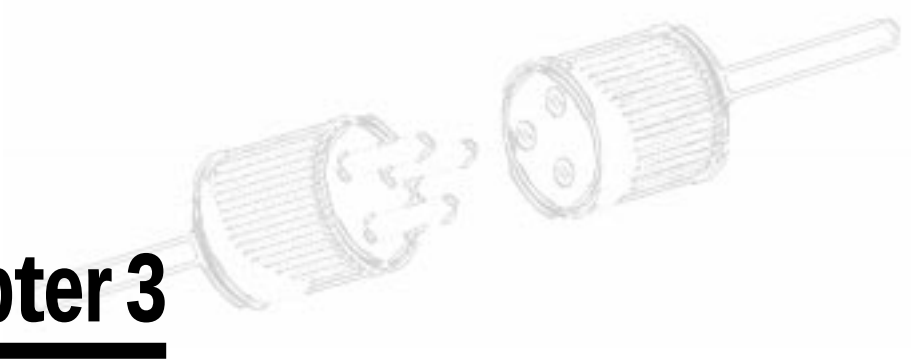

Chapter 3

Design with Interfaces

This chapter explores Java-style interfaces: what they are, why they are important, and how and when to use them.

3.1 What Are Interfaces?

Interfaces are the key to pluggability, the ability to remove one component and replace it with another. Consider the electrical outlets in your home: The interface is well-defined (plug shape, receptacle shape, voltage level, polarity for each prong); you can readily unplug a toaster and plug in a java-maker, and continue on your merry way.

Design with interfaces? Yes!

An *interface* is a collection of method signatures that you define for use again and again in your application. It's a listing of method

signatures alone. There is neither a common description, nor any source code behind these method signatures.*

An interface describes a standard protocol, a standard way of interacting with objects in classes that implement the interface.

Working with interfaces requires that you (1) specify the interface and (2) specify which classes implement that interface.

Begin with a simple interface, called IName (Figure 3-1). IName consists of two method signatures, the accessors getName and setName.

---

*Java expresses inheritance and polymorphism distinctly with different syntax. C++ expresses both concepts with a single syntax; it blurs the distinction between these very different mechanisms, resulting in overly complex, overly deep class hierarchies. (We design with interfaces regardless of language; Java makes it easier for us to express that design in source code.)

In Smalltalk, interfaces (called protocols) are agreed upon by convention and learned by reading source code. In C++, interfaces are implemented as classes with no default implementation (everything inside is declared as being “pure virtual”).

Java interfaces can also include constants. This provides a convenient way to package useful constants when programming, but it has no impact on effective design.

Within the Java Language Specification, a signature is defined in a narrower way, describing what a Java compiler must pay attention to when resolving overloaded methods. In that document, an interface consists of a method name and the number and types of parameters—not the return type, not the name of the parameters, and not any thrown exceptions. For overridden methods (in an extending/implementing class, using the same method name and the same number and types of parameters), a Java compiler checks to make sure that the return type is the same and the thrown exceptions are the same.

UML offers a definition with the same basic meaning as the one we use in this book: "An interface is a declaration of a collection of operations that may be used for defining a service offered by an instance."

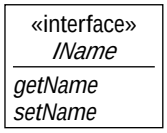

Figure 3-1. An interface.

By convention, interface names are capitalized: the IName interface. References to an object of a class that implements an interface are not capitalized: a name object, meaning, an object in a class that implements IName.

By one convention, interface names end with the suffix "-able," "-ible," or (occasionally) "-er."*

By another convention, interface names begin with the prefix "I."

By convention in this book, interface names begin with the prefix "I" and are followed by

- a noun, if it's an accessor interface
- a verb, if it's a calculation interface, or
- a noun or a verb, if it's a combination of interfaces.**

In Figure 3-1 the interface name is "I" + a noun.

---

*Requiring interface names to end in -able or -ible is a bit too complicated a convention. However, if you'd like to adopt this convention, take note of the following English-language spelling rules:

1. Drop a silent "e" before adding "-able."
2. Check a dictionary. If the spelling is not listed, look at other forms of the word to see which letter might make sense. (Again, this is a bit too complicated for day-to-day use.)

**Choose whatever prefix convention you prefer: I, I_, Int_; whatever. We prefer "I" (as long as it does not conflict with other prefix conventions of the project).

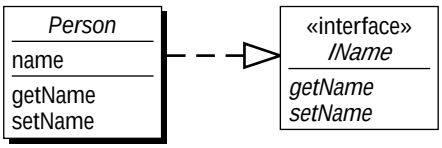

Figure 3-2. A class that promises to implement the IName interface.

In Java, an IName interface might look something like this:

```
public interface IName {
    String getName();
    voidsetName(String aName); }
```

A class that implements the IName interface promises to implement the "get name" and "set name" methods in a way that is appropriate for that class. The "get name" method returns the name of an object. The "set name" method establishes a new name for an object (Figure 3-2). A dashed arrow indicates that a class (at the tail of the arrow) implements the interface (at the head of the arrow).

The IName interface describes a standard way to interact with an object in any class that implements that interface.

This means that as an object in any class, you could hold an IName object (that is, objects within any number of classes that implement the IName interface). And you could ask an IName object for its name without knowing or caring about what class that object happens to be in.

3.2 Why Use Interfaces?

3.2.1 The Problem

Over the years, you may have encountered the classic barrier to:

- flexibility (graciously accommodating changes in direction)
- extensibility (graciously accommodating add-ons), and
- pluggability (graciously accommodating pulling out one class of objects and inserting another with the same method signatures).

Yes, this is a barrier within object-oriented design.

All objects interact with other objects to get something done. An object can answer a question or calculate a result all by itself, but even then some other object does the asking. In short, objects interact with other objects. That's why scenario views are so significant, because they model time-ordered sequences of interactions between objects.

The problem with most class diagrams and scenarios is that an object must be within a specified class.

Yet what is the element of reuse? It's not just a class. After all, objects in a class are interconnected with objects in other classes. The element of reuse is some number of classes, the number of classes in a scenario, or even more, the total number of classes contained in overlapping scenarios.

What's the impact, in terms of pluggability? If you want to add another class of objects, one that can be plugged in as a substitute for an object in another class already in a scenario, you are in trouble. There is no pluggability here. Instead, you must add associations, build another scenario, and implement source code behind it all.*

The problem is that each association and each message-send is hardwired to objects in a specific class (or class hierarchy), impeding pluggability, as well as extensibility and flexibility.

Traditionally, objects in a scenario are hardwired to each other. But if the "who I know" (associations) and "who I interact with" (messages) are hardwired to just one class of objects, then pluggability is nonexistent; adding a new class means adding the class itself, associations, and scenarios, in addition to making changes to other classes in the design and in source code.

---

*In C++, developers often implement monolithic class hierarchies with a base class that does nothing more than allow the ease of "pluggability" via base class pointers. This is a bulky and limited workaround compared to the elegance of Java interfaces.

3.2.2 A Partial Solution

We'd like a more flexible, extensible, and pluggable approach, one that would let us add in new classes of objects with no change in associations or message-sends.

There is a partial solution.

If you want to add a new class that is a subclass of one of the classes of objects participating in a scenario, you can do so without any problems. Show the generalization class in your scenario, add a note indicating that any object in a specialization class will do, and you are ready to go.

However, if inheritance does not apply, or if you have already used inheritance in some other way (keeping in mind that Java is a single inheritance language), then this partial solution is no solution at all.

3.2.3 Flexibility, Extensibility, and Pluggability—That's Why

Interfaces enhance, facilitate, and even make possible the flexibility, extensibility, and pluggability that we so desire.

Interfaces shift one's thinking about an object and its associations and interactions with other objects.

Challenge Each Association Strategy: *Is this association hardwired only to objects in that class (simpler), or is this an association to any object that implements a certain interface (more flexible, extensible, pluggable)?*

For an object and its associations to other objects ask, "Is this association hardwired only to objects in that class, or is this an association to any object that implements a certain interface?" If it's the latter, you are in effect saying, "I don't care what kind of object I am associated with, just as long as that object implements the interface that I need."

Interfaces also shift one's thinking about an object and the kinds of objects that it interacts with during a scenario.

Challenge Each Message-Send Strategy: *Is this message-send hardwired only to objects in that class (simpler), or is this a message-send to any object that implements a certain interface (more flexible, extensible, pluggable)?*

For each message-send to another object ask, "Is this message-send hardwired only to objects in that class, or is this a message-send to any object that implements a certain interface? If it's the latter, you are in effect saying, "I don't care what kind of object I am sending messages to, just as long as that object implements the interface that I need."

So, when you need flexibility, specify associations (in class diagrams) and message-sends (in scenarios) to objects in *any* class that implements the interface that is needed, rather than to objects in a *single* class (or its subclasses).

Interfaces loosen up coupling, make parts of a design more interchangeable, and increase the likelihood of reuse—all for a modest increase in design complexity.

Interfaces express "is a kind of" in a very limited way, "is a kind that supports this interface." This gives you the categorization benefits of inheritance; at the same time, it obviates the major weakness of inheritance: weak encapsulation within a class hierarchy.

Interfaces give composition a much broader sphere of influence. With interfaces, composition is flexible, extensible, and pluggable (composed of objects that implement an interface), rather than hardwired to just one kind of object (composed of objects in just one class).

Interfaces reduce the otherwise compelling need to jam many, many classes into a class hierarchy with lots of multiple inheritance. In effect, using interfaces streamlines how one uses inheritance: use interfaces to express generalization-specialization of

method signatures (behavior); use inheritance to express generalization-specialization of interfaces implemented—along with additional attributes and methods.

Interfaces give you a way to separate method signatures from method implementations. So you can use them to separate UI method signatures from operating-system dependent method implementations; that's exactly what Java's Abstract Windowing Toolkit (AWT) and Swing do. You can do the same for data management, separating method signatures from vendor-dependent method implementations. You can also do the same for problem-domain objects, as you'll see later in this chapter.

Sound-bite summary: Why use interfaces? Interfaces give us a way to establish associations and message-sends to objects in any class that implements a needed interface, without hardwiring associations or hardwiring message-sends to a specific class of objects.

The larger the system and the longer the potential life span of a system, the more significant interfaces become.

3.3 Factor-out Interfaces

Factoring out every method signature into a separate interface would be overkill—you'd make your object models more complex and your scenarios way too abstract.

In what contexts should you apply interfaces?

You can factor out method signatures into interfaces in a variety of contexts, but the following are the four contexts in which interfaces really help:

Factor out repeaters.

Factor out to a proxy.

Factor out for analogous apps.

Factor out for future expansion.

3.3.1 Factor Out Repeaters

Begin with the simplest use of interfaces: to factor out common method signatures to bring a higher level of abstraction (and an overall visual simplification) to a class diagram. This is a modest yet important use of interfaces.

Factor Out Repeaters Strategy: *Factor out method signatures that repeat within your class diagram. Resolve synonyms into a single signature. Generalize overly specific names into a single signature. Reasons for use: to explicitly capture the common, reusable behavior and to bring a higher level of abstraction into the model.*

Look for repeating method signatures and factor them out.

Example: calcTotal in one class, calcTotal in another class.

Factor out that method signature into an ITotal interface.

Mark each class as one that implements the ITotal interface.

Now look for method signatures that are synonyms. Pick a common method signature and factor it out.

Example: calcTotal in one class, determineTotalAmount in another class. Same behavior.

Pick a synonym: calcTotal.

Factor out that method signature into an ITotal interface.

Mark each class as one that implements the ITotal interface.

Next take each method signature and generalize it. (But be careful not to generalize to the point of obscurity; a method name like "process it" or "calculate it" would not be very helpful, would it?)

Then look for method signatures that are synonyms; finally, pick a common method signature and factor it out.

> Example: calcSubtotal in one class, calcTotal in another class, calcGrandTotal in another class.
>
> Pick a synonym: calcTotal.
>
> Factor out that method signature into an ITotal interface.
>
> Mark each class as one that implements the ITotal interface.

When factoring out interfaces, you also need to consider the return types and the parameter types; they must match up, too. In fact in a class diagram, you could include a complete method signature:

> return type + method name + parameter types + exceptions

However, including all of that information in a class diagram takes up far too much screen real estate. It is far better to have an effective class diagram of the design plus source code with fine-grained details, side by side.

3.3.1.1 Example: The Lunch Counter at Charlie's Charters

Okay then, apply the "Factor Our Repeaters" strategy. Consider a point-of-sale application for the lunch counter at Charlie's Charters.

Build an initial class diagram (Figure 3-3).

In Java, it looks like this:

```
public class Customer {
✂
    // methods / public / conducting business
    public BigDecimal howMuch() { /* code goes here */ }
✂
}
```

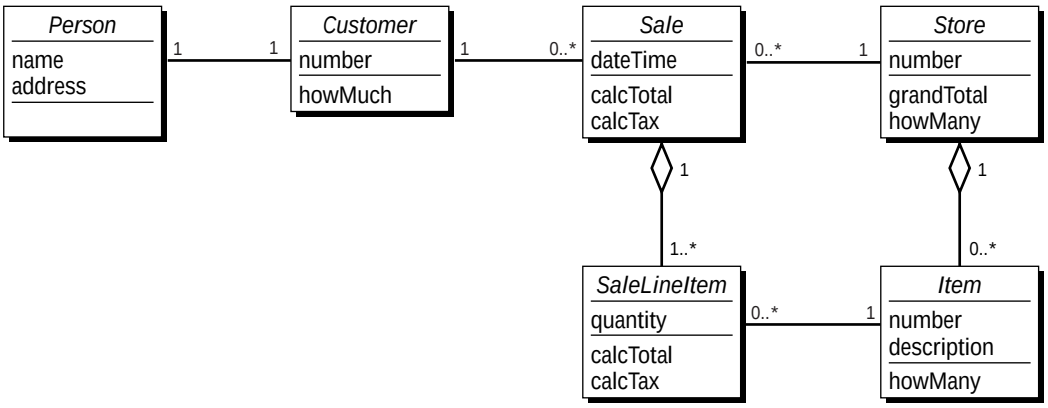

Figure 3-3. Repeating method signatures.

```
public class Sale {
⁓
    // methods / public / conducting business
    public BigDecimal calcTotal() { /* code goes here */ }
    public BigDecimal calcTax() { /* code goes here */ }
⁓
}
public class SaleLineItem {
⁓
    // methods / public / conducting business
    public BigDecimal calcTotal() { /* code goes here */ }
    public BigDecimal calcTax() { /* code goes here */ }
⁓
}
public class Store {
⁓
    // methods / public / conducting business
    public BigDecimal grandTotal() { /* code goes here */ }
    public int howMany() { /* code goes here */ }
⁓
}
public class Item {
⁓
    // methods / public / conducting business
```

```
    public int howMany() { /* code goes here */ }
✂
}
```

Applying the "factor out repeaters" strategy:

You can factor out calcTotal without any problem.

Now look for synonyms.

The methods calcTotal and howMany could be synonyms, but they have distinct meanings here (adding monetary units versus tallying some items, respectively).

Moreover, the return types don't match. This is a problem. We could check the return types to see if they too are synonyms; or we could try generalizing each return type to see if that helps. In this case, however, calcTotal returns a BigDecimal number; howMany returns an integer. You cannot combine different method signatures into a single interface method signature.

Keep looking. The calcTotal and howMuch methods are synonyms, and the return types match (both return a BigDecimal value). One or the other will do just fine; choose calcTotal and factor it out.

Looking further, grandTotal is a specialized name for calcTotal. Use calcTotal for both.

What are the common method signatures? Let's see:

- howMany—occurs twice
- calcTax—occurs twice
- calcTotal, how much (synonyms here)—occurs four times.

You can factor out those common method signatures, using these interfaces:

- ICount—how many
- ITax—calcTax
- ITotal—calcTotal.

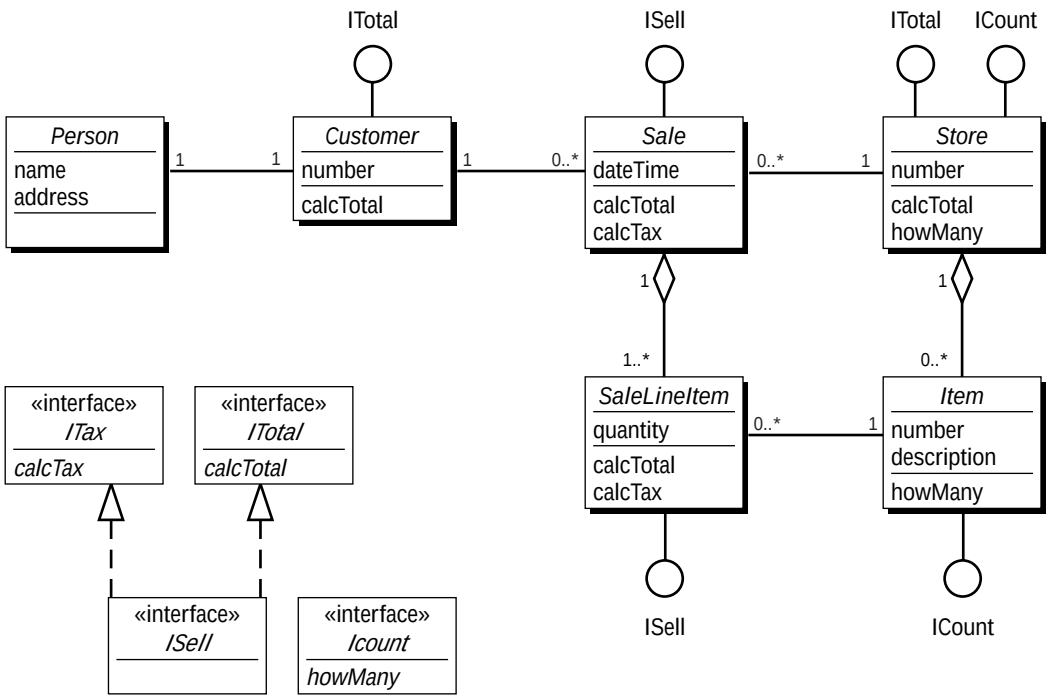

Figure 3-4. Factor out repeating method signatures.

You can go a step further. What common interface combinations are we using?

- ITotal, ITax—occur together, twice.

So you can combine those two interfaces, with this result:

- ISell—ITotal, ITax.

The result? See Figure 3-4. The "lollipops" indicate interface implementers. Another way to indicate interface implementers is with implement links, dashed arrows from implementers to interfaces. Convention: use interface links until they overpower your class diagram, then switch over to lollipops to avoid link overload.

