The what, why, and how of a subsystem

by Fredrik Ferm
Software Engineering Specialist
Rational Software, Sweden
IBM Software Group

I could easily have titled this article "Component-based development using subsystems," because many authors who write about component-based development are simply describing how to divide the system into replaceable units. They call these units components, and in UML they are called subsystems; the two are equivalent. Although the UML (Unified Modeling Language) concept of a subsystem is very useful for design modeling, its use is still not universal, in part because many people do not fully understand how to use subsystems and the benefits of doing so.

Many of the students who take my OOAD (Object Oriented Analysis and Design) classes come from an implementation background. I have found that they can more easily understand subsystems by exploring implementation details rather than treating subsystems only as a design concept. Therefore, this article will provide my views on subsystems, focusing on why they are beneficial and how to use them.

Simplified view of a subsystem

This section briefly describes my view of a subsystem in simple terms; see the Appendix to learn more about complications relating to these simple terms.

Informally speaking, a subsystem is a cross between a package and a
class. It behaves like a **package** in that it **groups other model elements**, but it also behaves like a **class** because it has a **specified behavior**.

To say that a subsystem behaves like a package implies that it has an inside view and an outside view. **Inside** the subsystem are a number of other model elements (e.g., classes) that collaborate to fulfill the behavior for which the subsystem is responsible. From the **outside** of the subsystem these classes are not necessarily visible; in fact, it is a very good idea to make them invisible. From the outside, inner details of the subsystem are not interesting, and it is best to treat the subsystem as a single unit.

To say that the subsystem behaves like a class means that it has **behavior**, which is specified through one or more interfaces. This means the subsystem itself can be asked to perform the operations in the interface(s), and when that happens the elements inside the subsystem will collaborate to fulfill that behavior. The subsystem client (i.e., the model element that asked the subsystem to perform the behavior) will not see this collaboration; instead, it will appear to the client as if the subsystem itself performed the operation.

The difference between a subsystem and a package is that, for a package, a client asks **some element inside the package** to fulfill a behavior; for a subsystem, a client asks **the subsystem itself** to fulfill the behavior.

A subsystem is a design time concept, but we can extend the concept of a subsystem into runtime. To do this, we need a mechanism that provides an instance of a subsystem interface to clients when they need it. We will present ideas for such a mechanism later on.

### Modeling subsystems

As we noted above, subsystems have an external view (what is known to the outside world) and an internal view (what is known only to the subsystem).

#### External view

In UML, a subsystem is modeled as a package with the stereotype **subsystem**. The operations of the subsystem are shown in the one or more interface(s) that the subsystem realizes.

A useful convention is to have interface names start with a capital "I." Also, if there is only one interface on the subsystem, it can be named `IsubsystemName`, as in Figure 1. If these interfaces were implemented directly in a programming language, their names would also have to comply with any constraints imposed by the language.
Subsystems and their interfaces can be shown graphically in one of two forms, as shown in Figure 1 (the left and right parts of the figure show the same information). The representation on the left shows the canonical form; the interface looks like a stereotyped class, and the realization relationship between the subsystem and the interface is represented as a true realization arrow. The representation on the right is in the elided form; the interface is shown as an icon that is sometimes called "lollipop notation," and the realization relationship is shown as a solid line.

**Internal view**

Inside the subsystem are a number of model elements that collaborate to fulfill the behavior of the subsystem's interface. A useful design pattern here is the Façade pattern\(^1\), which manifests itself in one class\(^2\) that is used as an entry point into the subsystem. It is this class that implements the interface of the subsystem and delegates the behavior to other elements inside the subsystem. This class could be modeled as a class that is stereotyped with façade, as shown in Figure 2.

**Figure 2: Internal view of a subsystem**

Figure 2 shows the facade class and some other classes that reside inside the subsystem. These classes collaborate to fulfill the behavior of the operations in the interface of the subsystem. In this example, the class MySubsystemFacade plays the role of the facade. Note that this follows a good convention: Name this class with the same name as the subsystem itself plus the word Facade. If the subsystem has more than one interface, name it as the interface it implements (minus the "I") plus the word
Facade. Also note that the classes AHelper and AnotherHelper are simply examples of classes with which the facade class collaborates in realizing the subsystem’s behavior.

