Chapter 6
Composition

This chapter covers:

Concepts and Principles: Divide and conquer, law of Demeter;
Programming Mechanisms: Aggregation, delegation, cloning;
Design Techniques: Sequence diagrams, combining design patterns;
Patterns and Antipatterns: GOD CLASS†, MESSAGE CHAIN† COMPOSITE, DECORATOR, PROTOTYPE, COMMAND

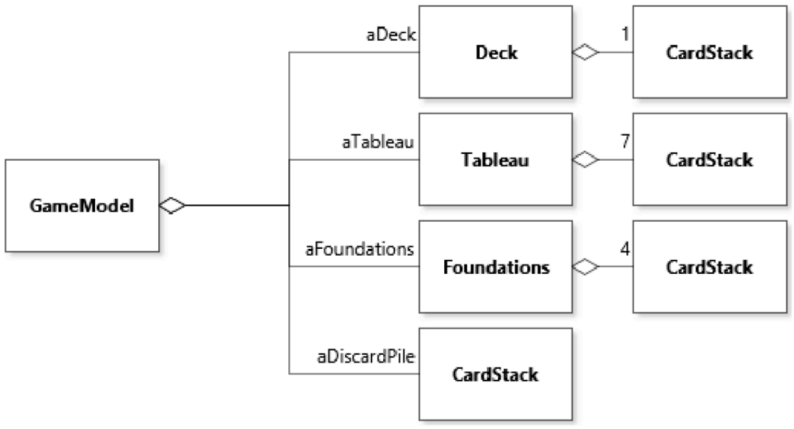
6.1 Composition and Aggregation

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.

```
public class CardStack implements Iterable<Card>
{
  private final List<Card> aCards = new ArrayList<>();

  public void push(Card pCard)
  { assert pCard != null && !aCards.contains(pCard);
    aCards.add(pCard);
  }

  public Card pop()
  { assert !isEmpty();
    return aCards.remove(aCards.size()-1);
  }

  public Card peek()
  { assert !isEmpty();
    return aCards.get(aCards.size()-1);
  }

  public void clear() { aCards.clear(); }
  public boolean isEmpty() { return aCards.size() == 0; }
  public Iterator<Card> iterator() { return aCards.iterator(); }
}
```

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`:

```
public boolean isVisibleInTableau(Card pCard)
{
  return aTableau.contains(pCard) && aTableau.isVisible(pCard);
}
```

6.2 The Composition Design Pattern

```
public interface CardSource
{
  /**
   * Removes a card from the source and returns it.
   *
   * @return The card that was removed from the source.
   * @pre !isEmpty()
   */
  Card draw();

  /**
   * @return True if there is no card in the source.
   */
  boolean isEmpty();
}
```

[Class Definition]

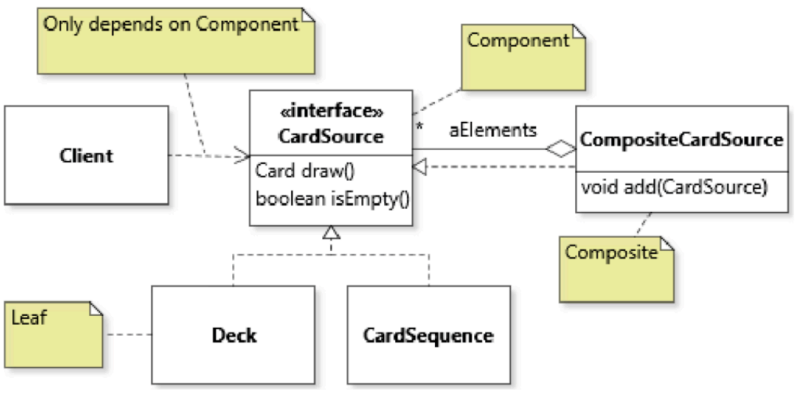
```
public class Deck implements CardSource { ... }
public class MultiDeck implements CardSource { ... }
public class FourAces implements CardSource { ... }
public class FaceCards implements CardSource { ... }
public class DeckAndFourAces implements CardSource { ... }
```

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]

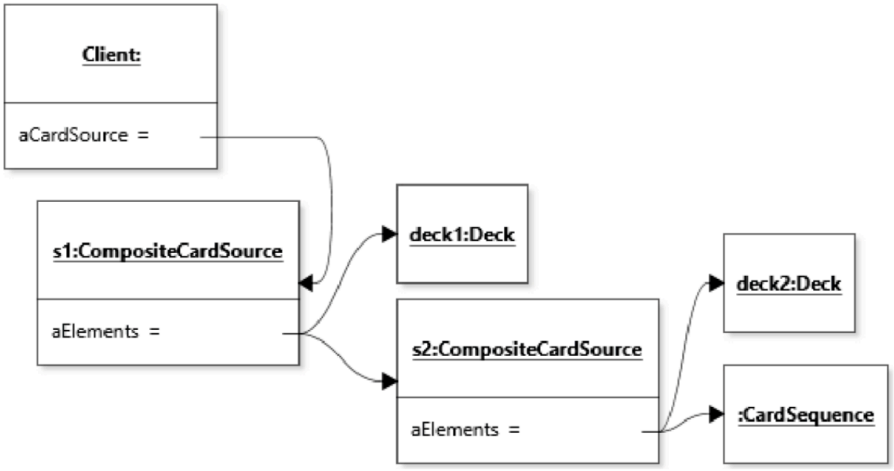
The fundamental idea to support this approach is to define a class that represents multiple `CardSources` while still behaving like a single one.