In Java, it looks like this:

```
public interface ICount {
      int howMany(); }

public interface ITotal {
      BigDecimal calcTotal(); }

public interface ITax {
      BigDecimal calcTax(); }

public interface ISell extends ITotal, ITax {}
public class Customer implements ITotal {
      // methods / public / ITotal implementation
      public BigDecimal calcTotal() { /* code goes here */ }
}
public class Sale implements ISell {
      // methods / public / ISell implementation
      public BigDecimal calcTotal() { /* code goes here */ }
      public BigDecimal calcTax() { /* code goes here */ }
}
public class SaleLineItem implements ISell {
      // methods / public / ISell implementation
      public BigDecimal calcTotal() { /* code goes here */ }
      public BigDecimal calcTax() { /* code goes here */ }
}
public class Store implements ITotal, ICount {
      // methods / public / ITotal implementation
      public BigDecimal calcTotal() { /* code goes here */ }
      // methods / public / ICount implementation
      public int howMany() { /* code goes here */ }
}
```

```
public class Item implements ICount {
↩
    // methods / public / ICount implementation
    public int howMany() { /* code goes here */ }
↩
}
```

Especially note this:

```
public interface ISell extends ITotal, ITax {}
```

Here, an interface extends two other interfaces. Is this Multiple inheritance?

Well, yes and no.

Yes, the new interface is a combination of the other two interfaces. Yes, ISell is a special kind of ITotal and a special kind of ITax.

No, it's not inheritance; only method signatures are involved. There is absolutely no implementation behind these method signatures.

We really don't think of it as inheritance, either.

We think of interfaces as useful method-signature descriptions, ones that we can conveniently mix and match with the "extends" keyword to provide pluggability.

One way to visualize it is to picture a stack of index cards; each card has an interface name and its method signatures on it; grab whatever combination is useful to you (ITotal, ITax); name that useful combination (ISell)—especially if it is reusable.

3.3.1.2 Example: Simplify and Identify Object-Model Patterns

Together with David North, we have cataloged 31 object-model patterns: templates of objects with stereotypical responsibilities and interactions. Those patterns are documented at www.oi.com/handbook and (more thoroughly) in the book, *Object Models: Strategies, Patterns, and Applications*.

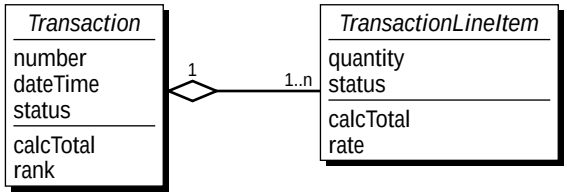

Figure 3-5. The transaction-transaction line item object-model pattern.

One of the more puzzling matters has been how to show these patterns within source code. Some have proposed adding extra classes of objects to manage each pattern, but that seemed like overkill somehow.

Interfaces offer an interesting twist. And the simplest use of interfaces, factoring out common method signatures, takes on some added significance.

Consider the transaction pattern called "transaction–transaction line item" (Figure 3-5).

Other patterns use attributes and methods with exactly the same names. So everything can be factored out into interfaces.

For full impact, first add in attribute accessors (Figure 3-6).

In Java, it looks like this:

```
public class Transaction {
✂
    // attributes / private
    private int number;
    private Date dateTime;
    private String status;

    // attributes / private / associations
    private Vector transactionLineItems = new Vector();

    // methods / public / conducting business
    public float calcTotal() { /* code goes here */ }
```

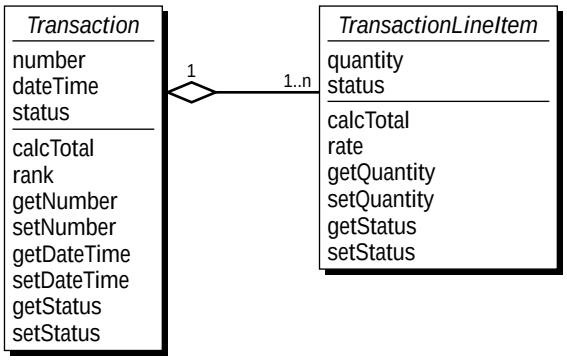

Figure 3-6. An object-model pattern with attribute accessors.

```
    public Enumeration rank() {
        /* return an enumeration with ranked transaction line items */
        /* code goes here */ }

    // methods / public / accessors for attribute values
    public int getNumber() { return this.number; }
    public void setNumber(int aNumber) { this.number = aNumber; }
    public Date getDateTime() { return this.dateTime; }
    public void setDateTime(Date aDateTime)
        { this.dateTime = aDateTime; }
    public String getStatus() { return this.status; }
    public void setStatus(String aStatus) { this.status = aStatus; }
✂
}

public class TransactionLineItem {
✂
    // attributes / private
    private int quantity;
    private String status;

    // attributes / private / associations
    private Transaction transaction;

    // methods / public / conducting business
    public float calcTotal() { /* code goes here */ }
    public int rate() { /* code goes here */ }
```

```
    // methods / public / accessors for attribute values
    public int getQuantity() { return this.quantity; }
    public void setQuantity(int aQuantity) { this.quantity = aQuantity; }
    public String getStatus() { return this.status; }
    public void setStatus(String aStatus) { this.status = aStatus; }
✂
}
```

Second, apply the "factor out repeaters" strategy (Figure 3-7).

In Java, it looks like this:

```
public interface IRank {
    Enumeration rank(); }
```

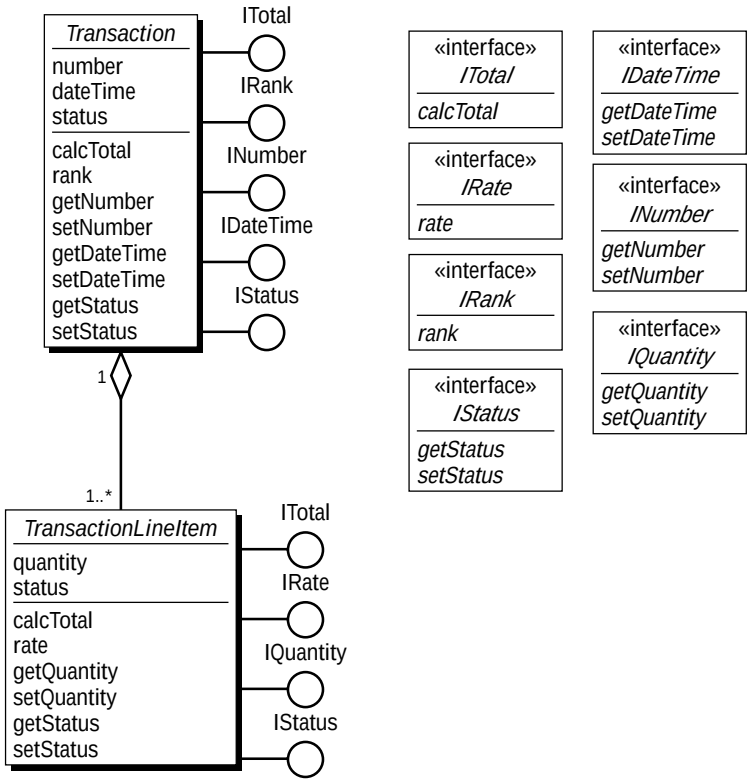

Figure 3-7. Factor out repeaters.

```
public interface IRate {
      int rate(); }

public interface ITotal {
      float calcTotal() ; }

public interface INumber {
      int getNumber();
      void setNumber(int aNumber); }

public interface IDateTime {
      Date getDateTime();
      void setDateTime(Date aDate); }

public interface IQuantity {
      int getQuantity();
      void setQuantity(int aQuantity); }

public interface IStatus {
      String getStatus();
      void setStatus(String aStatus); }

public class Transaction
      implements IRank, ITotal, INumber, IDateTime, IStatus {
✂
            // class definition here
✂
}

public class TransactionLineItem
      implements IRate, ITotal, IQuantity, IStatus {
✂
            // class definition here
✂
}
```

Now, go for the gold: factor out the interfaces within each "pattern player," making pattern players explicit in the design (and ultimately, in source code). See Figure 3-8.

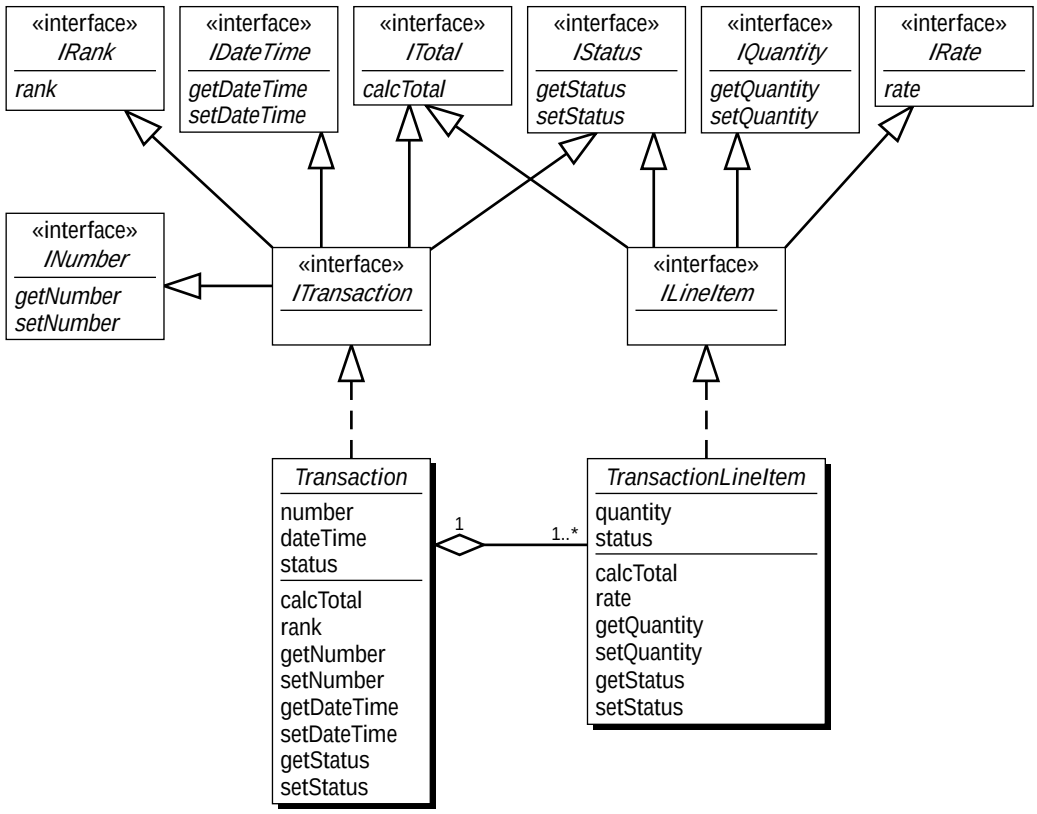

Figure 3-8. Factor out completely, so you can mark out pattern players.

In Java, it looks like this:

```
public interface ITransaction
      extends ITotal, IRank, INumber, IDateTime, IStatus {}

public interface ILineItem
      extends ITotal, IRate, IQuantity, IStatus {}

public class Transaction implements ITransaction {
```

```
✂
        // class definition here
✂
}

public class TransactionLineItem implements ILineItem {
✂
        // class definition here
✂
}
```

3.3.2 Factor Out to a Proxy

Factor Out to a Proxy Strategy: *Factor out method signatures into a proxy, an object with a solo association to some other object. Reason for use: to simplify the proxy within a class diagram and its scenarios (Figure 3-9).*

3.3.2.1 Recognizing a Proxy

Another way to bring interfaces into your design is to factor out method signatures into a proxy. A proxy is one who acts as a substitute on behalf of another. Consider person and passenger in Charlie's Charters' reservation system, this time with get and set accessors included (Figure 3-9).

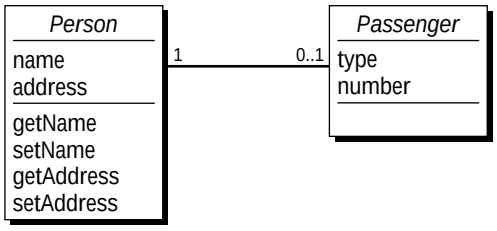

Figure 3-9. Person with accessors.

In Java, it looks like this:

```
public class Person {
    // attributes / private
    private String name;
    private String address;

    // attributes / private / associations
    private Passenger passenger;

    // methods / public / accessors for attribute values
    public String getName() { return this.name; }
    public void setName(String aName) { this.name = aName; }
    public String getAddress() { return this.address; }
    public void setAddress(String anAddress)
        { this.address = anAddress; }

    // methods / public / accessors for association values
    public void addPassenger(Passenger aPassenger) {
        this.passenger = aPassenger; }
    public void removePassenger() { this.passenger = null; }
    public Passenger getPassenger() { return this.passenger; }
}

public class Passenger {
    // attributes / private
    private int number;
    private String type;

    // attributes / private / associations
    private Person person;

    // methods / public / accessors for attribute values
    public String getNumber() { return this.number; }
    public void setNumber(int aNumber) { this.number = aNumber; }
    public String getType() { return this.type; }
    public void setType(String aType)
        { this.type = aType; }
```

```
        // methods / public / accessors for association values
        public Person getPerson() { return this.person; }

        // constructors
        // notice that there is no *default* constructor; a passenger must have
        // a corresponding person object.
        public Passenger(Person aPerson) {
            // implicit call to superclass constructor super();
            this.person = aPerson; }
✂
}
```

Passenger has a "one and only one" association with a person object. Whenever an object (Passenger) has a "one and only one" association with another object (Person), then that object (Passenger) can act as a proxy for the other (Person).

3.3.2.2 Life without a Proxy

Proxy? Why bother? Well, consider this "before" picture, where you don't have one object acting as a proxy for another. Suppose that you've identified a passenger object, and would like to know its name and address. What does the scenario look like? Ask a passenger, delegate to a person—explicitly. Again and again. There must be a better way to deal with this (Figure 3-10.)

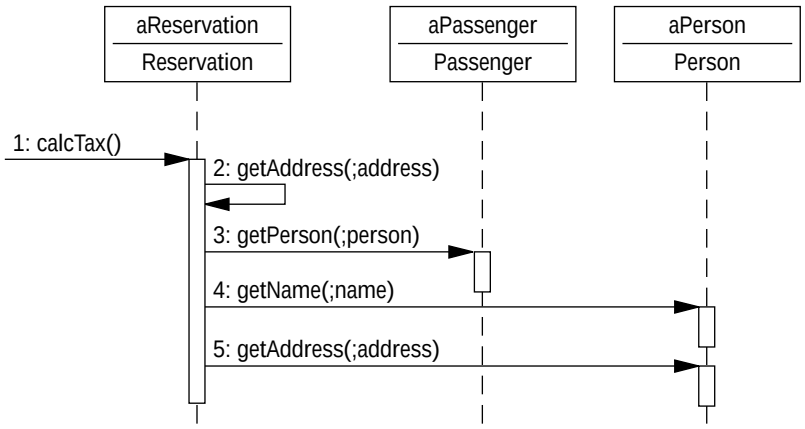

Figure 3-10. Asking a passenger for its person object, then asking a person object for its name and address.

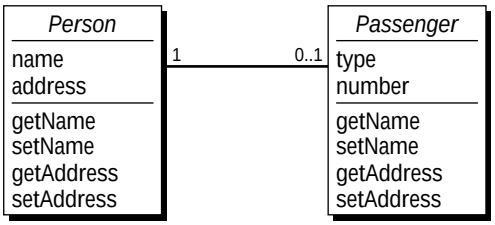

Figure 3-11. Person and Passenger, both with accessors.

3.3.2.3 Life with a Proxy

A proxy answers questions on behalf of another, and it provides a convenient interface. See Figure 3-11.

A proxy-based scenario is shown in Figure 3-12.

In Java, it looks like this:

```
public class Passenger {
✂
    // methods / public / accessors for Person's attribute values
    public String getName() { return this.person.getName(); }
    public void setName(String aName) { this.person.setName(aName); }
    public String getAddress() { return this.person.getAddress(); }
```

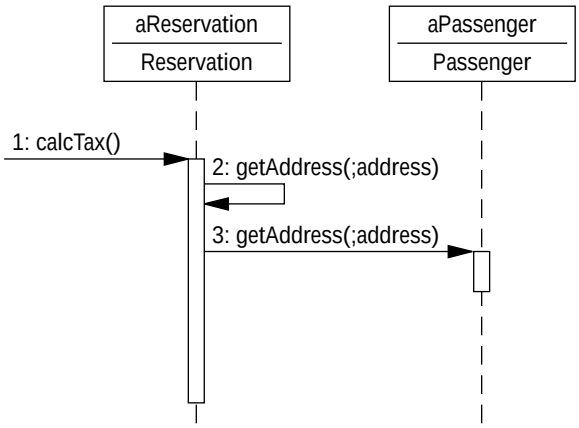

Figure 3-12. Asking a proxy for what you need.

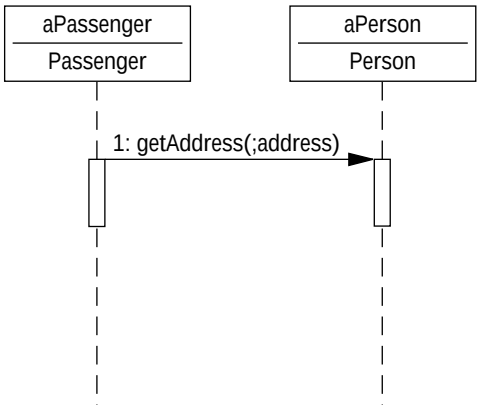

Figure 3-13. Behind the scene: a proxy interacting with the one it represents (boring).

```
    public void setAddress(String anAddress)
        { this.person.setAddress(anAddress); }
✂
}
```

Now you can ask a passenger for its name and address rather than asking a passenger for its person object and then interacting with that person object.

Yes, a passenger object still privately interacts with its person object. We could show that interaction, as illustrated in a separate scenario view (Figure 3-13).

But that really is rather boring and not something we would normally sketch out.

Hence, with a proxy, scenarios become simpler; the details about whomever is being represented by the proxy are shielded from view, letting the important stand out, improving effective communication—a good thing.

3.3.2.4 Introducing a Proxy Interface

Now let's bring interfaces into the picture. Factoring out commonality yields Figure 3-14.

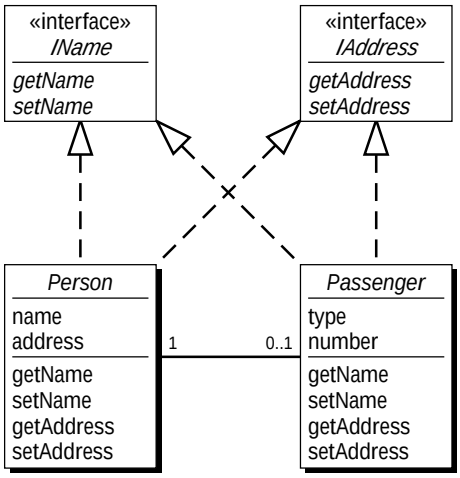

Figure 3-14. Person and Passenger, with common interfaces.

