Rules Are for Fools, Patterns Are for Cool Fools

By Mahesh Dodani

"I wasn't a very good designer, but then I read some books on patterns—and now I am a real good designer." This quote from one of the participants of the Patterns Fish Bowl discussion at last years' OOPSLA made me cringe. Here is the sad confession of a cool fool with a tool—Pat Terna Buser.

PAT'S CONFESSION

Hi, my name is Pat and I have been abusing patterns for over 18 months now. 18 months ago I was just a happy programmer, hacking my way through code and willing the code to work through long hours of programming. My only design technique was to "start with something that is close to what I wanted, and then just modify it like hell until it works." I was usually left alone in my dark programming world, given specific functionality to program, test, and integrate into the system.

Then into my programming world walked this knight in shining armor—Jerry. Jerry was the ultimate designer, prancing around the developers with the Gang of Four design patterns book[1] in his hand, using it as a bible to preach to us about good design techniques and lead us from our dark world into the hallowed land of well-designed systems and reusability. Jerry even insisted that everyone call him (and spell his name) JERR to show respect to the Gang of Four.

JERR showed us that everything about our code could be refactored using patterns. As soon as we recast our code using these patterns, our code became better—easier to understand by others who knew the secret code of patterns defined in the Book, easier to modify, easier to test, and easier to maintain.

I took to patterns like fish to water. I realized that patterns were my redemption and could lead me from the dark dungeons of programming to the cool world of designing. I memorized each of the patterns in the Book and was able to recite the intent, explain the motivation for using the pattern, and describe the structure of the pattern and the relationships to other patterns exactly as stated in the Book. I went through each line of code I had ever written and rewrote it using patterns. I was merciless in my application of patterns.

After applying patterns to my own code, I became a pattern evangelist in my company. I badgered each programmer about his code. How come you didn't use pattern X in this part of the code? (Obviously, its application would make the design much better, and make it easier for the rest of us to use.) I forced programmers to present their code in review sessions and hounded them for not using patterns. Programmers in my department feared these review sessions.

I was able to easily convince my project managers how much better our code was, measured in terms of the number of patterns used. This measure was used as a metric for reusability. They were able to parlay this reusability metric to propel our project into the best-of-breed within our company. We were invited to explain our methods and approaches to other projects within the company. We became the center of competency within our company, and everyone looked to me as their main designer.

I went from a lowly programmer to a well-paid and respected designer in just a

few months. My ability to turn our bad code to highly reusable code spread around the industry. I was asked to speak at several conferences regarding my use of patterns. I became a pattern celebrity! I got offers from several companies to come and help make their code more reusable, and teach them to use and speak the wonderful language of patterns. I gave up my job and became a consultant. I was paid exorbitant sums to work on converting systems to reusable systems by applying patterns. I used the number of patterns as a reusability metric, and assured success using this metric. I was able to guarantee during my period with the company that the number of patterns used in their code would grow dramatically.

The more I used patterns, the more I was addicted to their use. I was ruthless about the application of patterns wherever possible, regardless of the consequences to the code. My main intent was to see how many patterns I could apply to a single class (and its associated subclasses). Through hard work and long hours, I was able to come up with my own "pattern" of how to apply as many patterns from the Book to a single class.

I will explain my "pattern" for applying patterns below. I assume that you have familiarity with all the patterns in the Book.

Let's take a simple example, and I will show you my "pattern." The last project I worked on for a bank included the `Account` class and its associated subclasses, shown in Figure 1. Here is my "pattern" for applying patterns:

For the instance variables, apply the State Pattern
The State Pattern can be applied to each (or a subset) of the instance variables of a `Class`. For `Account`, the balance instance variable defines its state. Therefore, use the State Pattern to create an `AccountState` class that defines the same methods provided by `Account (deposit, withdraw, calcInterest)`. Each subclass of `AccountState (Open, Overdrawn, Closed)` redefines the implementation of the methods to reflect the behavior of the methods dependent on the state of the `Account`. The `Account` class has a state variable that references the current state instance. Changes to the state of the `Account` will result in the appropriate `AccountState` instance to be referenced by the state variable.