**Encapsulation**

The main goal of a subsystem is to encapsulate behavior. When the subsystem’s clients (which are outside the subsystem) request the subsystem to perform a service, they should not be connected directly to the implementation of that service (which is inside the subsystem). Instead, the clients should access the subsystem only through an interface that is also on the outside. Implementation of the behavior should be totally encapsulated from these clients. One way to enforce the rule that clients should connect through the interface is to make all classes inside the subsystem invisible to outside classes. If we extend this rule to implementation -- in other words, using package visibility in Java or implementation visibility in C#, a compile time check of the encapsulation will be performed.

**Why use subsystems?**

The major benefits of using subsystems to design a system fall into two categories, which we will describe below.

**Architectural benefits**

Subsystems help architects greatly in realizing many of a software system’s architectural qualities:

- Insulation from change
- Replaceable implementation
- Dynamic replacement
- Abstraction
- Reuse

We will discuss these qualities in the following subsections.

**Insulation from change**

Insulation from change is a consequence of encapsulation. If we manage to completely encapsulate a behavior, the rest of the system -- that is, the set of subsystem clients -- will not be affected by changes to the subsystem internals. Of course, this will be true only if the interface remains unchanged.

For this reason, architects often use subsystems to interface with an external system or a COTS (Commercial Off-The-Shelf) component, because a subsystem can insulate the new system from changes in the interfacing external system or COTS component. Figure 3 shows this
Figure 3: Subsystem as a proxy for an external system

If a change is made to the External System shown in Figure 3, that change will -- like any other change -- propagate backwards through dependencies or other relationships. This means the change will propagate to the inside of the subsystem External System Encapsulator, which may also have to be changed.

If this subsystem were instead a package that clients could access directly, these changes would propagate further, to the clients, which might have to be changed, and might then propagate more changes within the system. In the worst case, the whole system would be affected by a single change to the external system.

However, because the External System Encapsulator is a subsystem, the clients do not depend directly on it but only on its interfaces; so the propagation of changes stops inside the subsystem and does not travel any further into the system. That is because the interface is not dependent on the subsystem -- instead, the subsystem depends on the interface. Remember that changes propagate backwards through relationships, but in our example there are no relationships coming into the subsystem that would make the changes propagate out from it.

As we have seen, a huge benefit of using subsystems is that they insulate the rest of the system from changes in interfacing external systems -- changes over which the architect of the new system may have little or no control.

Replaceable implementation

If no subsystem clients depend directly on the subsystem internals, it will be easy to change those internals or even to replace the entire subsystem implementation. None of the clients will be affected by such a change or replacement, provided the interface remains unchanged.
The technique of replacing a subsystem implementation with another implementation of the same interface(s), as shown in Figure 4, can be used to upgrade a part of the system, change algorithms, react to changes in external systems, and so on.

![Diagram showing subsystem implementation replacement](image)

**Figure 4: Replacing a subsystem Implementation with another subsystem Implementation**

A consequence of this replaceability is that the subsystem interface has to be defined outside of the subsystem. If it were defined inside, it would be a part of the subsystem internals, so replacing the subsystem internals could also mean replacing the interface. This would affect all clients—and possibly force changes to those clients— which would violate the replaceability we are trying to achieve. However, as long as the interface is defined outside of the subsystem, it does not need to be replaced in this situation, and the clients will not be affected.

There are further benefits to keeping the interface outside of the subsystem internals: The interface can be controlled separately from the subsystem internals, which prevents inadvertent changes to the interface when only the implementation is supposed to change. Furthermore, multiple implementations of a subsystem can coexist in the design model, and they can share the interface.

If we want clients to depend only on an interface, then we need a mechanism that instantiates the correct subsystem implementation when a client wants to connect to the subsystem. We will explore this mechanism later in this article.

**Dynamic replacement**

If we extend the subsystem concept to implementation—in other words, if we use a subsystem to represent components in the runtime system—we may be able to dynamically replace an implementation at runtime. To do this, we must do the implementation in a language that has dynamic loading, such as Java or C#.

We simply install the new implementation into the running system and
also update the mechanism that retrieves instances of the subsystem’s interface to the clients. Then, any clients requesting an instance of the subsystem interface get an instance of the new implementation instead of the old one. As the clients stop using the old subsystem implementation, they release their old instance, and the system gradually moves to using the new implementation instead of the old one -- without having to do a system restart.

