A hierarchy that acts like a class

Bobby Woolf

WHILE RECENTLY CONSULTING for a client, I developed a miniframework that incorporated several design patterns. These patterns combined to make a rather complex hierarchy of subtypes act like a single class. Besides providing useful functionality that the client and their customers needed (which never hurts!), the framework successfully demonstrated the following techniques:

- what I call "how to do a case statement in Smalltalk," the Objects from States pattern[1]
- an example of the Reusability through Self-Encapsulation pattern language[2]
- a variation of the Factory Method pattern[3]
- an example of what I call the "Null Object" pattern, also known as NoWorker[4] and Null Representation[5]
- limited amounts of visual behavior in domain objects

The framework itself is an example of a case statement using Self-Encapsulation. In it are a set of Factory Methods and a Null Object. Even though it is a framework of domain objects, it still contains some application model behavior used to display the objects in a view. Not only does the framework show simple examples of these patterns, but it also shows how to combine individual patterns together to solve more complex problems.

THE PROBLEM

Part of this client's system consisted of a questionnaire the user could display so that he could answer it. Analysis revealed that the domain objects were Questionnaire, which contained a list of Questions, each of which had exactly one Answer. Further requirements gathering discovered that there were three different ways a user could answer a question: most questions needed yes/no answers; some needed one of a list of possible answers; others needed freeform text answers. The view could easily indicate these different ways of answering using check boxes, combo boxes, and input field widgets, respectively.

Most of the domain code had already been implemented for me. There were already Questionnaire, Question, and Answer classes, and the container relationships between the three had been defined. The problem was the difficulty supporting the three different ways of answering and the three different kinds of visual widgets used to input and display answers.

The solution that had been implemented required a lot of fairly ugly code in the Answer class. The class had three instance variables: yesOrNo, selection, and responseText. Each instance only used one of those variables; the other two were always nil. The displayString method printed-out each variable's value as long as it wasn't nil; because only one variable was not nil at any given time, that's the only one that was printed. What the implementation did not solve was displaying a particular Answer as different widgets. When I started on the project, testing methods like isYesNo, isSelection, and isText were about to be introduced.

SMALLTALK CASE STATEMENT

Obviously Answer was becoming too complex. The solution I devised was to expand the Answer class into a hierarchy of classes:

```
Answer ()
    BooleanAnswer (yesOrNo)
    EnumerationAnswer (responseChoices, responseIndex)
    TextAnswer (responseText)
```

This way each question could have the appropriate type of answer: boolean, symbol, or text. None of the Answer objects wasted any instance variables. Each class knew what its instance variables' types were and how to handle them. This factored the complexity of handling these different possibilities into separate classes so that the decisions each class had to make were actually quite simple.

The reason I call this hierarchy an example of a Smalltalk case statement is that it eliminates the need for testing methods like isYesNo, isSelection, and isText. An example of a case statement method would be something like

```
Answer>>visualWidget
    self isYesNo ifTrue: ["Use a check box."].
    self isSelection ifTrue: ["Use a combo box."].
    self isText ifTrue: ["Use an input field."].
```

This is poor object-oriented (O-O) style. Sometimes developers clamor for Smalltalk to have a case statement, usually because they're trying to write code like this. Although all of us write code like this sometimes, it is best avoided. Factoring the class into a hierarchy allowed me to eliminate the testing methods and simplify the code like this:

```
Answer>>visualWidget
    ^self subclassResponsibility
BooleanAnswer>>visualWidget
    "Use a check box."

EnumerationAnswer>>visualWidget
    "Use a combo box."

TextAnswer>>visualWidget
    "Use an input field."
```

This is a case statement via polymorphism and inheritance. Just send the message and whatever implementer gets run is the correct case. This forces the "testing" to be encapsulated within the hierarchy where it can easily be reused. If the results of the testing need to be changed, the code is easier to maintain because it is so cleanly encapsulated. Finally, the differences between the peer classes are easy to see; just look at the methods they implement instead of inheriting.

This case statement framework is also easy to extend to add new cases. For an example of this, see the "Null Object" section of this article.

SELF-ENCAPSULATION

A problem this hierarchy introduced is that now each particular Question instance had to know what kind of Answer instance it had. To avoid this problem, I wanted all of the concrete classes to be polymorphically equivalent. This means that they would all have the same common interface so that I could generally treat any instance as an Answer without regard to which concrete subclass it was.

