GRASP
(General Responsibility Assignment Software Patterns/Principles)

- GRASP: 5 basic patterns
  - Information Expert
  - Creator
  - Controller
  - Low Coupling (evaluation pattern)
  - High Cohesion (evaluation pattern)

  Plus:
  - Polymorphism,
  - Pure fabrication
  - Indirection
  - Protected Variation

Plus GOF: Adaptor, Factory, Singleton, Strategy, Facade

Polymorphism

Larman meaning (quoting Coad):

“giving the same name to services in different objects”

An underlying principle for good design, underlying many specific software design patterns, such as Adapter (GoF)

Polymorphism

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 conditional 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?

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.

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

Example POS (Point of Sale)

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.

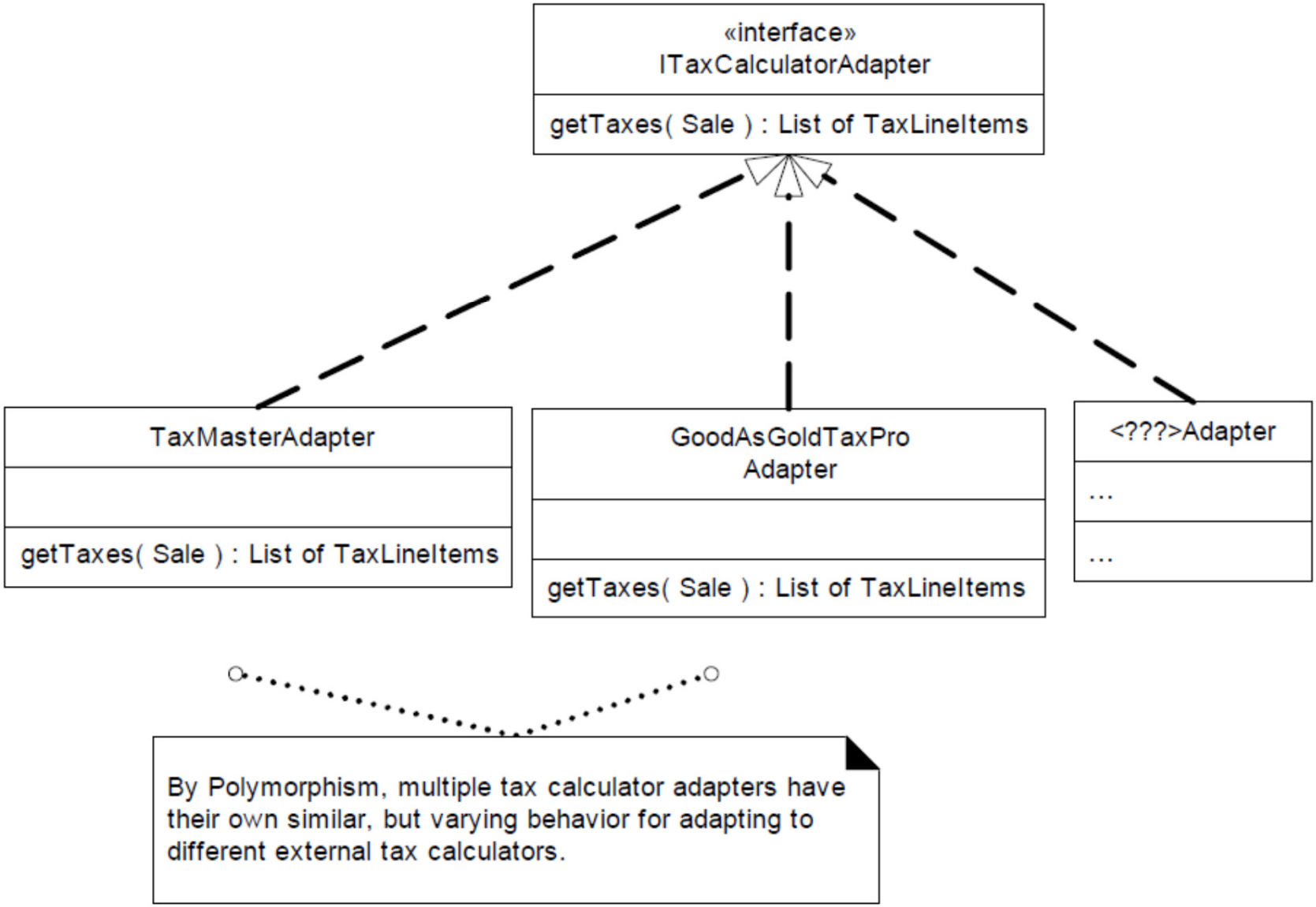
What objects should be responsible for handling these varying external tax calculator interfaces?