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()`

```
public boolean isEmpty()
{
  for( CardSource source : aElements )
  {
    if( !source.isEmpty() )
    { return false; }
  }
  return true;
}
```

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.

```
public Card draw()
{
  assert !isEmpty();
  for( CardSource source : aElements )
  {
    if( !source.isEmpty() )
    {
      return source.draw();
    }
  }
  assert false;
  return null;
}
```

Because `CompositeCardSource.draw()` is an implementation of the interface `CardSource.draw()`, it has the same preconditions as the interface method. Thus, it does not need to deal with the case where a call is made to `draw` from an empty card source, even if this is a composite. In the first line of the method, we assert that `!isEmpty()`. Here, the call to `isEmpty()` would be to method `isEmpty` of class `CompositeCardSource`, so the following code could be assumed to always find a method to draw.[1] This assumption is further encoded with the **`assert false;`** statement, which encodes the developer's assumption that if the precondition is respected, the execution should not reach this point. The following **`return null;`** statement serves no purpose besides making the code compilable.

[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:

```
public CompositeCardSource
{
  private final List<CardSource> aElements;

  public CompositeCardSource(List<CardSource> pCardSources)
  {
    aElements = new ArrayList<>(pCardSources);
  }
}
```

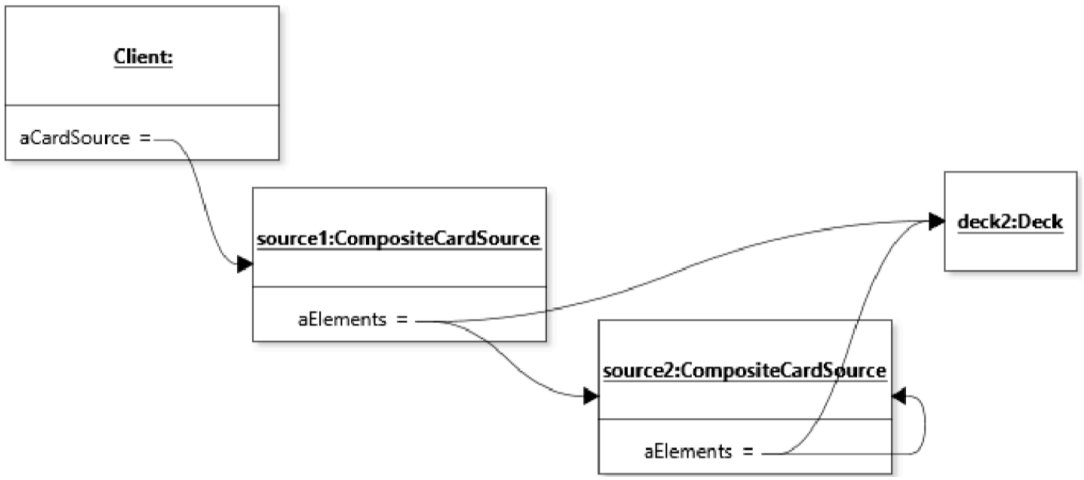
  Here we use of the copy constructor, to avoid leaking a reference to the private collection structure

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.

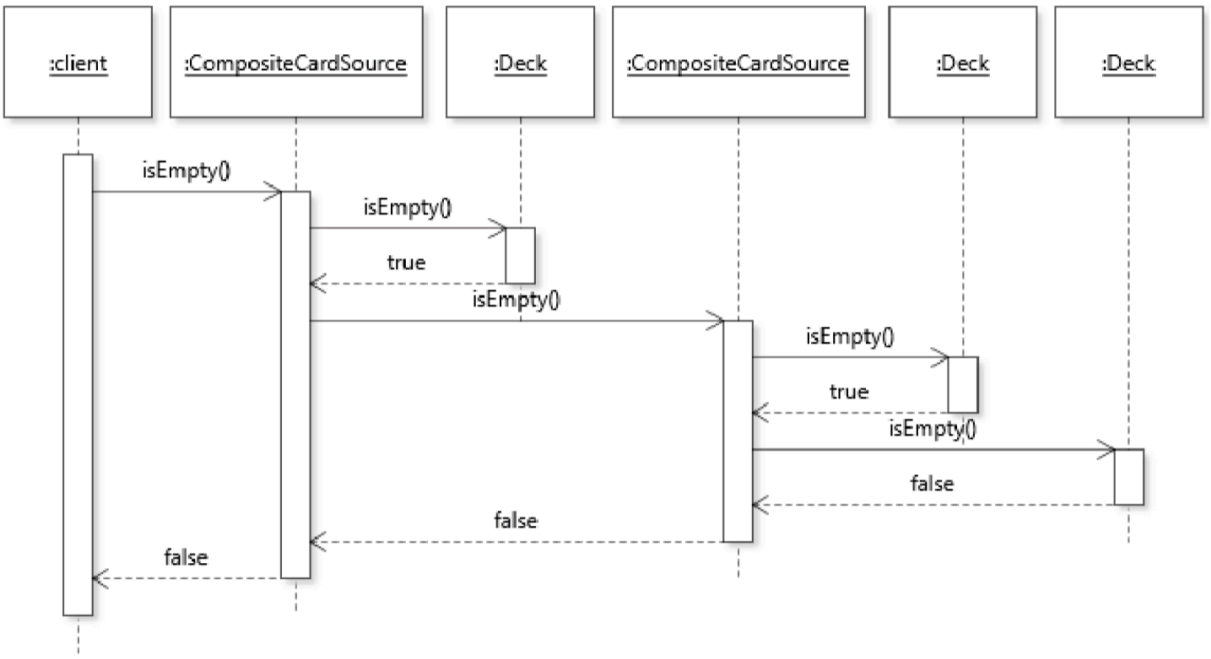
Because these concerns are context-dependent, their solution will depend on the specific design problem at hand. However, it is important to be aware that simply creating a well-designed composite class is not sufficient to have a correct application of the COMPOSITE. For example, with the design of 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`.

