

DESIGNING COMPONENT-BASED FRAMEWORKS USING PATTERNS

Framework solutions help create reusable, approachable software architectures.

GRANT LARSEN

AS I WALKED THROUGH THE ENTRANCE GATE I CAUGHT MY first glimpse of the structure. It appeared to be simple with symmetrical lines and was surrounded by gardens and pathways that made it very approachable. As I got closer to the Taj Mahal I was impressed by the depth and detail of its surface and its ability to reflect light through its many gems, giving it the ability to adapt to its changing environment.

Today we seek many of these same architectural qualities in software. We strive to have well-formed lines and boundaries in the software architecture with known interfaces and yet to be simple enough to be approachable. Add to that the ability to adapt to changing environments and implementations, and we would have an architecture that possesses many of the “-ilities” corporations seek today.

Designing systems using components and proven solutions elevates the abstractions at which engineers work. Productivity and quality are the two main drivers this approach brings. Productivity will be positively affected by using larger abstractions for analysis, design, and development. Quality will be positively affected by reusing known, proven solutions and the components that implement them.

There are several approaches for providing automation to meet a business need, including a) turnkey solutions, b) framework solutions, and c) custom development solutions. This article will focus on framework solutions.

I will discuss and illustrate the use of the UML for designing component-based frameworks using proven solutions (patterns). To do this, I present a brief description of the interrelationships of patterns, frameworks, and components followed by a

collection of models and code creating and extending a framework.

Patterns of Productivity

Designing systems using proven solutions such as patterns and frameworks improves the productivity of software engineers. Achieving productivity and artifact improvement affects several stakeholders in the organization, directly and indirectly. The list of stakeholders tends to include: software architects, developers, IT managers, business managers, and ultimately the customer.

Software patterns have been strongly influenced by the architect Christopher Alexander. He often describes a quality that patterns have, but this quality has no name. This quality makes the pattern useful, understandable, relevant, and living.

Within the software industry several pundits have articulated and presented initial reusable artifacts and patterns. Some of the early pattern work, mostly concerning human interface patterns, was done by Ward Cunningham and Kent Beck in 1987. Several years later more discussions at conferences such as OOPSLA were held with individuals such as Erich Gamma and Richard Helm. In the early 1990s the “Gang of Four” organized themselves and began

FRAMEWORKS PATTERNS in the UML

their efforts articulating and classifying their collective design experiences. Other experts have been closely involved with these efforts such as Desmond D'Souza, Norm Kerth, Peter Coad, and Bruce Anderson.

Patterns and frameworks provide the medium for reusing proven solutions and elevating the abstractions whereby engineers communicate and produce solutions. Components provide the packaging for the patterns and frameworks. So components provide the pathways for making content, pattern, or framework reuse possible.

Grady Booch defines components as “a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces” [1]. Components are more approachable and easier for other engineers to reuse if additional metadata is captured on them. This metadata can include answers to questions such as: what benefits does the component provide, what are its liabilities, what are its dependencies, and so forth.

From the original UML submission document to the OMG the following definition is provided for patterns: “Pattern is a synonym for a template collaboration that describes the structure of a design pattern. Design patterns involve many nonstruc-

tural aspects, such as heuristics for their use and lists of advantages and disadvantages. Such aspects are not modeled by the UML and may be represented as text or tables” [5]. A more casual definition is: patterns are descriptions of and solutions to recurring problems.