Example POS (Point of Sale)

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

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.

Polymorphism

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. But need to evaluate if required.

Benefits of Polymorphism

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

Pure Fabrication

Problem:

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 software 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.

Pure Fabrication

Solution:

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.

Example PoS system

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:

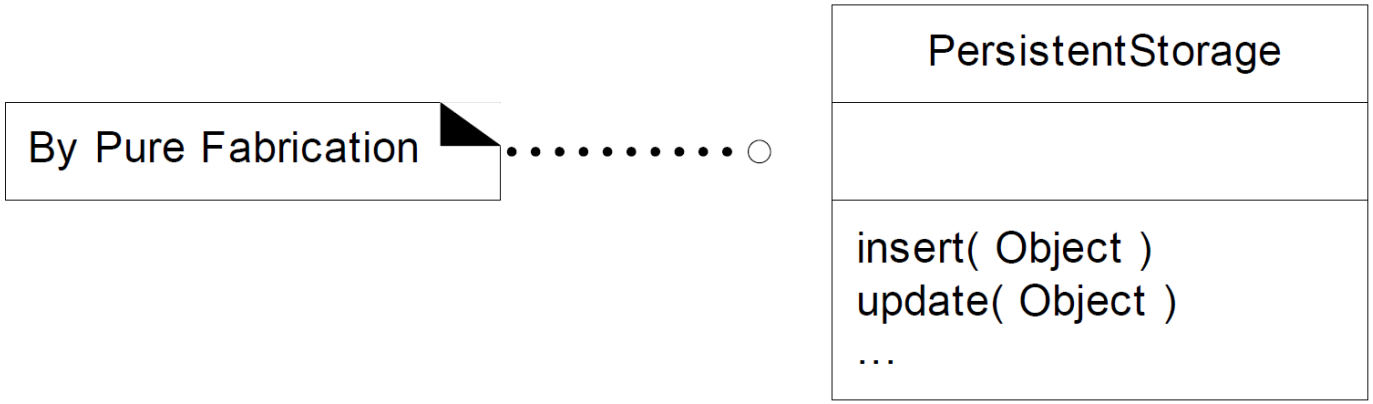
- The task requires a relatively large number of supporting database-oriented operations, none related to the concept of sale-ness, so the *Sale* class becomes incohesive.
- *Sale* class has to be coupled to the relational database interface (such as JDBC in Java technologies), so its coupling goes up. 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.

Example PoS system

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*.

This class is a Pure Fabrication—a figment of the imagination.

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 purpose of storing or inserting objects in a persistent storage medium.
• The *PersistentStorage* class is a very generic and reusable object.

Example the POS system

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.

Indirection

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?

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.

Example the POS system

The TaxCalculatorAdapters (in the Polymorphism example) 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.

Example in the POS system

PersistentStorage in the Pure Fabrication example is used to decouple the *Sale* from the relational database services and 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“ (David Wheeler)

Protected Variations

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?

Solution:

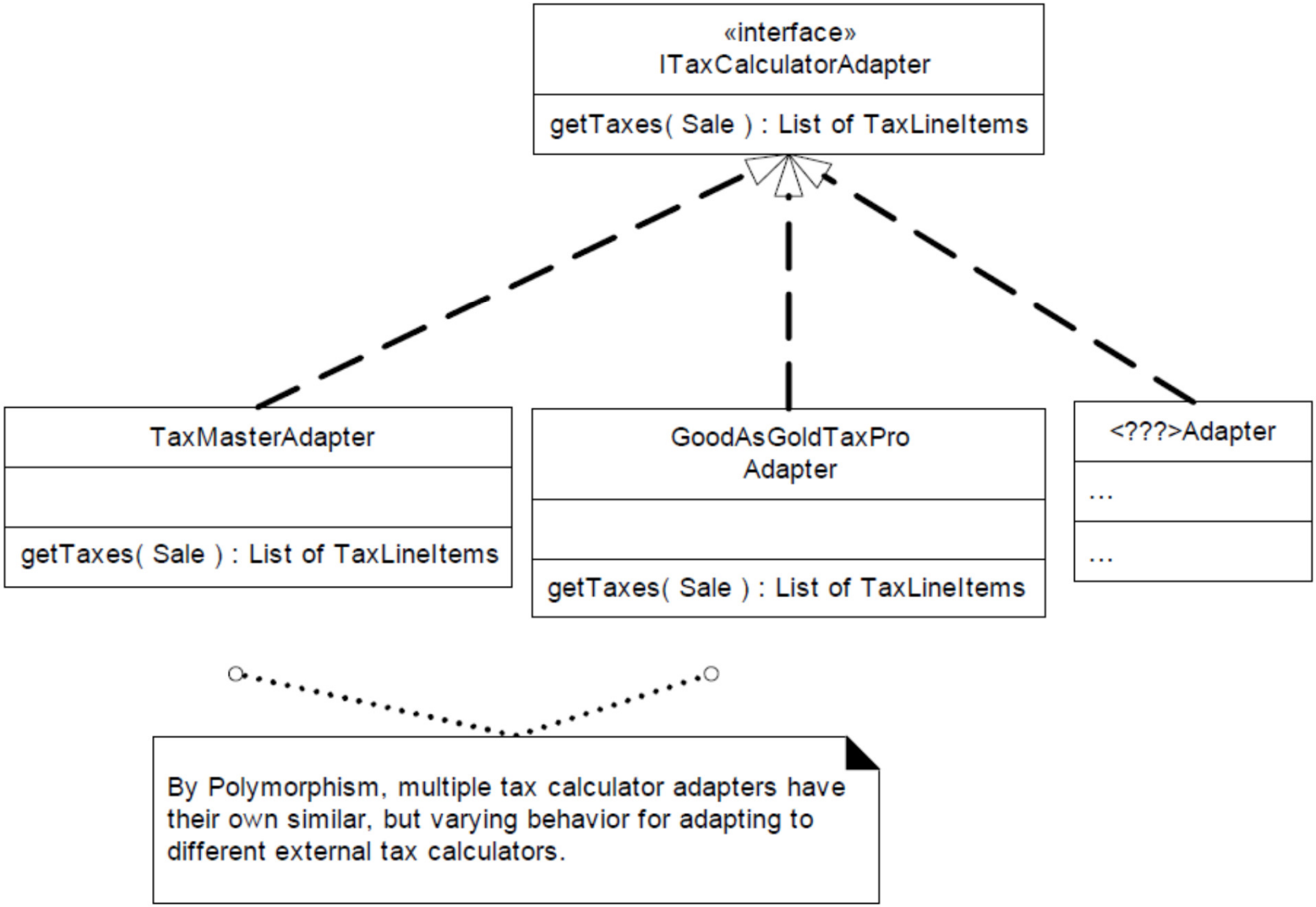
- Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them.

("interface" is used in the broadest sense of an access view; it does not literally only mean something like a Java interface)

PV Example

The previous external tax calculator problem and its solution with Polymorphism also illustrate Protected Variations. 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.


«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.

PV core mechanisms

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 runtime. 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 data variation.

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. variant, reading it in, and reasoning with it.

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-language 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, *aCircle.radius* may invoke a *radius():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** formalizes the principle of protection against variations in different implementations of an interface, or subclass extensions of a superclass.

"What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when ol is substituted for o2 then S is a subtype of *T [Liskov88].*

LSP example

- 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 taxLineItems = calculator.getTaxes( sale );
// ...
```

  }

For this method *addTaxes,* no matter what implementation of *ITaxCalculatorAdapter* 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.

The Law of Demeter/ Don't Talk to Strangers

- 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.

Don't Talk to Strangers:

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.

class Register

```
{
private Sale sale;
public void slightlyFragileMethod() {
// sale.getPayment() sends a message to a "familiar"
// 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.

The Law of Demeter/ Don't Talk to Strangers

More Fragile:

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

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 Law of Demeter/ Don't Talk to Strangers

- Lieberherr et al on the Demeter project devised this Law of Demeter (Don't Talk to Strangers) 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.
- Special case of Protected Variation
- But depends on stabilty of object structure; for standard libraries and mature designs, structure can be stable

Protected Variation

Can consider:

- **variation point—**Variations in the existing, current system or requirements, such as the multiple tax calculator interfaces that must be supported.
- **evolution point—**Speculative points of variation that may arise in thefuture, but which are not present in the existing requirements.

Care with evolution points – problem of over generalizing, with unnecessary cost

Protected Variation

Benefits:

- 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 patterns are developed to achieve Protected Variation

Related Ideas

- Information Hiding (Parnas 72)

"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."

- Open-Closed Principle (Meyer 88)

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

GRASP Summary

- GRASP: patterns
  - Information Expert
  - Creator
  - Controller
  - Low Coupling (evaluation pattern)
  - High Cohesion (evaluation pattern)

  - Polymorphism,
  - Pure fabrication
  - Indirection
  - Protected Variation