To do this, I defined the common interface in Answer with messages like response, response:, and displayString. In Answer, each of these methods returned the subclassResponsibility error. Each subclass implemented the messages appropriately in terms of its state. For example, here's how EnumerationAnswer handled the response aspect:

```
EnumerationAnswer>>response
    | index |
    index := self responseIndex.
    ^index == 0
        ifTrue: [nil]
        ifFalse: [self responseChoices at: index]

EnumerationAnswer>>response: newResponse
    self responseIndex:
        (self responseChoices
            identityIndexOf: newResponse)
```

This is an example of the Reusability through Self-Encapsulation pattern language, albeit an extremely simple example. The language shows how to implement an extensible yet well-encapsulated hierarchy. An abstract superclass defines the interface the subclasses will follow. The interface is implemented in terms of a small number of kernel methods that each subclass must implement appropriately. As the language suggests, Answer defines the hierarchy's interface and leaves the implementation details to the subclasses.

This could be considered a variation of the Factor a Superclass pattern.[6] That pattern starts with a number of peer classes and factors their shared variables and behavior into a common superclass so that the subclasses do not duplicate each other's variables and behavior. What I did with Answer was the same process in reverse; I started with one class and factored it into many subclasses. I use the superclass to define a common interface rather than implement common behavior, so my efforts are more reminiscent of Self-Encapsulation than Factor a Superclass.

FACTORY METHOD

Another problem the Answer hierarchy introduced is the matter of assuring that the right kind of Answer instance gets assigned to each Question. How does a Question phrased as a yes/no question get a BooleanAnswer? How does one with a list of possible answers specify that it needs not only an EnumerationAnswer but the list of choices as well?

To solve this problem, I introduced the following messages into Question: useYesNoAnswer, usePossibleAnswers:, and useTextAnswer. This way, as each Question was created, the answer details could be specified as well. Here are some examples:

```
question1 := (Question text: 'Do chickens have lips?')
        useYesNoAnswer.
question2 := (Question text: 'Are you lazy?')
        usePossibleAnswers:
            #(#always #sometimes #never).
question3 := (Question text: 'How old are you?')
        useTextAnswer.
```

I wanted to hide the complexity of the Answer hierarchy and maintain the illusion that it was still just one class. This way the one class could actually manage the other classes and their use. This will also encapsulate this management within the class. Because this is not a complicated hierarchy, its management is fairly simple. Answer's instance creation protocol just has to allow for creating each kind of answer. Here are the methods that do this:

```
Answer class>>yesNoAnswer
    ^BooleanAnswer new

Answer class>>possibleAnswers: answerList
    ^EnumerationAnswer new responseChoices: answerList


Answer class>>textAnswer
    ^TextAnswer new
```

Question in turn just delegates the Answer creation to that class:

```
Question>>useYesNoAnswer
    self answer: Answer yesNoAnswer
Question>>usePossibleAnswers: answerList
```

```
    self answer: (Answer possibleAnswers: answerList)

Question>>useTextAnswer
    self answer: Answer textAnswer
```

The three instance creation methods in Answer are examples of the Factory Method pattern, or at least a variation thereof. Gamma et al. say that the "Factory Method lets a class defer instantiation to subclasses." A classic example in Smalltalk-80 is the way View defines the method defaultControllerClass. Each subclass of View subimplements defaultControllerClass to return the class for its controller. Thus defaultControllerClass is a Factory Method.

The Factory Methods in Answer are yesNoAnswer, possibleAnswers:, and textAnswer. Because they are not standard protocol that is overridden in subclasses, they are not standard Factory Method examples. However, they are a variation on the same theme because they use message sends to hide the existence of the various Answer subclasses, as well as their names and interfaces. As far as a collaborator like Question is concerned, there is only one Answer class (not a hierarchy) and it is able to act in these various ways. This encapsulates the hierarchy and simplifies its interface to the rest of the system.

I cannot claim to have invented this technique. In VisualWorks, Filename uses it to determine which of its subclasses to use. Similarly, CompositePart uses it to determine which Wrapper class to use.