The key to a successful transition is the mechanism for retrieving the interface, which we will discuss later on.

**Abstraction**

The subsystem "hides" its inside details from the outside world. This means that the clients of the subsystem need not worry about the inner workings of how a certain task is performed; their only concern is what task is being performed. Subsystems enable us to raise the level of abstraction in our system; this, in turn, helps us understand the system.

**Reuse**

All of the aforementioned benefits are internal to the system being developed. But designing with subsystems also has broader benefits. Correctly packaging subsystem behavior and functionality aids in reuse between systems. Subsystems contain packaged behavior with a specified interface; because they abstract out internal details, it is easy for other systems to call these subsystems.

**Process benefits**

In addition to the architectural benefits we have just described, subsystems also provide benefits that relate to the software development process and project management, which we will describe below.

**Delay of detailed design**

At the start of a design effort, an architect typically wants to do a coarse-grained, or "rough," organization of the system into smaller building blocks that will collaborate to fulfill system functionality requirements. In detailing the collaboration between these different blocks, the architect defines the responsibilities of each block and the other blocks with which it communicates. When the collaborations between the building blocks have been established, the architect can go on to designing the internals of each block as if the block were a system of its own.

It is easy for architects to model these building blocks as subsystems. The abstraction benefit we discussed earlier enables them to delay doing a detailed design of the inner parts of the subsystem until a later stage. To detail interactions among the parts, it is enough to specify the subsystem interfaces and what other subsystems each of the subsystems may use. Designing with subsystems enables us to use a top-down process; we can show the "big picture" before moving into specific details. This approach is
sketched in Figure 5.

Figure 5: System broken down into subsystems and classes

Depending on the complexity of the system being built, this approach can be used recursively at more than one level. The internals of the top-level subsystems can be made up of lower-level subsystems, which can be made up of even lower-level subsystems, and so on. The process is scalable to just about any level of complexity.  

Parallel development

Once the system has been divided into subsystems, the development of the individual subsystem internals can be split and assigned to different team members, thereby allowing parallel development of the individual subsystems.

Each subsystem is, of course, not an island of its own. It will depend on services provided by other subsystems. But if the subsystems are
developed in parallel, these services may not yet be implemented. Designing the internals of a subsystem requires that interfaces of dependent subsystems be available. But if we also want to implement the design into code and unit test this code either before, or in parallel with, the implementation of the other subsystem, we need to go further. For the unit tests to work, our subsystem has to get some kind of response from the other subsystems that it depends upon.

Therefore, when implementation starts, the first step each subsystem developer should take is to create a "dummy" implementation of all operations in the interfaces of the subsystem. This dummy implementation may not do anything meaningful, but it should at least accept a call to an operation of the subsystem and possibly answer with a default answer. When a developer tries to implement and unit test his or her subsystem, it will then be possible to connect to the outside world (i.e., other subsystems being developed in parallel), and receive some kind of answer. As the development of the subsystems progresses gradually and incrementally, more mature implementations of each subsystem will become available; these will simply replace the prior implementations, as we discussed earlier.

The key to making this technique work is to ensure that the set of subsystems and their interfaces and interdependencies are defined and reasonably stable. This has to be done before any parallel development can start.

How to use and implement a subsystem

Now that we have discussed what subsystems are and why you should use them, let's focus on how to implement and use them. We'll begin by explaining how to model the internals of subsystems. The modeling can be done with a collaboration of modeling elements called subsystem realizations.

Subsystem realizations

When designing the insides of subsystems, our task is to define a set of design elements that collaborate to fulfill a specific behavior. This is very similar to the task of use-case design, so we can use an approach that is analogous to designing use cases. Table 1 shows these analogies.

