Chapter

22

GRASP: MORE PATTERNS FOR ASSIGNING RESPONSIBILITIES

Luck is the residue of design.

— Branch Rickey

Objectives

- Learn to apply the remaining GRASP patterns.

Introduction

Previously, we explored the application of the first five GRASP patterns:

- Information Expert, Creator, High Cohesion, Low Coupling, and Controller

The final four GRASP patterns are:

- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

Once these have been explained, we will have a rich and shared vocabulary with which to discuss designs. And as some of the "gang-of-four" (GoF) design patterns (such as Strategy and Factory) are also introduced (in subsequent chapters), that vocabulary will grow. A short sentence such as, "I suggest a Strategy generated from a Factory to support Protected Variations and low coupling with respect to **<X>"** communicates lots of information about the design, since pattern names tersely convey a complex design concept.

This chapter introduces the remaining GRASP patterns, a learning aid of fundamental principles by which responsibilities are assigned to objects and objects are designed.

Subsequent chapters introduce other useful patterns and apply them to the development of the second iteration of the NextGen POS application.

1 Polymorphism

Solution When related alternatives or behaviors vary by type (class), assign responsibility for the behavior—using polymorphic operations—to the types for which the behavior varies.[1]

Corollary: Do not test for the type of an object and use conditional logic to perform varying alternatives based on type.

Problem How to handle alternatives based on type? How to create pluggable software components?

Alternatives based on type—Conditional variation is a fundamental theme in programs. If a program is designed using if-then-else or case statement conditional logic, then if a new variation arises, it requires modification of the case logic. This approach makes it difficult to easily extend a program with new variations because changes tend to be required in several places—wherever the conditional logic exists,

Pluggable software components—Viewing components in client-server relationships, how can you replace one server component with another, without affecting the client?

Example In the NextGen POS application, there are multiple external third-party tax calculators that must be supported (such as Tax-Master and Good-As-Gold Tax-Pro); the system needs to be able to integrate with different ones. Each tax calculator has a different interface, and so there is similar but varying behavior to adapt to each of these external fixed interfaces or APIs. One product may support a raw TCP socket protocol, another may offer a SOAP interface, and a third may offer a Java RMI interface.

What objects should be responsible for handling these varying external tax calculator interfaces?

1. **Polymorphism** has several related meanings. In this context, it means "giving the same name to services in different objects" [Coad95] when the services are similar or related. The different object types usually implement a common interface or are related in an implementation hierarchy with a common superclass, but this is language-dependent; for example, dynamic binding languages such as Smalltalk do not require this.

Since the behavior of calculator adaptation varies by the type of calculator, by Polymorphism we should assign the responsibility for adaptation to different calculator (or calculator adapter) objects themselves, implemented with a polymorphic *getTaxes* operation (see Figure 22.1).

These calculator adapter objects are not the external calculators, but rather, local software objects that represent the external calculators, or the adapter for the calculator. By sending a message to the local object, a call will ultimately be made on the external calculator in its native API.

Each *getTaxes* method takes the *Sale* object as a parameter, so that the calculator can analyze the sale. The implementation of each *getTaxes* method will be different: *TaxMasterAdapter* will adapt the request to the API of Tax-Master, and so on.

