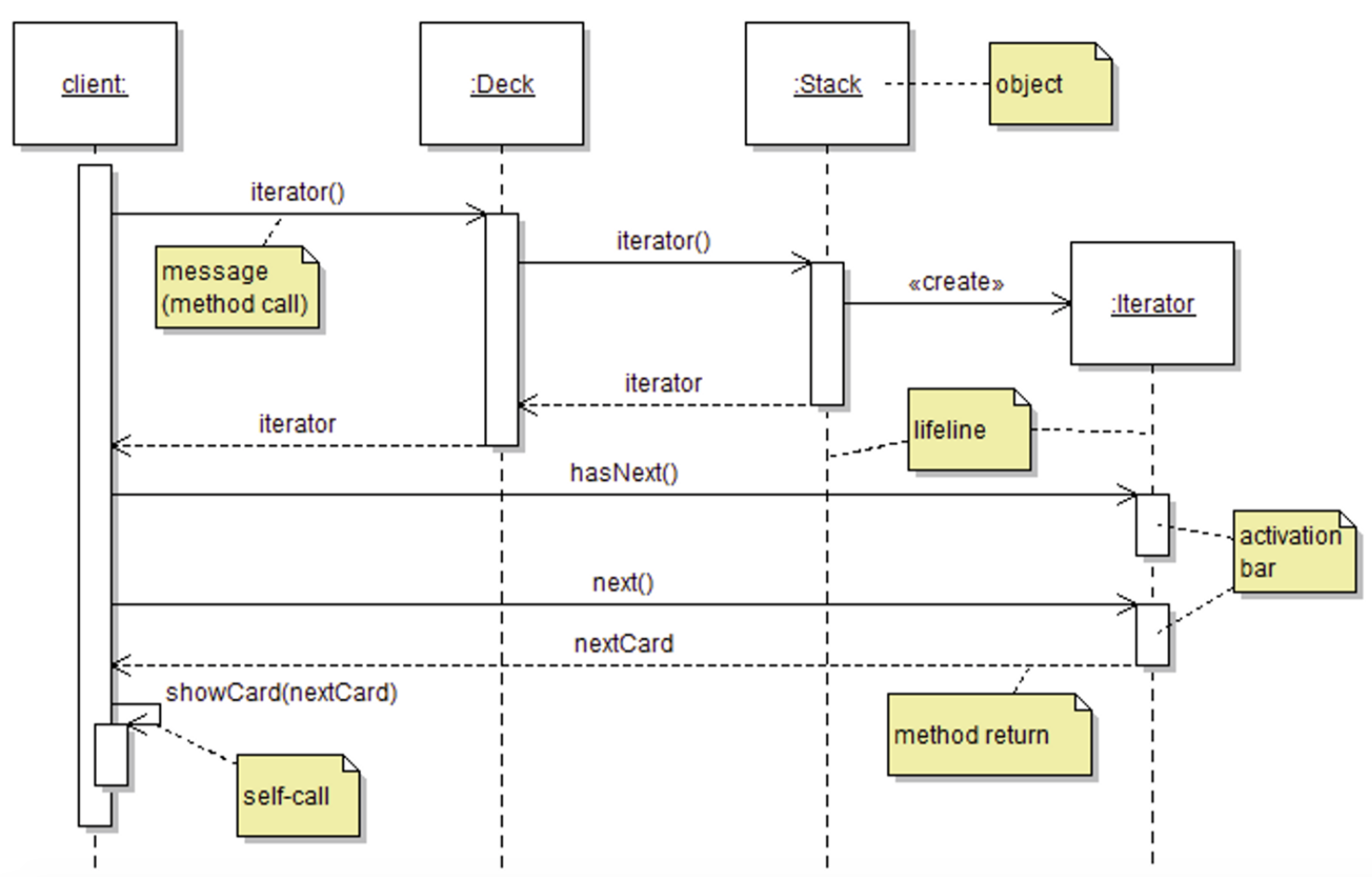
It is also possible to model *the return of control out of a method back to the caller*. This is represented with a **dashed directed arrow**. Return edges are optional. I personally only use them to aid understanding when there are complex sequences of messages, or to give a name to the value that is returned to make the rest of the diagram more self-explanatory. Here for example, I included return edges to provide the rationale for subsequent calls in the sequence (given that the execution terminates as soon as isEmpty() returns false).