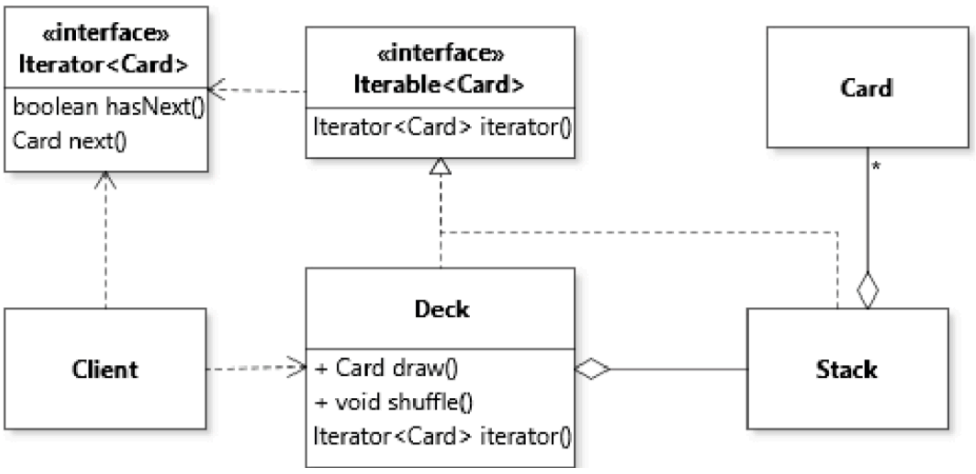


Each rectangle at the top of the diagram represents an object. An object in a sequence diagram is also referred to as implicit parameter, because it is the object upon which a method is called. Consistently with other UML diagrams that represent the system at run time, the object names are underlined 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.

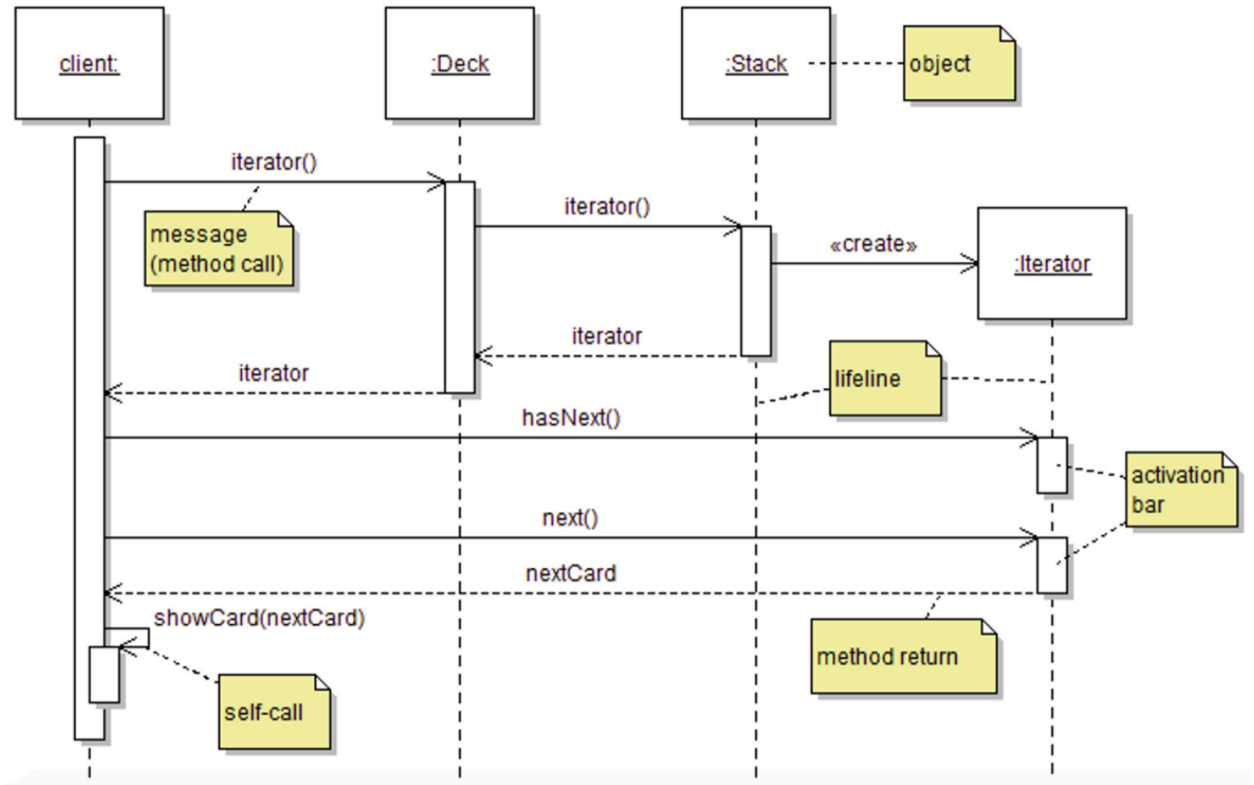
Here the `CompositeCardSource` and `Deck` objects are represented using their concrete type because the only other option is CardSource, which makes the information in the diagram less self-explanatory.
Messages between objects typically correspond to method calls. Messages are represented using a **directed 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.*

```
public class LoggingDeck implements CardSource
{
  private final CardStack aCards = ...

  public Card draw()
  {
    Card card = aCards.pop();
    System.out.println(card);
    return card;
  }

  public boolean isEmpty() { return aCards.isEmpty(); }
}
```

[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.