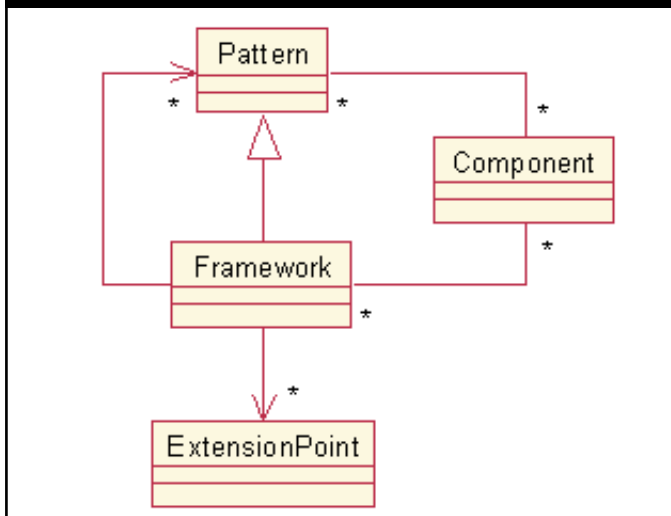
Frameworks are closely related to patterns and reside as a stereotype in the UML. Grady Booch defines a framework as: “An architectural pattern that provides an extensible template for applications within a domain” [1]. The conceptual model shown in Figure 1 elevates frameworks as first class citizens.

The model in Figure 1 illustrates that frameworks are also solutions to recurring problems, but they are more than that. Frameworks have a collection of extension points whereby their behavior can be augmented. Furthermore, a framework can have any number of patterns implemented within it.

Extension points are another way of describing the “slots, knobs, and dials that you must adjust in order to adapt the framework to your context” [1]. Extension points describe where and how the framework is extended and customized. Extension points will be illustrated in the Observable Party Account framework later in this article.

When applying a pattern typically you map the

Figure 1. Pattern, Framework, Component metamodel.



roles/participants of the pattern into the actual model elements that will fulfill that role. Any given model element may participate in multiple roles at any given point in time. Frameworks can be applied in a similar way. However, frameworks also can be extended and modified to augment their default behavior. As shown in Figure 1, a component can implement many patterns and frameworks, and a pattern or framework can be implemented across many components.

Architectural patterns or frameworks provide descriptions of the software architecture. Typically frameworks provide key decision points in the software to help comprise the architecture. In the UML there are several ways to view software architecture, often referred to as the five views of software architecture. I will illustrate only the logical view of the architecture for the Observable Party Account framework.

Frameworks may achieve a quality, with a name, that includes both horizontal and vertical characteristics—perhaps you can call this quality “hertical.” Specifically these frameworks possess enough horizontal characteristics to be useful across multiple contexts, while providing sufficient vertical qualities to deliver some domain knowledge for a given industry.

When developing a system continually look for opportunities to create elements of it as extensible frameworks. First there is the activity of creating the framework and secondly the activity of packaging the framework. However, properly selecting the right frameworks to build is a key activity. Designing, building, and packaging extensible frameworks adds to the development time and therefore to the cost. So consider the downstream benefit of what you are about to produce.

There are several tasks to design and implement a

framework, these include:

1. Identify the specific domain(s) for which the framework will apply;
2. Determine the key use cases the framework will support;
3. Identify the known set of actors interacting with the framework;
4. Determine existing patterns or other proven solutions to aid framework development;
5. Design the key interfaces and components of the framework; mapping roles and actors into interfaces;
6. Provide a default implementation of the interfaces in the framework;
7. Describe and document the extension points of the framework; and
8. Create the test cases and plans for the framework.

The preceding tasks for designing and implementing a framework produce several artifacts. Here are some of those artifacts to consider for packaging a framework, making it possible for other engineers to use:

- a) Architecture document
- b) Components implementing the framework
- c) Defined extension points
- d) Framework characteristics
- e) Framework code
- f) Framework quality measurements
- g) Requirements document and database
- h) Models
- i) Snapshots
- j) Test cases
- k) Test data and test drivers

I won’t cover any more detail on the artifacts listed here. However, though some artifacts are more valuable than others, it is important that the majority of these artifacts are provided for the engineers who will be using your framework to insure consistency, quality, and usability. This is often one of the greatest disappointments in reuse: the inability to *easily* pick up, understand, and apply the “reusable” item. Framework, component, and pattern reuse increases when the reusable “thing” is approachable, predictable, and easy to understand.

Designing and Building the Framework

The following example illustrates the process of using the UML to design a component-based framework using existing patterns. Since a framework can

have many patterns implemented within it, a collection of patterns from various authors including Gamma et al., Hay, and Fowler [2–4] are modeled as interoperable components to form the Observable Party Account framework.