To explore some of the additional modeling features of sequence diagrams and their potential, let us model the use of an iterator in the ITERATOR pattern (see Section 3.6). Figure 6.6 shows the class diagram of the specific application of ITERATOR I model with a sequence diagram.



This diagram shows a version of the Deck class that relies on a collection type Stack to store cards. Both the Deck and the Stack are iterable. The client code, represented as class Client, can refer to instances of class Deck as well as the iterators they return.

Let us look at what happens when the client code makes a call to Deck.iterator(). Figure 6.7 is the sequence diagram that models a specific execution of Deck.iterator() within client code.



The iterator() message to a Deck instance leads to the call being **delegated** to the Stack object. The Stack object is responsible for creating the iterator. It is also possible to show the creation of an instance by placing it lower in the diagram, as in the case here for the Iterator object. The label “iterator” is used on the return edge from both iterator() calls to show (indirectly) that it is the same object being propagated back to the client. In this diagram I also included a return edge from the next() method and labeled it “nextCard” to show that the returned object is the one being supplied to the subsequent self-call (a method called on an object from within a method already executing with this object as implicit parameter).

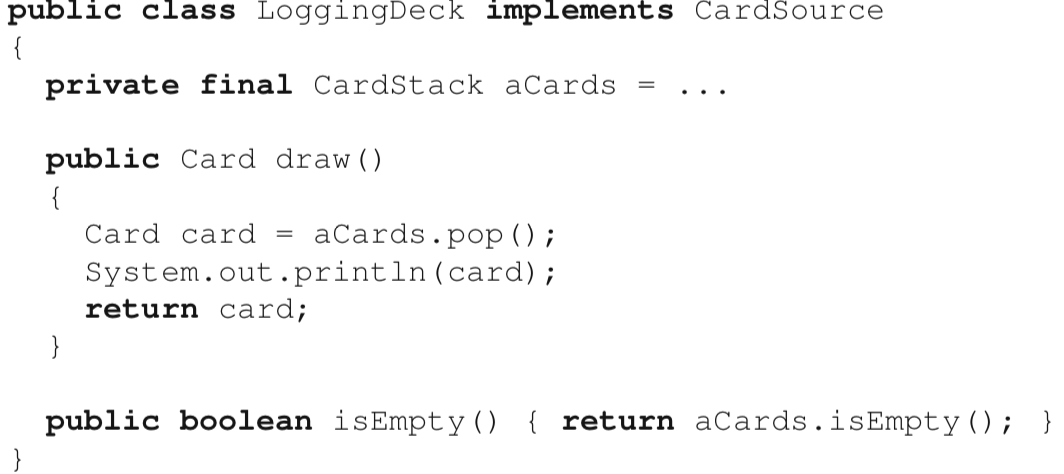
In terms of representing types, here the Deck object is represented using its concrete type, but the label deck:Iterable<Card> would have been a valid option as well. For the Iterator object I used the interface supertype because in practice the concrete type of this object is anonymous and does not really matter.

* The distinction between models and complete source code applies to sequence diagrams as well. First, a sequence diagram models a specific execution, not all executions. In the above example, a different execution could have received false from hasNext() and not called next(), or called next() twice, etc. These options are not represented, because they are different scenarios.
* Second, sequence diagrams will naturally omit some details of the execution of the code. We use sequence diagrams to show how objects interact to convey a specific idea.

6.4 The DECORATOR Design Pattern

In the example of a CardSource, we could imagine that in some cases we might want to print a description of each card drawn on the console or in a file (a process called *logging*). As another example, we might want to keep a reference to every card drawn from a certain source (i.e., *memorizing* the drawn cards).

