System Analysis and Design

L19. GRASP: More Objects with Responsibilities

Four More GRASP Patterns

- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

GRASP的四个新模式

Polymorphism

- **The problem** is how to handle alternatives based upon a type (does this look familiar from databases?) Also, how to create pluggable software components?
- A related problem is pluggable software components

Alternatives based on type

基于类的选项

- One way to handle type-based alternatives is with conditionals: if…else or switch…case statements 用条件语句实现选择
- If a program is designed using conditional logic, then if a new variation arises, it requires modification of the case logic, often in many places. 如果发生变化，在很多条件处需要修改。
- 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. 这样不能容易地扩展。

可插拔组件

Pluggable software components

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

可以把一个部件可以用一个新的组件换掉，不影响其它任何的

Polymorphism – Pluggable Components

多态性可以实现可更换组件.

- Tax calculator uses a standard **interface**, the TaxCalculatorAdapter, to call any of the **actual calculators**.

核心是标准的接口(不变)和可选的多种具体实现

- In one of my systems, I defined a set of standard methods such as getCreditLimit and the like, that interfaced to different accounting systems.

标准的方法组构成了接口的内容,各记账系统的需实现该接口.

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.

使用多态性，将变化的行为赋予多态的各个实现类型。

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

不要使用条件逻辑实现基于类型的选择。

NextGen Problem: How Support Third-Party Tax Calculators?

- Multiple external third-party tax calculators 各种税计算器.
- The system needs to be able to integrate with different ones. 系统需要集成各种不同的计算方法.
- Each tax calculator has a different interface so need to adapt to each of these external fixed interfaces.
  - One may support a raw TCP socket protocol
  - another may offer a SOAP interface
  - a third may offer a Java RMI interface

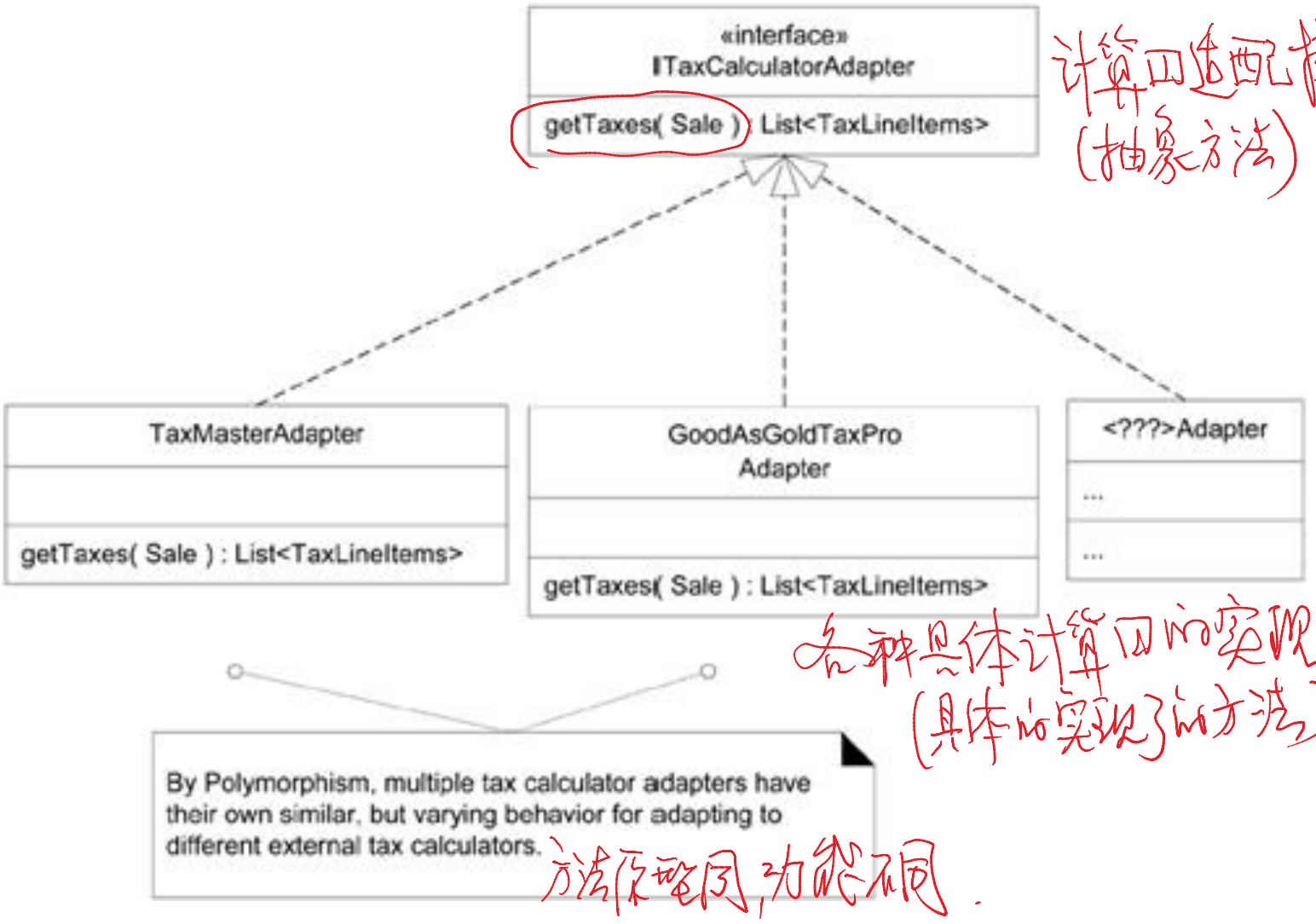
  } 各种不同的接口.
- What objects should be responsible for handling these varying external tax calculator interfaces?