For each method, apply the Strategy Pattern
The Strategy Pattern can be applied to each method. For example, the `calcInterest` method can be implemented using several algorithms (e.g., simple interest, compound interest, fixed interest, and variable interest). Define an `InterestCalculator` strategy class with a single method for calculating interest. Define subclasses of the `InterestCalculator` strategy class that defines implementations of the various interest calculators. The `Account` class (or the `AccountState` class) has an instance variable strategy that references the particular interest calculator that applies. Each call for `calcInterest` calls the method in the strategy instance to calculate the interest passing in the context of the `Account` (e.g., `balance`).

For a grouping of instances, apply the Chain of Responsibility Pattern
The Chain of Responsibility Pattern can be applied to a grouping of instances to allow the handler of the request to be determined dynamically. For example, since `Account` instances are grouped by the `AccountPortfolio` (for a given `Customer` of a `Bank`), we can use the Chain of Responsibility Pattern to determine which `Account` instance to withdraw money from to handle an overdrawn account automatically. Each account instance grouped in an `AccountPortfolio` is chained to other accounts in a successor chain. The successor chain can be defined using a criteria that is meaningful to the `Bank`. When an account is overdrawn, the money to handle the overdrawn account is handled by withdrawing money from the `Account` in the successor chain that has enough money to handle the overdrawn amount.

For a grouping of instances, apply the Composite, Iterator, and Visitor Patterns
The Composite Pattern can be applied to a grouping of instances to organize it

into a tree-like structure (which include group and leaf nodes) and treat the objects uniformly. The Iterator Pattern allows group nodes to iterate through their subnodes, while the Visitor Pattern allows the separation of the operation to be performed on the Composite structure. For this example, the `AccountPortfolio` represents the root of the Composite structure. We can (and should) introduce several group nodes to structure the leaf `Account` nodes. The grouping of `Accounts` may be done by account type (e.g., savings accounts versus checking accounts), by interest type (e.g., no interest accounts versus simple interest accounts versus compound interest accounts), or by customer type (e.g., individual's accounts versus joint accounts versus children's accounts). Once we have defined these group nodes, it is straightforward to apply the Composite Pattern, where the common operations include showing the accounts' activity and the details of the accounts for printing statements, and determining the cumulative balance for accounting. The Iterator Pattern is applied whenever a group node iterates through its leaf nodes for all of the above operations. Note that the operations for printing statements are orthogonal to the basic behavior for `Accounts`. Therefore it is suitable to use the Visitor Pattern to implement these operations.

For clients of objects, apply the Observer Pattern
The Observer Pattern can be applied when a group of clients are dependent on changes to an object that they are observing. The clients register themselves as interested in (particular) changes in the object, and when the object changes it sends update messages to the registered clients. In this example, we have several options for applying the Observer Pattern. The `AccountPortfolio` and/or `Customer` can maintain a total balance of all their accounts and can register themselves as `Observers` on the `Account` objects. Any view (from a user interface perspective) of the `Account` or `AccountPortfolio` would need to register itself as interested in changes. Whenever an `Account` changes, it will send update messages to the observers, which will in turn use the `getBalance` method (for example) to get the change and update their information accordingly.

For clients of objects, apply the Proxy Pattern
The Proxy Pattern can be applied to allow different clients to interface to the services of an object in different ways and to allow distributed clients to access the services of an object. The Proxy Pattern allows a proxy to represent appropriate services to a client, accept a message from the client, invoke the appropriate method in the object, and return results to the client. For this example, develop proxy objects for both `AccountPortfolio` (for services applying to groups of `Account` objects) and `Account` (for services applying to accounts). So, a `TellerAccount` proxy object would present teller services to account objects, including `withdraw, deposit`, and `balance`; while a `TellerAccountPortfolio` proxy object would present teller services that apply to account groups, including transfer of funds between accounts.