**(1) Specialized class** - *Design one class for each type of feature we want to support.*

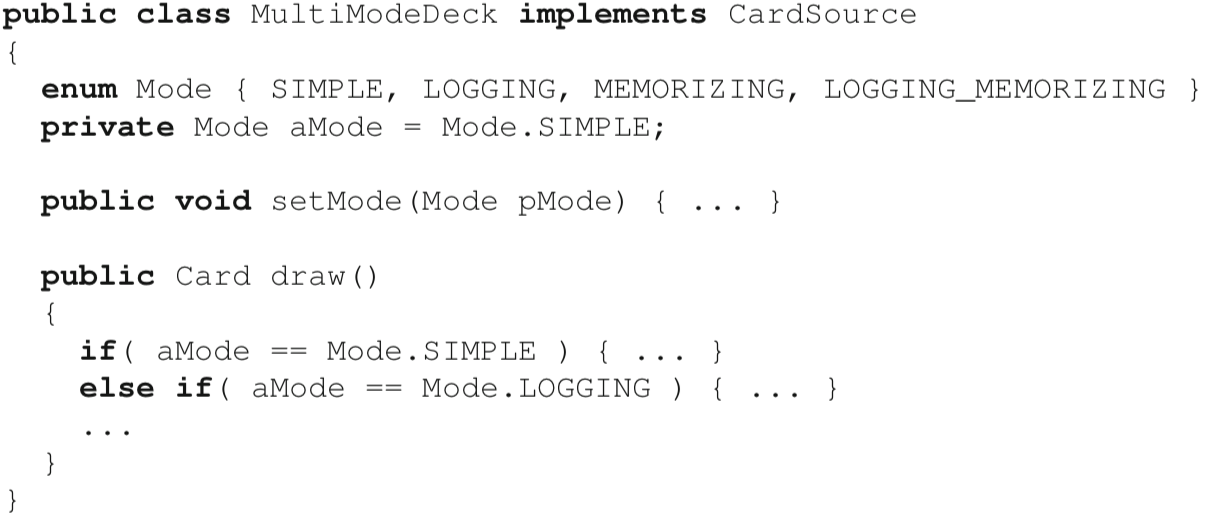


[Drawbacks]

It offers no flexibility for toggling features on and off at run-time. In other words, it is not easily possible to turn a normal deck into a “memorizing” deck, or to start logging the cards drawn at some arbitrary point in the execution of the code. In Java, it is impossible to change the type of an object at run-time, so the only option would be to initialize a new object and copy the state of the old object into a new object which has the desired features.

**(2) multi-mode class**

We provide all possible features within one class, and include a *flag* value to represent the “mode” the object of the class is in.