什么对象负责处理这些不同的接口?

Polymorphism in Adapting to Different External Tax Calculators

- By Polymorphism, assign the responsibility for **adaptation** to different calculator objects themselves, implemented with a **polymorphic** *getTaxes* operation
- 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.

Polymorphism in Adapting to Different External Tax Calculators



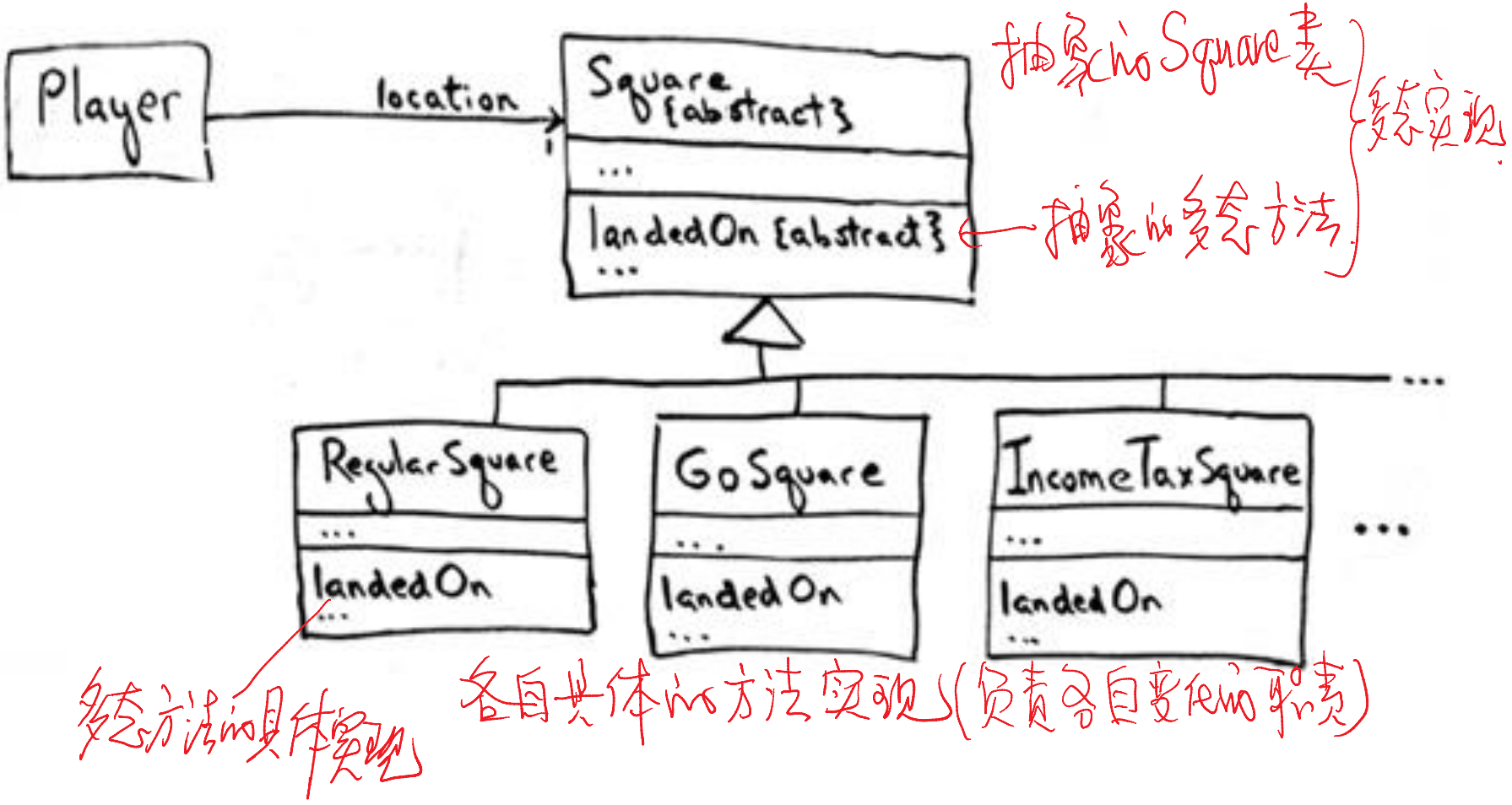
Monopoly Problem

- When a player lands on the Go square, they receive $200.
- There's a different action for landing on the Income Tax square, and so forth.
- There is a different rule for different types of squares. 不同类的方格对应不同的规则