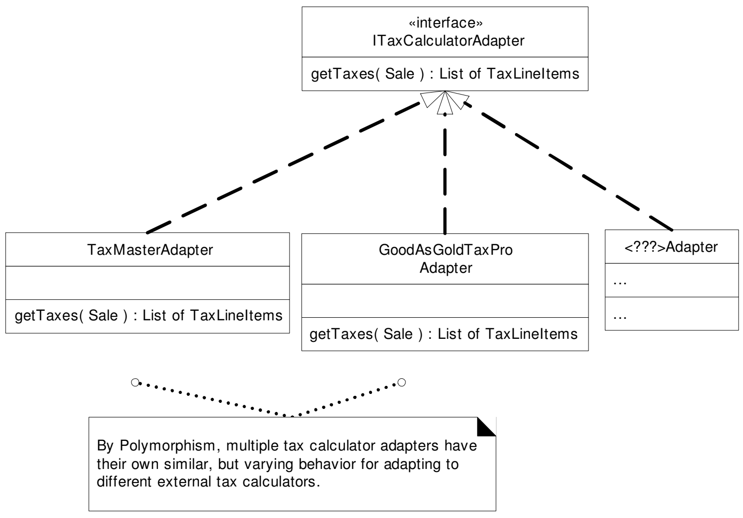

«interface»
ITaxCalculatorAdapter
getTaxes(Sale) : List of TaxLineItems
TaxMasterAdapter
getTaxes(Sale) : List of TaxLineItems
GoodAsGoldTaxPro Adapter
getTaxes(Sale) : List of TaxLineItems
<???>Adapter
...
...
By Polymorphism, multiple tax calculator adapters have their own similar, but varying behavior for adapting to different external tax calculators.

Figure 22.1 Polymorphism in adapting to different external tax calculators.

UML notation—Figure 22.1 introduces some new UML notation for specifying **interfaces** (a descriptor of operations without implementation), interface implementation, and for "collection" return types; Figure 22.2 elaborates. A UML **stereotype** is used to indicate an interface; a stereotype is a mechanism to categorize an element in some way. A stereotype name is surrounded by guillemets symbols, as in «interface». Guillemets are special *single-character* brackets most widely known by their use in French typography to indicate a quote; but to quote Rumbaugh, "the typographically challenged could substitute two angle brackets (« ») if necessary" [RJB99].

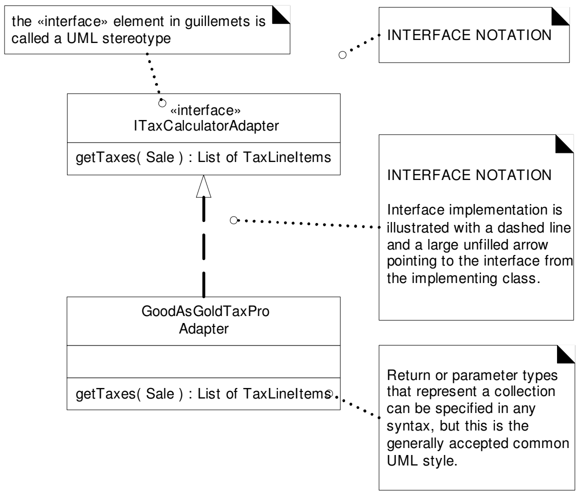

the «interface» element in guillemets is called a UML stereotype
INTERFACE NOTATION
«interface»
ITaxCalculatorAdapter
getTaxes(Sale) : List of TaxLineItems
INTERFACE NOTATION
Interface implementation is illustrated with a dashed line and a large unfilled arrow pointing to the interface from the implementing class.
GoodAsGoldTaxPro Adapter
getTaxes(Sale) : List of TaxLineItems
Return or parameter types that represent a collection can be specified in any syntax, but this is the generally accepted common UML style.

Figure 22.2 UML notation for interfaces and return types.

Polymorphism is a fundamental principle in designing how a system is organized to handle similar variations. A design based on assigning responsibilities by Polymorphism can be easily extended to handle new variations. For example, adding a new calculator adapter class with its own polymorphic *getTaxet* method will have minor impact on the existing design.

Sometimes, developers design systems with interfaces and polymorphism for speculative "future-proofing" against an unknown possible variation. If the variation point is definitely motivated by an immediate or very probable variability then the effort of adding the flexibility through polymorphism is of course rational. But critical evaluation is required, because it is not uncommon to see unnecessary effort being applied to future-proofing a design with polymorphism at variation points that in fact are improbable and will never actually arise. Be realistic about the true likelihood of variability before investing in increased flexibility.

- Extensions required for new variations are easy to add.
- New implementations can be introduced without affecting clients.

- Protected Variations
- A number of popular GoF design patterns [GHJV95], which will be dis cussed in this book rely on polymorphism, including Adapter, Command, Composite, Proxy, State, and Strategy.

Choosing Message, Don't Ask "What Kind?"

Fabrication

Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept—something made up, to support high cohesion, low coupling, and reuse.

