Editor: Martin Fowler ■ ThoughtWorks ■ fowler@acm.org

To Be Explicit

Martin Fowler

Software is an odd medium in which to construct something. Because few physical forces make you design one way or another, many design decisions sadly resist any form of objective analysis. Where design counts is often not in how the software runs but in how easy it is to change. When how it runs is important, ease of change can be the biggest factor in ensuring good performance.

This drive toward changeability is why it's so important for a design to clearly show what the program does—and how it does it. After all, it's hard to change something when you can't see what it does. An interesting corollary of this is that people often use specific designs because they are easy to change, but when they make the program difficult to understand, the effect is the reverse of what was intended.

Attributes and dictionaries

Let's say we want a person data structure. We can accomplish this by having specific fields, as Figure 1 shows. Of course, to make this work, we must define the variables in the person class. Like many modern languages, Ruby provides a dictionary data structure (also knows as a *map*, *associative array*, or *hash table*). We could use Ruby instead to define the person class, using the approach in Figure 2. (This is slower, but let's assume this section of code isn't performance critical.)

Using a dictionary is appealing because it lets you change what you store in the person without changing the person class. If you want to add a telephone number, you can do it without altering the original code.

Despite this, the dictionary doesn't make it easier to modify the code. If I'm trying to use the person structure, I can't tell what is in it. To learn that someone's storing the number of dependents, I must review the entire system. If the number of dependents is declared in the

```
class Person
      attr_accessor :lastName, :firstName, :numberOfDependents
end

def frag1
      martin = Person.new
      martin.firstName = "Martin"
      martin.lastName = "Fowler"
      martin.numberOfDependents = 1
      print (martin.firstName, " ", martin.lastName, " has ",
                  martin.numberOfDependents, " dependents")

end
```

Figure 1. Explicit fields (using Ruby).

0740-7459/01/$10.00 © 2001 IEEE

Figure 2. Dictionary fields (using Ruby).

```
class Person
       attr_accessor :data
       def initialize()
              @data = {}
       end
end

def frag2
       martin = Person.new
       martin.data["firstName"] = "Martin"
       martin.data["lastName"] = "Fowler"
       martin.data["numberOfDependents"] = 1

       print (martin.data["firstName"]," ",
                  martin.data["lastName"], " has ",
                  martin.data["numberOfDependents"],
                  " dependents")
end
```

class, then I only have to look in the person class to see what it supports.

The key principle is that explicit code is easier to understand—which makes the code easier to modify. As Kent Beck puts is, the explicit code is intention revealing.

This dictionary example is small in scale, but the principle holds at almost every scale of software design.

Events and explicit calls

Here's another example, on a slightly bigger scale. Many platforms support the notion of events to communicate between modules. Say we have a reservation module that, when canceled, needs to get a person module to send email to that person.

We can do this using an event, as Figure 3 shows. We can define interesting events that affect a reservation, and any object that wants to do anything when an event occurs can build a handler to react when it occurs. This approach is appealing because you need not modify the reservation class to get something else to happen when you cancel a reservation. As long as other objects put handlers on the event, it's easy to extend the behavior at these points.

```
public delegate void ReservationHandler (IReservation source);
public class Reservation ...
       public String Id;
       public event ReservationHandler Cancelled;
       public Person client {
              get {
                     return client;
                  }
                  set {
                     value.AddReservation(this);
                  }
              }
              public void Cancel(){
                     Cancelled (this);
              }

public class Person ...
       public String EmailAddress;
       public readonly ArrayList reservations;

       public void SendCancellationMessage(Reservation arg) {
              // send a message
       }

       public void AddReservation(Reservation arg) {
              //invoke SendCancellationMessage when the cancelled event occurs on arg
                     arg.Cancelled +=
                     new ReservationHandler(SendCancellationMessage);
              }
```

Figure 3. Cancellation using events (using C#).

However, there is a cost to using events—I can't see what happens at cancellation by reading the code in the cancellation method. To find out what happens, I have to search for all the code that has a handler for the event. The explicit code for this (see Figure 4) clearly shows in the cancel method the consequences of cancellation, at the cost of modifying the reservation class when I need to change the behavior.

I've seen a few code examples that use events heavily, and the problem is that it's hard to determine what the program does when you call a method. This becomes particularly awkward when you're debugging, because behavior pops up suddenly in places you don't expect.

I'm not saying that you shouldn't use events. They let you carry out behavior without changing a class, which makes them useful when working with library classes you can't modify. They are also valuable because they don't create a dependency from the class triggering the event to the one that needs to react. This lack of a dependency is valuable when the two classes are in different packages and you don't want to add a dependency. The class case of this is when you want to modify a window in a presentation when some domain object changes. Events let you do this while preserving the vital separation of the presentation and domain.

Those forces both suggest events, but in their absence, the lack of explicitness of events becomes more dominant. So, I would be reluctant to use events between two application classes that can be aware of each other.

As you can see, explicitness is not always the dominant force in design decisions. In this example, packaging and dependency forces are also important. People often underestimate the value of explicitness. There are times when I would add a dependency to make code more explicit, but, as always with design, each situation has its own trade-offs to consider.

Data-driven code and explicit subclasses

My final example is on yet a bigger scale. Consider a discounting scheme for orders that uses different discounting plans. The blue plan gives you a fixed discount of 150 if you buy goods from a particular group of suppliers and the value of your order is over a certain threshold. The red plan gives you a 10 percent discount when delivering to certain US states.

Figure 5 presents explicit code for this. The order has a discounter with specific subclasses for the blue and red cases. The data-driven version in Figure 6 uses a generic discounter that is set up with data when the order is created.