- The principle advises us *to create a polymorphic operation for each type for which the behavior varies.* 按原理我们需要为每个类创建操作.

How to Design for Different Square Actions?

- It varies for the types (classes) *RegularSquare, GoSquare*, and so on.
- What is the operation that varies? It's what happens when a player lands on a square.
- Thus, a good name for the polymorphic operation is ***landedOn***.
- Therefore, by Polymorphism, we'll create a separate class for each kind of square that has a different *landedOn* responsibility, and implement a *landedOn* method in each.

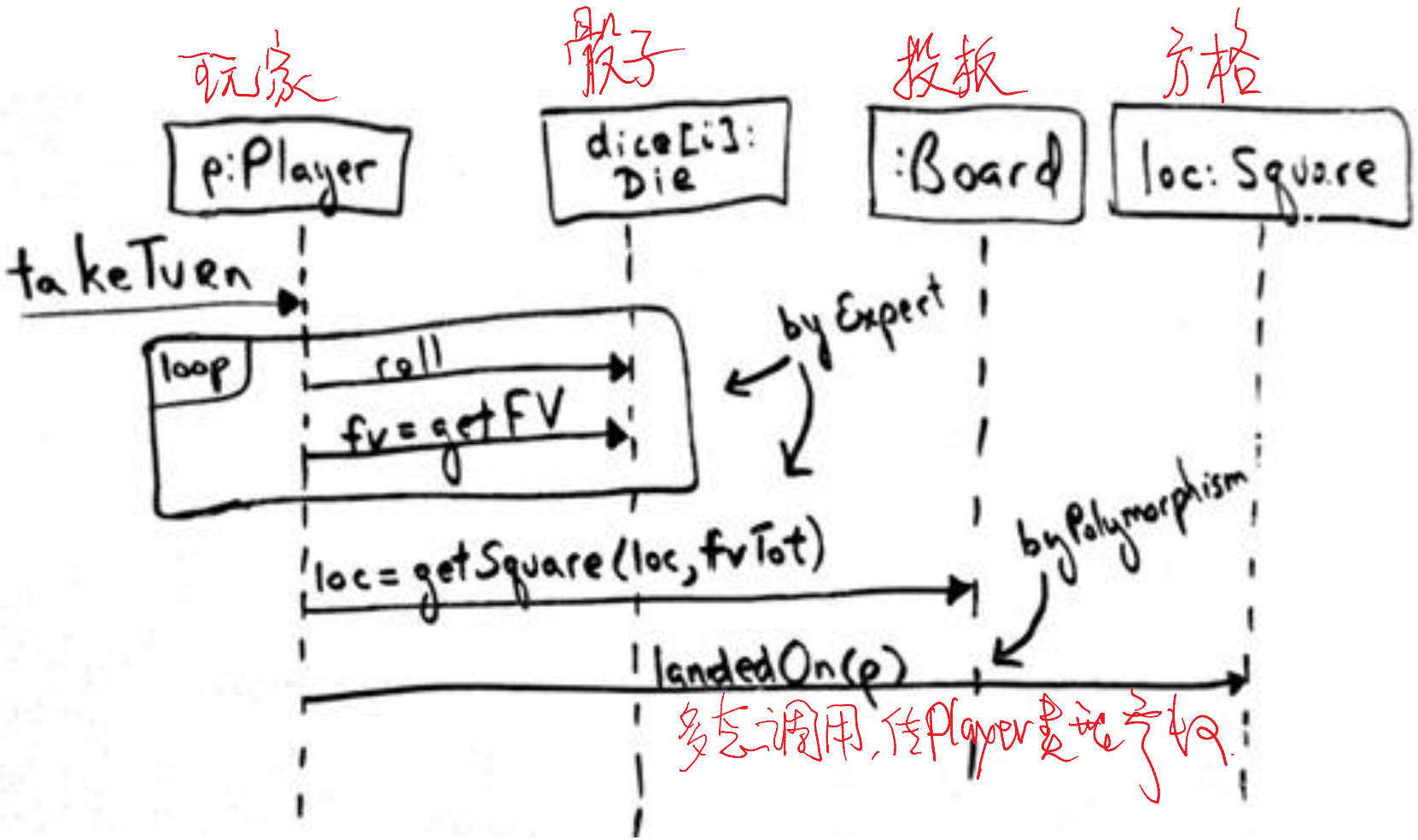
Applying Polymorphism to the Monopoly Problem