Such a class is *a fabrication* of the imagination. Ideally, the responsibilities assigned to this fabrication support high cohesion and low coupling, so that the design of the fabrication is very clean, *or pure*—hence a pure fabrication.

Finally, a pure fabrication implies making something up, which we do when we're desperate!

What object should have the responsibility, when you do not want to violate High Cohesion and Low Coupling, or other goals, but solutions offered by Expert (for example) are not appropriate?

Object-oriented designs are sometimes characterized by implementing as soft- ware classes representations of concepts in the real-world problem domain to lower the representational gap; for example a *Sale* and *Customer* class. However, there are many situations in which assigning responsibilities only to domain layer software classes leads to problems in terms of poor cohesion or coupling, or low reuse potential.

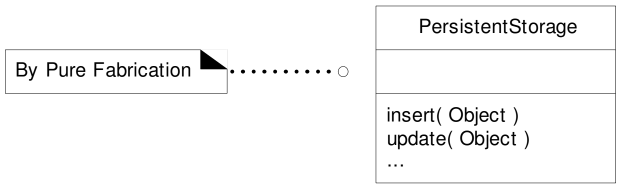
For example, suppose that support is needed to save *Sale* instances in a relational database. By Information Expert, there is some justification to assign this responsibility to the *Sale* class itself, because the sale has the data that needs to be saved. But consider the following implications:

- The task requires a relatively large number of supporting database-oriented operations, none related to the concept of sale-ness, so the *Sale* class becomes incohesive.
- The*Sale* class has to be coupled to the relational database interface (such as JDBC in Java technologies), so its coupling goes up. And the coupling is not even to another domain object, but to a particular kind of database interface.

- Saving objects in a relational database is a very general task for which many classes need support. Placing these responsibilities in the *Sale* class suggests there is going to be poor reuse or lots of duplication in other classes that do the same thing.

Thus, even though *Sale* is a logical candidate by virtue of Information Expert to save itself in a database, it leads to a design with low cohesion, high coupling, and low reuse potential—exactly the kind of desperate situation that calls for making something up.

A reasonable solution is to create a new class that is solely responsible for saving objects in some kind of persistent storage medium, such as a relational database; call it the *PersistentStorage.*[2]' This class is a Pure Fabrication—a figment of the imagination.



Notice the name: *PersistentStorage.* This is an understandable concept, yet the name or concept "persistent storage" is not something one would find in the Domain Model. And if a designer asked a business-person in a store, "Do you work with persistent storage objects?" they would not understand. They understand concepts such as "sale" and "payment." *PersistentStorage* is not a domain concept, but something made up or fabricated for the convenience of the software developer.

This Pure Fabrication solves the following design problems:

- The*Sale* remains well-designed, with high cohesion and low coupling.
- The*PersistentStorage* class is itself relatively cohesive, having the sole pur pose of storing or inserting objects in a persistent storage medium.
- The*PersistentStorage* class is a very generic and reusable object.

Creating a pure fabrication in this example is exactly the situation in which their use is called for—eliminating a bad design based on Expert, with poor cohesion and coupling, with a good design in which there is greater potential for reuse.

Note that, as with all the GRASP patterns, the emphasis is on where responsibilities should be placed. In this example the responsibilities are shifted from the *Sale* class (motivated by Expert) to a Pure Fabrication.

2. In a real persistence framework, more than a single pure fabrication class is ultimately necessary to create a reasonable design. This object will be a front-end facade on to a large number of back-end helper objects.

The design of objects can be broadly divided into two groups:

1. Those chosen by **representational decomposition.**
2. Those chosen by **behavioral decomposition.**

For example, the creation of a software class such as *Sale* is by representational decomposition; the software class is related to or represents a thing in a domain. Representational decomposition is a common strategy in object design and supports the goal of reduced representational gap. But sometimes, we desire to assign responsibilities by grouping behaviors or by algorithm, without any concern for creating a class with a name or purpose that is related to a real-world domain concept.

A good example is an "algorithm" object such as a *TableOfContentsGenerator,* whose purpose is (surprise) to generate a table of contents and was created as a helper or convenience class by a developer, without any concern for choosing a name from the domain vocabulary of books and documents. It exists as a convenience class conceived by the developer to group together some related behavior or methods, and is thus motivated by *behavioral decomposition.*

To contrast, a software class named *TableOfContents* is inspired by *representational decomposition,* and should contain information consistent with our concept of the real domain (such as chapter names).