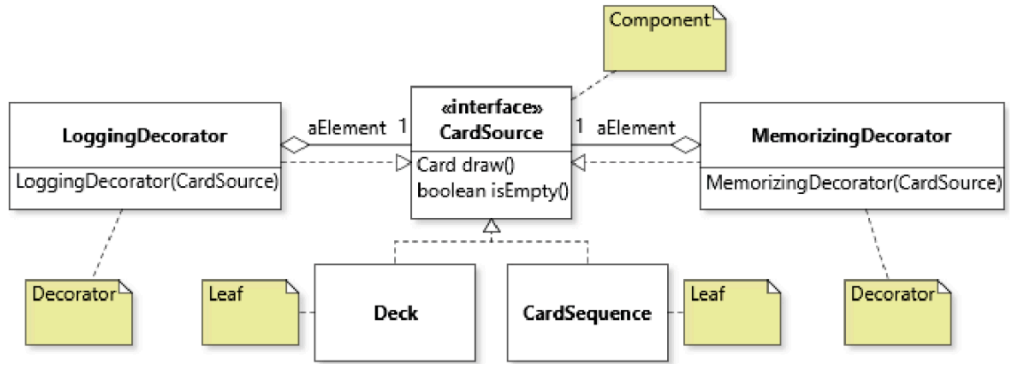
(2) multi-mode class
We provide all possible features within one class, and include a *flag* value to represent the "mode" the object of the class is in.

```
public class MultiModeDeck implements CardSource
{
  enum Mode { SIMPLE, LOGGING, MEMORIZING, LOGGING_MEMORIZING }
  private Mode aMode = Mode.SIMPLE;

  public void setMode(Mode pMode) { ... }

  public Card draw()
  {
    if( aMode == Mode.SIMPLE ) { ... }
    else if( aMode == Mode.LOGGING ) { ... }
    ...
  }
}
```

[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.

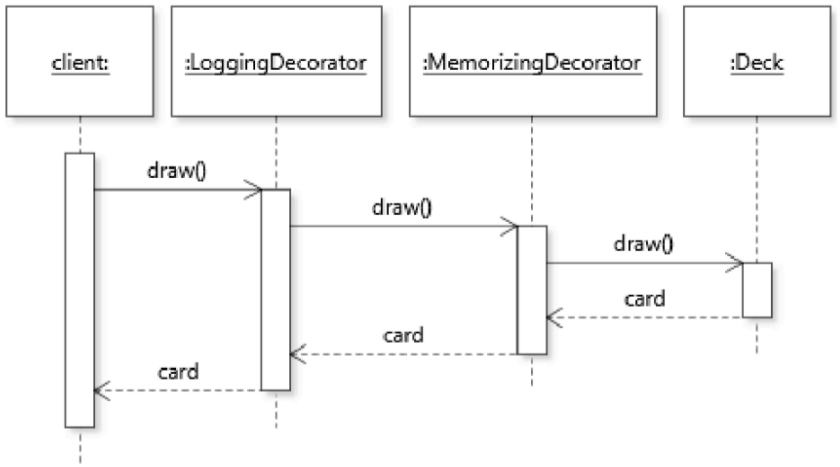


In terms of solution template, the DECORATOR looks very much like the COMPOSITE, except that instead of a composite class we have some decorator classes.
Indeed, the design constraints of the decorator class are similar as those of the composite class:

- 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.

```
public class MemorizingDecorator implements CardSource
{
  private final CardSource aElement;
  private final List<Card> aDrawnCards = new ArrayList<>();

  public MemorizingDecorator(CardSource pCardSource)
  { aElement = pCardSource; }

  public boolean isEmpty()
  { return aElement.isEmpty(); }

  public Card draw()
  {
    // 1. Delegate the original request to the decorated object
    Card card = aElement.draw();
    // 2. Implement the decoration
    aDrawnCards.add(card);
    return card;
  }
}
```

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.

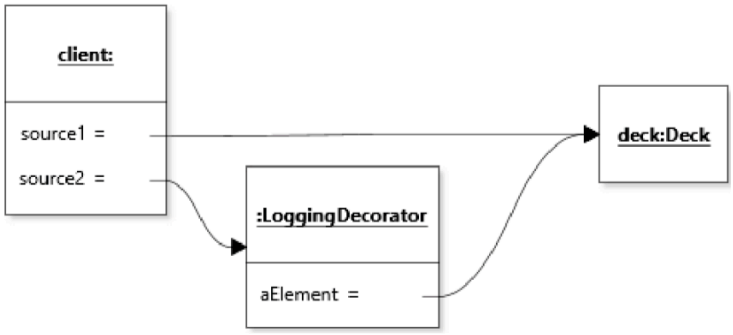


[Constraint]
An important constraint when using the DECORATOR is that for the design to work, decorations must be independent and strictly additive.

- 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.

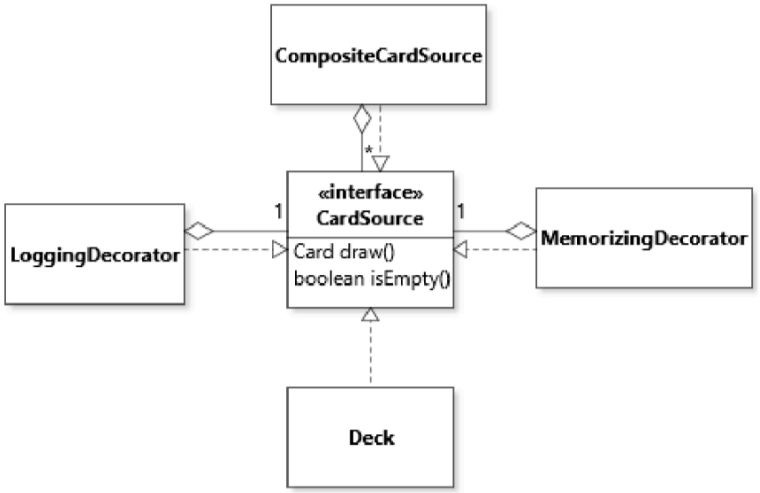
When implementing the DECORATOR design pattern in Java, it is a good idea to specify as `final` the field that stores a reference to the decorated object, and to initialize it in the constructor. A common expectation when using the DECORATOR is that a decorator object will decorate the same object throughout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