Only a few of the models are shown in this example; no other artifacts are illustrated. However, the major activities of designing and building a framework are identified throughout the example.

The Observable Party Account framework will manage the elements of posting and processing financial transactions for a variety of business entities and accounts. Furthermore, the framework will provide fundamental business object relationships and will provide a mechanism for extension and customization. The framework may be instantiated for key elements of a billing system or other related systems. Here I have performed task 1 listed previously: *Identify the specific domain(s) for which the framework will apply.*

Not all use cases of this framework are covered. However, for the purposes of this example the “happy day” scenarios of the following use cases are discussed:

- Retrieve Account Balance
- Monitor Account Activity

This addresses task 2: *Determine the key use cases the framework will support.*

The use cases are influenced by the following actors:

- Customer
- Customer Service Rep

This addresses task 3: *Identify the known set of actors interacting with the framework.*

From David Hay I will use elements of the Party [4] pattern; from Martin Fowler I will use elements of the Account [2] pattern and from Gamma et al. I will use elements of the Observer [3] pattern. Each of the patterns’ static structures will be illustrated with inter-

Figure 2a. Interfaces on Hay Party pattern.

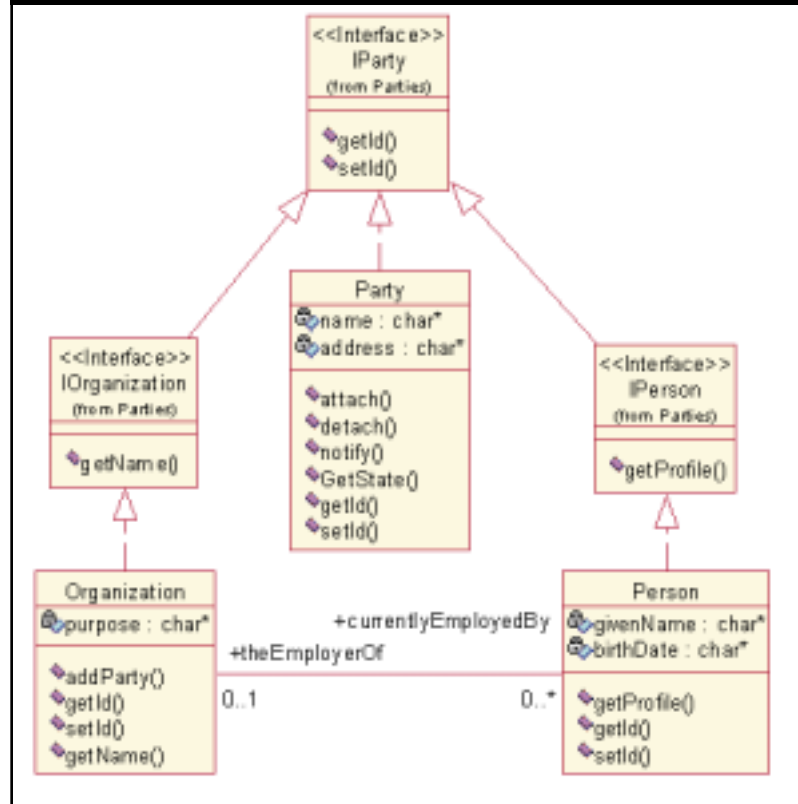


Figure 2b. Interfaces on Fowler Account pattern.