<p>| Table 1: Analogies between modeling use-case realizations and subsystem realizations |</p>
<table>
<thead>
<tr>
<th>Model element</th>
<th>Use-case design</th>
<th>Subsystem design</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UML Collaboration</strong></td>
<td>Use-case design may use a UML collaboration that is stereotyped as use-case realization.</td>
<td>Subsystem design may use a UML collaboration that is stereotyped as subsystem realization.</td>
</tr>
<tr>
<td><strong>Sequence Diagram and/or Collaboration Diagram</strong></td>
<td>A use-case realization contains at least one sequence diagram and/or collaboration diagram per flow of the use case.</td>
<td>A subsystem realization contains at least one sequence diagram and/or collaboration diagram per interface operation.</td>
</tr>
<tr>
<td><strong>Class Diagram</strong></td>
<td>A use-case realization contains at least one class diagram showing the static structure of all modeling elements involved in the use-case collaboration.</td>
<td>A subsystem realization contains at least one class diagram showing the static structure of all modeling elements involved in the subsystem collaboration.</td>
</tr>
<tr>
<td></td>
<td>This diagram is often called Participants.</td>
<td>This diagram is called Participants.</td>
</tr>
</tbody>
</table>

A use-case realization shows how a set of design elements (from possibly the whole model) performs a specific task together. For interactions involving subsystems, the use-case realization usually represents the subsystem by its interface. However, when doing the subsystem realization, we have a different perspective: We look upon the model from inside the subsystem. Therefore, the subsystem realization is expressed in terms of design elements that reside inside the subsystem. If the subsystem implementation uses services from the outside, the subsystem realization represents these services in terms of design elements from the outside (classes, packages, or interfaces of other subsystems).

The structure of a subsystem and its subsystem realization is shown in Figure 6, which describes a subsystem for communicating with the external bank system for an ATM machine. The interface has two operations: queryPIN and sendTransaction; as we can see, the subsystem realization has two sequence diagrams with these names.
Of course, the subsystem BankSystemInterface should also contain any subsystem-internal classes that are part of the implementation of the operations of the subsystem. Actually, both use-case realizations and subsystem realizations are UML collaborations, possibly stereotyped to use-case realization and subsystem realization. This works if we use IBM Rational XDE™ for modeling, because it supports the use of collaborations. However, if we use IBM Rational Rose®, then we must bend the rules of UML a little, because Rational Rose does not support collaborations. Therefore, a convention is to model these collaborations through use cases instead. We can model the use-case realization and subsystem realization as use cases stereotyped with use-case realization and subsystem realization, respectively.

If we want to show the traceability from the subsystem realization to the interface(s) that it realizes, we can draw a diagram such as Figure 7. The proper UML relation would be a realization arrow (the use-case realization realizes the interface[s]). This works in IBM Rational XDE, but not in IBM Rational Rose; as noted above, we use stereotyped use cases for our subsystem realizations, and UML does not allow us to draw a realization arrow from a use case to an interface. We have to use a plain dependency instead. This is shown in Figure 7, which is the class diagram called "Traceability to interface" in Figure 6.
Since the artifacts involved in use-case design and subsystem design are analogous, our subsystem design process can be similar to the process for designing use cases. The process involves creating sequence -- and/or collaboration diagrams with collaborating objects. The objects in the collaborations are mapped to classes placed in a structure inside the subsystem, and this structure is shown in a "Participants" class diagram. The main difference between subsystem design and use-case design is the focus. Subsystem design looks at a single subsystem, but use-case design looks at a single use case of the complete system.

**Subsystem analysis and design**

IBM Rational Unified Process®, or RUP®, recommends that use-case design be divided into two steps: analysis and then design. Since subsystem artifacts are analogous to use-case artifacts, we could similarly analyze subsystems before we design them. This is not directly supported in RUP (there is no subsystem analysis activity, just a subsystem design activity), but, if needed, the analysis concept could easily be extended to subsystems as well.

We would do *subsystem analysis* for the same reason we would do use-case analysis: to deal with complexities that are not easily handled in a single step. Therefore, subsystem analysis would be most beneficial for large and complex subsystems. As we noted above, this "mini-process" (use-case analysis -> use-case design -> subsystem analysis -> subsystem design) could be extended in as many levels of abstraction as needed.

For use-case analysis, the RUP recommends using three analysis class stereotypes: boundary, control, and entity. By correspondence, when analyzing subsystem internals, we would also use these analysis class stereotypes.