Alternate solution: Question hierarchy

When analyzing the requirements and designing a solution, I considered developing a Question hierarchy as well as an Answer hierarchy. This would have incorporated Factory Method more directly by using subclassing (as the pattern suggests). The Question class would have defined a method like defaultAnswer in terms of defaultAnswerClass. Then Question subclasses would override defaultAnswerClass to return the appropriate Answer subclass.

For example:

```
Question>>defaultAnswer
    ^self defaultAnswerClass new

Question>>defaultAnswerClass
    ^Answer

BooleanQuestion>>defaultAnswerClass
    ^BooleanAnswer
```

and so forth for EnumerationQuestion>>defaultAnswerClass and TextQuestion>>defaultAnswerClass. In fact, I did implement Question>>defaultAnswerClass in preparation for such a protocol.

This combining of dual hierarchies is an example of the Bridge pattern, where an abstraction is decoupled from its implementation by implementing it in two hierarchies. The two hierarchies can be extended independently, and because they are connected by a standard interface, most any pair of instances from the two hierarchies can work together.[3]

The problem with Question and Answer hierarchies is that the classes are not decoupled from each other. There is a one-to-one correspondence between the classes in the two hierarchies: BooleanQuestion/BooleanAnswer, EnumerationQuestion/EnumerationAnswer, and TextQuestion/TextAnswer. Anytime a new class was added to one hierarchy, a corresponding class just like it would need to be added to the other hierarchy, e.g., RangeQuestion would require SliderAnswer. So these hierarchies are not truly decoupled; in fact, they require duplicate effort to extend both hierarchies.

Another problem with the Question hierarchy is that subclasses would not have behaved differently from their superclass. All Questions were essentially the same, even though they expected different types of answers. Hopefully, those that claimed to need a yes/no answer were phrased as a yes/no question, but there was no way to enforce this in BooleanQuestion.

In the end, although an Answer hierarchy looked promising, a similar Question hierarchy not only wasn't helpful, but was in fact counterproductive. The Question subclasses would not have introduced any helpful behavior but would have required duplicate effort. Thus a Question hierarchy was not necessary.

Tangent topic: SelectionState class?

Kent Beck recently wrote an excellent column, "Clean code: Pipe dream or state of mind?"[7] In it, he describes how he factored out a State Object using two classes, SingleSelectionState and GroupSelectionState, a terrific solution to the problem he was facing. He might have taken the solution one step further by using the Factory Method variation described here. Using it, he would introduce an abstract class, SelectionState.

The SelectionState class would define the interface for all SelectionState instances (SingleSelectionState, GroupSelectionState, and eventually DelegationSelectionState). It would also serve as the hierarchy's interface to the rest of the system (collaborators such as SelectionTool). Then methods like SelectionTool>>setSelectionState: could be moved into SelectionState (probably as SelectionState>>setSelectionState: aFigure). setSelectionState: is a fairly messy method that must contain a lot of knowledge about the classes in the SelectionState hierarchy. Notice that when Kent introduced an additional class, DelegationSelectionState, he had to rewrite this method. This messiness indicates that the method should be encapsulated within the hierarchy, which moving it to SelectionState would do. Also, if setSelectionState: were ever needed by another collaborator that was not a SelectionTool, the method would be available for reuse.

NULL OBJECT

Although requirements gathering discovered fairly early on that there were three types of answers—yes/no, list of choices, and freeform text—we discovered later that there was actually a fourth, hidden case to be considered. Some "questions" in the questionnaire were actually not ques-

tions per se, but headings for subsequent questions. Such a question might be "Check each of the following traits that describes you:"

Why not Heading?

The problem this introduces is that such a heading is more of a Heading object than a Question object. Both have text, but headings don't have answers the way questions do. But then how should this be displayed in a view?

The questionnaire was displayed as a table with two main columns, question and answer. Every row was expected to have two aspects that would be displayed in the two columns. Thus Heading needed to have an answer aspect just like Question, and the heading's answer would need to be able to display itself the way an Answer can.

A Null Object supports an extensive, customized interface while encapsulating how to "do nothing."

This caused Heading to work just like a Question, so I found no need for a separate Heading class. This solution may be an example of improperly letting the view define the domain; iterating over the design might produce a better one. Yet I feel that the solution described below turned out pretty well and may in fact be the most graceful way to hide the exceptions to some otherwise simple and uniform rules.