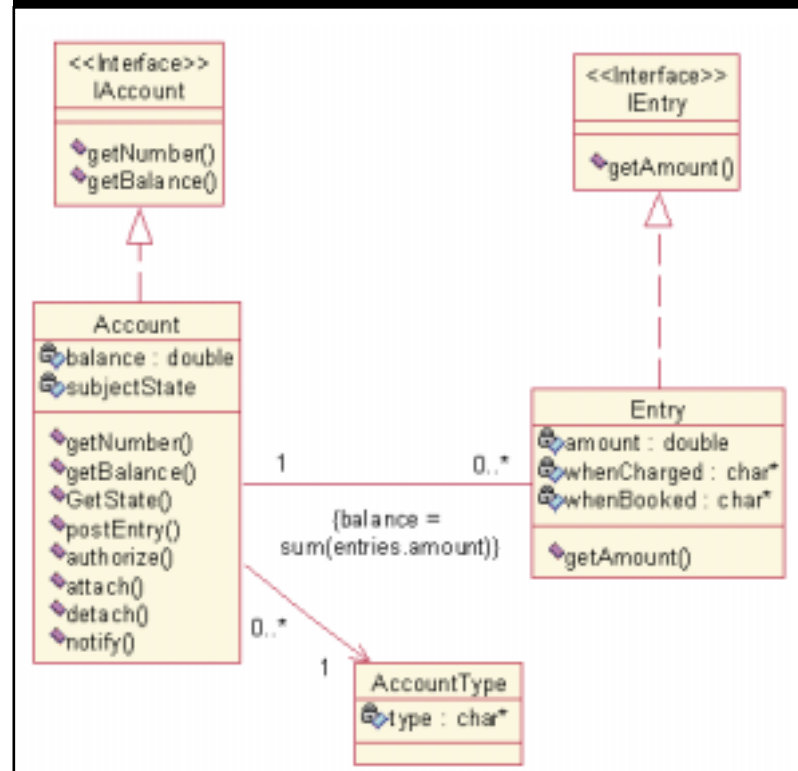
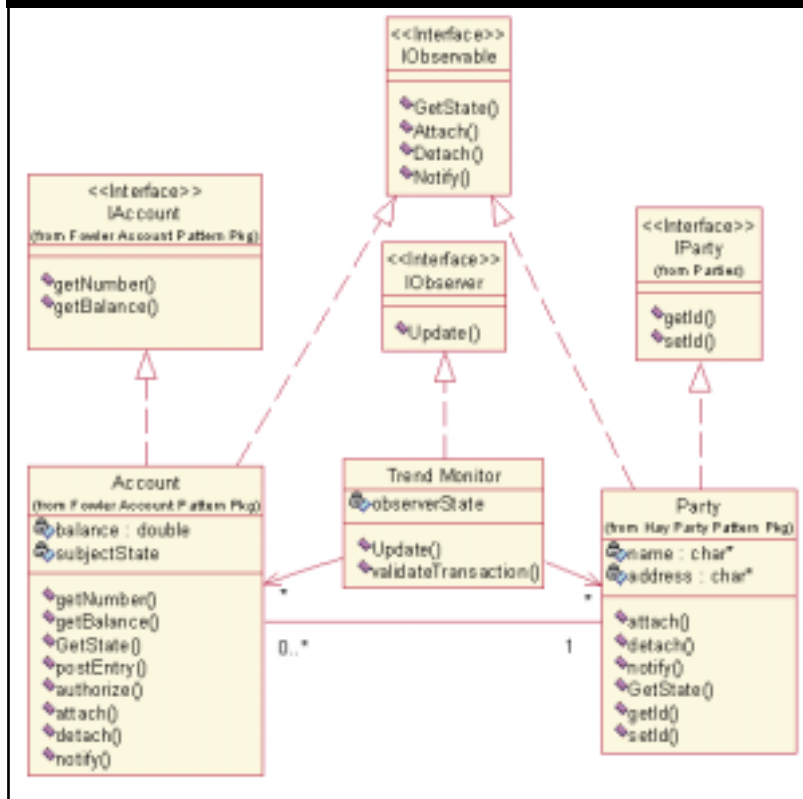


Figure 3. Interfaces for Gamma Observer pattern.



faces and components implementing them. These components and the defined extension points create the framework. For reasons of space, the dynamic semantics of the framework are not illustrated, however, any serious framework development effort relies heavily upon that view.