The subsystem facade (the class that directly implements the interface) could play a combined role of boundary and controller. It is a *boundary* in the sense that it is a communication port into the subsystem. It is a *controller* in the sense that it coordinates and delegates behavior to other elements inside the subsystem. Alternatively, the facade could simply play the role of boundary, and another class would be the controller of the
External dependencies

To reuse a subsystem implementation, we must know what that subsystem implementation needs to work properly. This information is available in the design model, but for ease of reuse, it helps if the information is summarized in one place. Therefore, each subsystem package should contain a diagram showing its external dependencies, as shown in Figure 8.

Technically, a subsystem can be dependent on anything, such as simple packages in the system being built, the interfaces of other subsystems, and external systems/subsystems. However, depending on all of these things is not good. To achieve the reuse potential of a subsystem, the subsystem should be dependent on as few things as possible. Moreover, those few things should not be regular packages of the system under development, but rather interfaces of other subsystems. Otherwise, we could not reuse the subsystem without also reusing parts of our system's internal structure. This would mean that the system using our subsystem would be dependent on our internal structure, and, eventually, all systems would become tightly dependent on each other, which would lead to the anti-pattern of stovepipe systems. If a subsystem absolutely needs to depend on a package, we must ensure that the package is very general and globally accessible, because in such cases the package would probably become part of the system that is using the subsystem anyway.

Figure 8: External dependencies of a subsystem

The diagram in Figure 8 lists the external dependencies of the subsystem MySubsystem. As we can see, this subsystem is dependent on the following things:

- A subsystem that is external to the system being developed
(External Subsystem).
This is an acceptable dependency.

- A subsystem that is *internal* to the system being developed (Other Internal Subsystem).
  This is an acceptable dependency.

- A *globally accessible package* (Global Package).
  This is an acceptable dependency as long as the globally accessible package is accessible to all other systems that want to reuse MySubsystem. Otherwise, we should find another solution.

- A *package inside the system being developed* (Internal Package).
  This is not an acceptable dependency, because it prohibits any other system from reusing MySubsystem without also reusing the Internal package.

### Parameters as interfaces

Operations in the subsystem interface may have parameters and/or return values. For a subsystem to be truly reusable, the classes of these parameters should not be internal to the system being developed. If we break this rule, we create a dependency from the subsystem interface and its facade class to some other internal part in our system, which in turn creates a dependency from the subsystem to that internal part of our system. This would force reuse of parts of our system as well as the subsystem, when what we want to achieve is reuse of the subsystem alone.

![Diagram](image)

**Figure 9: Billing example before reuse**

Consider the example shown in Figure 9: Our system has a subsystem that takes care of billing (Billing System). This subsystem has an interface with an operation that needs a customer as a parameter. The
value of the parameter indicates to what customer the bill should be submitted. The customer parameter is of a type that is defined inside our system (Customer). The client(s) of the subsystem (Subsystem Client) is dependent on the interface (since it calls operations on it) and on the Customer class, because calling the operation involves dealing with a certain customer object.

When the First system was finished and development started on the next system, the development team found they needed the Billing System functionality again. In such a case it makes sense to reuse the Billing System subsystem in the second system also. This situation is shown in Figure 10.

![Figure 10: Billing example after reuse (bad)](image)

In this example, the subsystem has been broken out of the First System. The new client inside the second system is dependent on the interface of the subsystem, but for the same reasons stated above, it is also dependent on the Customer class. This is not optimal, since it forces the second system to be dependent on the first system if it wants to reuse the subsystem.

To alleviate this problem I propose the following solution: Create a new interface that represents the parameter type. This interface should be globally accessible in the same way as the subsystem interface; any system that wants to use the subsystem must implement this interface with its own implementing class. The internals of the subsystem will know nothing about the implementing classes; they will only see that the parameters are of the interface type. Therefore, the subsystem will not be dependent on any system that wants to use it -- it will only be dependent
on its set of interfaces. This solution for our example is shown in Figure 11.

![Figure 11: Billing example after reuse (better)](image)

Note that this figure introduces a new interface (ICustomer) to represent the customer type. The implementation of the subsystem uses only this interface type instead of any of the implementing types. Any system that wants to reuse the subsystem must implement its own customer type, which should realize the interface ICustomer. This ensures that the two systems are independent of one other; they depend only on the type interface and the subsystem interface.