Identifying a class as a Pure Fabrication is not critical. It is an educational concept to communicate the general idea that some software classes are inspired by representations of the domain, and some are simply "made up" as a convenience for the object designer. These convenience classes are usually designed to group together some common behavior, and are thus inspired by behavioral rather than representational decomposition.

Said another way, a Pure Fabrication is usually partitioned based on related functionality, and so is a kind of function-centric or behavioral object.

Many existing object-oriented design patterns are examples of Pure Fabrications: Adapter, Strategy, Command, and so on [GHJV95].

As a final comment worth reiterating: Sometimes a solution offered by Information Expert is not desirable. Even though the object is a candidate for the responsibility by virtue of having much of the information related to the responsibility, in other ways, its choice leads to a poor design, usually due to problems in cohesion or coupling.

- High Cohesion is supported because responsibilities are factored into a fine grained class that only focuses on a very specific set of related tasks.
- Reuse potential may increase because of the presence of fine-grained Pure Fabrication classes whose responsibilities have applicability in other applications.

Behavioral decomposition into Pure Fabrication objects is sometimes overused by those new to object design and more familiar with decomposing or organizing software in terms of functions. To exaggerate, functions just become objects. There is nothing inherently wrong with creating "function" or "algorithm" objects, but it needs to be balanced with the ability to design with representational decomposition, such as the ability to apply Information Expert so that a representational class such as *Sale* also has responsibilities. Information Expert supports the goal of co-locating responsibilities with the objects that know the information needed for those responsibilities, which tends to support lower coupling. If overused, Pure Fabrication could lead to too many behavior objects that have responsibilities *not* co-located with the information required for their fulfillment, which can adversely affect coupling. The usual symptom is that most of the data inside the objects is being passed to other objects to reason with it.

Related Patterns and Principles

- Low Coupling.
  TT· , „ , ·
- High Cohesion.
- A Pure Fabrication usually takes on responsibilities from the domain class that would be assigned those responsibilities based on the Expert pattern.
- All GoF design patterns [GHJV95], such as Adapter, Command, Strategy, and so on, are Pure Fabrications.
- Virtually all other design patterns are Pure Fabrications.

22.3 Indirection

Solution Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled.

The intermediary creates an *indirection* between the other components.

Problem Where to assign a responsibility, to avoid direct coupling between two (or more) things? How to de-couple objects so that low coupling is supported and reuse potential remains higher?

Examples TaxCalculatorAdapter

These objects act as intermediaries to the external tax calculators. Via polymorphism, they provide a consistent interface to the inner objects and hide the variations in the external APIs. By adding a level of indirection and adding polymorphism, the adapter objects protect the inner design against variations in the external interfaces (see Figure 22.3).

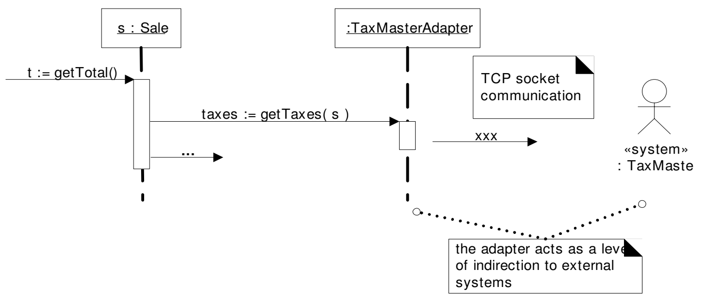


Figure 22.3 Indirection via the adapter.

PersistentStorage

The Pure Fabrication example of decoupling the *Sale* from the relational database services through the introduction of a *PersistentStorage* class is also an example of assigning responsibilities to support Indirection. The *PersistentStorage* acts as a intermediary between the *Sale* and the database.