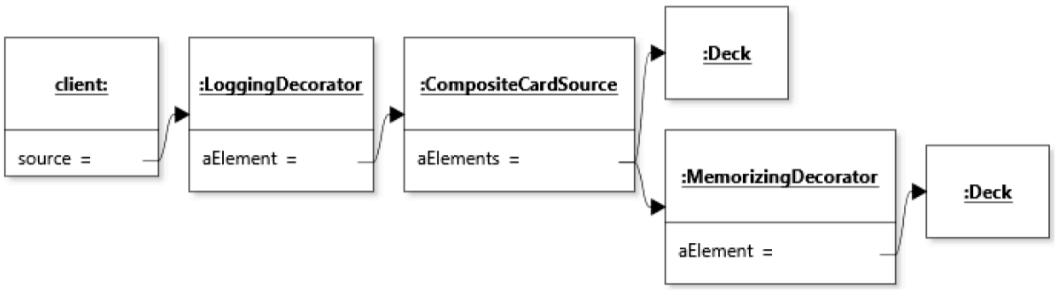
Finally, an important consequence of decorating objects using the DECORATOR is that decorated objects lose their identity. In other words, because a decorator is itself an object that wraps another object, a decorated object is not the same as the undecorated object.



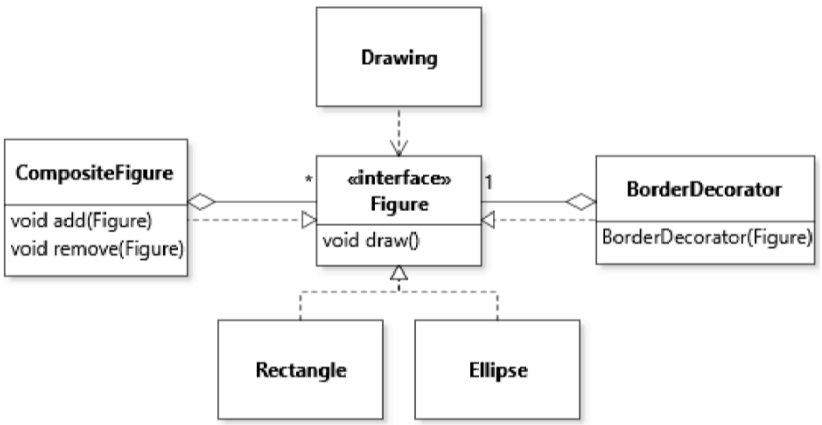
Although source1 and source2 conceptually refer to the *same* card source, the decorated version does not have the same identity as the undecorated version. In other words, source1 != source2. This issue of identity loss could be a problem in a code base where, for example, object comparison relies on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

```
public void draw()
{
  for( Figure figure : aFigures ) { figure.draw(); }
}
```

DECORATE

```
public void draw()
{
  aFigure.draw();
  // Additional code to draw the border
}
```

a clean sequence of one delegation followed by a decoration.

6.6 Polymorphic Object Copying

[Constraint of Copy Constructor]

```
Deck deckCopy = new Deck(pDeck);
```

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.

```
public class CardSourceManager
{
  private final List<CardSource> aSources;

  public List<CardSource> getSources()
  {
    // return a copy of aSources;
  }
}
```

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.

```
if( source.getClass() == Deck.class )
{ return new Deck((Deck) source); }
else if( source.getClass() == CardSequence.class )
{ return new CardSequence((CardSequence) source); })
else if( source.getClass() == CompositeCardSource.class )
{ return new CompositeCardSource((CompositeCardSource) source); }
...
```

- 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.)*

```
public class Deck implements Cloneable ...
```

This allows other objects to check whether an object can be cloned, e.g.:

```
if( o instanceof Cloneable ) { clone = o.clone(); }
```

Overriding Object.clone()

```
public Deck clone() ...
```

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

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.

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.[5]

Calling `super.clone()`

The overridden clone method needs to create a new object of the same class. Although the normal way to create an object is to use a constructor, creating the cloned object should *not* be done through a constructor. The proper way to create the cloned object is by calling the method `clone()` defined in class `Object` by using the call **`super`**`.clone()`. How a super call works exactly is detailed in Section 7.4. For now, the important thing to know is that the proper way to obtain a clone of the implicit parameter to a call to method `clone()` is as follows:

```
public Deck clone()
{
  // NOT Deck clone = new Deck();
  Deck clone = (Deck) super.clone();
  ...
}
```

The statement **`super`**`.clone()` calls the `clone()` method in the superclass, which here means method `Object.clone()`. This method is very special. It uses metaprogramming features (see Section 5.4) to return an object of the class from where the call to the method originates. This is special because although the method is implemented in the library class `Object`, it still returns a new instance of class `Deck`.[6]

Method `Object.clone()` is also special because it does not create a new instance of the class by internally calling the default constructor (sometimes there is not even a default constructor). Instead, it reflectively creates a new instance of the class initialized by making a shallow copy of all the instances fields. Whenever a shallow copy is not sufficient, the overridden `clone` method must perform additional steps to more deeply copy some of the fields.

---

[5] This feature, called covariant return types, is available since Java 5. The feature allows overriding methods to have a return type that is more specific than the return type of the method they override.

[6] To understand why the use of a constructor is dangerous, we need to consider the possibility that the method `clone` might be inherited, a topic covered in Chapter 7. If an inherited implementation of `clone()` uses a constructor to create the cloned object, the method will return an object of the wrong type if called on an instance of the subtype. For example, if we have a subclass of `Deck` called `SpecialDeck` and call `clone` on an instance of `SpecialDeck`, the returned object will be of class `Deck`, not `SpecialDeck`. With **`super`**`.clone()`, the cloned object would be an instance of `SpecialDeck` even if the `clone()` method is inherited from `Deck`.