This addresses task 4: *Determine existing patterns or other proven solutions to aid framework development.*

In its simplest form the Hay Party Pattern has three classes, Party, Organization, and Person. Organization and Person inherit from Party. Organization has an association to many Persons. In Figure 2a the interfaces comprising the pattern and a simple implementation with classes are illustrated. The important relationships to note are the UML realization relationships, which are all of the relationships that point to an «Interface». Party, Organization, and Person each realize their respective interfaces. IPerson and IOrganization inherit from IParty. I chose these interfaces because each represents key roles that many entities could play at any given point in time.

This addresses task 5: *Design the key interfaces and components of the framework; mapping roles and actors into interfaces.* This activity is performed several times for each of the components in the example described here.

The pattern is implemented as the Parties compo-

nent (see Figure 5a) containing the classes implementing these interfaces. This component will collaborate with other components (which implement other patterns) to provide the basic capabilities of the framework.

Figure 2b illustrates some basic classes and relationships of a modified view of the Fowler Account pattern. The Account is of one AccountType at any given point in time and maintains a balance across a collection of related entries.

The pattern in Figure 2b is implemented as the Accounts component (see Figure 5a). This component will collaborate with other components (which implement other patterns) to provide the basic capabilities of the framework.

One other requirement is that the framework needs to provide the capabilities for an account's activity to be monitored. I have used the Gamma Observer pattern to declare which classes will be observed, and which will do the observing. The partici-

pants or roles of the Observer pattern are mapped into the existing classes of the framework and some of the framework's classes will realize/implement the IObservable and IObservable interfaces shown here. Classes implementing the IObservable interface can have IObservers attached to them for notification and update.

In Figure 3 the Trend Monitor class plays the role of the Concrete Observer, and Account and Party classes play the role of Concrete Subject from the Gamma Observer pattern. I decided to have Account and Party realize/implement the IObservable interface. I could have inherited the IObservable interface from the IAccount and IParty interfaces. However I opted for looser coupling and did not feel convinced that the notion of being "observable" should tie so closely to the semantics of *all* accounts and parties and any other classes that implemented the IAccount and IParty interfaces.

This decision has some programmatic implications—causing the developer to write a few extra keywords on the implementation class.

```

public class Account implements IAccount, IObservable
{
    ...
}
    
```

However, this decision does provide greater flexibility and potential reuse by not enforcing observable semantics to be an innate part of the interfaces of the business objects. I did not implement this pattern in its own component, but rather had the classes and interfaces of existing components play the various roles of the pattern.

Figure 4 illustrates all of the interfaces and classes comprising the framework. The framework's component interfaces are above the blue line. The implementation of those interfaces, which comprise the basic behavior of the framework, are below the blue line.

This addresses task 6: *Provide a default implementation of the interfaces in the framework.*

Frameworks may be delivered purely as interface specifications and some implementation-less components. This approach may be used if the client of the framework has unique implementation requirements. However, most frameworks provide basic implementation.

There are several techniques for extending the behavior of frameworks, including:

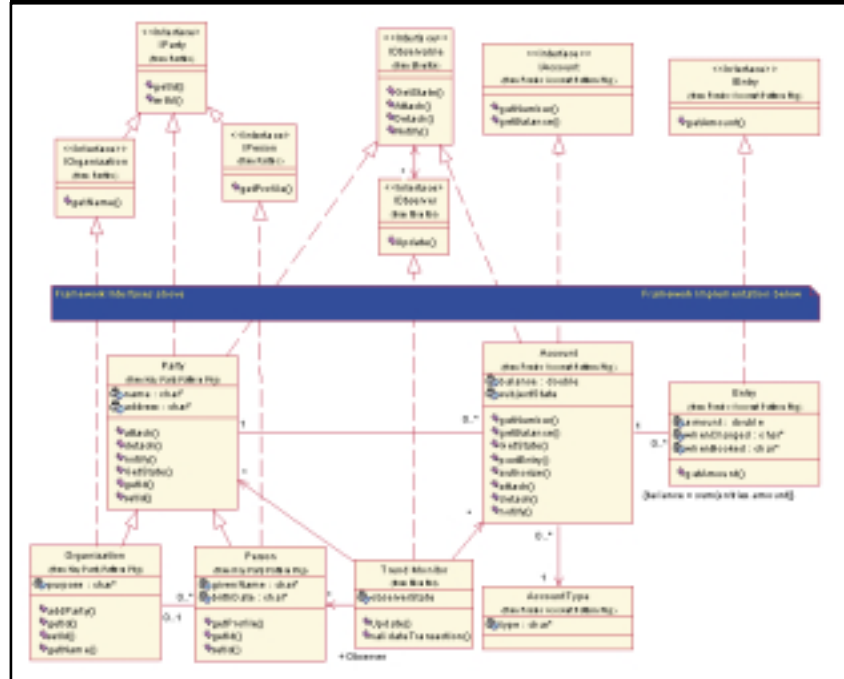
- Implementing an interface in the framework directly, meaning bypassing provided implementation;
- Inheriting from a class in the framework and augmenting the behavior; and
- Aggregating a framework class and augmenting the behavior.