"Most problems in computer science can be solved by another level of indirection" is an old adage with particular relevance to object-oriented designs. [3]

Just as many existing design patterns are specializations of Pure Fabrication, many are also specializations of Indirection. Adapter, Facade, and Observer are examples [GHJV95]. In addition, many Pure Fabrications are generated because of Indirection. The motivation for Indirection is usually Low Coupling; an intermediary is added to decouple other components or services.

- Lower coupling between components.
- Protected Variations
- Low Coupling
- Many GoF patterns, such as Adapter, Bridge, Facade, Observer, and Mediator [GHJV95].
- Many Indirection intermediaries are Pure Fabrications.

3. If any adage is old in computer science! I have forgotten the source (Parnas?). Note there is also the counter-adage: "Most problems in performance can be solved by removing another layer of indirection!"

1 Protected Variations

Solution Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them.

Note: The term "interface" is used in the broadest sense of an access view; it does not literally only mean something like a Java or COM interface.

Problem How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?

Example For example, the prior external tax calculator problem and its solution with Polymorphism illustrate Protected Variations (Figure 22.1). The point of instability or variation is the different interfaces or APIs of external tax calculators. The POS system needs to be able to integrate with many existing tax calculator systems, and also with future third-party calculators not yet in existence.

By adding a level of indirection, an interface, and using polymorphism with various *ITaxCalculatorAdapter* implementations, protection within the system from variations in external APIs is achieved. Internal objects collaborate with a stable interface; the various adapter implementations hide the variations to the external systems.

Discussion Protected Variations (PV) was first published as a pattern by Cockburn in [VCK96], although this very fundamental design principle has been around for decades under various terms.

Mechanisms Motivated by PV

PV is a root principle motivating most of the mechanisms and patterns in programming and design to provide flexibility and protection from variations.

At one level, the maturation of a developer or architect can be seen in their growing knowledge of ever-wider mechanisms to achieve PV, to pick the appropriate PV battles worth fighting, and their ability to choose a suitable PV solution. In the early stages, one learns about data encapsulation, interfaces, and polymorphism—all core mechanisms to achieve PV. Later, one learns techniques such as rule-based languages, rule interpreters, reflective and metadata designs, virtual machines, and so forth—all of which can be applied to protect against some variation.

For example:

Core Protected Variations Mechanisms

Data encapsulation, interfaces, polymorphism, indirection, and standards are motivated by PV. Note that components such as brokers and virtual machines are complex examples of indirection to achieve PV.

Data-Driven Designs

Data-driven designs cover a broad family of techniques include reading codes, values, class file paths, class names, and so forth, from an external source in order to change the behavior of, or "parameterize" a system in some way at run-time. Other variants include style sheets, metadata for object-relational mapping, property files, reading in window layouts, and much more. The system is protected from the impact of data, metadata, or declarative variations by externalizing the variant, reading it in, and reasoning with it.

Service Lookup

Service lookup includes techniques such as using naming services (for example, Java's JNDI) or traders to obtain a service (for example, Java's Jini, or UDDI for Web services). Clients are protected from variations in the location of services, using the stable interface of the lookup service. It is a special case of data-driven design.

Interpreter-Driven Designs

Interpreter-driven designs include rule interpreters that execute rules read from an external source, script or language interpreters that read and run programs, virtual machines, neural network engines that execute nets, constraint logic engines that read and reason with constraint sets, and so forth. This approach allows changing or parameterizing the behavior of a system via external logic expressions. The system is protected from the impact of logic variations by externalizing the logic, reading it in, and using an interpreter.

Reflective or Meta-Level Designs

An example of this approach is using the *java.beansJntrospector* to obtain a *Beanlnfo* object, asking for the getter *Method* object for bean property X, and calling *Method, invoke.* The system is protected from the impact of logic or external code variations by reflective algorithms that use introspection and meta-lan-guage services. It may be considered a special case of data-driven designs.

Uniform Access

Some languages, such as Ada, Eiffel, and C#, support a syntactic construct so that both a method and field access are expressed the same way. For example, *adrcle.radius* may invoke a *radiusQ:float* method or directly refer to a public field, depending on the definition of the class. We can change from public fields to access methods, without changing the client code.

The Liskov Substitution Principle *(LSP)*

LSP [LiskovSS] formalizes the principle of protection against variations in different implementations of an interface, or subclass extensions of a superclass.