For example, a reasonable implementation of `clone()` of a class `Deck` could look like this:

```
public Deck clone()
{
  Deck clone = (Deck) super.clone();
  clone.aCards = new CardStack(aCards);
  return clone;
}
```

In this code, the `clone()` method obtains a new instance of `Deck` by using `super.clone()`. However, this would result in a shared (shallow-copied) reference to the value of the field `aCards`, as illustrated in Figure 6.14. Because this outcome would break encapsulation and most likely is incorrect, the clone method makes a new copy of the `CardStack`, this time using a copy constructor given that for `CardStack` there is no requirement to use polymorphic copying.

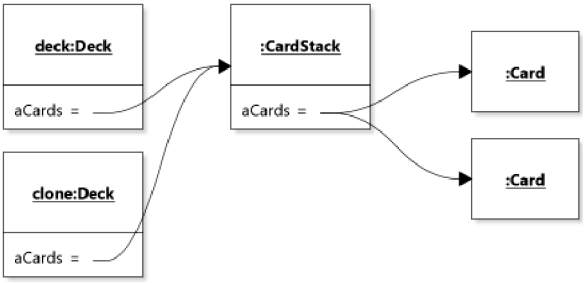


Fig. 6.14 Object diagram of a cloned instance of `Deck`

Catching `CloneNotSupportedException`

Unfortunately, the code in the previous fragment will not compile because `super.clone()` declares to throw a checked exception of type `CloneNotSupportedException`. This exception is only raised if `super.clone` is called from within a class that is not declared to implement `Cloneable` (directly or transitively). If we properly declare our class `Deck` to be `Cloneable`, then we are assured that this exception will never be raised, and we can safely squash it by catching it and doing nothing.

```
public Deck clone()
{
  try
  {
    Deck clone = (Deck) super.clone();
    clone.aCards = new CardStack(clone.aCards);
    return clone;
  }
  catch( CloneNotSupportedException e )
  {
    assert false;
    return null;
  }
}
```

The reason for this awkward exception-handling requirement is that because the default cloning behavior implemented by `Object.clone` (shallow copying) is potentially not appropriate for a class (like `Deck`), a programmer should not be able to call `clone` "accidentally". Having to catch the exception is supposed to force programmers to remember this, although it is not clear to what extent this mechanism is successful.

Adding `clone()` to an Interface

The last, optional, step when using cloning is to add the `clone` method to the super type of a hierarchy of classes whose objects we want to clone. Unfortunately, the `Cloneable` interface does not include the `clone()` method, so that the `clone()` method will not automatically be visible to clients of `Cloneable` types. In other words, this means that if we declare `CardSource` to be cloneable:

```
CardSource extends Cloneable ...
```

it does not mean that we can do:

```
CardSource clone = cardSource.clone();
```

To support this idiom, we need to add `clone()` to the interface (and implement it in all concrete subtypes). A version of `CardSource` meant to be cloneable would thus look like this:

```
public interface CardSource extends Cloneable
{
  Card draw();
  boolean isEmpty();
  CardSource clone();
}
```

As a review example, here is how we would make class `CompositeCardSource` cloneable, assuming that `CardSource` has the above definition

```
public class CompositeCardSource implements CardSource
{
  private final List<CardSource> aSources;

  ...

  public CardSource clone()
  {
    try
    {
      CompositeCardSource clone =
        (CompositeCardSource) super.clone();
      clone.aSources = new ArrayList<>();
      for( CardSource source : aSources )
      {
        clone.aSources.add(source.clone());
      }
      return clone;
    }
    catch(CloneNotSupportedException e ) { return null; }
  }
}
```

With this definition, and assuming all subtypes of `CardSources` properly implement `clone()`, the `clone()` method in `CompositeCardSource` will recursively make a deep copy of all card sources in the composite object.

6.7 The PROTOTYPE Design Pattern

The ability to copy objects polymorphically, as seen in the previous section, is a powerful feature that can be used for a variety of purposes in composition-based designs. One specialized use of polymorphic copying is to support *polymorphic instantiation*. Let us continue with the context of a `CardSourceManager` class, introduced in the previous section. We now wish to add an additional service to the class: a method to return a default card source.

```
public class CardSourceManager
{
  public CardSource createCardSource() { ... }
}
```

The implementation of `createCardSource()` can be trivial if we hard-code the specific type of source to return (e.g., `return new Deck();`). However, what if we want to make it possible to configure `CardSourceManager` so that it is possible to obtain *any* type of `CardSource`, and to change the default card source at run time? In this case, the problems are very similar to the ones discussed in the previous section (i.e., the use of a SWITCH STATEMENT† structure destroys the benefits of polymorphism, etc.).

To create a default card source, one option better than a SWITCH STATEMENT† would be to use metaprogramming (see Section 5.4), for example by adding a parameter to `createCardSource()` of type `Class<T>`, which specifies the type of the card source to add. Although workable, solutions of this nature tend to be fragile and require a lot of error handling.

Another option is to rely on a polymorphic copying mechanism and create new instances of an object of interest by copying a *prototype object*. This idea is captured as the PROTOTYPE design pattern. The context for using the PROTOTYPE is the need to create objects whose type may not be known at compile time. The solution template involves storing a reference to the prototype object and polymorphically copying this object whenever new instances are required.

For the `CardSource` scenario, the application of the PROTOTYPE would look like this:

```
public class CardSourceManager
{
  private CardSource aPrototype = new Deck(); // Default

  public void setPrototype( CardSource pPrototype )
  { aPrototype = pPrototype; }

  public CardSource createCardSource()
  { return aPrototype.clone(); }
}
```

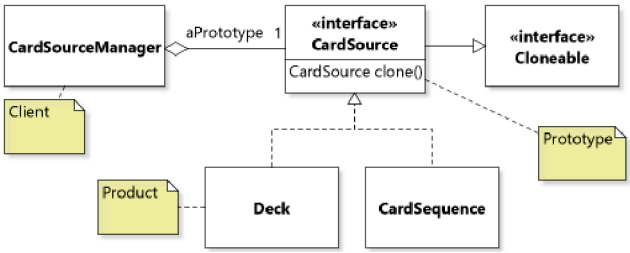


Fig. 6.15 Sample application of the PROTOTYPE, with the name of roles indicated in notes

Figure 6.15 shows a class diagram that summarizes the key aspects of the solution template, and indicates the role various elements play in the application of the pattern. The client is a generic mention to represent any code that needs to perform instantiation. The prototype is the abstract element (typically an interface) whose concrete prototype can be switched at run-time. The *products* are the objects that can be created by copying the prototype.