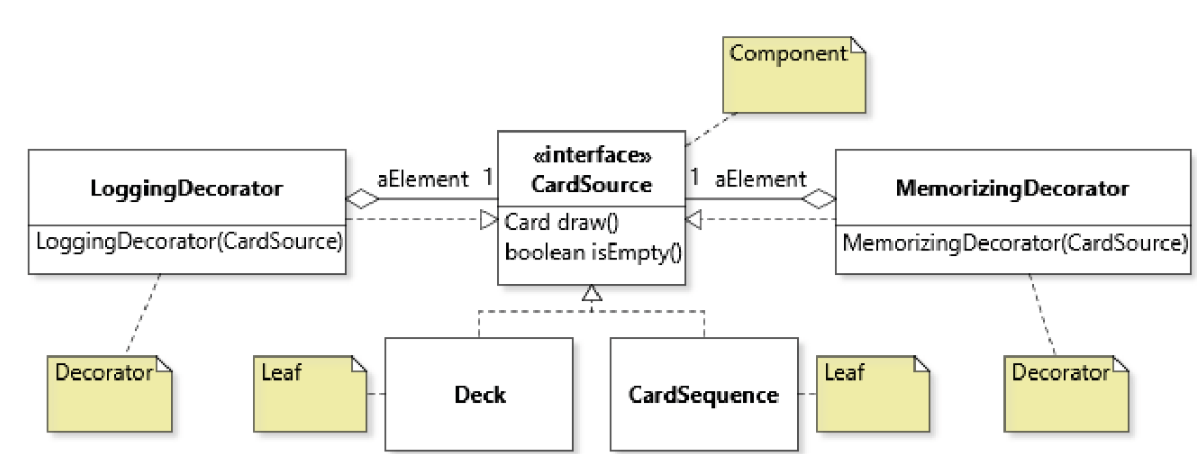


[Drawbacks]

Although the multi-mode class solution does allow one to toggle features on and off at run-time, it contravenes the important principles presented in Chapter 4 by inducing elaborate state spaces for objects that should otherwise be fairly simple. It also violates the principle of separation of concerns by tangling the behavior of different features within one class, or even a single method. In the extreme, it can turn a class intended to represent a simple concept into a GOD CLASS†. As a consequence of its complexity, the multi-mode class solution also suffers from a lack of extensibility. To add a new feature, we need to add yet more code and branching behavior to account for new modes. With, say, 10 features, it is easy to imagine how the code would become a nightmare of case switches and an instance of SWITCH STATEMENT†.

**(Solution) DECORATE**

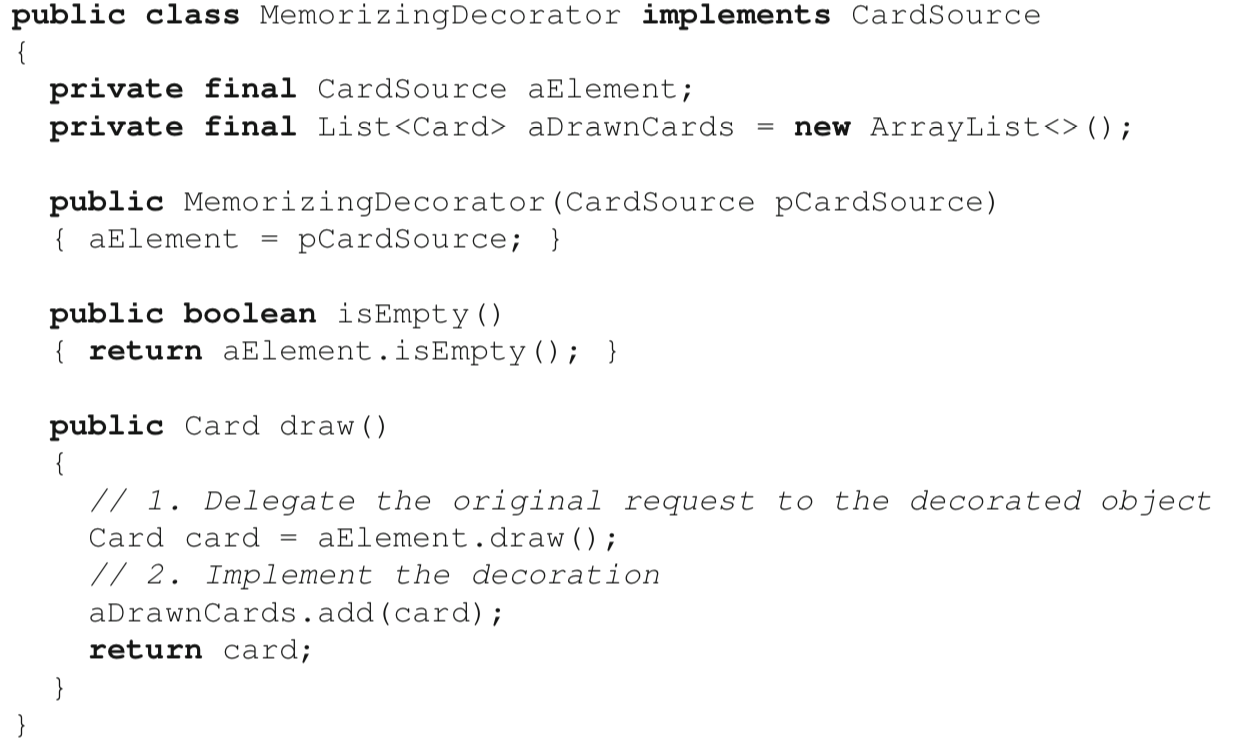
The context for using the pattern is a design problem where we want to “decorate” some objects with additional features, while being able to treat the decorated objects just like any other object of the undecorated type.