In Java, it looks like this:

```
public interface IName {
      String getName();
      void setName(String aName);
}

public interface IAddress {
      String getAddress();
      void setAddress(String anAddress);
}

public class Person implements IName, IAddress {
      // class definition here
}

public class Passenger implements IName, IAddress {
      // class definition here
}
```

You can combine these two interfaces as shown in Figure 3-15.

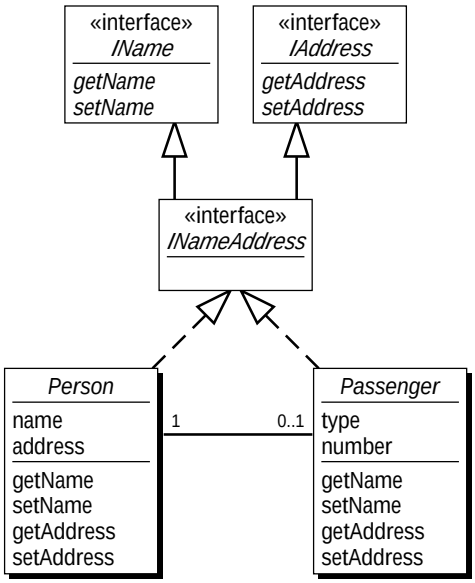

Figure 3-15. Person and Passenger with a single, combined interface.

In Java, it looks like this:

```
public interface INameAddress extends IName, IAddress {}

public class Person implements INameAddress {
    // class definition here
}

public class Passenger implements INameAddress {
    // class definition here
}
```

Now bring agent into the picture (Figure 3-16).

In Java, it looks like this:

```
public class Person implements INameAddress {
```

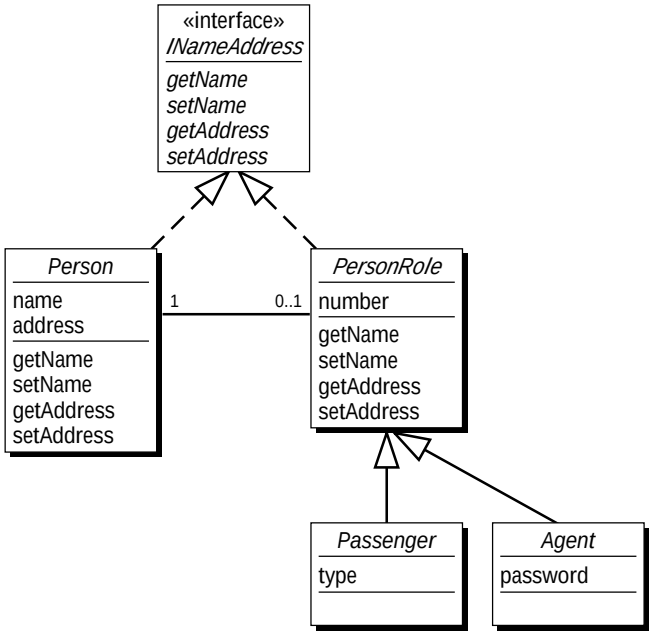

Figure 3-16. A person is composed of one or more person roles; a person role specializes into different kinds of person roles.

```
{
        // class definition here
}
}

public abstract class PersonRole implements INameAddress {
{
        // class definition here
}
}

public class Passenger extends PersonRole {
{
        // class definition here
}
}

public class Agent extends PersonRole {
{
```

```
        // class definition here
⁓
}
```

Now consider a NameAddressUI object.

It's a user interface (UI) object, one that contains a number of smaller, handcrafted or GUI–builder-generated UI objects: text fields, buttons, scrollable lists, and the like.

In addition, and more importantly (from an object-modeling perspective), a NameAddressUI object knows some number of objects in classes that implement the INameAddress interface.

The real power is that the NameAddressUI is not hardwired to objects in just one class. Instead, it works with objects from any class that implements the INameAddress interface (Figure 3-17).

In Java, it looks like this:

```
public class NameAddressUI {
⁓
    // attribute / private / association
    private Vector nameAddresses = new Vector();

    // method / public / accessor for object association values
    public void addNameAddress(INameAddress aNameAddress) {
        // only add objects of the type INameAddress to the vector
        this.nameAddresses.addElement(aNameAddress) ; }
⁓
}
```

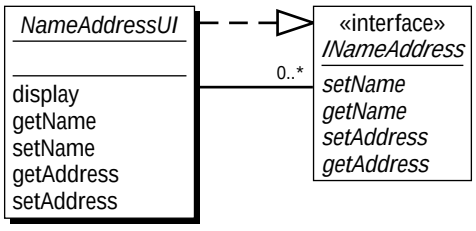

Figure 3-17. Each name-address UI object is composed of a collection of INameAddress objects.

Impact: interfaces change the very nature of an association, of one object knowing other objects. As an object, one's perspective shifts from, "I hold a collection of sale objects" to "I hold a collection of ISell objects," meaning, objects in *any* class that implements the ISell interface. Intriguing!

Here a UI object holds a collection of objects from any class that implements a specific interface. This shifts an object-model builder's attention to "what interface does that object need to provide?" rather than "what class(es) of objects should I limit myself to?"

With interfaces an object model gains better abstraction and simpler results. The implementation also benefits from this simplification.

Now, take a look at the corresponding scenario (Figure 3-18).

Additional impact: interfaces change the heart and soul of working out dynamics with scenarios. A scenario is a time-ordered sequence of object interactions. Now, as an object in a scenario, one's perspective shifts from, "I send a message to a sale object" to "I send a message to an ISell object," meaning, an object in *any* class that implements the ISell interface. Doubly intriguing!

In this scenario, a UI object sends a message to any object in a class that implements the needed interface. For the receiving object, it no longer matters where its class is in the class hierarchy, and it no longer matters if its class spells out a different implementation (time vs. size tradeoffs will always be with us).

With interfaces, your attention shifts from "what class of objects am I working with now?" to "what's the interface and what's the interface that I need from whatever kind of object I might work with, now or in the future?"

With interfaces, you spend more time thinking about the interfaces that you need, rather than who might implement that interface.

With interfaces, each scenario delivers more impact. Redundancy across related scenarios goes down.

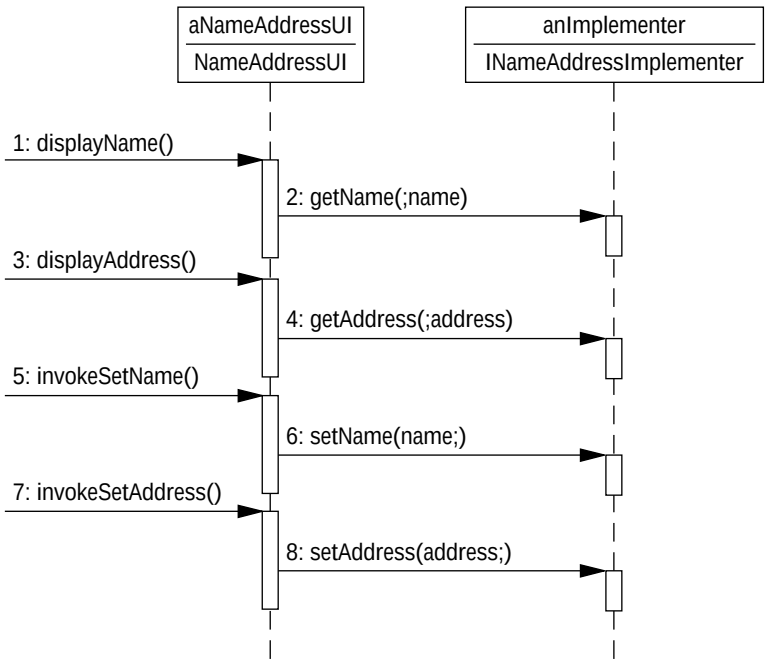

Figure 3-18. A name-address UI object, interacting with an INameAddress object.

What's the impact of interfaces? Reuse within the current app and greater likelihood of reuse in future apps. In addition, you gain simplified (easier to develop and maintain) models that are flexible, extensible, and support pluggability.

This is a nice outcome for relatively modest effort.

3.3.3 Factor Out for Analogous Apps

Factor Out for Analogous Apps Strategy: *Factor out method signatures that could be applicable in analogous apps. Reason for use: to increase likelihood of using and reusing off-the-shelf classes.*

You can use the "factor out repeaters" strategy to increase the level of abstraction within a class diagram and its scenarios within the problem domain you are currently working.

The "factor out for analogous apps" strategy takes an even broader perspective. You can use this strategy to achieve use and reuse across a family of analogous applications.

Here's how.

3.3.3.1 Categorize to Your Heart's Content

You can categorize business apps in different ways. If inheritance were your only categorization mechanism, you could go absolutely crazy. How could you decide upon just one or just a few ways to categorize what you are working on?

Now you have interfaces. You can use them to categorize classes of objects in multiple ways, across a variety of dimensions.

Consider business apps. Two key (yet certainly not all-inclusive) categories are sales and rentals. In a sales system, some goods are sold for a price. So we could categorize certain classes of objects as being sellable, perhaps reservable, too.

In a rental system, talent, equipment, or space is rented for a date or for an interval of time; the goods are still there, and are rented again and again and again. Here, we could classify certain classes of objects as being rentable, and perhaps reservable, too.

3.3.3.2 Categorize Charlie's Charters Business

How do we categorize Charlie's Charters business? Charlie's Charters is in the rental business: it rents space on a scheduled flight for a specific date.

For a flight description on Charlie's Charters, we can reserve space on a scheduled flight. We can ask it if a seat is available; we can ask it to reserve a seat; and we can ask it to cancel a reservation (Figure 3-19).

Now consider a UI object who knows one or more flight description objects. Without interfaces, it looks like Figure 3-20.

The corresponding scenario is shown in Figure 3-21.

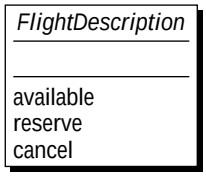

Figure 3-19. Methods for reserving space on a scheduled flight.

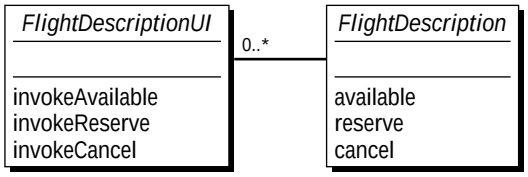

Figure 3-20. A UI class, custom crafted for a flight description.

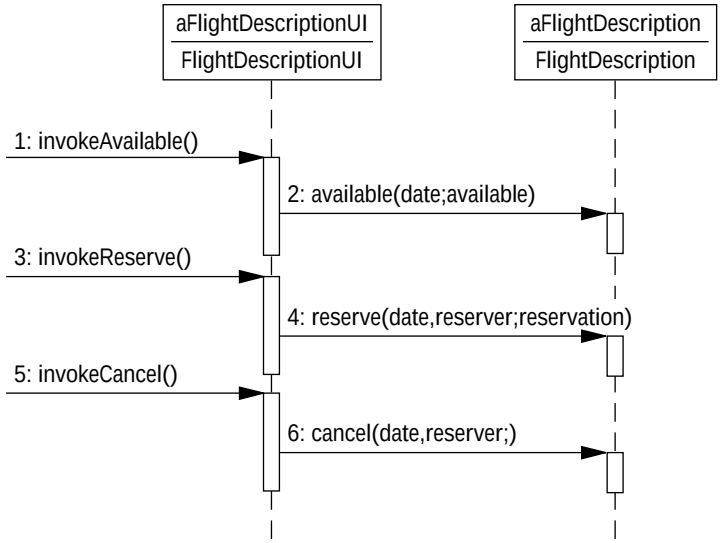

Figure 3-21. UI objects, interacting with objects in just one class (hardwired object interactions).

3.3.3.3 How Can Interfaces Help in This Context?

Charlie's Charters is a no-frills airline. It reserves space on a scheduled flight; it does not reserve specific seat numbers. (Adding SeatMap, Seat, and SeatAssignment classes would take care of that—not a big deal.)

For the Charlie's Charters app, we are interested in reserving space for a given date. We could use an interface called IDateReserve (see Figure 3-22).

We need to add the passenger as a parameter for reserve and cancel. However, since we want this interface to be general, the parameter type should be that of an Object. Let's give it the name "reserver,"—and so we have:

reserve (date, reserver) and

cancel (date, reserver).

Here is what it looks like in Java:

```
public interface IDateReserve {
    boolean available(Date aDate);
    Object reserve(Date aDate, Object reserver);
    boolean cancel(Date aDate, Object reserver); }
```

Code notes: available and cancel return boolean results. Reserve returns an object, keeping the interface flexible (we aren't needlessly limiting the interface to objects in a specific class or its subclasses). The object that gets that returned object must cast the result into whatever kind of object it expects to get back.

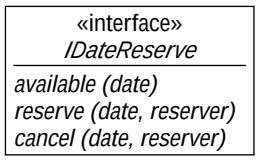

Figure 3-22. The IDateReserve interface.

Note that the method signatures are generalized a bit, so they can be applied within any system that has IDateReserve elements within it.

Why bother extracting this analogous interface? Simply put, we are looking for an interface that makes it easy for objects that know how to interact with that interface to "plug in" and make use of that interface. Having off-the-shelf UI components that sport commonly used interfaces saves design, development, and testing time. Very nice indeed.

For example, if you have an object that knows how to interact with an object in any class that implements IDateReserve, then you can use and reuse that object in any app with IDateReserve objects in it. Note that all you care about is the interface; you are free from having to consider the specific class or classes of objects that you might want to interact with. This gives new-found freedom within object-oriented design.

3.3.3.4 An Aside: Some Related Interfaces

A variation on this theme is IDateTimeReserve, which is not needed at Charlie's because a flight description specifies a time of departure. However, if we needed it, it would look like Figure 3-23.

Consider analogous systems such as other rental businesses.

For video rentals, you'd reserve a title for a date (for example, this Saturday). This is another case in which you could use that same IDateReserve interface.

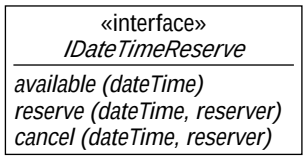

Figure 3-23. The IDateTimeReserve interface.

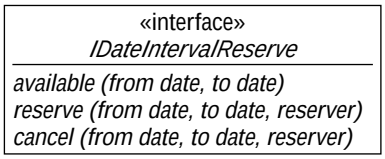

Figure 3-24. The IDateIntervalReserve interface.

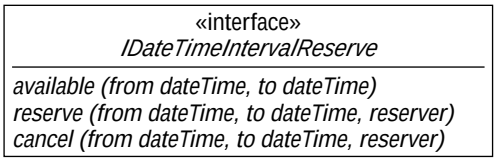

Figure 3-25. The IDateTimeIntervalReserve interface.

For hotel rooms, you'd be interested in reserving a certain kind of room (concierge level) for an interval of time (for example, from the fifth to the ninth). You could use an interface called IDateIntervalReserve (Figure 3-24).

For car rentals, you'd reserve a certain kind of car (full-size four-door) for an interval of time (for example, from the fifth at 5 PM until the ninth at 9 PM). You could use an interface called IDateTimeIntervalReserve (Figure 3-25).

3.3.3.5 Using IDateReserve for Charlie's Charters

For Charlie's Charters you need an IDateReserve interface as shown in Figure 3-26.

You can use or reuse any object that knows how to interact with an object in a class that implements the IDateReserve interface.

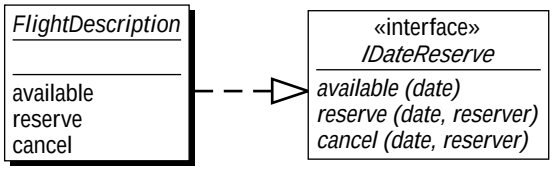

Figure 3-26. The flight description class implements the IDateReserve interface.

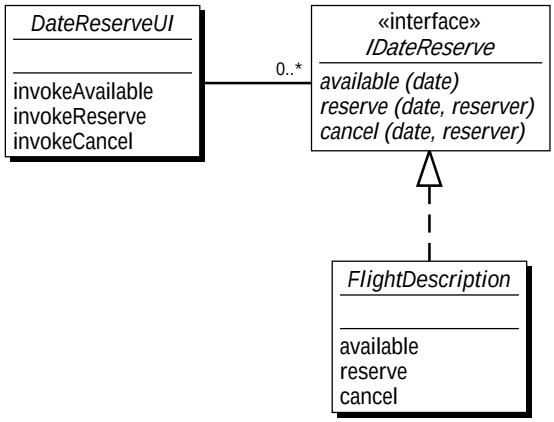

Figure 3-27. UI objects, connected to objects in classes that implement a given interface (flexible associations).

For example, a "date reservation" user interface could interact with an object in any class that implements IDateReserve—a flight reservation object, a video title object, and so on.

With interfaces you get new found flexibility. Now UI objects can connect with an object in any class that implements the correct interface (Figures 3-27 and 3-28).

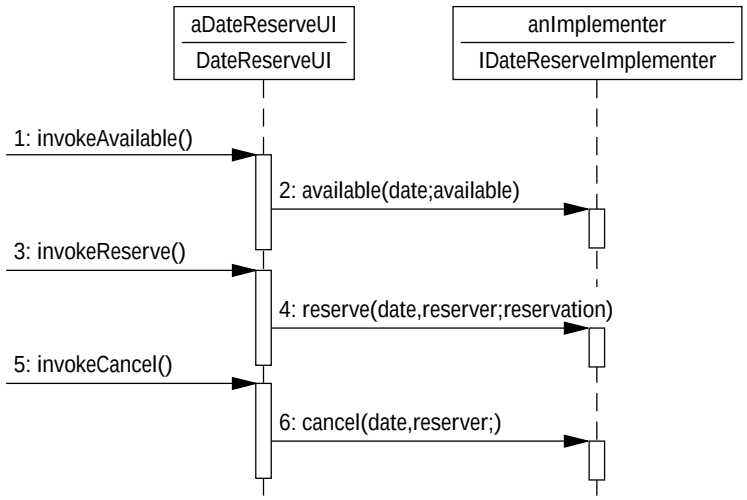

Figure 3-28. UI objects, interacting with objects in classes that implement a given interface (flexible object interactions).

With interfaces, our attention shifts from "what class of objects can I interact with?" to "what's the interface that I can interact with?"

3.3.6.6 Using IDateReserve in Other Apps

Let's consider another date reservation example. Suppose you are designing a system for a temporary help business in which each worker and each piece of equipment is reservable for a date. In this case, a "daily work order" object can interact with any objects in classes that implement the IDateReserve interface (Figures 3-29 and 3-30).

Today, a daily work order might be a collection of workers and pieces of equipment. Next year, it might be a collection of workers, pieces of equipment, and workspace.

What is the impact of change ?