Extending a framework using interfaces. The default behavior of a framework may be circumvented entirely by providing your own implementation. Using this approach, you may extend this framework with a SavingsAccount class as shown here:

```
public class SavingsAccount implements IAccount
{
...
}
```

Extending a framework using inheritance. This is perhaps the simplest approach to extending and customizing a framework. You may introduce a new Savings Account class that can participate in the

Figure 4. Framework classes and interfaces.



framework by merely deriving from the framework's Account class. Using this approach you may extend this framework as shown here:

```
public class SavingsAccount extends Account
{
...
}
```

Extending a framework using aggregation. This approach can be the most loosely coupled approach to augmenting a framework. It uses delegation and composition to extend the framework by having a class contain one of the framework's classes. For instance you may create a Savings Account class that contains an instance of the framework's Account class. However, this approach may remove your extension from participating in event loops or other interfaces in the framework. Using this approach you may extend this framework as shown here:

```
import Account;
public class SavingsAccount
{
private:
    Account acct = null;
}
```

The options for extending a framework are largely dependent on the form in which it is delivered to you. If the framework is a black box component frame-

Figure 5a. Framework components and interfaces.

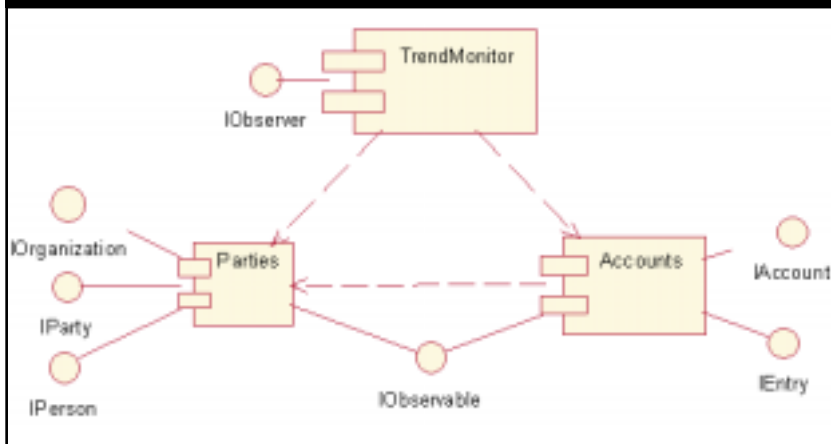
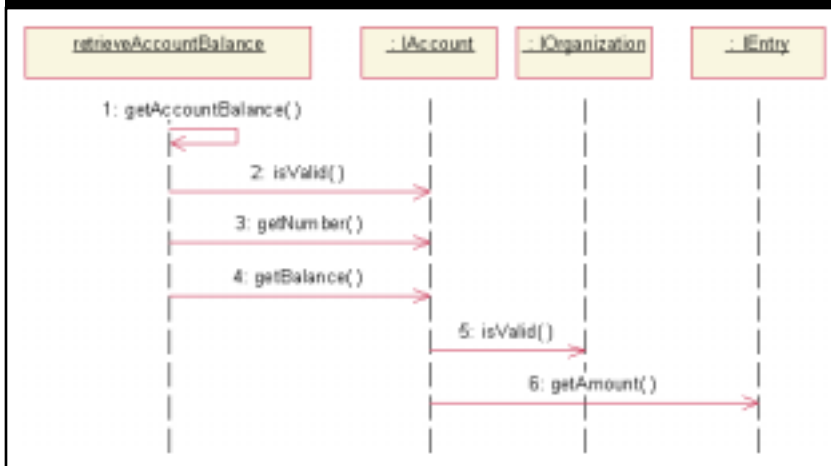