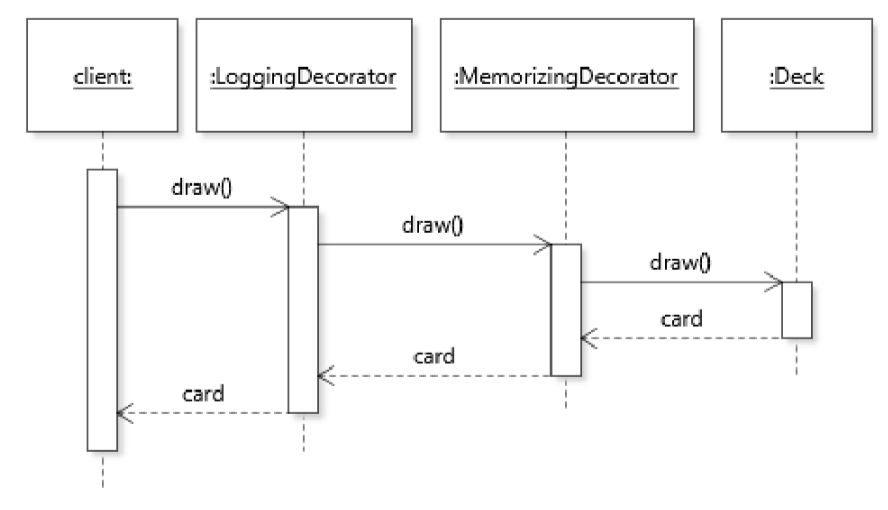
In terms of solution template, the DECORATOR looks very much like the COMPOSITE, except that instead of a composite class we have some decorator classes.

Indeed, the design constraints of the decorator class are similar as those of the composite class:

* A decorator aggregates one object of the component interface type (CardSource in the example). Using the component interface type is important, as it allows the decorator to “decorate” any other kind of components, including other decorators (and composites).
* It implements the component interface. This is basically what allows decorator objects to be treated by the rest of the code in exactly the same way as leaf elements.



Delegation sequence when using a DECORATOR where we decorated a Deck with a MemorizingDecorator, and then again with a LoggingDecorator, so that the final behavior of draw() will be to memorize, log, and return the next card in the card source.



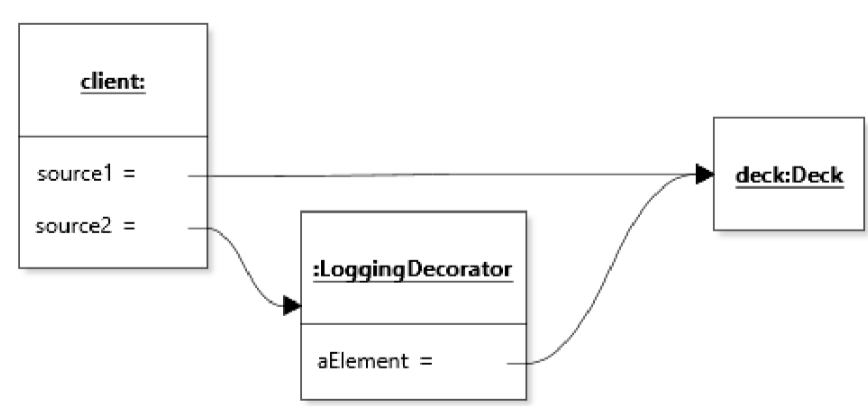
[Constraint]

An important constraint when using the DECORATOR is that for the design to work, decorations must be independent and strictly additive.