The Dynamic Design for Monopoly

- How should the interaction diagrams evolve?
- What object should send the *landedOn* message to the square that a player lands on?
- Since a *Player* software object already knows its location square (the one it landed on), then by the principles of Low Coupling and by Expert, class *Player* is a good choice to send the message, as a *Player* already has visibility to the correct square.

Applying Polymorphism



Discussion

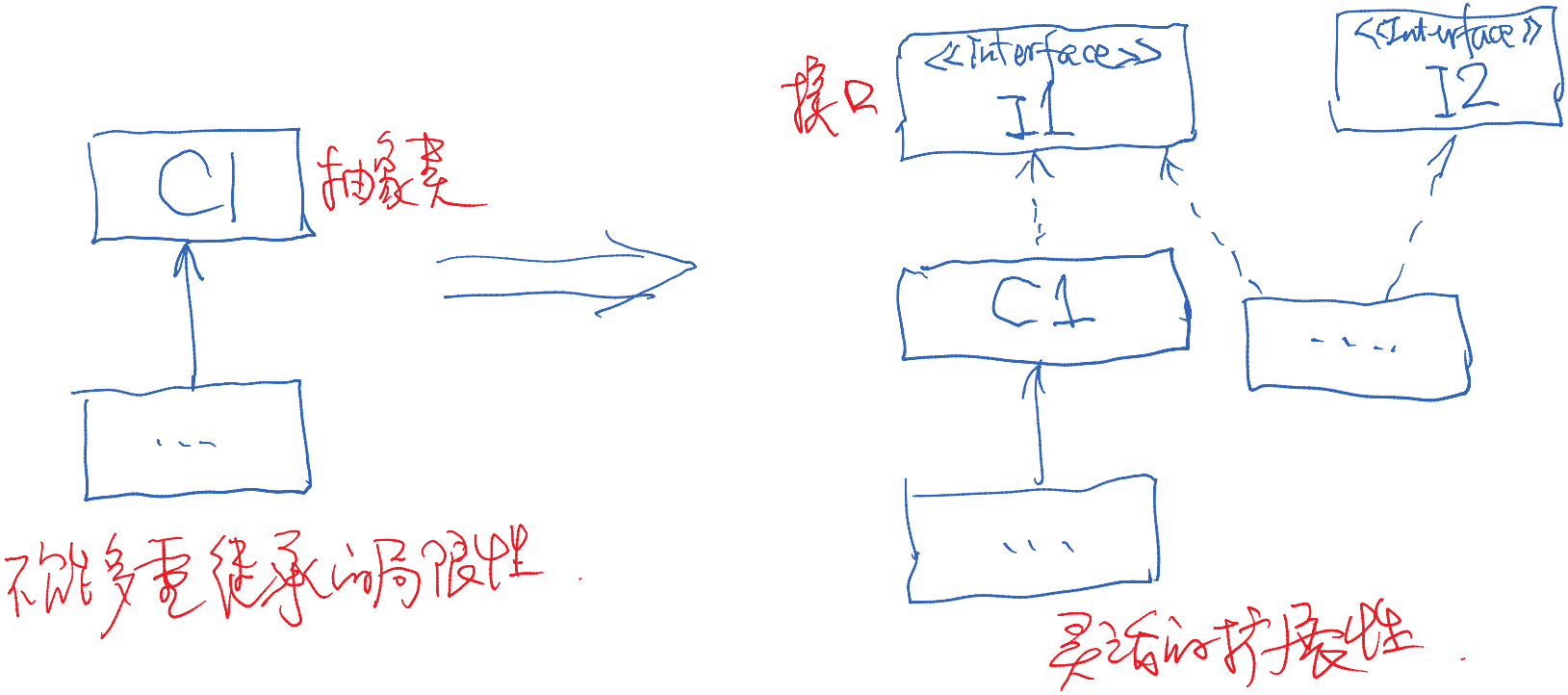
- Polymorphism is a fundamental principle in designing how a system is organized to handle similar variations. 用于设计对象的变化的处理.
- A design based on assigning responsibilities by Polymorphism can be easily extended to handle new variations. 用多态的设计的扩展.
- For example, adding a new calculator adapter class with its own polymorphic *getTaxes* method will have minor impact on the existing design. Adapter是, polymorphic方法. 对已有的设计影响很小.