Add a new class to your object model: Workspace. Be sure it implements the IDateReserve interface. Connect it with whatever associations it might need (Figure 3-31).

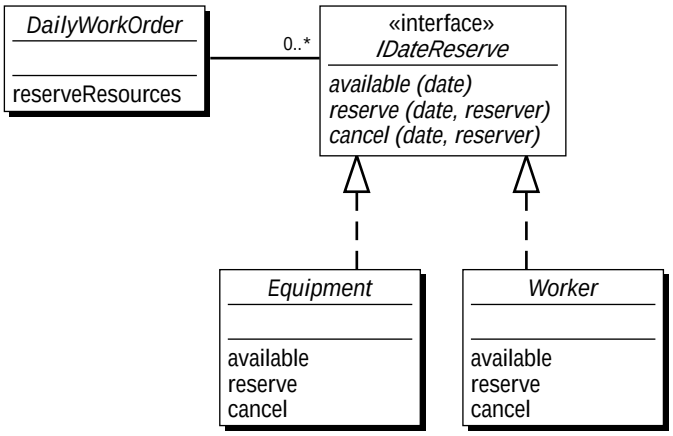

Figure 3-29. Each daily work order object is composed of a collection of IDateReserve objects.

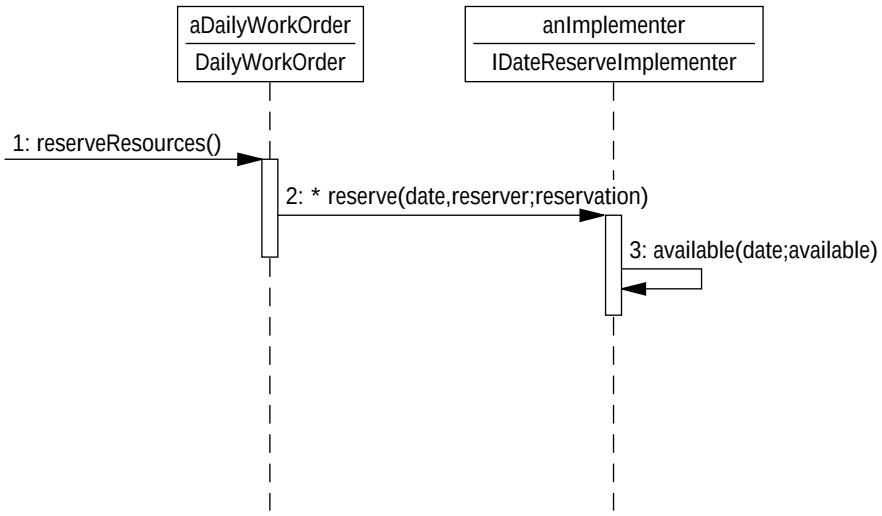

Figure 3-30. Each daily work order object interacts with its IDateReserve objects.

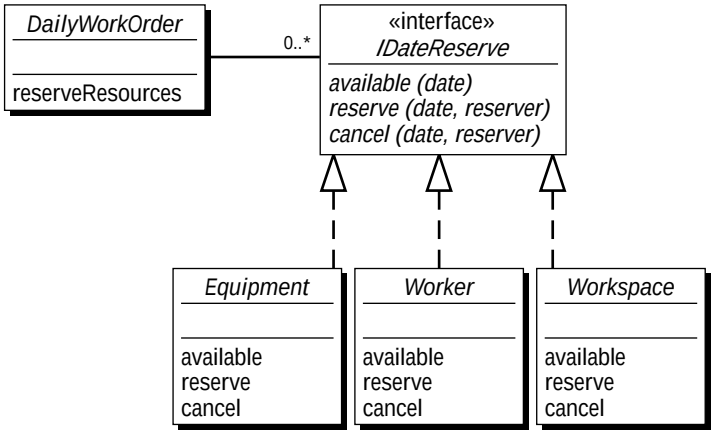

Figure 3-31. Each daily work order object is *still* composed of a collection of IDateReserve objects.

No change to your scenario is needed. The interaction between a daily work order and its IDateReserve objects remains exactly the same.

A daily work order holds a collection of IDateReserve objects. What if it also holds other objects in that collection, objects from classes that don't implement IDateReserve? In this case, a daily work order object can ask an object if it is an instance of IDateReserve. If it is, the daily work order object can then use the interface to interact with that object.*

The point of all this is expandability. By using interfaces, your class diagram and scenarios are organized for change. Instead of being hardwired to a limited number of classes of objects, your design can accommodate objects from present or future classes, just as long as these classes implement the interface(s) that you need.

3.3.4 Factor Out for Future Expansion

Factor Out for Future Expansion Strategy: *Factor out method signatures now, so objects from different classes can be graciously accommodated in the future. Reason for use: to embrace flexibility.*

You can use interfaces as a futurist, too. What if you are wildly successful on your current project? Simply put, the reward for work well done is more work.

So what is next? What other objects might you deal with in the future, objects that could "plug in" more easily, if you could go ahead and establish a suitable interface now?

You can add such interfaces to improve model understanding now and point to change flexibility for the future (hey, this might even get you a pay raise). And you can demonstrate to your customer that your model is ready for expansion—just send more money!

---

*In C++, information about what class an object is in is called run-time type information (RTTI). In Java and Smalltalk, information about what class an object is in is a standard query that can be asked of any object.

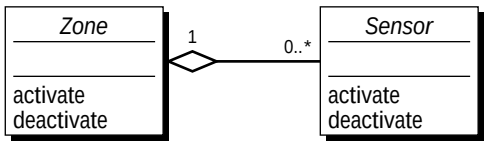

Figure 3-32. A zone and its sensors.

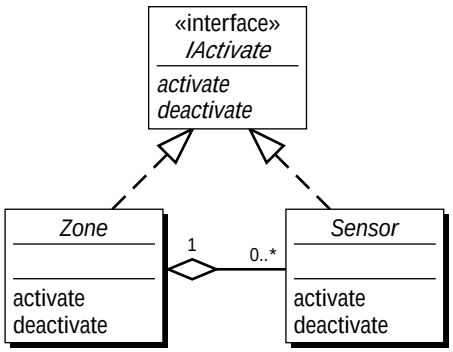

Figure 3-33. Factoring out a common interface.

3.3.4.1 Factoring Out for the Future of Zoe's Zones

Take a look at a zone and its sensors (see Figure 3-32).

Factor out common method signatures into a new interface (see Figure 3-33).

Now adjust the class diagram, so a zone holds a collection of IActivate objects (Figure 3-34).

Go even further: an IActivate object consists of other IActivates (Figure 3-35).

However, this is going a bit too far. An IActivate is an interface; it has no attributes, it has no associations. So showing an association with a constraint on an interface really is going a bit too far. You cannot require an interface to implement an association.

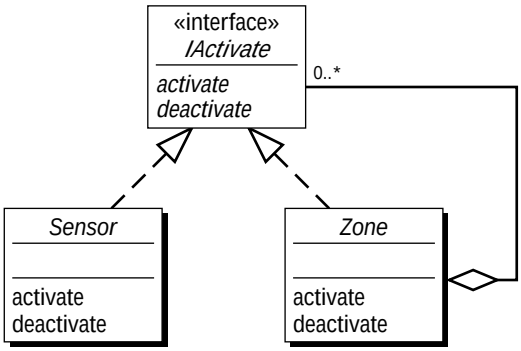

Figure 3-34. A zone and its collection of IActivates.

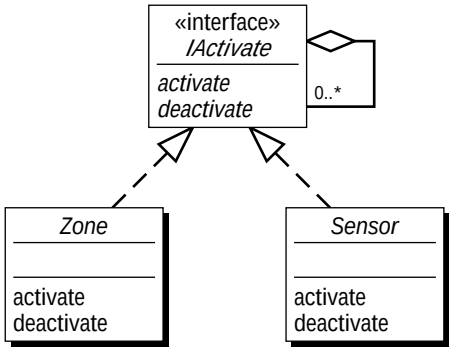

Figure 3-35. An IActivate and its collection of IActivates (too far).

Now, what you *can* do is use method naming conventions that imply attributes and methods:

- get/set method signatures imply attributes

  getStatus and setStatus

- add/remove method signatures imply associations

  addIActivate and removeIActivate.

By using the add/remove naming convention, we end up with a new, improved IActivate interface (Figure 3-36).

Figure 3-37 depicts a corresponding scenario, showing add, activate, and deactivate. Zone is an example of an IActivateGroupImplementer, Sensor is an example of an IActivateImplementer.

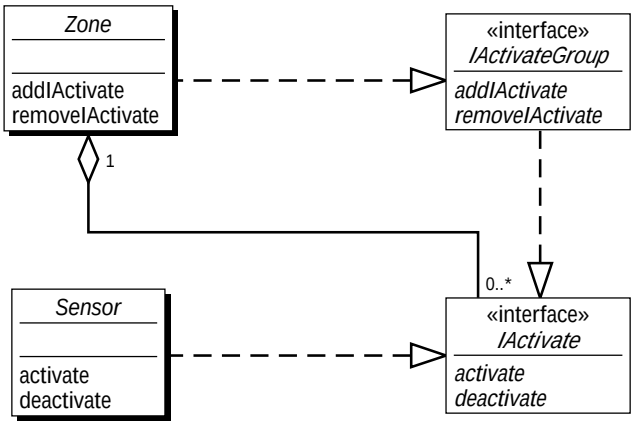

Figure 3-36. An IActivate and adding/removing IActivates.

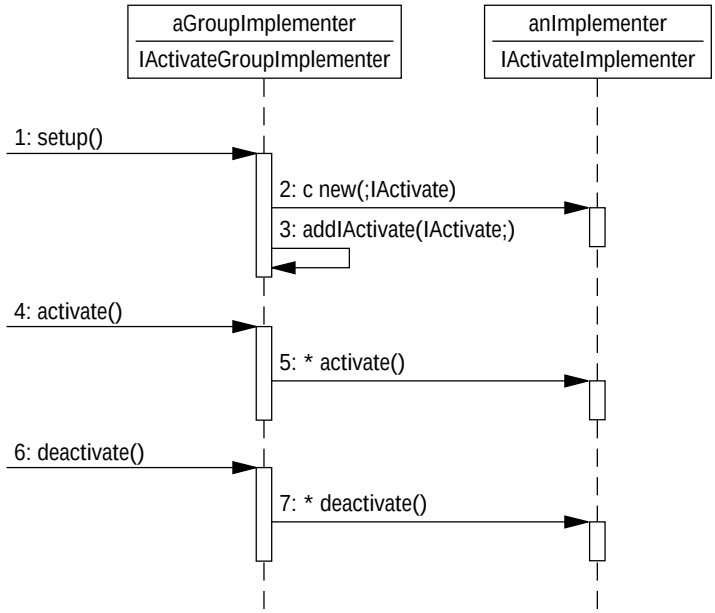

Figure 3-37. An IActivateGroup interacting with its IActivates.

In Java, it looks like this:

```
public interface IActivate {
      void activate();
      void deactivate(); }
```

```
public interface IActivateGroup extends IActivate {
      void addIActivate(IActivate anIActivate);
      void removeIActivate(IActivate anIActivate); }

public class Sensor implements IActivate {
✂
      // methods / public / IActivate implementation
      public void activate() { /* code goes here */ }
      public void deactivate() { /* code goes here */ }
✂
}

public class Zone implements IActivateGroup {
✂
      // attributes / private / associations
      private Vector activates = new Vector();

      // methods / public / IActivateGroup implementation
      public addIActivate(IActivate anIActivate) {
            this.activates.addElement(anIActivate); }
      public removeIActivate(IActivate anIActivate) {
            this.activates.removeElement(anIActivate); }
      public void activate() {
            // iterate through the vector of "IActivates" and ask each one to
            // activate itself
            Enumeration activateList = this.activates.elements();
            while (activateList.hasMoreElements()) {
                  // must cast the element to IActivate
                  IActivate anIActivate = (IActivate)activateList.nextElement();
                  anIActivate.activate(); }
            }
      public void deactivate() {
            // iterate through the vector of "IActivates" and ask each one to
            // deactivate itself
            Enumeration activateList = this.activates.elements();
            while (activateList.hasMoreElements()) {
                  // must cast the element to IActivate
                  IActivate anIActivate = (IActivate)activateList.nextElement();
                  anIActivate.deactivate(); }
            }
✂
}
```

3.3.4.2 Flexibility, Extensibility, and Pluggability for Zoe's Zones

One aspect of flexibility, extensibility, and pluggability is being able to combine objects that you are already working with in new ways—combinations that you might not have anticipated at first.

Now a zone could be a collection of other zones, which could be a collection of sensors. And a sensor could be a collection of other sensors. Nice.

A sensor could be a collection of zones, but this would probably not make much sense. Interfaces allow you to express what kind of behavior must be supported. However, reasonableness applies when it comes to deciding what to plug together!

Another aspect of extensibility is being able to add in new classes of objects: ones that you can anticipate now, and ones that may surprise you in the future.

Look at the interfaces that you are establishing and consider what other classes of objects might implement that same interface at some point in the future.

For zones and sensors, you might look ahead to additional IActivates: switches, motors, conveyor belts, and robot arms (Figure 3-38).

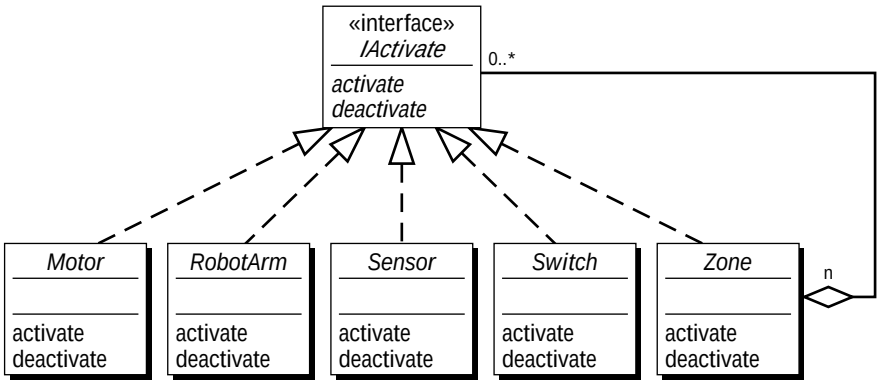

Figure 3-38. Adding in some new IActivates—flexibility, extensibility, pluggability.

When the time comes, you could simply add the new classes, new interface implementers, to the class diagram.

The scenario stays exactly the same as before, up to the point of sending a message to an interface implementer.

3.4 A Short Interlude: Where to Add Interfaces

Okay, so at this point, you might be beginning to wonder about when and where to use an interface. After all, for a fully flexible design, you could include interfaces *everywhere:*

- An interface for every method signature, separating signature from implementation
- An interface for every method signature
- An interface at each end of an association, so each end of the association is not hard-wired to objects in just one class
- An interface for every method call, so you can plug in an alternative implementation of that method any time you choose to.

Very flexible? Yes. Very unwieldy? Yes—and that is the problem. If you set off to build the most flexible software in the universe, you will most definitely run out of time, budget, and resources before you get there. "As flexible as possible" is not a reasonable design objective.

So where does it make sense to add in an interface? Where should you invest in designing-in flexibility? Here is a strategy on this very matter:

Where to Add Interfaces Strategy: *Add interfaces at those points in your design that you anticipate change: (1) Connect with an interface implementer rather than with an object in a specific class; (2) Send a message to an interface implementer rather than to an object in a specific class; and (3) Invoke a plug-in method rather than a method defined within a class.*

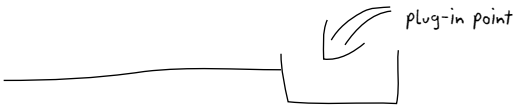

Figure 3-39. Interfaces let you specify the plug-in points, the points of flexibility, within your design.

You've already seen the first two parts of this strategy; the third part is coming up later in this chapter.

There's a picture of this strategy in our minds that needs to somehow get in print. The next four figures visually express this strategy.

When you add an interface, you are adding in a plug-in point, a place where you can plug-in any object from any class that implements the interface (Figure 3-39). Think of an interface as a plug-in point, like a socket on a circuit board.

Rather than connect with objects in a specific class, you can connect with objects in any class that implements an interface (Figure 3-40). One might put it this way: The association connects to a plug-in point.

Rather than send a message to an object in a specific class, you can send a message to an interface implementer (Figure 3-41). Then you can plug in to that plug-in point absolutely any object from any class that implements that interface. You get added flexibility, at the points where you need it (or anticipate that you need it).

Sometimes you might need to vary the implementation of a method. That is to say, you need the ability to unplug one algorithm

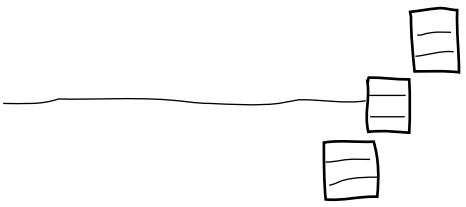

Figure 3-40. Connect with an interface implementer not limited to just objects in a single class.

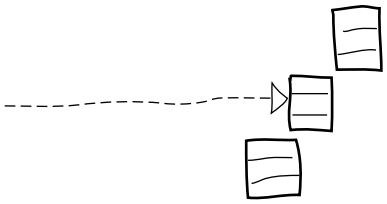

Figure 3-41. Send a message to an interface implementer not limited to just objects in a single class.

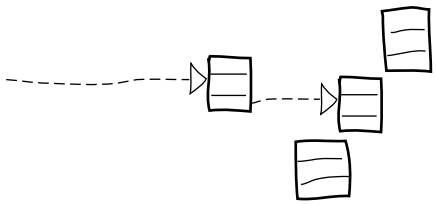

Figure 3-42. Delegate to an interface implementer so you won't be restricted to the methods defined in just one class.

and plug-in another one (Figure 3-42). We are not suggesting general-purpose function blobs or master controllers here (we get a wee bit queasy whenever we see a class name ending with the suffix –er). Yet there are times where a plug-in point for some pluggable behavior brings some algorithmic flexibility that we've found quite helpful.

Where should you add interfaces? Add interfaces to those places where you, as a designer, see the cost-justified need for association flexibility, messaging flexibility, and algorithmic flexibility.

Having completed this short interlude, let's continue with more strategies for interface-centric design.

3.5 Design-in Interfaces

The previous section presented four strategies on how to *factor out* interfaces, that is to say, extract out interfaces from an evolving object model. And factoring-out seems like a good way to begin working with interfaces. Indeed, that's how we got started.

Yet a better approach is to *design in* interfaces in the first place. In other words, look for and establish interfaces all along the way, right as you build your object model. Then, as a crosscheck, you can use the factoring-out strategies to check your interface design and to find additional opportunities for adding flexibility using composition and interfaces.

This two-pass approach—design in and then factor out—mirrors what we've done in practice, the discovery process we've gone through on a variety of projects.