To go ahead with this solution, we have to extend the set of interfaces that "belong" to the subsystem. Traditionally, we say that only interfaces such as IBillingSystem "belong" to the subsystem, but I think we should extend that "belonging" to interfaces such as ICustomer. Then, we can divide the set of interfaces for the subsystem into two groups:

- **Traditional subsystem interfaces**, which specify the operations of the interface. I propose that we call these "Operational" interfaces. In Figure 11, IBillingSystem belongs to this group.

- **New subsystem interfaces**, which represent parameters, return types, and so on, from the operational interfaces. As this group represents types that are handled by the subsystem, I propose that we call it the "Type" interfaces group. In Figure 11, ICustomer belongs to this group.

**Mechanism for retrieving interfaces**

When I talk about subsystems to my analysis and design students, they inevitably ask: "But how will I be able to access the subsystem? There is only an interface. Where will I get that from?"
The simplest solution might be to have the subsystem clients instantiate the subsystem facade when they need to access the subsystem. In Java or C#, this means that they use the `new` operator to instantiate a new object. Unfortunately, however, this is undesirable; it would mean that the subsystem clients are "glued" to a particular implementation of the subsystem, because they have to know the fully qualified name of the subsystem facade and also any parameters that it needs for instantiation. If there is an implementation change, all clients may have to be updated; this would negate the encapsulation that the subsystem is there to provide.

A better idea would be to have a mechanism that instantiates the subsystem facade class and delivers this instance to the subsystem clients. Before delivery, the instance would be typecast to the type of the subsystem interface, so that no client would know anything about the internal parts of the subsystem. This would decouple all clients from the subsystem internals.

**Realizing benefits**

The retrieval mechanism is at the heart of the replaceable implementation and dynamic replacement benefits we discussed previously. To realize those benefits, we have to design our retrieval mechanism with the benefits in mind.

We can realize the *replaceable implementation* benefit if the mechanism is configurable in the specific class it instantiates when a client asks for an instance of the facade. To change subsystem implementation, we would simply change the configuration of the mechanism and thereby instantiate another specific class to clients.

We can realize the *dynamic replacement* benefit by making the configuration mentioned above configurable in runtime. As the system is running, we can reconfigure the retrieval mechanism and thereby change what specific class of object is returned to the clients from this point in time and forward.

It is important to note that the mechanism cannot be a part of the subsystem itself; instead, it has to be defined outside of the subsystem. If it were inside, the clients would then be dependent upon something internal to the subsystem, which would violate the encapsulation that the subsystem is there to provide.

We could either have one retrieval mechanism per subsystem or group the subsystems together and use a single retrieval mechanism. If we choose the second alternative, we could also make a globally accessible mechanism that all of our systems and subsystems use, and which could then be reused in all of the organization's projects.

**Relaxed encapsulation**

When we discussed encapsulation earlier, we said that all elements in the subsystem should not be accessible from the outside world, and that
would mean using package visibility in Java or internal visibility in C#. The problem with this approach, however, is that the retrieval mechanism needs access to the facade class in order to retrieve an instance of it. If the facade is visible only inside the subsystem, the mechanism will not be able to access it, because the mechanism is not part of the subsystem -- it is outside of it.

To get around this, we must relax the encapsulation rule a little: We must make the facade class public and therefore available to the outside world. When doing this, we must take great care so that no outside client (except the mechanism) instantiates the facade directly. Instead, they should go through the mechanism. If a client (except the mechanism) instantiates a facade directly, that would obviate the whole idea of an encapsulated subsystem, since the client would be directly dependent on the subsystem internals instead of just being dependent on the external interface(s).

Therefore, we have to set up a design and programming restriction that prohibits anything (except the mechanism) from directly accessing subsystem proxies. This restriction will not be checked at compile time, so we have to rely on the judgment of our designers and programmers and perhaps also on code reviews, IBM Rational Rose/IBM Rational XDE scripts and/or inspections, and so on, to make sure that the restriction is not breeched.

**Conclusion**

Subsystems represent a modeling concept that has a great value: They enable us to encapsulate behavior, which in turn allows for replaceable implementations, a raised level of abstraction, simpler reuse, parallel development, and so on. In the world of software development, we need to do a better job of spreading the word about these substantial benefits of subsystem architectures.

To help my students understand how to design with and implement subsystems within their own systems, I present some of the usage strategies discussed in this article, including retrieval mechanisms and type interfaces. My experience is that explaining these concepts greatly increases their understanding of the subsystem concept and encourages them to use subsystems in their architectures.