* The main benefit of the DECORATOR is to support attaching features in a flexible way, sometimes in unanticipated configurations. For this reason, use of the pattern should not require client code to respect elaborate combination rules.
* As for being additive, this means that the DECORATOR pattern should not be used to remove features from objects. The main reason for this constraint is that it would violate a fundamental principle of object-oriented design introduced in Chapter 7.

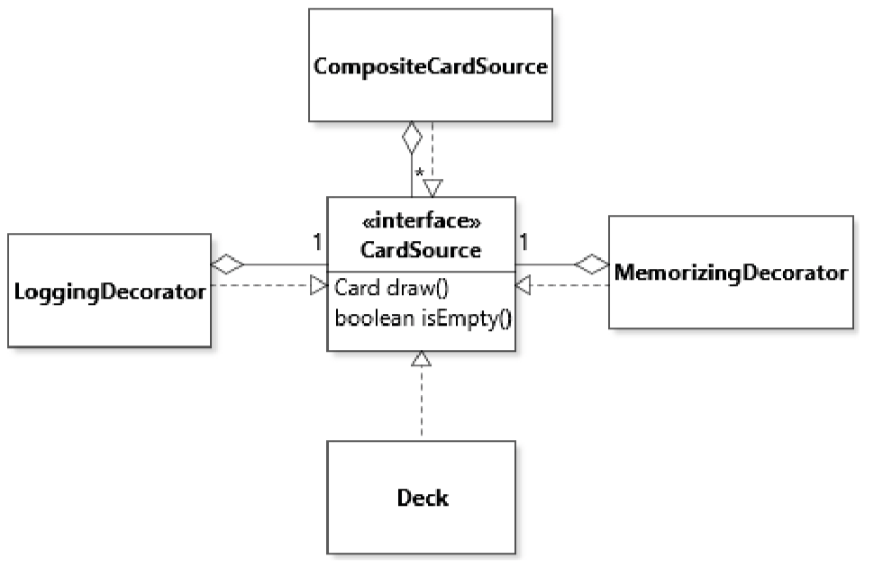
When implementing the DECORATOR design pattern in Java, it is a good idea to specify as final the field that stores a reference to the decorated object, and to initialize it in the constructor. A common expectation when using the DECORATOR is that a decorator object will decorate the same object throughout its lifetime

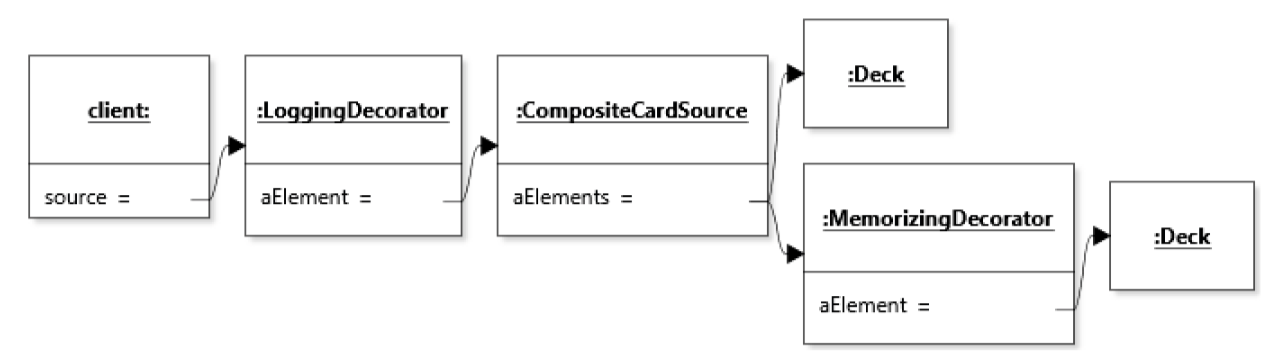
Finally, an important consequence of decorating objects using the DECORATOR is that decorated objects lose their identity. In other words, because a decorator is itself an object that wraps another object, a decorated object is not the same as the undecorated object.