Keep in mind why you design with interfaces. It's all about *substitution,* being able to substitute an object in one class for an object in another class. In fact, each interface within a design embodies two kinds of substitution: plug-in substitution or interaction substitution.

Plug-in substitution means that you can interchangeably put in any object that implements the required interface.

Interaction substitution means that you can send messages to an object in one class as if it were an object in some other class. (Inheritance is one way to do this; yet interfaces let you do this even when the classes you are working with are not directly related via inheritance.) The receiving object might do the work itself, or it might ask another object to do the real work for you (delegation). Either way, the work gets done.

Here is a list of the design-in strategies:

- Design-in interfaces based on common features
- Design-in interfaces based on role doubles
- Design-in interfaces based on behavior across roles
- Design-in interfaces based on collections and members
- Design-in interfaces based on common interactions
- Design-in interfaces based on intra-class roles
- Design-in interfaces based on a need for plug-in algorithms
- Design-in interfaces based on a need for plug-in feature sequences

Let's consider and apply these strategies one by one—and then in combination with one another.

3.5.1 Design-in Interfaces Based on Common Features

As soon as you first write up a features list, you can identify important interfaces for your model. Here's what it takes:

Design-In, from Features to Interfaces Strategy:

1. *Look for a common feature, one you need to provide in different contexts.*
2. *Identify a set of method names that correspond to that feature.*
3. *Add an interface.*
4. *Identify implementers.*

It's time for an example—this time from Larry's Loans. Larry and his fellow loan sharks are very interested in both loan applicants and borrowers (loan-account holders).

Consider this excerpt from Larry's features list:

1. Total outstanding balances for a borrower
2. Total outstanding balances for an applicant (may or may not be a borrower)
3. List accounts and limits for a borrower (a limit is the maximum amount one can borrow, as in a credit-card limit)
4. List accounts and limits for an applicant (may or may not be a borrower)

Jot down some common method names:

```
totalBorrowingBalance
listAccountsAndLimits
```

Now define some interfaces.

If you planned using either name in any context, you could define three interfaces: one for one method signature, one for the other method signature, and one as a combination of the other two interfaces. If you did this with every interface, though, you'd end up with a needless explosion in the number of interfaces in your design. Flexible, yet not simple.

At the other extreme, if these two method names were so closely related that you'd always want them implemented in tandem, then you could say it all in one interface. Simple, not needlessly flexible.

So consider the middle ground between these two extremes. In this case, choose the middle ground: "total" in many contexts, "list and total" in other contexts (Figure 3-43).

In Java syntax, it's:

```
public interface ITotalBorrowingBalance {
    BigDecimal totalBorrowingBalance();
}
public interface IAccount extends ITotalBorrowingBalance {
    Enumeration listAccountsAndLimits();
}
```

Next, build a class diagram around those interfaces. Loan applicant and borrower can implement a common interface. Borrower will do the real work; a loan applicant will delegate its work to its corresponding borrower (Figure 3-44).

Note that Borrower defines the real work to be done; each loan applicant simply sends a message to its corresponding borrower

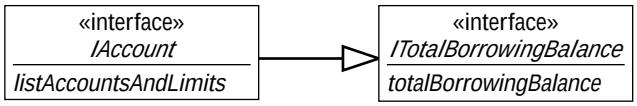

Figure 3-43. Feature-inspired interfaces.

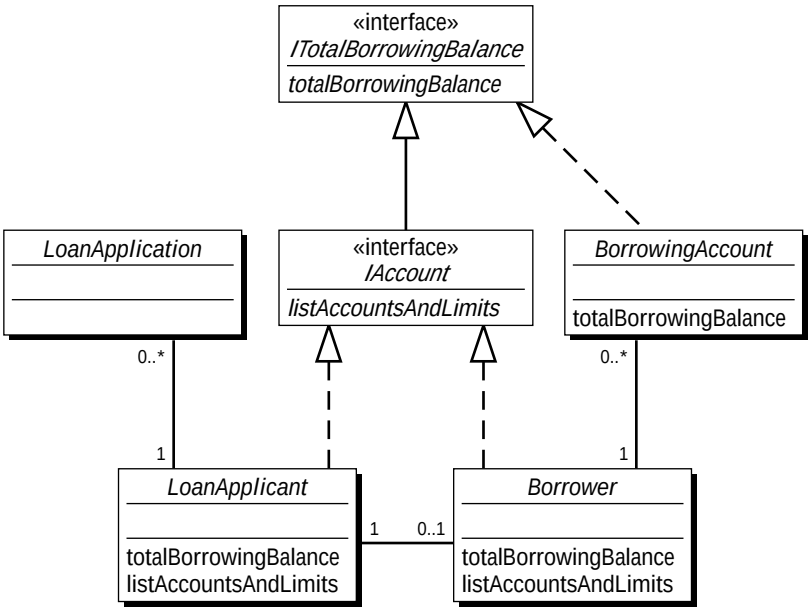

Figure 3-44. From features to an interface to a class diagram.

object, letting it do all the work (delegation at work once again). See Figure 3-45.

Now you can take any loan applicant or any borrower and ask the same question: what is your loan–borrowing balance? So problem-domain containers (like bank) and user-interface objects (like loan-balance lists) can readily work with objects from either class in exactly the same way.

Going from features to interfaces gives you a way to:

- categorize similar functionality.

  Similar in name, with the potential for some behind-the-scenes delegation

- explicitly represent those categorizations.

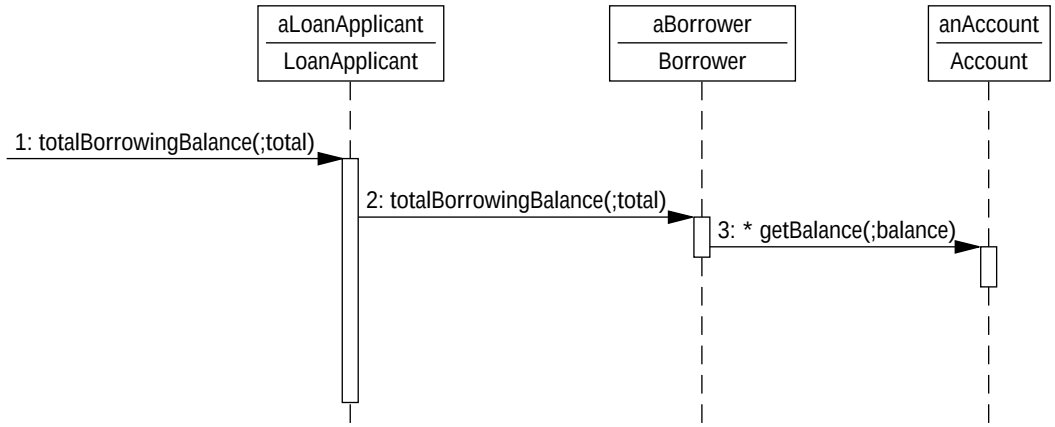

Figure 3-45. A loan applicant delegates to a borrower; a borrower does the real work.

3.5.2 Design-in Interfaces Based on Role Doubles

For each role in your model, you can name an interface in its honor and then let other roles offer that same interface. All of the others will delegate the real work back to the original role-player.

The strategy looks like this:

Design-in, from Role Doubles to Interfaces Strategy:

1. *Take a role and turn its method signatures into a role-inspired interface.*
2. *Let another role (a "role double") offer that same interface by:*
   - *implementing that interface, and*
   - *delegating the real work back to the original role player.*

Larry's Loans is most especially interested in borrowers. For Larry and his cohorts, borrower is the most important role (after all, their business and profits come from their borrowers). For a given borrower, Larry needs to see the total approved lending limits and a listing of the lending limits available to that borrower.

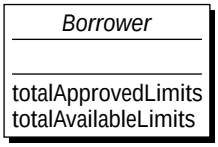

Figure 3-46. Begin with a role.

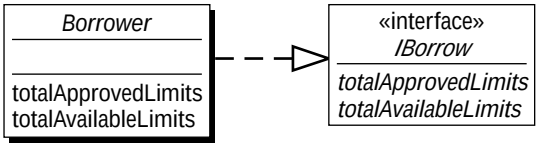

Figure 3-47. An interface corresponding to a role.

First, select a role. The borrower role looks something like Figure 3-46.

Next, add a role-inspired interface (Figure 3-47).

Finally, let another role implement that interface, delegating the real work back to the original role player (Figure 3-48).

Composition and interfaces work hand-in-hand. A loan applicant may play the role of a borrower (composition). Both loan applicant and borrower provide the same interface.

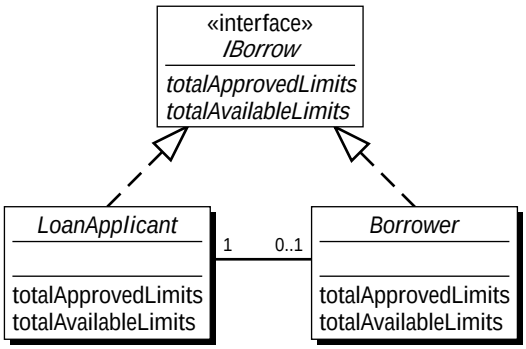

Figure 3-48. Role doubles.

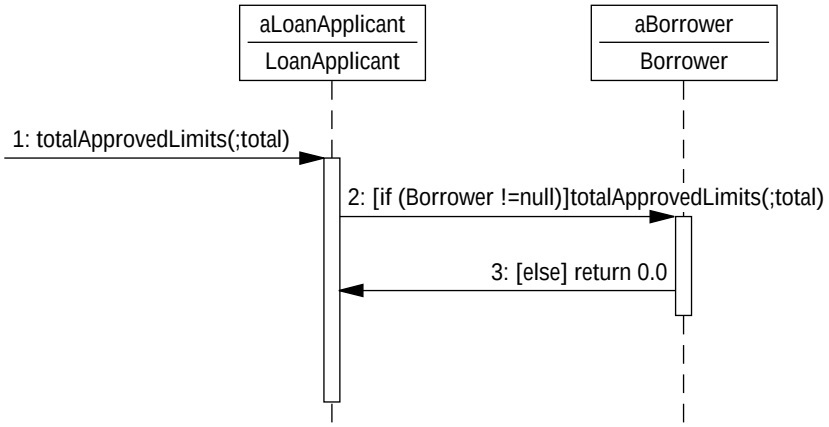

Figure 3-49. Ask a loan-applicant object; it delegates to its borrower object (If it has one).

Actually, the model indicates that a loan applicant *might* play the role of a borrower. If someone asks a loan-applicant object to "total its approved limits" when it has no corresponding borrower object, the loan-applicant object simply returns zero (and avoids messaging a borrower altogether).

Figure 3-49 shows how a loan-applicant delegates to its borrower.

Expressed in Java, IBorrow, Borrower, and LoanApplicant look like this:

```
public interface IBorrow {
    BigDecimal totalApprovedLimits();
    BigDecimal totalAvailableLimits();
}
public Borrower implements IBorrow {
✂
    public BigDecimal totalApprovedLimits() {/*real work*/}
    public BigDecimal totalAvailableLimits() {/*real work*/}
✂
}
```

```
public LoanApplicant implements IBorrow {
✂
    private Borrower borrower; /*add/remove with add/remove methods*/
    public BigDecimal totalApprovedLimits() {
        if (this.borrower != null)
            return this.borrower.totalApprovedLimits(); /*delegate*/
        else return (new BigDecimal (0.0));
    }
    public BigDecimal totalAvailableLimits() {
        if (this.borrower != null)
            return this.borrower.totalAvailableLimits(); /*delegate*/
        else return (new BigDecimal (0.0));
    }
✂
}
```

Interfaces give you a way to treat an object in one class just as if it were an object in some other class. And that's a good thing; it allows you to focus on time-ordered sequences of object interactions that are more important, more revealing, more able to help you improve the model you are working on.

3.5.3 Design-in Interfaces Based on Behavior Across Roles

Another place to design-in interfaces is to look at a party and consider what it does across its collection of party roles.

The same principle applies to a place (for example, an airport) and its roles (day operations, night operations); it also applies to a thing (an aircraft) and its roles (military or civilian).

3.5.3.1 Should a Party Support Role-at-a-Time Interfaces?

Back to the party for now. A party has a collection of party roles. Should the party itself offer single-role interfaces (Figure 3-50)?

If so, then a party could delegate to its loan applicant and the loan applicant could delegate to its borrower (Figure 3-51).

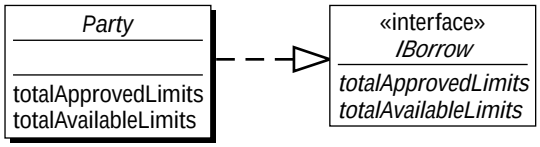

Figure 3-50. Should party offer single-role interfaces?

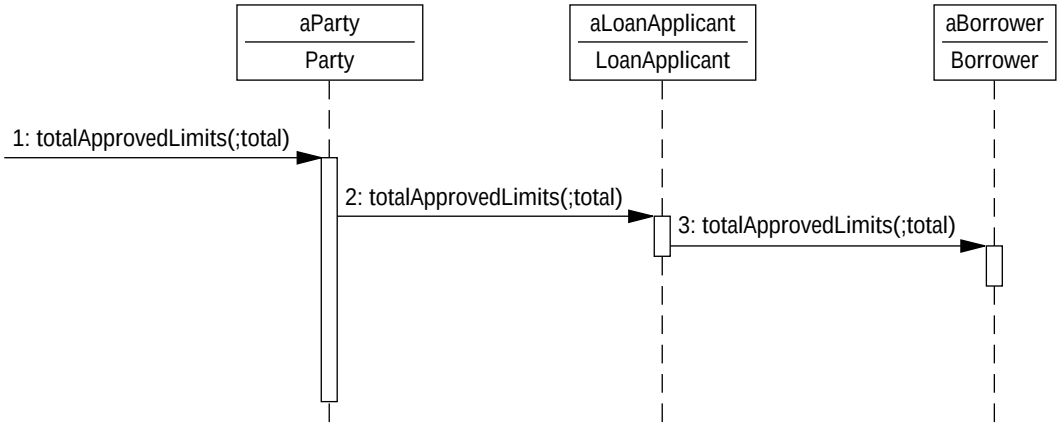

Figure 3-51. Party interacting with specific roles.

Yet why ask a party questions that apply to just a role? You don't need to. Instead, in scenarios, you end up interacting directly with a borrower object or a loan applicant object.

Hence, should you add an IBorrow interface to a party? No way. Why? Adding that interface needlessly complicates a party; you just don't need it.

Again, don't add single-role-specific interfaces to a party (this would make party too complicated, especially enterprise-wide); for single-role-specific interaction, let the party interact with the role's methods itself, rather than through a separately defined interface.

So when might you add a role-related interface to a party? Ever?

3.5.3.2 Should a Party Support "Behavior Across Roles" Interfaces?

Consider this: What does a party object do best? Simply put, it enforces behavior across its many roles.

Let's explore this a bit further.

A party might have a number of roles. Here, a party can have two roles; one of those roles might have a subsequent role (see Figure 3-52).

In general, a party might have many roles (loan applicant, borrower, shareholder, lender, manager, executive, and so on). To support a party's need to interact across its collection of roles, each role could implement a common role interface. In this case, that common role interface is IAuthorize (see Figure 3-53).

Note the designed-in flexibility: it's easy to drop in a new role, as long as that role implements the common interface(s) for party to iterate over. Party does what party does best: enforce business rules that apply across its roles.

Should a party support "behavior across roles" interfaces? Yes!

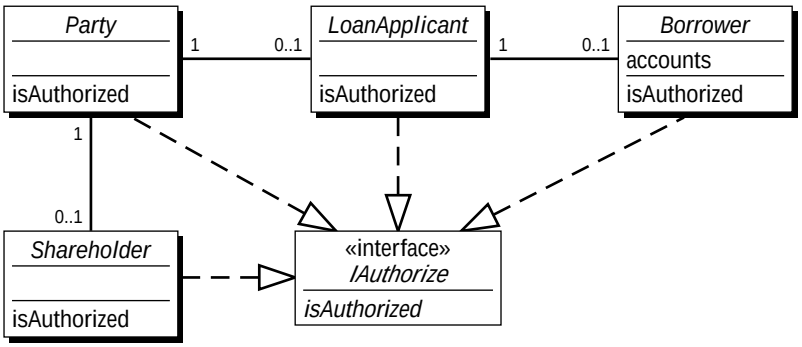

Figure 3-52. An interface for behavior across roles.

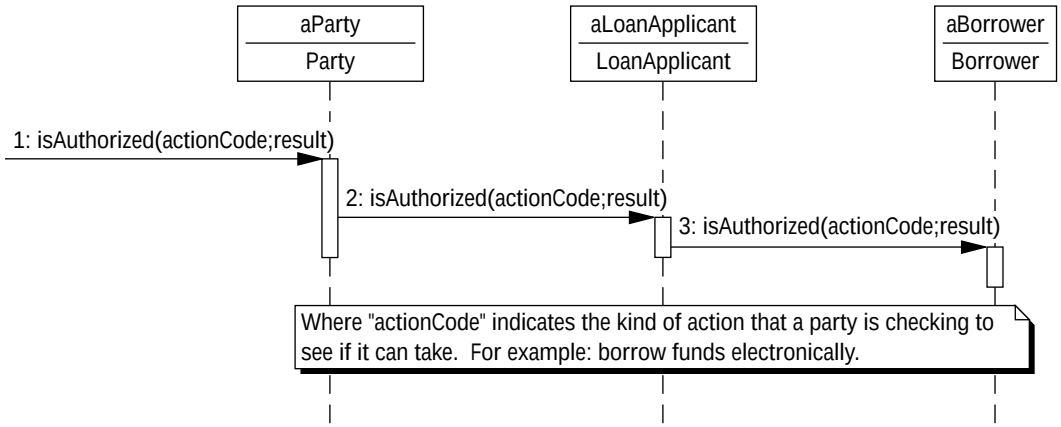

Figure 3-53. A party interacts with its role; that role interacts with a subsequent role.

3.5.4 Design-in Interfaces Based on Collections and Members

With each class you add to your model, you can add depth by considering what interfaces it needs for it to do its job as a collection itself, and then as a member within some other collection. Here's how:

Design-in, from Collections and Members to Interfaces Strategy:

1. *Does your object hold a collection of other objects? If so:*
   a. *Consider the potential "across the collection" method signatures.*
   b. *If other collections might offer the same set of method signatures, then design in that common interface.*
2. *Is your object a member within a collection? If so:*

   If that object needs to provide an interface similar to the collections it is in, then design in that common interface.
3. *Identify implementers.*

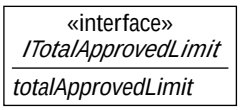

Figure 3-54. A collection-inspired interface (i).