When to Design with Interfaces?

什么时候使用接口实现多态？

- Polymorphism implies the presence of abstract superclasses or interfaces in most OO languages. 多态性需要抽象基类或接口的支持.
- When should you consider using an interface?
- The general answer is to introduce one when you want to support polymorphism without being committed to a particular class hierarchy. 支持多态又不想局限于特定的类层次.

How to Design with Interfaces?



- Code to interface.

Benefits of Polymorphism

- Extensions required for new variations are easy to add. 易于扩展。
- New implementations can be introduced without affecting clients. 易于升级(新实现)

另一个GRASP模式：纯虚构件

Pure Fabrication

Problem:

- What object should have responsibility when you don't want to violate High Cohesion and Low Coupling or other goals, but solutions offered by Expert (for example) aren't appropriate? 保持高内聚低耦合等要求下，如何分配职责？（Expert方法此时不适用）
- Having classes that represent only domain-layer concepts in some situations leads to problems. 仅有表示领域概念类不完备的，有时常要引入其它类

Cohesive
Coupling, reuse.

Pure Fabrication

Solution:

- Assign a highly cohesive set of responsibilities to a **convenience class** that does not represent a domain object, but which supports high cohesion, low coupling, and reuse.
- Called “fabrication” because it is “made up,” not immediately obvious

NextGen Problem: Saving a Sale Object in a Database

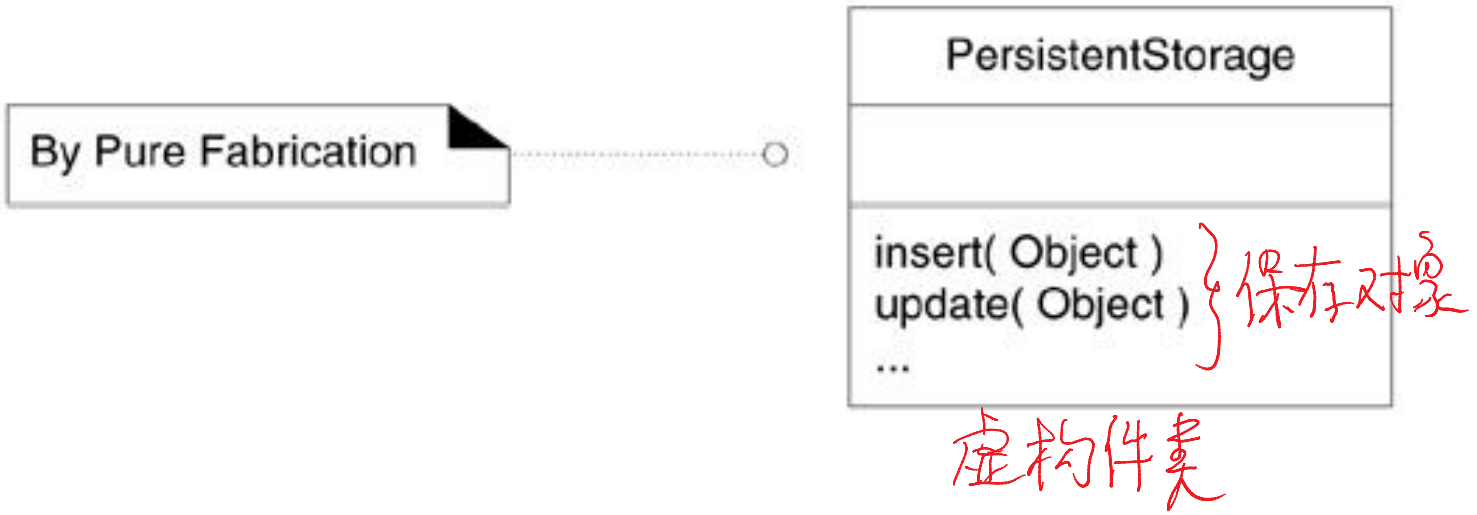
- Support is needed to save *Sale* instances in a relational database. 保存Sale的实例
- By Information Expert, there is some justification to assign this responsibility to the *Sale* class itself, because the sale has the data that needs to be saved. Sale类被分配职责(Expert)