**References**

**Books**

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Articles


Acknowledgments

I give my thanks to Björn Reinius for discussing these concepts with me as well as reviewing a large part of this article. I also give my thanks to Stefan Bergström, who inspired me to write about these subjects. Furthermore, I would like to thank Bruce MacIsaac, Bran Selic, and Peter Eeles for reviewing the article, and also Catherine Southwood and Marlene Ellin for copyediting it.

Notes

1 The Façade pattern is described in a number of publications, including Erich Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

2 Actually, the Facade pattern could be implemented by using more than one facade class -- objects of different classes would implement different parts of the interface. The rest of this article will use a single class for simplicity.

3 Provided the subsystem interface remains unchanged.

4 This process idea is based on the ideas presented in Maria Ericsson, "Developing Large-Scale Systems with the Rational Unified Process." Rational Whitepaper, 2000.

5 Actually, the UML specifies a way to represent subsystem realizations in a special compartment of the subsystem symbol. But I suggest this other approach for two reasons: First, there is no tool support for the UML method, and, second, it limits the subsystem interface to only have one realization, and that is not acceptable.

6 This rule should not be taken to its extreme. Doing so would mean that a subsystem isn’t dependent on anything at all, and the only practical way to realize that is to design a single monolithic component, which is not a good idea. Instead, a subsystem should be dependent on a few other well-defined subsystems, each with a special purpose.

7 A possible extension to this idea would be to provide a default implementation of the ICustomer interface, and make this implementation globally accessible (or possibly replace the ICustomer interface with that implementing class). This would make the burden of reusing the subsystem smaller, since it would not force clients to make their own implementation of the interface. But such an extension goes a little outside of the point I am trying to make. The important thing is that the types of the parameters should not be defined inside any of the systems that use the subsystem. Otherwise, there would be dependencies between the systems using the subsystem.

8 For further discussion about this confusion, see Joaquin Miller and Rebecca Wirfs-Brock,
Appendix: Taking a closer look

The subsystem explanation in this article is somewhat simplified. In this Appendix, we explore two complications: first, the problems of mixing design time concepts and runtime concepts, and second, the possibility that the outside and inside views are not necessarily the correct views.

Mixing of design time and runtime concepts

Much of the confusion about what a subsystem really is stems from the fact that the UML definition mixes design time concepts with runtime concepts. It states that a subsystem "Specifies a grouping of elements of which some constitute a specification of the behavior offered by the other contained elements." This definition does not clearly differentiate between groupings in a running system and those in a design model. Groupings in the design model are modeled through packages, but groupings in a running system should be modeled by associations/links. By not making this differentiation, this definition implies that the groupings of elements are similar at design time and runtime, and this is not necessarily true.

Nevertheless, in my view this mixing of design time and runtime concepts is quite beneficial in many cases. Using subsystems gives us an opportunity to organize our design model and running system in a similar way. Although in some cases this is not possible because of implementation language constraints, the topology of the distributed network, and so on, in general, using subsystems can greatly improve the understanding of our design model as well as the understanding of the running system. This means that the complication between design time subsystems and runtime subsystems should not force us to give up the benefits of using subsystems. Instead, I think we should try to bridge this gap, perhaps by using subsystems not only as a design concept, but also as an implementation concept, which is explored a little in this article.

Outside/inside versus specification/implementation

We discussed external and internal views above, but the UML specification does not actually talk about these views; instead, it talks about specification and implementation. Some design model elements are used to specify the behavior, and other (or the same) elements are used to implement this behavior.

In this article, we assume that the subsystem specification is what is seen in the external view -- largely the subsystem's interfaces -- and the implementation is what is seen in the internal view -- the design elements that reside inside the subsystem package.
This assumption works in many cases, but it is somewhat oversimplified because it overlooks the option of specifying a subsystem by building a reference implementation of it. If we build a reference implementation, that implementation would be part of the internal view of the subsystem, but it would also be part of the specification, which would violate the assumption.

Nonetheless, we can still say that the assumption holds. I have found that it makes the concept of subsystems easier to understand, and therefore this article is built on that assumption. If we use reference implementations to specify our subsystems, then the concepts explained in this article can easily be extended.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!