To begin with, does your object hold a collection of other objects? Collections are just about everywhere:

- If it's a party (person or organization), it has a collection of roles.
- If it's a role, it has a collection of moments or intervals.
- If it's a place or container, it has a collection of moments or intervals.
- If it's a moment or interval, it might have a collection of subsequent moments or intervals.
- If it's an item description, it might have a collection of corresponding specific items (actual things to keep track of).
- If it's an item description, it might have a collection of even more detailed item descriptions.

For an example, consider Larry's Loans.

An application is a collection of approvals; each approval sets an approved limit (one that might be more, less, or the same as the amount originally applied for).

As a collection, we could ask an application to total its approved limits. That's a generally useful interface (See Figure 3-54).

As a member in a collection, we could ask an object to compare its approved amount vs. the applied-for amount. An approval object would have to interact with its corresponding application object to work out the answer. That's another interesting addition to the interface we are working on (See Figure 3-55).

The corresponding class diagram looks like Figure 3-56.

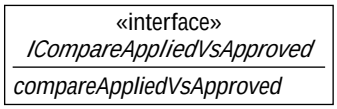

Figure 3-55. A collection-inspired interface (ii).

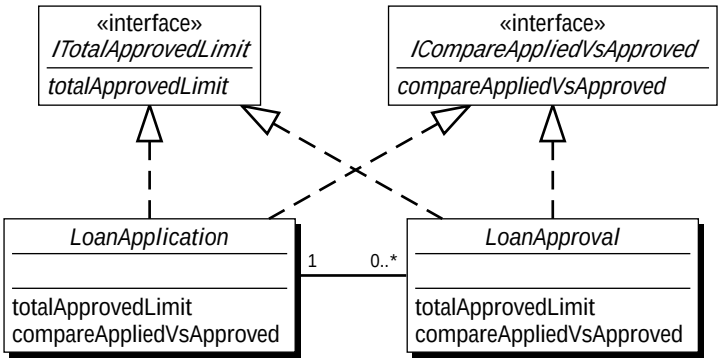

Figure 3-56. From a collection itself and a collection member: first to an interface, and then to a class diagram.

And the corresponding scenario looks like Figure 3-57.

The collection-inspired interface is ITotalApprovedLimit. Ask a loan application for its total approved limits and it interacts with each of its corresponding approval objects, returning the total approved limit. Ask a loan approval for its total approved limits and it simply returns its own approved amount.

In fact, you could plug in an object in any class that implements this interface in either column of the scenario.

Note this added twist. If you plug something into the left position, though, the interactions that follow might be different. It might, for example, implement its own "total approved limit" method and return the result to the sender. Or it might interact with some number of other objects, not shown in the scenario.

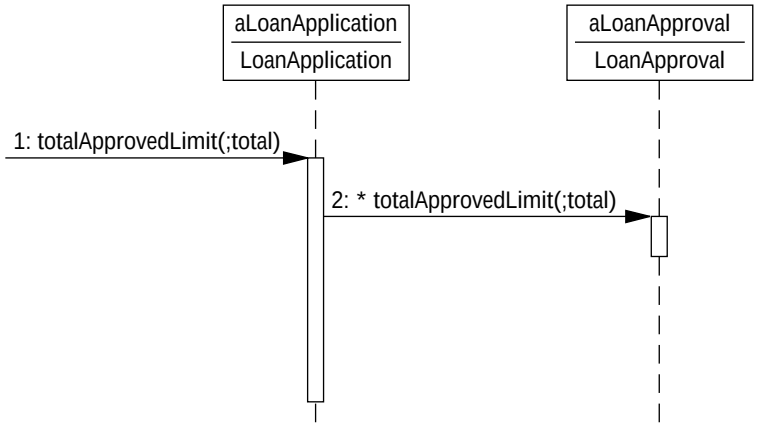

Figure 3-57. A loan application interacting with its loan approval(s).

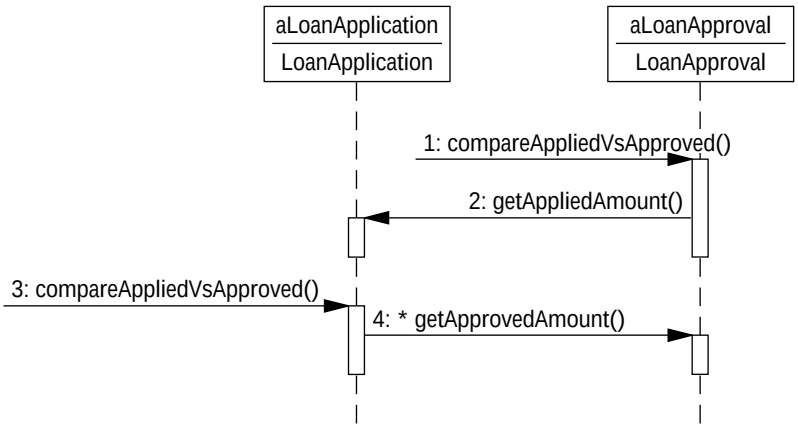

Figure 3-58. A loan approval interacting with its loan application.

On the other hand, take a look at the member-in-a-collection-inspired interface, ICompareAppliedVsApproved (Figure 3-58).

Note the two paths. Ask a loan approval object to compare applied vs. approved; it interacts with its one loan application, makes a comparison, and returns the result to you. Or ask a loan application object to compare applied vs. approved; it interacts with each of its loan approvals, totals the approved amounts, makes a comparison, and returns the result to you.

Examining collections and members gives you a way to establish common interfaces for both a collection and its members (first from a collection's perspective and then from a member's perspective). With the same interface, you end up with broader answers from a collection and more specific answers from a member. This brings more meaningful content to the model sooner, as well as provides a useful abstraction for thinking about and working with that added content.

3.5.5 Design-in Interfaces Based on Common Interactions

When working out dynamics with scenarios, you might come across similar interactions going on between one column and others (Figure 3-59).

When you see similar interactions, it's a good time to design in an interface. Why? To raise the abstraction level within your model. To explicitly capture the interaction commonality. To make the scenario "pluggable" (so you can unplug one interface implementer and plug in another one).

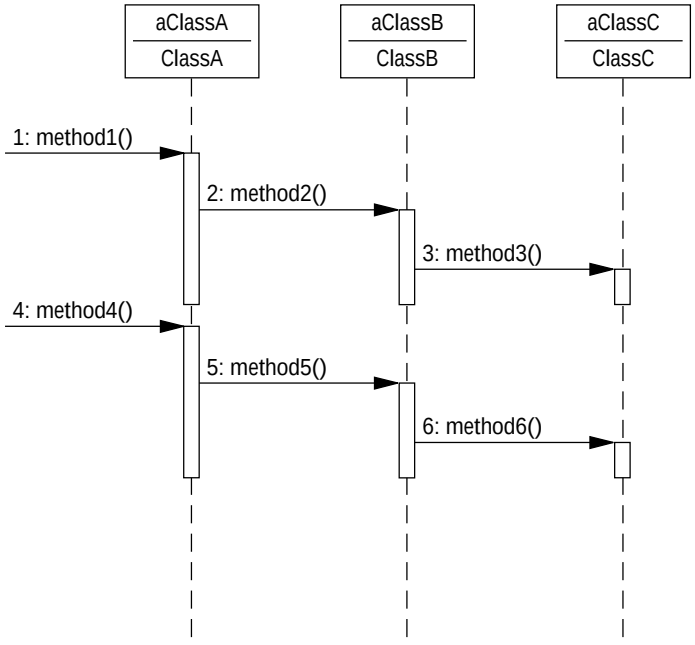

Figure 3-59. An example of similar interactions.

Here's the strategy:

Design-in, from Scenarios to Interfaces Strategy:

1. *Look for similar interactions.*
2. *Add an interface-implementer column.*

   Use this naming convention:

   I<what it does> Implementer.
3. *Add an interface: I<what it does>.*
4. *Identify implementers.*

Back to Larry's Loans we go, this time beginning with a scenario for assessing profit and risk, the two key aspects of any financial deal. For a loan applicant, we can assess risk. For a borrower, someone who has borrowed money from Larry's Loans, we can assess both profit and risk. The scenario looks like Figure 3-60.

Look at the similar interactions. The strongest similarity is in the assess-risk scenario. A loan applicant asks each of its applications to assess the risk it poses; then a loan applicant asks its corresponding borrower object to assess its risk.

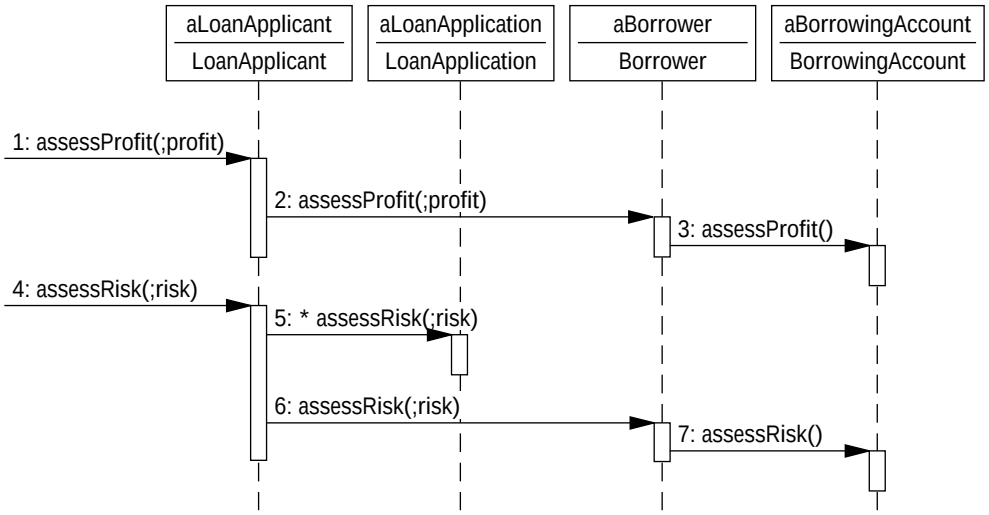

Figure 3-60. Scenarios with some similar interactions.

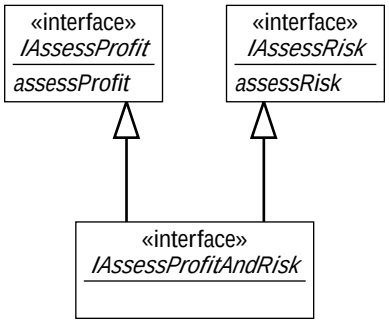

Figure 3-61. A scenario-inspired interface.

Another similarity in interaction occurs when a loan applicant asks a borrower to assess its risk or to assess its profit.

So take the opportunity to introduce new interfaces (Figure 3-61).

But wait. If assessing profit *always* has assessing risk as a companion (and it always does, in real life), then we really don't need an IAssess interface after all. Here's why:

Interface Granularity Strategy: *If a method signature can only exist with others, then add it directly to an interface definition with those others (no need for a separate, one-signature interface).*

You can use that strategy to keep your number of interfaces down to a more modest level, as in Figure 3-62.

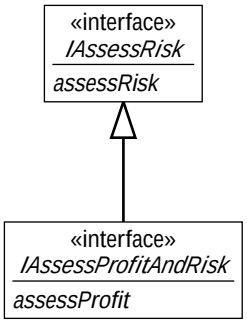

Figure 3-62. Interfaces, upon applying the "interface granularity" strategy.

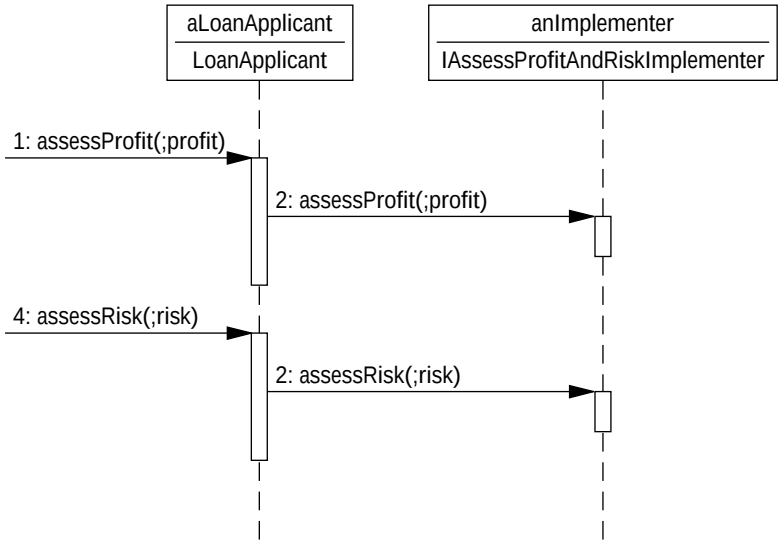

Figure 3-63. Scenarios with an interface implementer.

Okay, with the interfaces established, simplify the scenario itself. How? Use a single "interface implementer" column to represent the implementers of that interface: application, borrower, and borrowing account (Figure 3-63).

Taking these new interfaces into a class diagram, you get Figure 3-64.

So what does this mean? It means that:

- You can ask a borrower object to assess its risk. It does so by interacting with its borrowing account objects.
- You can ask an application object to assess its risk. It does so by interacting with whatever objects it knows (for example, credit reports).
- You can ask a loan-applicant object to assess its risk. It does so by interacting with its application objects and its borrower object.

Working from scenarios to interfaces gives you a way to classify objects with similar functionality and interactions, regardless of what classes those objects might be in, now or in the future.

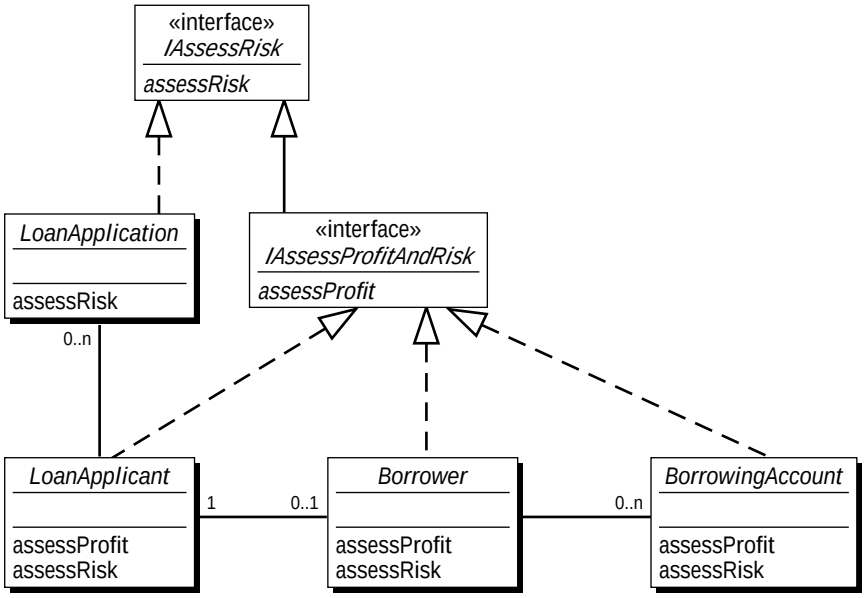

Figure 3-64. From scenarios to interfaces to a class diagram.

3.5.6 Design-in Interfaces Based on Intra-class Roles

Objects rarely do anything interesting by themselves. Most often, an object interacts with objects in other classes to get something done.

Yet sometimes objects within a class interact with other objects in that same class. How can you recognize when this is the case? Take the time to consider each class and the roles the objects in that class might play, interacting with other objects in that same class.

Design-in, from Intra-Class Roles to Interfaces Strategy:

1. *Identify roles that objects within a class can play.*
2. *Establish an interface for each of those roles.*
3. *Identify implementers.*

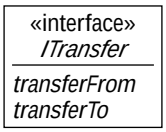

Figure 3-65. An interface inspired by examining intra-class roles.

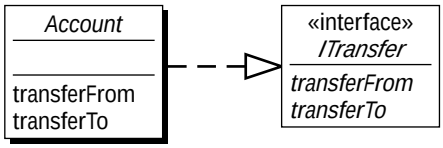

Figure 3-66. From intra-class roles to interfaces to class diagram.

At Larry's Loans, there are accounts. In an account transfer, one account acts as an origin ("transfer from") and another account acts as a destination ("transfer to").

Examining intra-class roles leads us to the interface in Figure 3-65.

So the Account class gets some added depth to it, as the implementer of the interface (Figure 3-66).

We end up with a scenario with interactions between two objects in the same class: a "transfer from" object and a "transfer to" object (Figure 3-67).

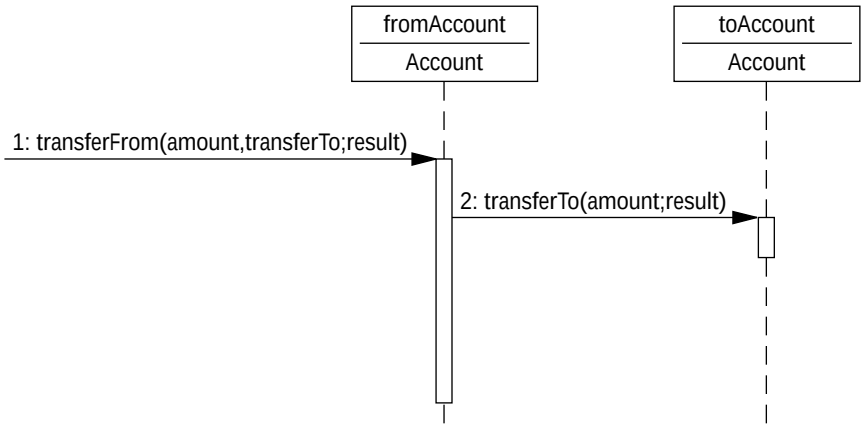

Figure 3-67. Objects in the same class interacting with each other.

This strategy uses interfaces in yet another way: this time, to abstract up major interaction categories when objects in the same class interact with each other in some collaborative way—and to explicitly model those interaction categories in a class diagram.

3.5.7 Design-in Interfaces Based on a Need for Plug-in Algorithms

When it comes to building better object models, once the overall model shape is in place, methods are where the action is, where strategic advantage comes into play.

An object model without methods is not very exciting. All you end up with is a well-structured data-holding system. An object model that is feature-rich and correspondingly method-rich represents strategic advantage, genuine business advantage in the global marketplace.

In a class, you define a method that applies to each object in that class. It's something that each object can do itself. Yes, there is some potential for variation in what that method does (based upon the state of that object and interactions with related objects), yet the algorithm for each variation is set.

What happens when you require far more algorithmic diversity for objects within a single class? When you find you need algorithmic flexibility, use an interface and some plug-in algorithms—a specific usage context for what is sometimes referred to as a strategy pattern [Gamma 95]. Here is the strategy:

When to Use Plug-in Algorithms and Interfaces Strategy: *Use a plug-in algorithm and interface when you find this combination of problems:*

- *An algorithm you want to use can vary from object to object within a single class*
- *An algorithm is complex enough that you cannot drive its variation using attribute values alone.*
- *An algorithm is different for different categories of objects within a class—and even within those categories (hence, adding a category-description class won't resolve this problem).*

- *An algorithm you want to use will be different over time and you don't know at this point what all those differences will be.*

When this happens, you can design-in an interface so you can plug-in the functionality you need, on an object-by-object basis.