To quote:

> What is wanted here is something like the following substitution property: If for each object *ol* of type S there is an object *o2* of type *T* such that for all programs *P* defined in terms of *T,* the behavior of P is unchanged when ol is substituted for o2 then S is a subtype of *T* [LiskovSS].

Informally, software (methods, classes, ...) that refers to a type *T* (some interface or abstract superclass) should work properly or as expected with any substituted implementation or subclass of T—call it *S.* For example:

```
public void addTaxes( ITaxCalculatorAdapter calculator. Sale sale ) {
    List taxLineltems = calculator.getTaxes( sale );
    // ... }
```

For this method *addTaxes,* no matter what implementation of *ITaxCalculator-Adapter* is passed in as an actual parameter, the method should continue to work "as expected." LSP is a simple idea, intuitive to most object developers, that formalizes this intuition.

Structure-Hiding Designs

In the first edition of this book, an important object design principle called **Don't Talk to Strangers** or the **Law of Demeter** [LieberherrSS] was expressed as one of the nine GRASP patterns. Briefly, it means to avoid creating designs that traverse long object structure paths and send messages (or talk) to distant, indirect (stranger) objects. Such designs are fragile with respect to changes in the object structures—a common point of instability. But in the second edition the more general PV replaced Don't Talk to Strangers, because the latter is a special case of the former. That is, a mechanism to achieve protection from structure changes is to apply the Don't Talk to Strangers rules.

Don't Talk to Strangers places constraints on what objects you should send messages to within a method. It states that within a method, messages should only be sent to the following objects:

1. The *this* object (or *self).*
2. A parameter of the method.
3. An attribute of *this.*
4. An element of a collection which is an attribute *of this.*
5. An object created within the method.

The intent is to avoid coupling a client to knowledge of indirect objects and the object connections between objects.

Direct objects are a client's "familiars," indirect objects are "strangers." A client should talk to familiars, and avoid talking to strangers.

Here is an example that (mildly) violates Don't Talk to Strangers. The comments explain the violation.

```
class Register
{
private Sale sale;

public void slightlyFragileMethod() {
    // sale.getPayment() sends a message to a "familiar" (passes #3)

    // but in sale.getPayment().getTenderedAmount()
    // the getTenderedAmount() message is to a "stranger" Payment

    Money amount = sale.getPayment().getTenderedAmount();

    // . . .
  }
    // . . .
  }
```

This code traverses structural connections from *a* familiar object (the *Sale)* to a stranger object (the *Payment),* and then sends it a message. It is very slightly fragile, as it depends on the fact that *Sale* objects are connected to *Payment* objects. Realistically, this is unlikely to be a problem.

But, consider this next fragment, which traverses farther along the structural path:

```
public void moreFragileMethod() {
    AccountHolder holder =
         sale. getPayment () . get Ac count () . getAccountHolder () ;

    // ...
 }
```

The example is contrived, but you see the pattern: Traversing farther along a path of object connections in order to send a message to a distant, indirect object—talking to a distant stranger. The design is coupled to a particular structure of how objects are connected. The farther along a path the program traverses, the more fragile it is.

Karl Lieberherr and his colleagues have done research into good object design principles, under the umbrella of the Demeter project. This Law of Demeter (Don't Talk to Strangers) was identified because of the frequency with which they saw change and instability in object structure, and thus frequent breakage in code that was coupled to knowledge of object connections.

Yet, as will be examined in the following "Speculative PV and Picking your Battles" section, it is not always necessary to protect against this; it depends on the instability of the object structure. In standard libraries (such as the Java libraries) the structural connections between classes of objects are relatively stable. In mature systems, the structure is more stable. In new systems in early iteration, it isn't stable.

In general, the farther along a path one traverses, the more fragile it is, and thus it is more useful to conform to Don't Talk to Strangers.