Figure 5b. Framework Retrieve Account Balance scenario.



work (no source code, binary code only) then the options for extending the framework are constrained to implementing the interfaces. However, if the framework is delivered as a white box component framework (source code), then source code level extensions are possible using techniques such as inheritance and aggregation.

This addresses task 7: *Describe and document the extension points of the framework.* I will skip the test cases referenced in task 8 for the purposes of this example.

Figure 5a illustrates the framework's components and their interfaces. The Trend Monitor component observes the activities of the Parties and Accounts components. The Accounts component provides the ability to manipulate account balances for a given party. The components implementing the patterns previously illustrated collaborate to realize the use cases and scenarios of the framework. The Retrieve Account Balance scenario is illustrated here. Figure 5b

illustrates a sequence diagram showing the interfaces of the framework's components retrieving an account's balance.

Using the Framework

Now that the framework has been designed, built, and extended, I will illustrate the usage of the framework. To do so I will use the inheritance form of extension on the framework for a specific domain. Suppose I need to create a billing system that generates marketing literature based on account activity for commercial organizations. This marketing literature needs to be generated as the bills are prepared. To do this, I can extend the Observable Party Account framework and create two new classes, CommercialOrg and LiteratureMonitor.

```

class CommercialOrg extends Organization
{
    // As the Accounts for this Party-
    // derived class are evaluated, which
    // comes as part of this framework,
    // invoke the getBalance() method
    // for each Account—if the balance
    // is over, say, $5000 then invoke
    
```

```

    // the notify() method on any attached monitor
    // observer objects.
    ...
}

```

```

class LiteratureMonitor extends TrendMonitor
{
    // Implement the update() method in this class to
    // generate the appropriate marketing literature based
    // on the level of Account activity for a given
    // CommercialOrg.
    ...
}


```

Using these two simple extension points of the framework I can reuse the existing framework while adding new features for the given application. The Observable Party Account framework in this example briefly illustrates some of the techniques for designing and building component-based frameworks through applying reusable patterns and modeling with the UML.

Conclusion

As I approached the Taj Mahal, I looked closely at its surface and discovered that it was laden with jewels. I was impressed with its symmetry from a distance and its ability to abstract the details while providing intimate views up close. Its demeanor adjusted according to the light, time of day, and weather.

Modeling frameworks and components with the UML provides the same benefits. It can give the high level views while providing the robustness and rigor for capturing the details up close. Though at times modeling can be tedious, the UML provides the medium for managing complexity and designing reusable, customizable frameworks.

Approaching your system's design using these techniques increases the likelihood of downstream reuse, increased productivity and return on investment, qualities with a name that provide value to most stakeholders. 

REFERENCES

1. Booch, G. *The Unified Modeling Language User Guide*. Addison Wesley, Reading, MA, 1998.
2. Fowler, M. *Analysis Patterns: Reusable Object Models*. Addison Wesley, Reading, MA, 1996.
3. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1995.
4. Hay, D. *Data Model Patterns: Conventions of Thought*. Dorset House, NY, 1995.
5. Rational Software and UML Partners. *UML Semantics and Appendices*, November 19, 1997.

GRANT LARSEN (glarsen@blueprint-technologies.com) is Director of Development with Blueprint Technologies in Englewood, CO.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

© 1999 ACM 0002-0782/99/1000 \$5.00