One nice conceptual benefit of the PROTOTYPE pattern is that normally the option to create objects of different types does not increase the amount of control flow (branching statements) in the client class. In a traditional, "mode-based" design, the `createCardSource()` method would have to check whether the object is in

a specific state to create, say, `Deck` card sources as opposed to other card sources, using a control statement such as an **`if`** statement. With the PROTOTYPE, this branching is done through polymorphism. As the code of the `createCardSource()` method shows, there is no such control statement: the method just makes a copy of whatever object is the current prototype. The Code Exploration section provides additional insights on the use of the PROTOTYPE in practice.

6.8 The COMMAND Design Pattern

Conceptually a *command* is a piece of code that accomplishes something: saving a file, drawing a card from a deck, etc. Intuitively the way to represent a command in code naturally maps to the concept of a function or method, since that is an abstraction that corresponds to a piece of code that will execute. As an example, we can consider the two main state-changing functionalities of a `Deck` class: to draw a card, and to shuffle the cards in deck. To exercise these features, we can just call methods:

```
deck.shuffle();
Card card = deck.draw();
```

However, now that we are studying designs that make principled use of objects, let us consider an alternative idea for representing commands, namely for objects to serve as *manageable units of functionality*. In sophisticated applications, there are many different contexts in which we might want to exercise a functionality such as drawing a card from the deck. For example, we might want to store a history of completed commands, so that we can undo them or replay them later. Or, we might want to accumulate commands and execute them all at once, in batch mode. Or, we might want to parameterize other objects, such as graphical user interface menus, with commands. Requirements such as these point to the additional need to *manage* functionality in a principled way. The COMMAND design pattern provides a recognizable way to manage abstractions that represent commands.

The class diagram of Figure 6.16 shows a sample application of the pattern. The `Command` interface defines an `execute` method and other methods to specify the services required by the clients to manage the commands. In the example, this includes an additional `undo()` method, but other designs may leave it out or have other required services (such as `getDescription()`, to get a description of the command).

The COMMAND pattern has a simple solution template. The template involves defining commands as objects, with an interface for commands that includes a method to execute the command. Another important part of the solution template is for the client to refer to commands through the interface. Despite the simplicity of the solution template, the COMMAND pattern is not necessarily an easy one to apply, because many important design choices induced by the pattern are implementation-dependent. Let us look at some examples from our scenario.

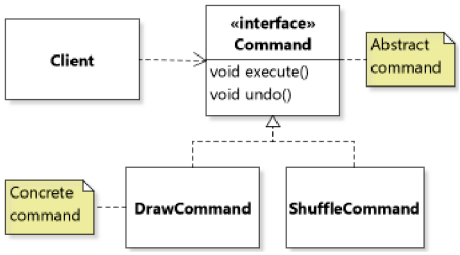


Fig. 6.16 Application of the COMMAND design pattern with the name of element roles in notes

- **Access to command target:** Command execution typically modifies the state of one of more objects, e.g., by drawing a card from a deck. The design must specify how the command gains access to the objects it must act on. Typically this is done by storing a reference to the target within the command object, but other alternatives are possible, including passing arguments to the `execute` method or using closures (see below);
- **Data flow:** In the typical solution template for COMMAND, the interface methods have return type **`void`**. The design must thus include a provision for returning the result of commands that produce an output, such as drawing a card from a deck;
- **Command execution correctness:** The code responsible for executing commands must ensure that the sequence of execution is correct. For example, the design needs to specify if commands can be executed more than once. The use of design by contract also leads to interesting implications. If commands call code with specified preconditions, the responsibility of respecting the preconditions is transferred to the code executing the command. In our case, a precondition is that we do not call `draw()` on an empty deck. The responsibility for ensuring that this precondition holds normally rests with the client code calling `Deck.draw()`. In the COMMAND pattern, we must ensure that code calling `Command.execute()` can ensure the precondition will be respected in cases where the command happens to be for drawing a card;
- **Encapsulation of target objects:** In some cases, a command object might require operations that are not available in the target object's public interface. For example, to undo the effect of calling `Deck.draw()`, it is necessary to push a card back onto the deck. In our running example class `Deck` does not have a `push` method. The design must include a solution to this issue. One possibility is to have a command factory method located in the class of the object the commands operate on. In our case, this would mean to add a `createDrawCommand()` method in class `Deck` (see below).
- **Storing data:** Some operations supported by commands require storing some data, something that also needs to be designed as part of the pattern's application. For example, in a design context where the undoing of commands is required, the effect of executing a command may have to be cached so that it can be undone. In our case, to undo the drawing of a card from a deck, it is necessary to remember which card was drawn. This information could be stored in the command object directly, or in an external structure accessible by the command object.