Perhaps another reason I rolled Heading into the Question class is that my deadline for completing this subsystem was rapidly approaching. It's funny how when I'm near a deadline, the current design I've already implemented can look much better than an alternative that requires rewriting a lot of code!

A heading's answer

Modeling the heading as a Question object, it had to have an Answer, but none of the three Answer subclasses applied. For this purpose, I developed a fourth Answer subclass called NullAnswer:

```
Answer ()
    NullAnswer ()
```

As a subclass of Answer, NullAnswer preserved the Answer interface, but did so without doing anything. Here are some examples of the methods it defined:

```
NullAnswer>>response
    ^nil

NullAnswer>>displayString
    ^'n/a'
```

NullAnswer is an example of what I call the Null Object pattern. I haven't seen documentation for this pattern published anywhere, but it is discussed fairly often. The pattern describes an object that shares the same interface as others of its type but that reacts to these methods by doing nothing. The trick is in designing, for each message, what doing nothing means. Typically it means getters that return nil or empty collections and display methods that show the object as null. Setter methods are usually ignored; they can create a real instance and substitute it for the Null Object, but this is more the behavior of a Proxy[3] than a Null Object.

The beauty of a Null Object is that it supports an extensive, customized interface and encapsulates the decisions about how it should "do nothing." nil is perhaps the most famous Null Object, but it doesn't really count because its interface is neither extensive nor customized. Yet programmers often use nil in a variable that hasn't been assigned yet. This leads to copious amounts of code that constantly check the variable for nil before sending it messages. This code can be simplified by assigning the variable a Null Object of the correct type and then sending the variable messages with impunity. Also, rather than each collaborator deciding what to do when the variable is nil, these decisions are encapsulated within the Null Object for reuse and consistency across all collaborators.

Adding NullAnswer

The infrastructure to support the new NullAnswer class was easy to introduce because the hierarchy was well encapsulated. It consisted of exactly one method:

```
Answer class>>nullAnswer
    ^NullAnswer new
```

Then collaborators, such as Question, just needed to tie into the expanded interface in a convenient way:

```
Question>>useNullAnswer
    self answer: Answer nullAnswer
```

Other collaborators could tie in just as easily.

VISUAL CODE IN DOMAIN OBJECTS

One of the distinguishing factors of the different types of answers is the way they were to be displayed. As described earlier, a question could be displayed as a check box, a combo box, or an input field. Also, a null answer would need to be displayed with a "do nothing" widget.

Much has been written recently about the importance of separating domain and application behavior, including by me.[8] Basically, domain objects represent core business behavior, while application objects know how to display domain objects in useful ways. This is the basic architecture I follow for all of my development, including the Questionnaire framework.

Questionnaire is the root of the domain framework described earlier. In turn, I also implemented a corresponding QuestionnaireUI class to represent Questionnaires. (Because I was developing in VisualWorks, QuestionnaireUI

was a subclass of ApplicationModel.) QuestionnaireUI was essentially a glorified table sort of widget where each row displayed a question. The table had two main columns, the question text and the question answer.

Why not AnswerUI?

The strength and limitation of this simple design are that there were no application models for the Questions and their Answers. I did not want these application models because the QuestionnaireUI itself was just a table, practically a TableView or a DataSetView (in VisualWorks). Just as those classes don't contain separate "RowView" and "CellView" classes, I didn't want QuestionnaireUI to contain numerous QuestionUI objects. Those would do nothing more than contain QuestionTextUI and AnswerUI objects, each of which would do little except to contain a single view such as InputFieldView or ComboBoxView. This seemed to me like an explosion of custom classes and do-practically-nothing objects, a complication I wanted to avoid. Once again, avoiding these classes appears to be the most graceful way to hide complicated exceptions to otherwise simple, uniform rules.

Had there been an AnswerUI class, it could have made the decision as to what kind of widget to use to display each kind of Answer (its domain model). Actually, this might have necessitated the need for a separate AnswerUI subclass for each Answer subclass. Then the AnswerUI hierarchy would have been tightly bound to the Answer hierarchy and duplicate effort would have been required to extend both in tandem. In any event, I did not have any AnswerUI class available. All I had was a cell in a table that was supposed to display an answer and an Answer domain object that contained the data for that cell.