Here is the strategy:

Design-in, from Plug-in Algorithms to Interfaces Strategy:

1. *Look for useful functionality you'd like to "plug in."*
2. *Add a plug-in point, using an interface.*
3. *Identify implementers.*

At Larry's Loans, we need a way to validate the terms of a borrowing account. Terms include restrictions on interest rate, compounding method, and prepayment penalties. And yet validating:

- is different for different kinds of terms.
- is different even for the same kind of term (so adding a "term category description" class won't help us here).
- is going to change over time, in ways we don't know in advance.

So begin tackling these problems by adding an interface (see Figure 3-68).

Now add the plug-in point in a class diagram (see Figure 3-69).

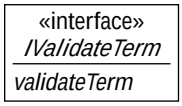

Figure 3-68. A plug-in inspired interface.

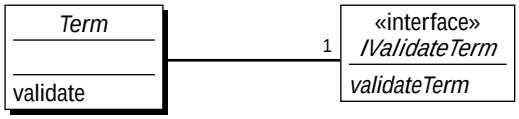

Figure 3-69. Adding a plug-in point.

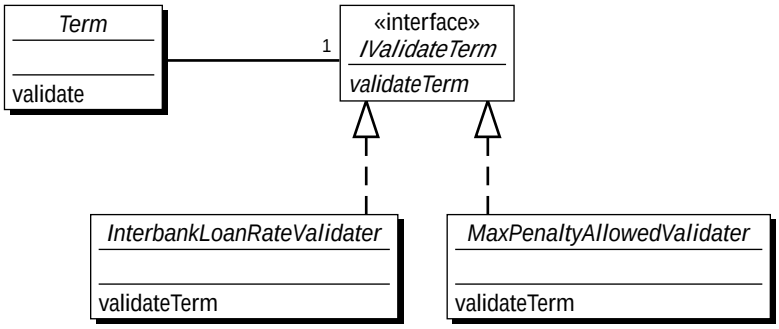

Figure 3-70. Adding something to plug-in at the plug-in point.

Next, take it a step further, adding something to actually plug into that plug-in point (see Figure 3-70).

An IValidateTerm object is an algorithm you can plug into a term object. For one term you might want to plug in an algorithm that compares the interest rate being charged with the interbank loan rate. For another term, you might want to plug in an algorithm that compares the penalty stated in that term with the maximum allowed by law within the applicable geopolitical region. And so on.

So you create a term object; create the appropriate validater; plug the validater into the term object; and you're ready to go.

The "validate a term" scenario looks like Figure 3-71.

Note that the term itself is passed along to the validater, so the implementer can in turn send messages to the term object, to get whatever the implementer needs to do its job (this kind of interaction is sometimes referred to as a "callback").

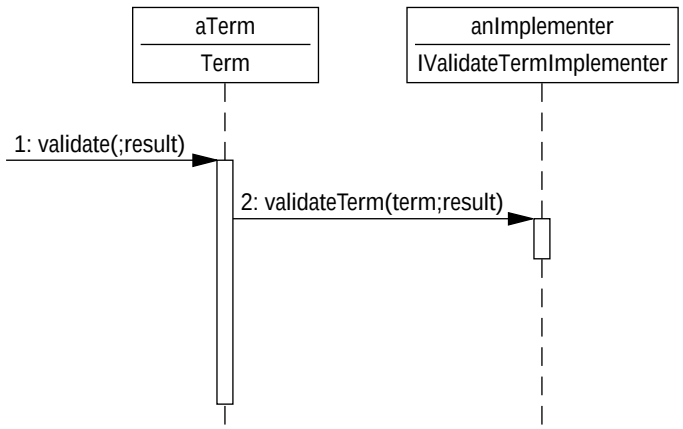

Figure 3-71. Delegating to a plug-in point.

Here's how to express it in Java:

```
public interface IValidateTerm {
      int validateTerm(Term term);
}
public class Term {
✂
      private IValidateTerm validater;
      public void addValidater (IValidateTerm validater) {
                  this.validater = validater; }
      public int validate() {
            return this.validater.validateTerm(this); }
✂
}
```

So what might happen when you plug in a specific term validater? It depends on the behavior of what you plug in, of course. Figure 3-72 is an example.

Designing-in interfaces at plug-in points gives you a way to design for current or anticipated algorithmic diversity for objects within a class, adding algorithmic flexibility to your design.

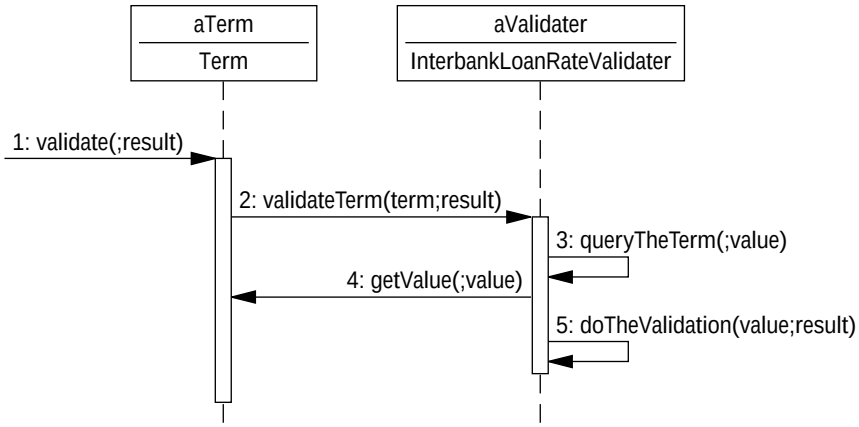

Figure 3-72. What happens once you ask a plug-in method to do its thing depends on that plug-in method.

3.5.8 Design-in Interfaces Based on a Need for Plug-in Feature Sequences

But first, it's time for another short interlude about designing with designs (plural). Plugging in feature sequences is another good reason for systematically designing interfaces.

3.5.8.1 Short Interlude: Effective Design Requires Designs

Design is all about making comparisons, tradeoffs, and judgments.

It's good engineering practice to develop and consider *several* design alternatives, then assess and select the one you'd like to use for the system under consideration. This occurs (or certainly should occur) again and again and again, throughout the design process.

Strategies and patterns are essential ingredients for assessing and selecting from design alternatives.

Working out dynamics with scenarios is one of the best ways to compare design alternatives.

Sometimes it's important to gather some empirical evidence: Design and build a small part of the system to work out what approach to take on your project.

Please note that this is not an exhortation to look for the *best* design, the one *true* design, or something close (it doesn't exist!). Please also note that you're not reading anything like, "Follow these steps and the very first design you come up with will be the best one." On the other hand, nor are you seeing an endorsement for design by committee—far from it. Good things get accomplished by small groups and even in a small group someone needs to play "chief architect" and make important decisions along the way.

By considering three possibilities, you can look for the best of that set. And nearly always the result will be better than grabbing the first design that pops into your head and running with it.

In fact, good designers often end up with a simpler overall result, with a sense of "Gee, that was obvious" or "I'm glad we got there, but it would have been nice to get there sooner." Such is the nature of good design. Yes, strategies and patterns get you there sooner. Yet this "collapsing of complexity" and "aha!" insight are an integral part of designing for even the very best software designers.

The best design work we do occurs with teams of about 10 designers. We routinely design in three parallel subteams. After 30 minutes of design, each sub-team presents work in progress. We learn from each other. And we come up with a far better design than any one subteam could produce on its own. Sometimes we pick one design out of the three. More often, we merge good ideas from all three. It takes a seasoned mentor to guide the process. A handbook of strategies and patterns is also an essential ingredient.

Now let's try designing with designs.

3.5.8.2 Feature Sequences

Many applications require feature sequences, also known as business events, business activities, or operational procedures. How should you model such sequences?

Consider the following example, adding the feature "make a sale" to Charlie's Charters.

> Feature: Make a sale.
>
> Initiate a new sale.
>
> Accept item and quantity.
>
> Accept method of payment.
>
> Accept amount tendered
>
> (make change, record the sale).

Design #1. We could introduce a "feature sequence (FS)" class, as in Figure 3-73.

The first design? Let a make-a-sale object do all the work. It grabs the data values it needs and does everything else. Low encapsulation, low distribution of responsibility, low resiliency to change.

Design #2. Let a make-a-sale object coordinate yet do absolutely no detailed work. It holds collections. It steps through the process of making a sale. It delegates everything it possibly can to the objects it interacts with. It takes care of a transaction's start/commit/abort. It delivers behavior across the collection of objects it knows and interacts with. Moderate encapsulation, moderate distribution of responsibility, and moderate resiliency to change. A reasonable design. Let's try for something even better.

Design #3. Let a problem-domain object do whatever it takes to coordinate making a sale. In other words, let a sale object make a sale (Figure 3-74).

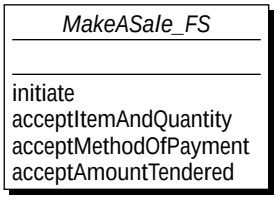

Figure 3-73. Try out a "feature sequence"?

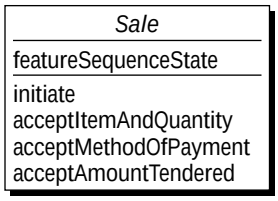

Figure 3-74. Let a sale coordinate the making of a sale?

High encapsulation, high distribution of responsibility across problem-domain objects, moderate resiliency to change.

Take a closer look at Design #3. How does it work? Consider the scenario in Figure 3-75.

Observation: In this design, human-interaction (HI) objects focus on doing one thing and one thing well: interaction!

Another observation: Rather than thinking of a human-interaction object as a sequence manager, think of it as an **interactive view** of the current states of some problem-domain objects. For example: a store object has a current state; it notifies its listeners (such as a store window object) whenever it changes state; a store window receives such notifications and updates itself accordingly.

Design #4. One more time! Let's take design #3 and improve upon it. The problem with design #3 is that it does not allow you to plug in variations on what it means to make a sale. Plug in? Yes, you can use an interface to define a useful plug-in point—a point in your design where you want to design in added flexibility. (This is another usage context for what is sometimes referred to as a strategy pattern.)

Figure 3-76 shows the Sale class with its corresponding plug-in point for plugging in whatever sequencing is needed to make a sale.

Figure 3-77 shows two implementers of the IMakeASale interface (when you create a sale object, you also "plug in" an object from a class that implements that interface).

Now every time someone asks a sale object to make a sale, that sale object delegates to whatever IMakeASale object it is holding

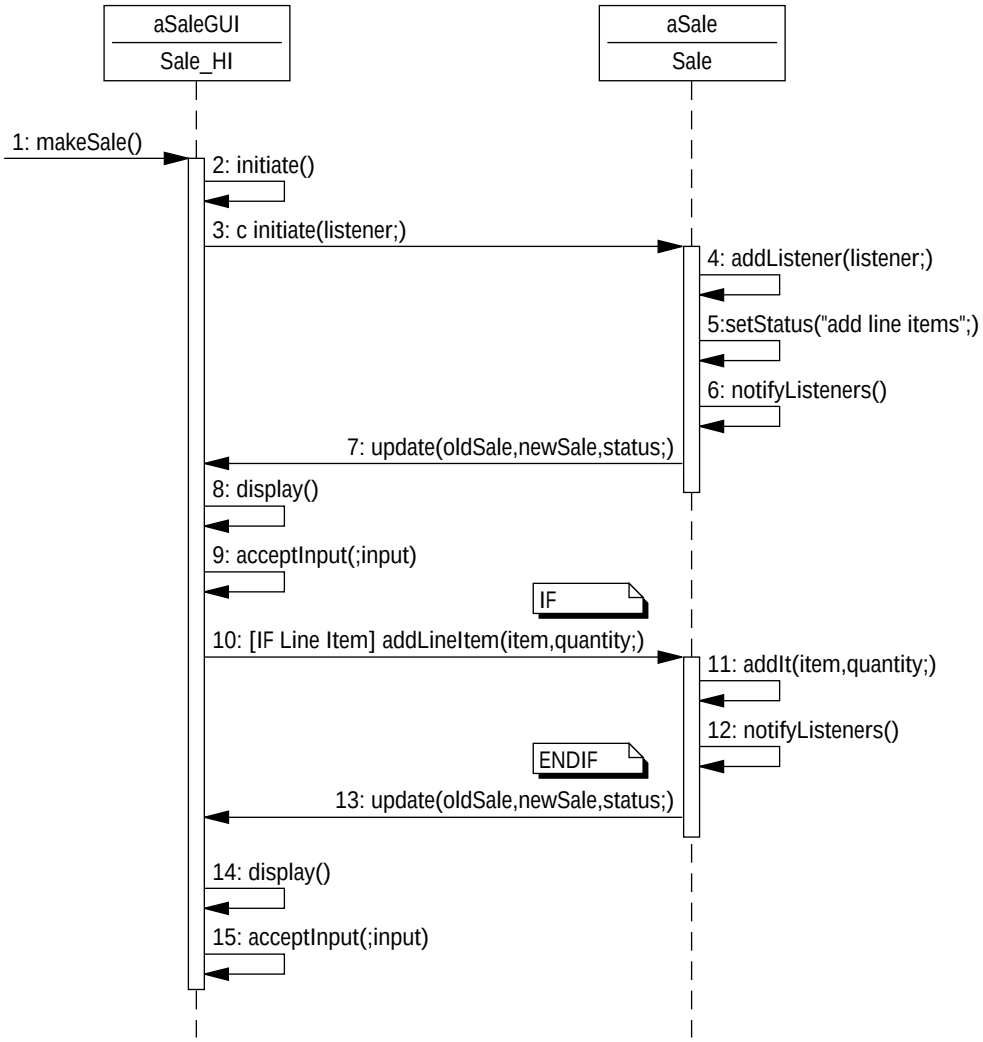

Figure 3-75. Establish a new sale (initial steps).

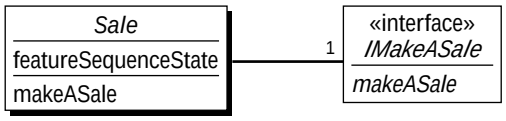

Figure 3-76. A sale with a process plug-in point.

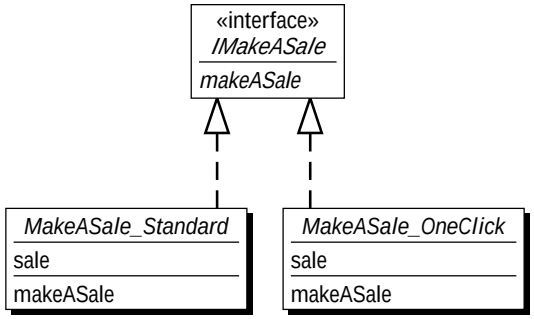

Figure 3-77. Plug in the process of your choice.

(meaning, whatever object from any class that implements IMakeASale).

Figure 3-78 is an example scenario view.

Note that the sale itself is passed along to the sequencer, so the sequencer can in turn send messages to the sale object, to get whatever the sequencer needs to do its job (another example of a "callback").

For the utmost in flexibility, Design #4 is the winner. It offers high encapsulation, high distribution of responsibility across problem-domain objects, high resiliency to change.

However, it's the winning design of the four designs only if you need such flexibility. Remember it's important to design in flexibility where you need it, not every single place you possibly can!

Wouldn't this fourth design be the obvious first choice for an experienced designer? In practice, we've found the answer to be no. The most flexible design is not always needed. Nor is it intuitively obvious to experienced designers (a team of designers worked on this sequence of designs and only one came up with design #4). Perhaps it's just that as designers we digest so much content that it takes us a while to see what's really needed within a design (as Jerry Weinberg wrote in *Secrets of Consulting,* know-when pays a lot more than know-how). A sense of "collapsing complexity" and "aha!" insight is still very much a part of everyday practice for even very experienced software designers, as it should be.

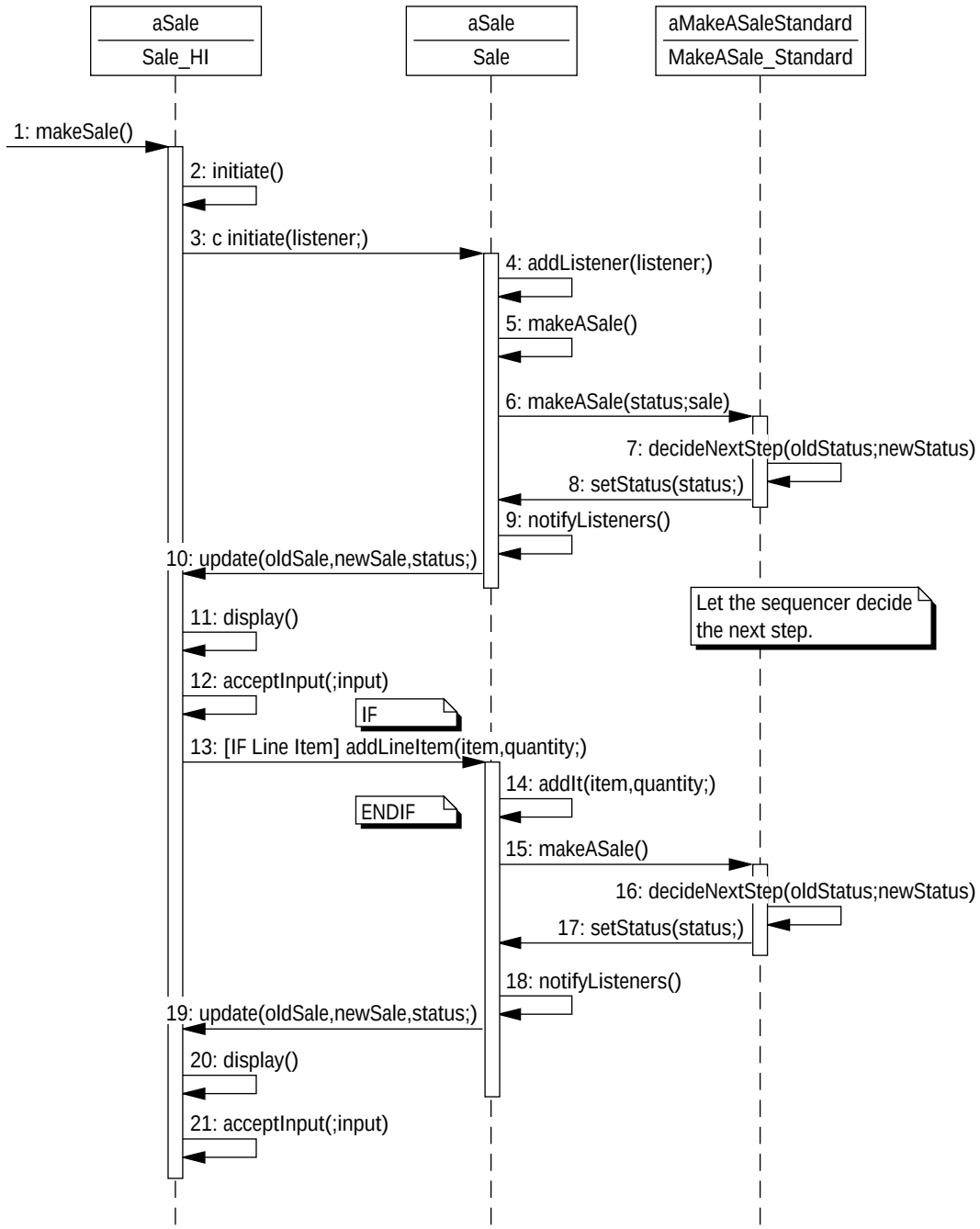

Figure 3-78. Make a sale with a sequencer.