To illustrate one point in the design space for each of the concerns above, the code below shows an example of how to support a command to draw cards from a deck. The key idea for this application of the pattern is to use a factory method to create commands that are instances of an anonymous class with access to fields of its outer instance (see Section 4.10). In this design, commands to operate on a `Deck` instance are obtained directly from the `Deck` instance of interest. To keep the example simple, I slightly modify the `Command` interface so that `execute()` returns an `Optional<Card>`, which allows some commands to return an instance of `Card` if applicable. The code also assumes commands are executed only once and undone in the inverse order of that in which they are executed.

```
public class Deck
{
  private CardStack aCards = new CardStack();

  public createDrawCommand()
  {
    return new Command()
    {
      Card aDrawn = null;
      public Optional<Card> execute()
      {
        aDrawn = draw();
        return Optional.of(aDrawn);
      }

      public void undo()
      {
        aCards.push(aDrawn);
        aDrawn = null;
      }
    }
  }
}
```

With this code, a new "draw" command is created, executed, and undone as follows:

```
Deck deck = new Deck();
Command command = deck.createDrawCommand();
Card card = command.execute().get();
command.undo();
```

When `command.execute()` executes, the code in the anonymous class calls `draw()` on the instance of `Deck` stored in variable `deck`, because anonymous classes retain a reference to their outer instance. The resulting card is then stored in a field of the anonymous class, which can then be used by the `undo()` method. The `undo` method also accesses the `Deck` instance through an implicit reference to its outer instance. Because the code is defined within class `Deck`, a reference to the private member `aCards` is possible. The Code Exploration section discusses another similar example application of the COMMAND pattern.

Independently of the specific way the pattern is applied, having command objects gives us much flexibility for managing how and when to execute commands on a `Deck` instance.

6.9 The Law of Demeter

When designing a piece of software using aggregation, one can often end up with long delegation chains between objects. For example, Figure 6.17 models the aggregation for card piles in the Solitaire example application.

Fig. 6.17 Aggregation structure for foundation piles in Solitaire

In this design, a `GameModel` object holds a reference to an instance of `Foundations` to manage the four piles of cards of a single suit. In turn, an instance of `Foundations` holds references to four `CardStack` instances, which are specialized wrappers around `List` objects, etc.

There are different ways to use such delegation chains. Figure 6.18 illustrates a hypothetical way to use the aggregation structure for adding a card to a pile.

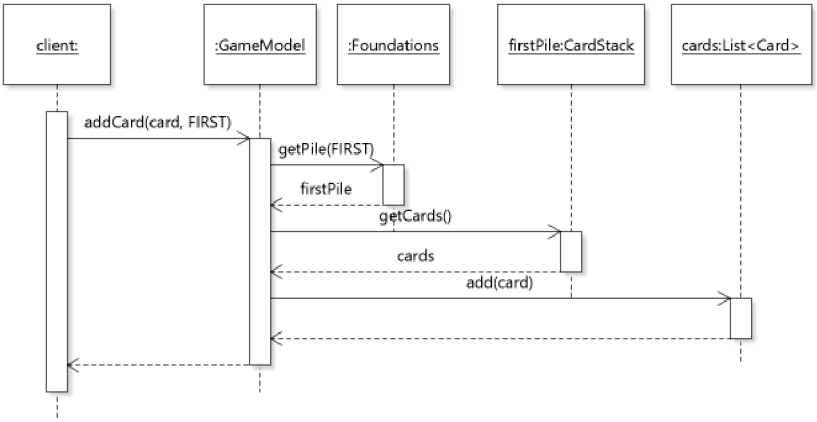


Fig. 6.18 Sample data structure access scenario for the Solitaire game design

In this design the `GameModel` is in charge of all the details of adding a card to a pile, and must handle every intermediate object in the delegation chain. As an

example, in this design the implementation of `addCard` in class `GameModel` would look like this:

```
aFoundations.getPile(FIRST).getCards().add(pCard);
```

This design violates the principle of information hiding by requiring the code of the `GameModel` class to know about the precise navigation structure required to add a card to the system. Although this might be obvious in the case of a `CardStack` returning its `List<Card>`, exactly the same argument can be made for `Foundations` returning one of its `CardStack`. However, the encapsulation quality of intermediate classes in aggregation chains is easier to overlook. The intuition that designs such as this one tend to be suboptimal is captured by the MESSAGE CHAIN† antipattern. The Law of Demeter is a design guideline intended to help avoid the consequences of MESSAGE CHAIN†. This "law" is actually a design guideline that states that the code of a method should only access:

- The instance variables of its implicit parameter;
- The arguments passed to the method;
- Any new object created within the method;
- (If need be) globally available objects.

So, to respect this guideline it becomes necessary to provide additional services in classes that occupy an intermediate position in an aggregation/delegation chain so that the clients do not need to manipulate the internal objects encapsulated by these objects. The solution in our example would be illustrated by Figure 6.19.

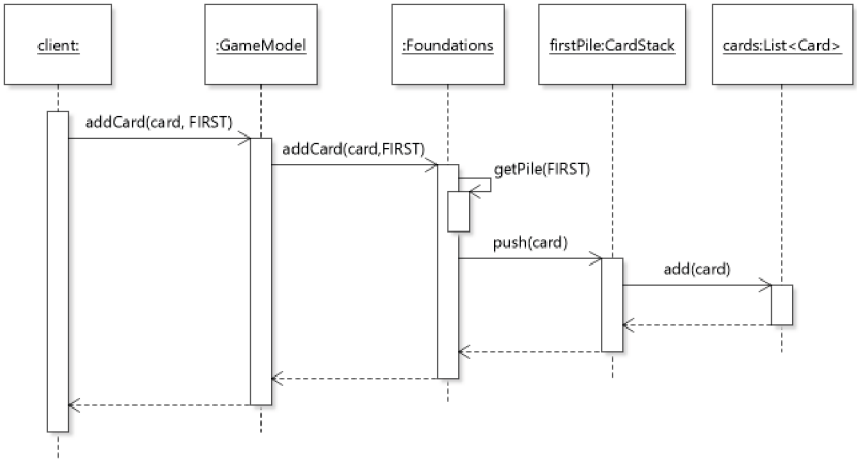


Fig. 6.19 Sample data structure access scenario for the Solitaire game design, which respects the Law of Demeter

In this solution, objects do not return references to their internal structure, but instead provide the complete service required by the client at each step in the delegation chain.