Sale having the Responsibility

- The task requires a large number of supporting database-oriented operations, so the *Sale* class becomes incohesive.
- The *Sale* class has to be coupled to the relational database interface, so its coupling goes up.
- 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.

Pure Fabrication 纯虚构件解决问题

- 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***. 引入新类做的纯虚构类
- This class is a Pure Fabrication, a figment of the imagination. 是一个虚构的类，与领域对象无关。



Pure Fabrication 解决的问题

This Pure Fabrication solves the following design problems:

- The *Sale* remains well-designed, with high cohesion and low coupling. Sale仍保持高聚低耦合
- The *PersistentStorage* class is itself relatively cohesive, having the sole purpose of storing or inserting objects in a persistent storage medium. 虚构类功能单一
- The *PersistentStorage* class is a very generic and reusable object. 可重用的类

Discussion

- The design of objects can be broadly divided into two groups:
  - By representational decomposition
  - By behavioral decomposition
- Most objects represent things in the problem domain, and so are derived by representational decomposition
- Sometimes it is useful to group methods by a behavior or algorithm, even if the resulting class doesn't have a real-world representation
- A Pure Fabrication can be partitioned based on related functionality, so it is a kind of function-centric or behavioral object.

Benefits

- 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.

Contraindications 不适用

- Behavioral decomposition into Pure Fabrication objects sometimes can be overused. 行为分解可能被滥用（过多的行为分解，导致类过于简单）
- Information Expert is often a better choice, since it has the information. Use with caution.

Information Expert 可能会更具高内聚.