Although source1 and source2 conceptually refer to the *same* card source, the decorated version does not have the same identity as the undecorated version. In other words, source1 != source2. This issue of identity loss could be a problem in a code base where, for example, object comparison relies on identity instead of equality.

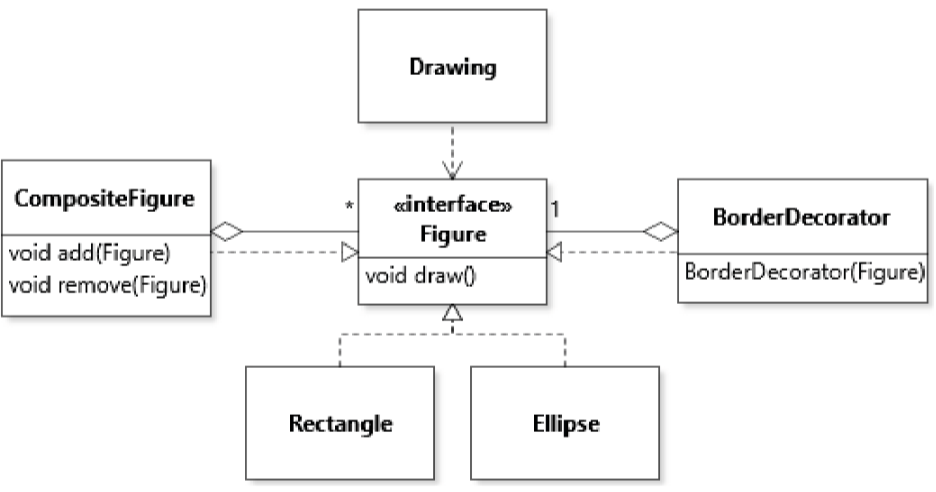
6.5 Combining COMPOSITE and DECORATOR



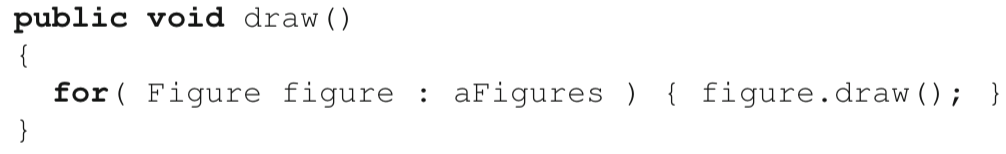


[Design Context]

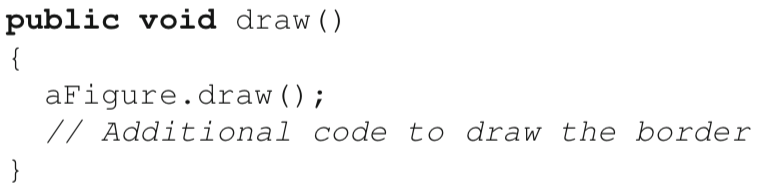
Development of some drawing feature (e.g., for a drawing tool or slideshow presentation application). In this scenario, the component type is a Figure with a draw() method or something like this. Leaf classes are concrete figures, such as rectangles, ellipses, text boxes, etc.



COMPOSITE



DECORATE



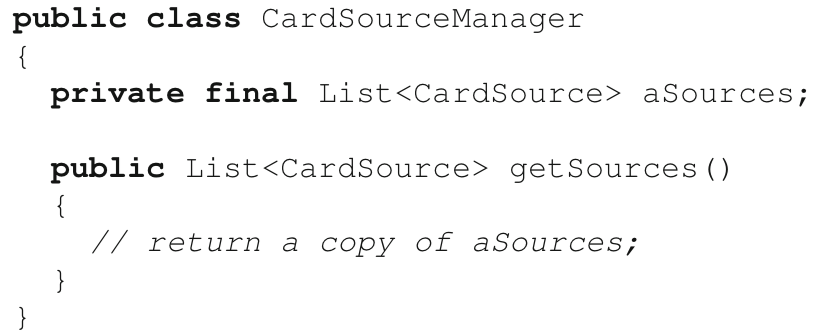
a clean sequence of one delegation followed by a decoration.