For each hierarchy of objects, apply the Façade Pattern
The Façade Pattern can be applied to define a single unified interface to access an entire subsystem of classes and interfaces. For this example, notice that the classes involved with `Accounts` have grown because of the application of the previous patterns. To hide this complexity from the users of `Accounts`, make the classes involved with `Accounts` into a subsystem and define an `Account Façade` object to expose the services provided by this subsystem to the outside world. This `Façade` object would include services such as `withdraw, deposit, transfer, balance, balancesOf`, etc.

For complex methods, apply the Bridge Pattern
The Bridge Pattern can be applied to any complex method to separate the interface from the implementation. For this example, the best candidate would be calculating interest on `Accounts`. Even for a particular type of interest calculation, e.g., compound interest, there may be several implementations. A bridge object can be used to separate the implementation of compound interest from the interface defined in the object (and using the Strategy Pattern).

For "legacy" objects, apply the Adapter Pattern

The Adapter Pattern can be applied to any "old" object to make it conform to a "new" interface, and therefore usable in the new context. In this example, it is very likely that `Account` objects have been defined (or existing legacy applications can be wrappered around an object to make it usable in an OO context). Define an adapter object to make these "legacy" objects conform to the new interface for `Account` and able to participate as part of an `AccountPortfolio`.

For creating objects as part of a family, apply the Abstract Factory, Builder, and Factory Method Patterns
The Abstract Factory and/or the Builder Patterns can be applied to build (create) a set of dependent objects. The difference lies in who is in charge of assembling the parts into a final product of related objects. In Abstract Factory the client is in charge, while the Builder returns a final product. The Factory Method can be used as an alternative, allowing sets of dependent objects to be created from an application perspective. For this example, the `Customer` will define a hierarchy of different kinds of customers that can hold accounts in the `Bank`, and who can open certain kinds of accounts in their account portfolios. Define a `Factory` class hierarchy to generate (create) the products provided by the `Bank`, which in turn create the appropriate combinations of `Customer`, `AccountPortfolio`, and `Account` objects. Either the Abstract Factory or the Factory Method can be used.

For creating similar objects, apply the Prototype Pattern
The Prototype Pattern can be applied to create objects from a prototypical object that allows clones (or copies) to be made. For this example, all `Account` objects belonging to an `AccountPortfolio` can be created from a prototypical `Account` object. Instead of creating `Account` from the class, create an `Account` prototype that is associated with each `AccountPortfolio`. Use this prototype to create `Account` objects for the `AccountPortfolio`.

There you have it, a "pattern" to apply patterns to any class hierarchy. I started with a single `Account` hierarchy, and showed you how to use my "pattern" to apply almost all of the patterns defined in the Book. If you use this approach, you can take an application written with no patterns and easily create an application full of patterns! I guarantee recognition from everyone around you, and fame and fortune.

THE DARK SIDE
Unfortunately, there is a "dark" side, which is the reason I am confessing my abuses to you. At first, I was content to wallow in the prestige, fame, and fortune that this abuse of patterns was getting me. Then I started reflecting on the systems that I had built, and realized that I didn't know if they were better or worse off after I was through with them. The Bank (that I mentioned above) was very happy when I was done "patternizing" their system. However, very soon after I was done, they called me because their developers were not able to make an easy change to the system. It was easy for me to get out of that problem, because I was able to point out that a) their developers needed time to mature into "pattern" thinking and using, and b) the problem that they identified could be resolved by introducing some more patterns into the application. They were convinced and hired me again to make the necessary changes and to train their developers! My God, will this madness ever end?

I have become more and more depressed as my "pattern-ploitation" of applications increases. I realize that I have several major problems that I cannot resolve with patterns. How do I know when a pattern is really applicable and will have the needed impact on the application? How do I know that the application of the pattern has made the application more reusable? Will the application become "better" after the patterns are applied? How do I define "better" and determine that the application has become "better"?

Can you please help me?

Reference
1. Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*,

Addison-Wesley, Reading, MA, 1994.