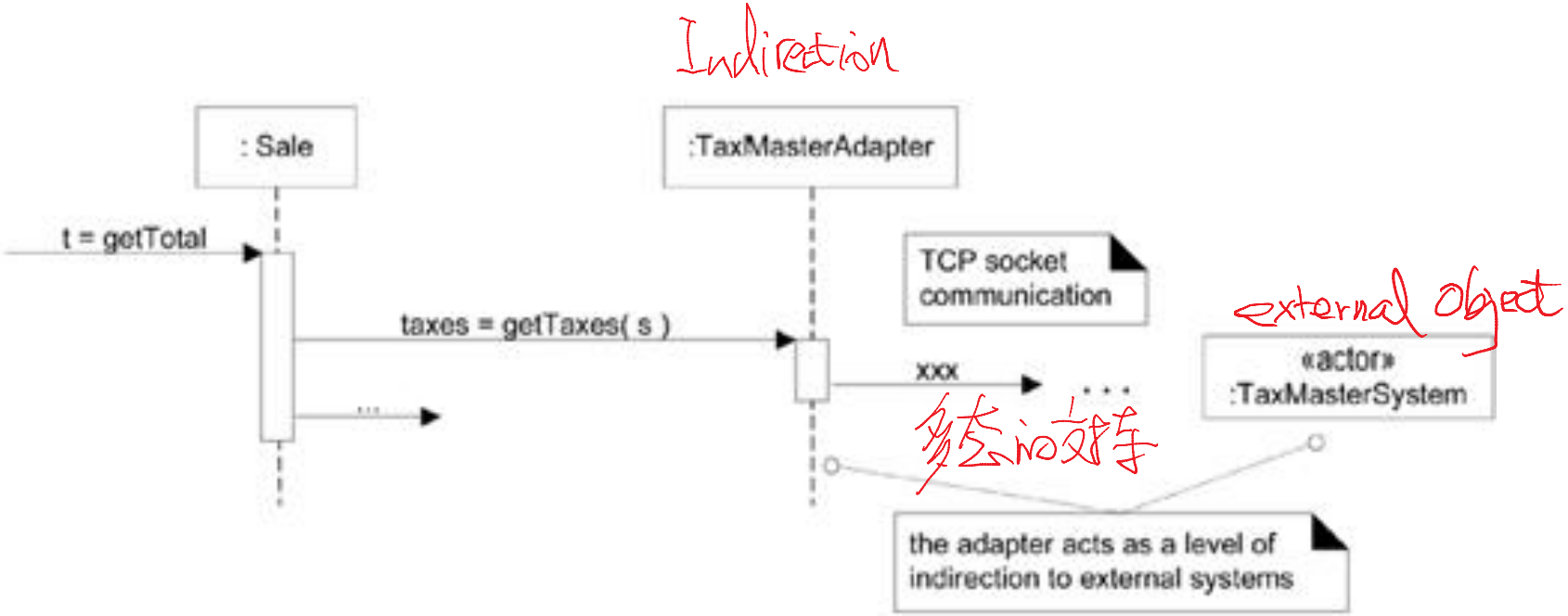
另一个GRASP模式

Indirection Pattern 间接模式

- **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 the chance of reuse is increased?
  在哪里分配职责，可以避免多个对象的直接耦合？如何分割对象，使其支持低耦合和高可重用性！
- **Solution**
  Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled.
  分配到一个中间对象，让其成为其它对象的媒介而避免直接耦合。

Example: TaxCalculatorAdapter

- Indirection via the adapter.
- By adding a level of indirection and adding polymorphism, the adapter objects protect the inner design against variations in the external interfaces



Discussion

- The Pure Fabrication example of *PersistentStorage* class is also an example of assigning responsibilities to support Indirection. PersistentStorage也是一个间接模式的例子.
- "Most problems in computer science can be solved by adding another layer of indirection." 间接层
- "Many performance problems can be solved by removing another layer of indirection."性能需要取消间接层
- The motivation (**Benefits**) for Indirection is usually Low Coupling 引入间接层的目的主要是为了低耦合.

另一个GRASP模式：保护的变化

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.

如何设计对象(子系统、系统)使其元素的变化或不稳定性不会影响其它元素？

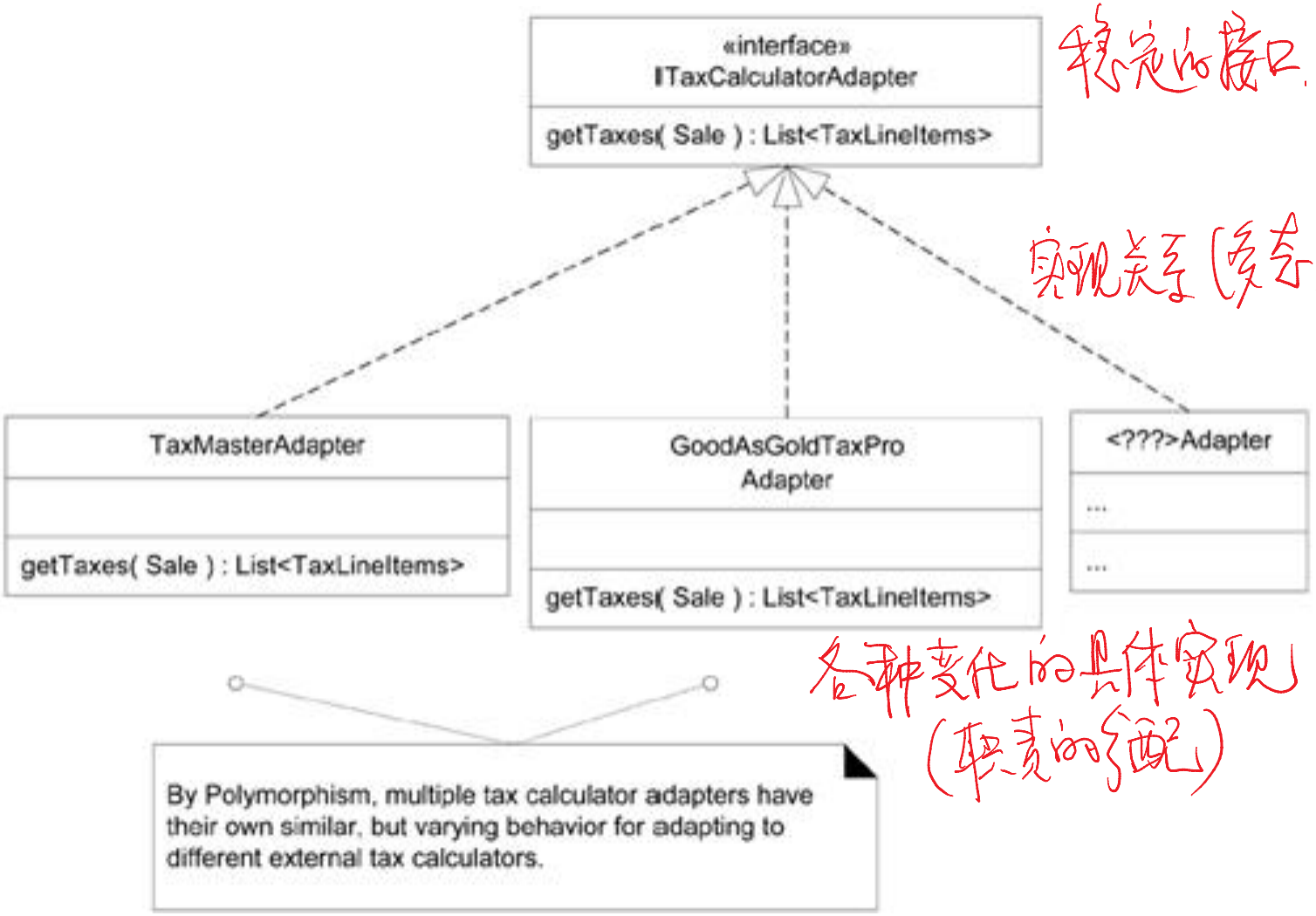
- **Solution:** Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them