Strictly obeying this law—protection against structural variations—requires adding new public operations to the "familiars" of an object; these operations provide the ultimately desired information, and hide how it was obtained. For example, to support Don't Talk to Strangers for the previous two cases:

```
// case 1
Money amount = sale.getTenderedAmountOfPayment( ) ;

// case 2
AccountHolder holder = sale.getAccountHolderOfPayment( ) ;
```

Caution: Speculative PV and Picking Your Battles

First, two points of change are worth defining:

- **variation point**—Variations in the existing, current system or require ments, such as the multiple tax calculator interfaces that must be sup ported.
- **evolution point**—Speculative points of variation that may arise in the future, but which are not present in the existing requirements.[4]

PV is applied to both variation and evolution points.

A caution: Sometimes the cost of speculative "future-proofing" at evolution points outweighs the cost incurred by a simple, more "brittle" design that is reworked as necessary in response to the true change pressures. That is, the cost of engineering protection at evolution points can be higher than reworking a simple design.

For example, I recall a pager message handling system where the architect added a scripting language and interpreter to support flexibility and protected variation at an evolution point. However, during rework in an incremental release, the complex (and inefficient) scripting was removed— it simply wasn't needed. And when I started OO programming (in the early 1980s) I suffered the disease of "generalize-itis" in which I tended to spend many hours creating superclasses of the classes I really needed to write. I would make everything very general and flexible (and protected against variations), for that future situation when it would really pay off—which never came. I was a poor judge of when it was worth the effort.

The point is not to advocate rework and brittle designs. If the need for flexibility and protection from change is realistic, then applying PV is motivated. But if it is for speculative future-proofing or speculative "reuse" with very uncertain probabilities, then restraint and critical thinking is called for.

4. In the UP, evolution points can be formally documented in **Change Cases;** each describes relevant aspects of an evolution point for the benefit of a future architect.

Novice developers tend toward brittle designs, intermediate developers tend toward overly fancy and flexible, generalized ones (in ways that never get used). Expert designers choose with insight; perhaps a simple and brittle design whose cost of change is balanced against its likelihood.

- Extensions required for new variations are easy to add.
- New implementations can be introduced without affecting clients.
- Coupling is lowered.
- The impact or cost of changes can be lowered.

- Most design principles and patterns are mechanisms for protected variation, including polymorphism, interfaces, indirection, data encapsulation, most of the GoF design patterns, and so on.
- In [Pree95] variation and evolution points are called "hot spots."

PV is essentially the same as the information hiding and open-closed principles, which are older terms. As an "official" pattern in the pattern community, it was named "Protected Variations" in 1996 by Cockburn in [VCK96].

Information Hiding

David Parnas's famous paper *On the Criteria To Be Used in Decomposing Systems Into Modules* [Parnas72] is an example of classics often cited but seldom read. In it, Parnas introduces the concept of **information hiding.** Perhaps because the term sounds like the idea of data encapsulation, it has been misinterpreted as that, and some books erroneously define the concepts as synonyms. Rather, Parnas intended information hiding to mean hide information about the design from other modules, at the points of difficultly or likely change. To quote his discussion of information hiding as a guiding design principle:

> We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others.

That is, Parnas's information hiding is the same principle expressed in PV, and not simply data encapsulation—which is but one of many techniques to hide information about the design. However, the term has been so widely reinterpreted as a synonym for data encapsulation that it is no longer possible to use it in its original sense without misunderstanding.

Open-Closed Principle

The **Open-Closed Principle** (OCP), described by Bertrand Meyer in [MeyerSS] is essentially equivalent to the PV pattern and to information hiding. A definition of OCP is:

> Modules should be both open (for extension; adaptable) and closed (the module is closed to modification in ways that affect clients).

OCP and PV are essentially two expressions of the same principle, with different emphasis: protection at variation and evolution points. In OCP, "module" includes all discrete software elements, including methods, classes, subsystems, applications, and so forth.

In the context of OCP, the phrase "closed with respect to X" means that clients are not affected if X changes. For example, "the class is closed with respect to instance field definitions" through the mechanism of data encapsulation with private fields and public accessing methods. At the same time, they are open to modifying the definitions of the private data, because outside clients are not directly coupled to the private data.

As another example, "the tax calculator adapters are closed with respect to their public interface" through implementing the stable *ITaxCalculatorAdapter* interface. However, the adapters are open to extension by being privately modified in response to changes in the APIs of the external tax calculators, in ways that do not break their clients.