How to display an Answer

Because the Questions and Answers did not have their own application model counterparts, there was no obvious place to put the code that decided how to display the different types of Answers. The way I solved this problem was to have the cell ask the domain object what widget should be used to display it. The domain object would return the widget and the table would display that widget in the corresponding cell.

This necessitated introducing the message visualWidget into the Answer hierarchy (as shown earlier). Each subclass would return an instance of the widget appropriate for itself. Thus visualWidget is another example of the Factory Method pattern, a more accurate example, since subclasses override the superimplementer.

Adding an application layer method like visualWidget into a domain layer object like Answer is certainly unusual, but not necessarily wrong. It was, after all, a single method, not a whole suite of behaviors that could easily become indistinguishable and inseparable from the domain behavior. Furthermore, its behavior is likely to be appropriate for any application layer that might be built on this domain, so there is little need to be able to swap one application object in for another.

Finally, I was still able to distinguish the application code from the domain code in this domain class using ENVY (ENVY/Developer, Object Technology International, Inc.). I defined the Answer hierarchy in the Domain application. Then I extended each of the hierarchy's classes in the UI application to add the visualWidget method. This not only clarified which Answer methods were for UI behavior, but it also meant that a developer could easily unload all UI code from the image—even that which the domain classes contained—by unloading the UI application.

CONCLUSION

One simple class in one minor part of a system turned out to employ a number of powerful O-O techniques. Answer was conceptually an uncomplicated little class that turned out to have multiple personalities. As the code to support those personalities grew, the need to expand the class into a hierarchy became apparent. But because the complexity of the hierarchy distracted from the simplicity of the class, the need to hide this complexity became apparent as well. I was able to develop this complex hierarchy with a simple, single class–like interface by using and combining the following techniques:

- Smalltalk Case Statement: This is what led to the hierarchy. Each case was represented as a separate class.
- Self-Encapsulation: This is what led to the abstract superclass. It defined the public interface that all subclasses would support so that various instances could be treated polymorphically.
- Factory Method: This hid the concrete subclasses so that they were never referenced from outside the hierarchy. The hierarchy's collaborators interfaced with the abstract superclass, telling it what behavior was expected from a new instance and trusting the superclass to return an appropriate instance.
- Null Object: This substituted as the answer for a question that did not need an answer. It supported the abstract superclass's interface and thus could be used just like any other concrete subclass. And it encapsulated the "do nothing" code so that all questions without answers would behave the same.
- Visual Code in Domain Objects: This acted as the application object for a domain object whose display was so simple that it did not need a separate application object. ENVY extensions demonstrated that separate objects are not the only way to separate independent layers of code.

I hope this successfully illustrates these techniques and shows how they may be used to solve real-world problems. I feel it is important not only that we document these techniques as reusable design patterns and pattern languages, but also that we show how they can be applied in practice to help develop better-quality software. I hope this experience report will prove useful to you. Please feel free to contact me at woolf@acm.org if you have any (tastefully phrased) questions or comments.

References

1. Beck, K. Death to case statements, part 2, THE SMALLTALK REPORT 3(4), 1994.
2. Auer, K. Reusability through self-encapsulation, Coplien, J.O. and D.C. Schmidt, Eds., PATTERN LANGUAGES OF PROGRAM DESIGN, Addison-Wesley, Reading, MA, 1995.
3. Gamma, E., R. Helm, R. Johnson, and J. Vlissides. DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE, Addison-Wesley, Reading, MA, 1995.
4. Hendley, G. The NoWorker pattern, available from the author at ghendley@ksccary.com
5. Carlini, G. Type and implementation, available from the author at giuliano@filenet.com
6. Beck, K. Inheritance: The rest of the story, THE SMALLTALK REPORT 3(1), 1993.
7. Beck, K. Clean code: Pipe dream or state of mind? THE SMALLTALK REPORT 4(8):20–22, 1995.
8. Woolf, B. Making MVC code more reusable, THE SMALLTALK REPORT 4(4):15–18, 1995.

Bobby Woolf is a Member of Technical Staff at Knowledge Systems Corp. in Cary, North Carolina. He is actively engaged in the patterns movement that is seeking to document common software development techniques. Comments are welcome at woolf@acm.org.