在变化处创建稳定的接口，将职责交给接口的实现
(接口只是职责的抽象表现)

Example

- The prior external tax calculator problem and its solution with Polymorphism 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.

Example


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

Discussion

- This is a *very* important, fundamental principle of software design! 非常重要和基础的软件设计原则
- Almost every software or architectural design trick in book is a specialization of Protected Variations.
  - data encapsulation
  - polymorphism
  - data-driven designs
  - interfaces
  - virtual machines
  - configuration files
  - operating systems
  - and much more

大量的变化保护(封装,隐藏)的特例.

Data-Driven Designs

- Data-driven designs cover techniques including 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

- Includes techniques such as using naming services (like Java Naming and Directory Interface (JNDI))
- Protects clients from variations in the location of services
- Special case of data-driven design

另一个例子：解释驱动的设计

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, constraint logic engines, etc.
- Allows changing 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.
- SQL stored functions; Excel formulas

Reflective or Meta-Level Designs

- An example of this approach is using the *java.beans.Introspector* to obtain a *BeanInfo* object, asking for the getter *or setter Method* object for bean property X, and calling *Method.invoke*.
  - Getting metadata from an external source (JavaBean)
  - JavaBean: Attributes + setter + getter
- 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 统一访问方式

- 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. 对域和方法以统一的方式访问
- For example aCircle.radius may invoke a radius() method or simply refer to the radius field, depending on the definition of the class.
- We can change from public fields to access methods, without changing the client code.

Liskov Substitution Principle

- **LSP** formalizes the principle of protection against variations in different implementations of an interface, or subclass extensions of a superclass.
- [**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 behavior of *P* is unchanged when *o1* is substituted for *o2* then *S* is a subtype of *T*
- **Translating:**
  software (methods, classes, …) that refers to a type *T* (some interface or abstract superclass) should work properly with any subclass of *T*

Liskov Substitution Principle

- No matter what implementation of *ITaxCalculatorAdapter* is passed in as an actual parameter, the method should continue to work "as expected."

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

Structure-Hiding Designs (Don't Talk to Strangers)

- Original version of Protected Variations.
- A method should only send messages to:
  - The *this* object (self)
  - A parameter of the method
  - An attribute of *this*
  - An element of a collection which is an attribute of *this*
  - 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.

在设计中限制对象对其它对象的访问

目的是避免对间接对象的访问，从而降低耦合性。

Two Points of Change

- **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 the future, but which are not present in the existing requirements.

Possible Problems with PV

- Over generalization:
  trying to protect against future variations by writing code that can be extended, when these variations will never happen
- The cost of engineering protection at evolution points can be higher than reworking a simple design.

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.

Information Hiding

- Hide information about the design from other modules, at the points of difficulty or likely change. (David Parnas) 在可能的变化点，隐藏设计的信息.
- Parnas's information hiding is the same principle expressed in PV
- It is not simply data encapsulation, which is but one of many techniques to hide information about the design.

Open-Closed Principle

- Modules should be open for extension and closed to modification in ways that affect clients. 对扩展开放，对修改关闭
- OCP includes all software components, including methods, classes, subsystems, applications, etc. 对各个层次有效