The generic discounter's advantage is that you can create new kinds of discounters without making new classes by writing code—if the new classes fit in with the generic behavior. For the sake of argument, let's assume they can. Is the generic case always the best choice? No, again because of explicitness. The explicit subclasses are easier to read and they make it easier to

Figure 4. An explicit reaction to cancel (using C#).

```
public class Reservation ...

       public String Id;
       public Person client;

       public void Cancel(){
              client.SendCancellationMessage(this);
       }
```

Figure 5. Explicitly programmed discount logic (using C#).

```
public class Order ...
       public Decimal BaseAmount;
       public String Supplier;
       public String DeliveryState;
       public Discounter Discounter;

       public virtual Decimal Discount {
              get {
                     return Discounter.Value(this);
              }
       }
}
abstract public class Discounter {
       abstract public Decimal Value (Order order);
}
public class BlueDiscounter : Discounter {
       public readonly IList DiscountedSuppliers = new ArrayList();
       public Decimal Threshold = 500m;
       public void AddDiscountedSupplier(String arg) {
              DiscountedSuppliers.Add(arg);
       }
       public override Decimal Value (Order order) {
              return (DiscountApplies(order)) ? 150 : 0;
       }
       private Boolean DiscountApplies(Order order) {
              return DiscountedSuppliers.Contains(order.Supplier) &&
                     (order.BaseAmount > Threshold);
       }
}
public class RedDiscounter : Discounter {
       public readonly IList DiscountedStates = new ArrayList();
       public void AddDiscountedState (String arg) {
              DiscountedStates.Add(arg);
       }
       public override Decimal Value (Order order) {
                     return (DiscountedStates.Contains(order.DeliveryState)) ?
                              order.BaseAmount * 0.1m : 0;
       }
}

// to set up a blue order
BlueDiscounter bluePlan = new BlueDiscounter();
bluePlan.AddDiscountedSupplier("ieee");

blue = new Order();
blue.Discounter = bluePlan;
blue.BaseAmount = 500;
blue.Supplier = "ieee";
```

```
public class GenericOrder : Order ...
       public Discounter Discounter;
       public override Decimal Discount {
              get {
                     return Discounter.Value(this);
              }
       }

public enum DiscountType {constant, proportional};

public class Discounter ...
       public DiscountType Type;
       public IList DiscountedValues;
       public String PropertyNameForInclude;
       public String PropertyNameForCompare;
       public Decimal CompareThreshold;
       public Decimal Amount;

       public Decimal Value(GenericOrder order) {
              if (ShouldApplyDiscount(order)) {
                     if (Type == DiscountType.constant)
                            return Amount;
                     if (Type == DiscountType.proportional)
                            return Amount * order.BaseAmount;
                     throw new Exception ("Unreachable Code reached");
              } else return 0;
       }

       private Boolean ShouldApplyDiscount(Order order) {
              return PassesContainTest(order) &&
                            PassesCompareTest(order);
       }
              private Boolean PassesContainTest(Order order) {
                     return DiscountedValues.Contains
                            (GetPropertyValue(order, PropertyNameForInclude));
              }
              private Boolean PassesCompareTest(Order order){
                     if (PropertyNameForCompare == null) return true;
                     else {
                            Decimal compareValue =
                                   (Decimal) GetPropertyValue(order, PropertyNameForCom-
pare);
                            return compareValue > CompareThreshold;
                     }
              }
              private Object GetPropertyValue (Order order, String propertyName) {
                     FieldInfo fi = typeof(Order).GetField(propertyName);
                     if (fi == null)
                            throw new Exception("unable to find field for " + property-
Name);
                     return fi.GetValue(order);
              }
       }
```

Figure 6. Data-programmed discount logic (using C#).

```
//to set up a blue order
       GenericDiscounter blueDiscounter = new GenericDiscounter();
       String[] suppliers = {"ieee"};
       blueDiscounter.DiscountedValues = suppliers;
       blueDiscounter.PropertyNameForInclude = "Supplier";
       blueDiscounter.Amount = 150;
       blueDiscounter.PropertyNameForCompare = "BaseAmount";
       blueDiscounter.CompareThreshold = 500m;
       blueDiscounter.Type = DiscountType.constant;

       blue = new Order();
       blue.BaseAmount = 500;
       blue.Discounter = blueDiscounter;
```

understand the behavior. With the generic case, you must look at the generic code and setup code, and it's harder to see what's happening—and even harder for more complicated bits of behavior. Of course, we can extend the generic order without "programming," but I'd argue that configuring that data is a form of programming. Debugging and testing are often both difficult and overlooked with data-driven behavior.

The generic case works when you have dozens of discounters. In such cases, the volume of code becomes a problem, while greater volumes of data are less problematic. Sometimes a well-chosen data-driven abstraction can make the logic collapse into a much smaller and easier-to-maintain piece of code.

Ease of deploying new code is also a factor. If you can easily add new subclasses to an existing system, explicit behavior works well. However, generic behavior is a necessity if new code means long and awkward compile and link cycles.

There's also the option of combining the two, using a data-driven generic design for most of the cases and explicit subclasses for a few hard cases. I like this approach because it keeps the generic design much simpler, but the subclasses give you a lot of flexibility when you need it.

Explicitness is not an absolute in design, but clever designs often become hard to use because they aren't explicit enough. In some cases, the cost is worth it, but it's always something to consider. In the last few years, I've tended to choose explicit designs more often because my views of what makes good design have evolved (hopefully in the right direction).

Martin Fowler is the chief scientist for ThoughtWorks, an Internet systems delivery and consulting company. Contact him at fowler@acm.org.