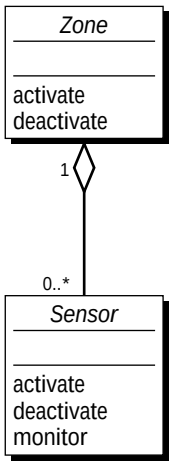

Figure 3-79. A simple yet inflexible diagram: zone and sensor.

3.6 Design with Interfaces: Applying Multiple Strategies

Now apply a number of strategies together, in concert. First apply them for Zoe's Zones. Then apply them for Charlie's Charters.

3.6.1 Designing-in Flexibility Is a Very Good Thing

Begin with a zone and sensor class diagram as in Figure 3-79.

This class diagram is a simple yet inflexible diagram. Why inflexible? Each association is hardwired to objects in a specific class. And each message-send will be hardwired to objects in a specific class.

Design in some flexibility. Review the list of interface strategies:

- Factor-out

  Repeaters

  Proxies

  Analogous apps

  Future expansion

- Design-in

  Common features

  Role doubles
  Behavior across roles

  Collections and members
  Common interactions
  Intra-class roles

  Plug-in algorithms
  Plug-in feature sequences.

Try out the last two in the list.

The monitoring algorithm might vary so greatly from sensor to sensor that we need a plug-in point. Apply the plug-in algorithms strategy.

The overall monitoring sequence for a zone might vary over time. Apply the plug-in feature sequences strategy.

The result looks like Figure 3-80.

Now a zone holds a collection of IActivates—sub-zones, sensors, or anything else that we might want to plug in over time (motors, robot arms, and the like).

And each sensor object gets its own monitoring algorithm. You can plug in the standard default algorithm or develop your own and plug it into the sensor objects you are working with.

Flexibility!

3.6.2 Yet There Usually Is a Design Tradeoff: Simplicity vs. Flexibility

Consider the class diagram in Figure 3-81.

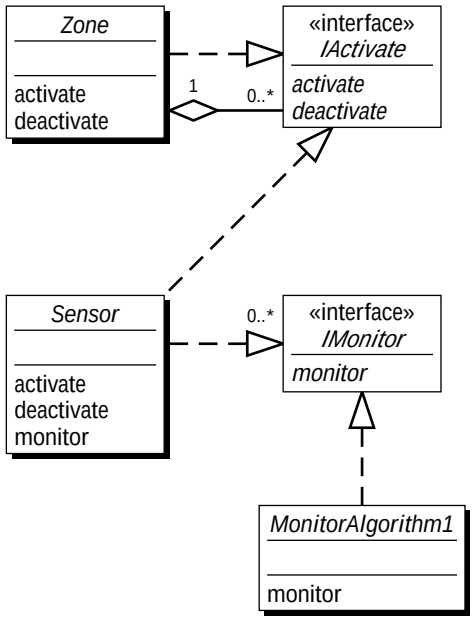

Figure 3-80. A more flexible yet more complex model; flexibility comes with a price.

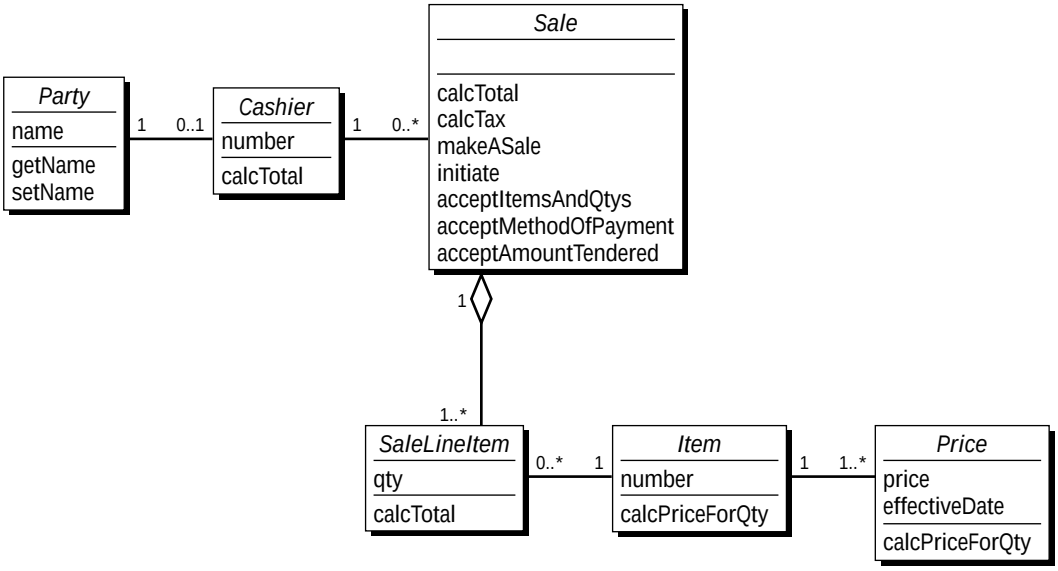

Figure 3-81. A simple class diagram for the retail part of Charlie's business.

This class diagram is simple yet inflexible. Why inflexible? Every association is hard-wired to a specific class. Every message-send will be hard-wired to objects in a specific class. It's simple; it's easy to implement (as long as requirements don't change too drastically along the way).

Now consider a variation on the theme (Figure 3-82).

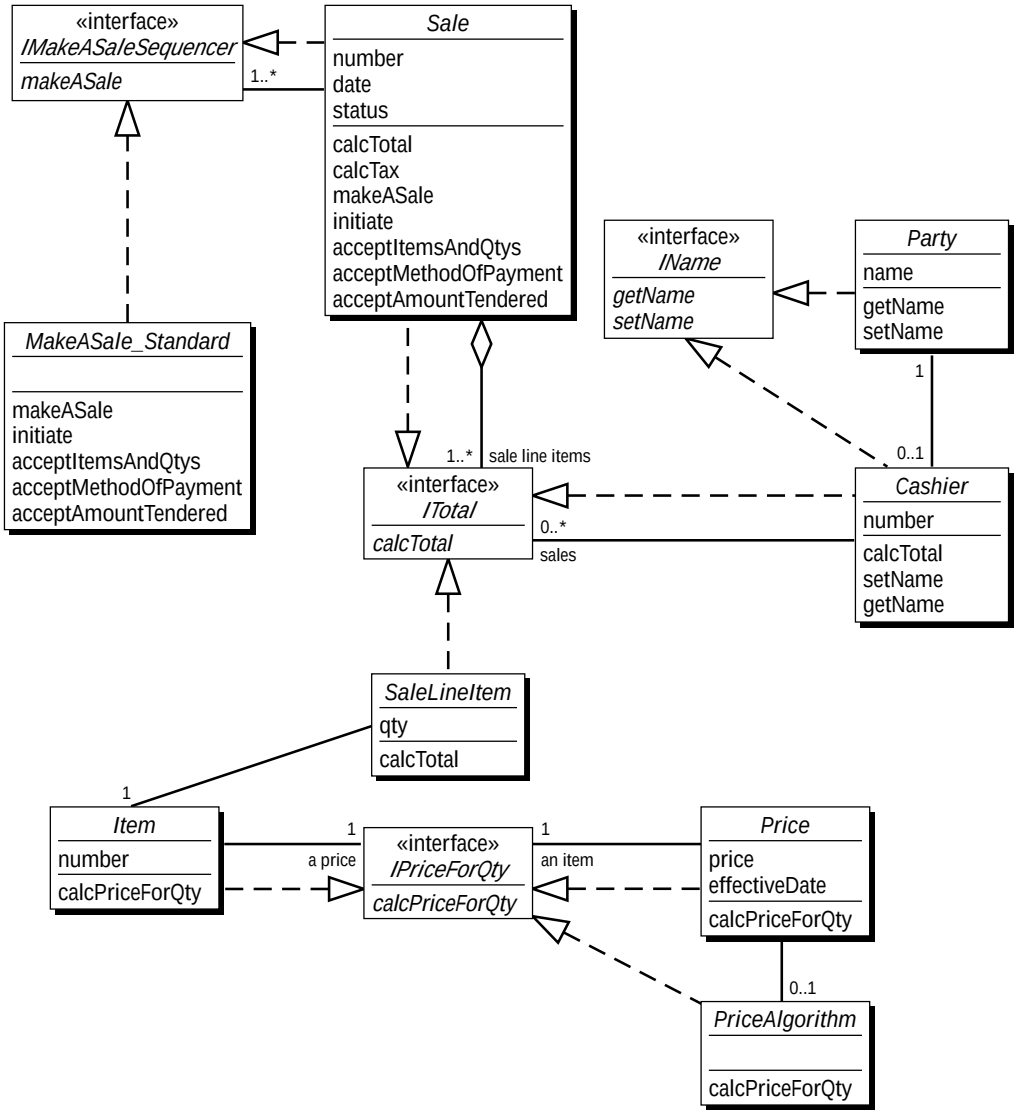

Figure 3-82. A more complicated yet more flexible class diagram for Charlie's business.

This class diagram is more complicated yet more flexible. The diagram includes plug-in points just at those points where the designer anticipates the need to design in some flexibility. A better design? Yes, *if* such flexibility fits within the capability, schedule, and cost constraints for the project at hand.

Add in plug-in points where you need them, where they pay for themselves by adding flexibility in exchange for increased model complexity.

3.7 Naming Interfaces Revisited

With more experience with interfaces in practice, we've thought more and more about what makes a good interface name. We've discovered some more things about useful interface names along the way.

You see, in practice, interface names give you a way to classify:

- The *kinds of classes* whose objects you want to plug into that plug-in point:

  IParty

  getAddress, setAddress

  getLegalName, setLegalName

  getNumber, setNumber

  addPartyRole, getPartyRole, removePartyRole

  with a "kinds of classes" interface name that follows this pattern:

  I<noun, just like a class name>

- Or the *kinds of behavior* you want such objects to exhibit

  ITotal (or ICalcTotal)

  calcTotal

  with a "kinds of behavior" interface name that follows this pattern:

  I<verb, just like a method name>

With these variations on "kinds of behavior:"

- an algorithm plug in point with:

  I<verb> Algorithm

  indicating you are expecting to plug in an algorithm.

- or, feature sequencers with:

  I<verb> Sequencer

  indicating that what you are expecting is indeed a sequencer.

Which approach is better? Or perhaps it is better to ask: which approach when?

The "kinds of classes" classification can be expressed with interfaces or with superclasses; after all, "is a kind of" is the central idea behind an effective subclass–superclass relationship. We usually take on the main problem-domain classification first, with inheritance. For other "kinds of classes" classifications, we use interface names built with a noun.

The "kinds of classes" interfaces work well for plug-in points at the end of an association or the end of a message—describing the full spectrum of interactions (more than just a "get" and a "set") along that path.

In contrast, the "kinds of behavior" interfaces work well for:

- little groupings of functionality within an "is a kind of" classification scheme, and
- the functionality required at an algorithmic plug-in point.

In practice, the most common "kinds of classes" names we use in object models correspond to the "pattern players" in the companion book, *Object Models: Strategies, Patterns, and Applications.* As class names, we can express those pattern players this way:

Party, PartyRole

Place, PlaceRole

Thing, ThingRole

Moment, Interval

LineItem, ItemDescription, SpecificItem

Yet often we find that we want one of these categories (e.g., Role) to offer the same interface as another (e.g., Party) and so we end up using interfaces for such overlaps—and inheritance when we don't:

IParty, PartyRole

IPlace, PlaceRole

IThing, ThingRole

Moment, Interval

LineItem, ItemDescription, ISpecificItem

Each of those "kinds of classes" classes and interfaces might consist of a number of little "kinds of behavior" interfaces, for example:

IParty: IAddress, IConnectPartyRole, INameLegal, INumber, IPhone

PartyRole: INumber, IConnectMoment

where:

IAddress: getAddress, setAddress

IConnectMoment: addMoment, getMoment, removeMoment

IConnectPartyRole: addRole, getRole, removeRole

INameLegal: getNameLegal, setNameLegal

INumber: getNumber, setNumber

Problem-domain objects need large-grain interfaces like IParty and IPlace—and occasionally a plug-in method interface like ITaxAlgorithm. In contrast, human-interaction objects often need fine-grained interfaces like IAddress and INumber.

Incidentally, using "kinds of classes" interfaces requires that some class provide the creation services for objects of that type, such a class is known as a factory (let it know what kind of object you need and it makes one up for you).

3.8 What Java Interfaces Lack

From a software designer's perspective, interfaces are the most important language construct in Java. As you've seen again and again in this chapter, interfaces open the door to remarkable design flexibility.

Yet, to make sure you leave this chapter with your feet firmly planted on *terra firma,* we wrap up this chapter with Java interface shortcomings.

Java interfaces specify method signatures and nothing more. And that's just not enough. James Gosling knows this; he included assertions in the last Java spec he wrote himself, took them out during a schedule crunch, and regrets having done so (as reported in an interview in *JavaWorld,* in March 1998).

We really should have syntax for three kinds of assertions, so we can express:

- The conditions for invoking a particular method (method preconditions)
- The conditions that an object satisfy at the end of a particular method (method postconditions)
- The conditions that an object must satisfy at the end of *any* method execution (commonly referred to as class invariants).

You see, there is a whole world of implied context for each and every plug-in point that you establish with an interface. Each plug-in point is like an integrated-circuit socket on a circuit board.

1. Someone decided the added flexibility was worth the added cost of establishing that plug-in point.
2. Someone expects that whatever is plugged into that socket will abide by certain rules and conventions.

For Java plug-in points, interfaces, it would be great if you could express that context explicitly, with programming language syntax for preconditions, postconditions, and assertions (the programming language Eiffel sets the standard here). We hope to see Java include such syntax at some point in the future.

3.9 Summary

In this chapter you've worked with interfaces: common sets of method signatures that you define for use again and again in your application.

Designing with interfaces is the most significant aspect of Java-inspired design because it gives you freedom from associations that are hardwired to just one class of objects and freedom from scenario interactions that are hardwired to objects in just one class. For systems in which flexibility, extensibility, and pluggability are key issues, Java-style interfaces are a must. Indeed the larger the system and the longer the potential life span of a system, the more significant interface-centric design becomes.

In this chapter, you've learned and applied the following specific strategies for designing better apps:

Challenge Each Association Strategy: *Is this association hardwired only to objects in that class (simpler), or is this an association to any object that implements a certain interface (more flexible, extensible, pluggable)?*

Challenge Each Message-Send Strategy: *Is this message-send hardwired only to objects in that class (simpler), or is this a message-send to any object that implements a certain interface (more flexible, extensible, pluggable)?*

Factor Out Repeaters Strategy: *Factor out method signatures that repeat within your class diagram. Resolve synonyms into a single signature. Generalize overly specific names into a single signature. Rea-*

sons for use: to explicitly capture the common, reusable behavior and to bring a higher level of abstraction into the model.

Factor Out to a Proxy Strategy: *Factor out method signatures into a proxy, an object with a solo association to some other object. Reason for use: to simplify the proxy within a class diagram and its scenarios (Figure 3-9).*

Factor Out for Analogous Apps Strategy: *Factor out method signatures that could be applicable in analogous apps. Reason for use: to increase likelihood of using and reusing off-the-shelf classes.*

Factor Out for Future Expansion Strategy: *Factor out method signatures now, so objects from different classes can be graciously accommodated in the future. Reason for use: to embrace change flexibility.*

Where to Add Interfaces Strategy: *Add interfaces at those points in your design that you anticipate change: (1) Connect with an interface implementer rather than with an object in a specific class; (2) Send a message to an interface implementer rather than to an object in a specific class; and (3) Invoke a plug-in method rather than a method defined within a class.*

Design-in, from Features to Interfaces Strategy:

1. *Look for a common feature, one you need to provide in different contexts.*
2. *Identify a set of method names that correspond to that feature.*
3. *Add an interface.*
4. *Identify implementers.*

Design-in, from Role Doubles to Interfaces Strategy:

1. *Take a role and turn its method signatures into a role-inspired interface.*
2. *Let another role (a "role double") offer that same interface by:*

- *implementing that interface, and*
- *delegating the real work back to the original role player.*

Design-in, from Collections and Members to Interfaces Strategy:

1. *Does your object hold a collection of other objects? If so:*
   a. *Consider the potential "across the collection" method signatures.*
   b. *If other collections might offer the same set of method signatures, then design in that common interface.*
2. *Is your object a member within a collection? If so:*

   If that object needs to provide an interface similar to the collections it is in, then design in that common interface.
3. *Identify implementers.*

Design-in, from Scenarios to Interfaces Strategy:

1. *Look for similar interactions.*
2. *Add an interface-implementer column.*

   Use this naming convention:

   I<what it does> Implementer.
3. *Add an interface: I<what it does>.*
4. *Identify implementers.*

Interface Granularity Strategy: *If a method signature can only exist with others, then add it directly to an interface definition with those others (no need for a separate, one-signature interface).*

Design-in, from Intra-class Roles to Interfaces Strategy:

1. *Identify roles that objects within a class can play.*
2. *Establish an interface for each of those roles.*
3. *Identify implementers.*

When to Use Plug-in Algorithms and Interfaces Strategy: *Use a plug-in algorithm and interface when you find this combination of problems:*

- *An algorithm you want to use can vary from object to object within a single class*
- *An algorithm is complex enough that you cannot drive its variation using attribute values alone.*
- *An algorithm is different for different categories of objects within a class—and even within those categories (hence, adding a category-description class won't resolve this problem).*
- *An algorithm you want to use will be different over time and you don't know at this point what all those differences will be.*

Design-in, from Plug-in Algorithms to Interfaces Strategy:

1. *Look for useful functionality you'd like to "plug in."*
2. *Add a plug-in point, using an interface.*
3. *Identify implementers.*