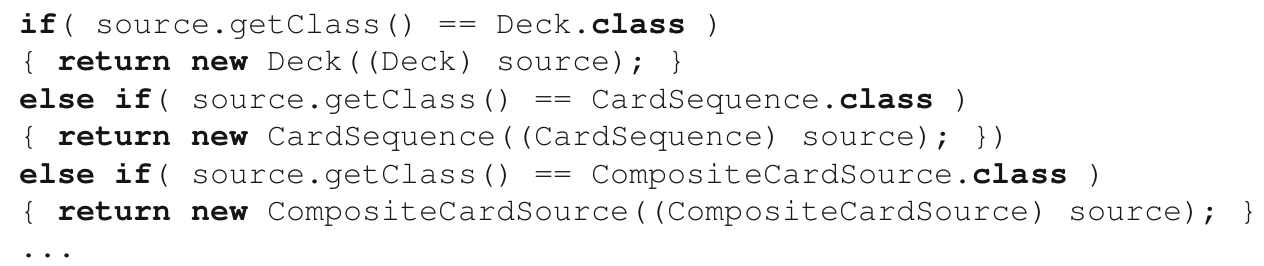
6.6 Polymorphic Object Copying

[Constraint of Copy Constructor]



A constructor call requires a static reference to a specific class (here, class Deck). In designs that make heavy use of polymorphism, this can turn out to be a problem.



If we want to return a copy of the list of card sources while protecting the class’s encapsulation, we would have to make a copy of every individual card source in aSources. However, because CardSource is an interface type that must be subtyped, we do not know the precise concrete types of the objects in the list aSources, so it is not easy to know what constructor to call. 

* Solutions of this nature are not recommended because they essentially void the benefits of polymorphism, namely, to be able to work with instances of CardSource no matter what their actual concrete type is.
* Moreover, this code is also an exmple of SWITCH STATEMENT† which completely destroys the extensibility of the design, as it would break as soon as a new subtype of CardSource is introduced into the design.
* Finally, it would be a complete mess to implement because some CardSource classes are just wrappers around other card sources. Specifically, because CompositeCardSource can aggregate any kind of card source, a copy constructor for this class would also need a branching statement like the above.

**Cloning**

Making an object cloneable involves four mandatory steps and a fifth, optional step:

1. Declaring to implement the Cloneable interface;

2. Overriding the Object.clone() method;

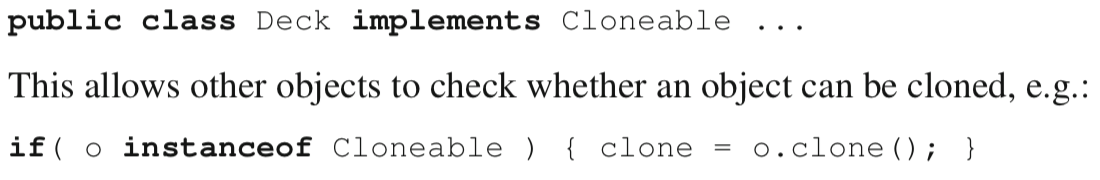
3. Calling super.clone() in the clone() method;

4. Catching CloneNotSupportedException in the clone() method;

5. Optionally, declaring the clone() method in the root supertype of a cloneable hierarchy.

**Declaring to Implement Cloneable**

Tag the class as cloneable using the Cloneable *tagging interface* *(A tagging interface is an interface with no method declaration, intended only to mark objects as having a certain property.)*



**Overriding Object.clone()**

When overriding clone, it is also recommend to change its return type from Object to the type of the class that contains the new clone method.



Method Object.clone() is declared to be protected, must be overridden with the public access modifier.