Chapter 25

GRASP: More Objects with Responsibilities

Four More GRASP Patterns

- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

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

Polymorphism

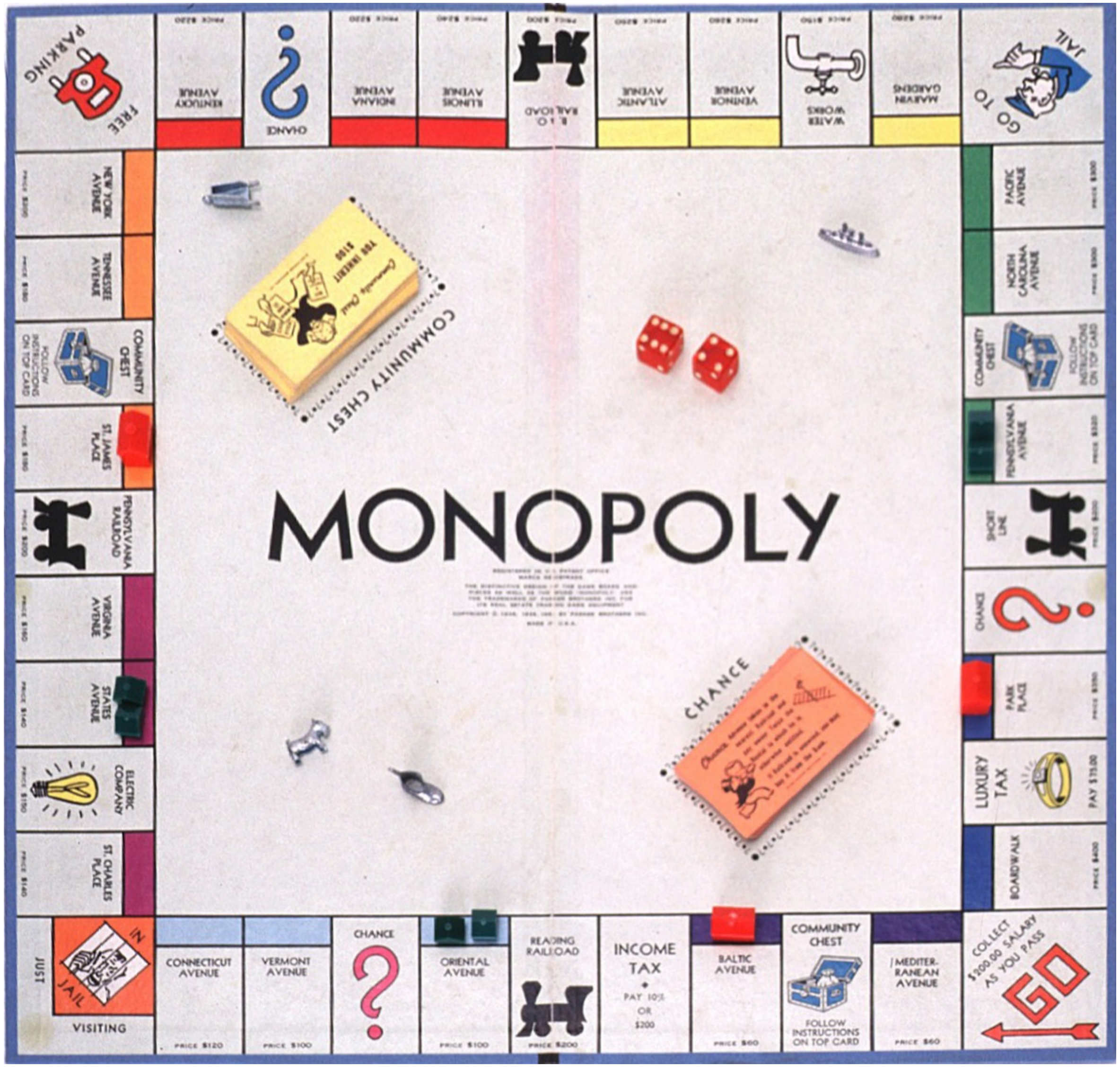
- One way to handle type-based alternatives is with conditionals: if…else or switch…case statements
- For example, the sqrt function has polymorphic variants for float and double
- (how does it really work?)
- (other examples?)

Polymorphism – Pluggable Components

- Tax calculator uses a standard interface, the TaxCalculatorAdapter, to call any of the actual calculators.
- In one of my systems, I defined a set of standard methods such as getCreditLimit and the like, that interfaced to different accounting systems.

Polymorphism

- Monopoly: Different square actions, therefore different classes of squares:
- Property
  - Normal, railroad, utility
- Tax
  - Income, Luxury
- Card
  - Chance, Community Chest


MONOPOLY
CONNECTICUT AVENUE
VERMONT AVENUE
CHANCE
ORIENTAL AVENUE
READING RAILROAD
INCOME TAX
BALTIC AVENUE
COMMUNITY CHEST
MEDITER-RANEAN AVENUE
JUST VISITING
IN JAIL
COLLECT $200.00 SALARY AS YOU PASS
GO
COMMUNITY CHEST
CHANCE

Polymorphism

- What are the common methods for squares that are different depending upon type?
- What are properties of squares that differ by type?

Pure Fabrication

- 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?
- Having classes that represent only domain-layer concepts leads to problems.

Pure Fabrication

- 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

Pure Fabrication

- Database operations are often put in a convenience class. Saving a *Sale* object might, by Expert, belong in the Sale class
- Using a "fabricated" class increases the cohesion in *Sale* and reduces the coupling

Pure Fabrication

Suppose we need to save instances of Sale in a relational database.

To which class in the model do you assign this responsibility?

Since Sale has the information to be saved, Sale would be suggested by Information Expert.

> *To manage data going to and from a relational database will require a large number of operations … insert, delete, update, select, rollback, commit, buffer management, …*

Pure Fabrication

The Expert pattern suggests that Sale is the expert, and so should be the class to do this.

There's a lot involved - save, retrieve, commit, rollback

- what about LineItems? When you save a Sale do you save LineItems too?

We would end up adding a lot to Sale that has nothing to do with *Sale*-ness … Sale becomes less cohesive and more highly coupled to more non-domain classes. Sale will become much more complex, and not so focused.

Pure Fabrication

Pure Fabrication suggests to create a new class for these new responsibilities

PersistentStorage
insert()
delete()
update()
...

PersistentStorage is a fabrication; it is made up from your imagination; it cannot be found in the Domain Model

Sale remains well-designed - high cohesion, low coupling

PersistentStorage class is relatively cohesive - sole purpose is to store/retrieve objects to/from a relational database

Pure Fabrication

Example: see pages 73-76 of Patterns in Java, Volume 2; Mark Grand; John Wiley & Sons

Fig 4.14 shows an initial assignment of responsibilities, where the ServiceManager does scheduling and invoicing.

Fig 4.15 shows the fabrication of a new class, InvoiceGenerator, which results in higher cohesion/less coupling.

System manages a field service organization:

•Organization sends technicians who install and repair equipment on service calls to other organizations that use the equipment

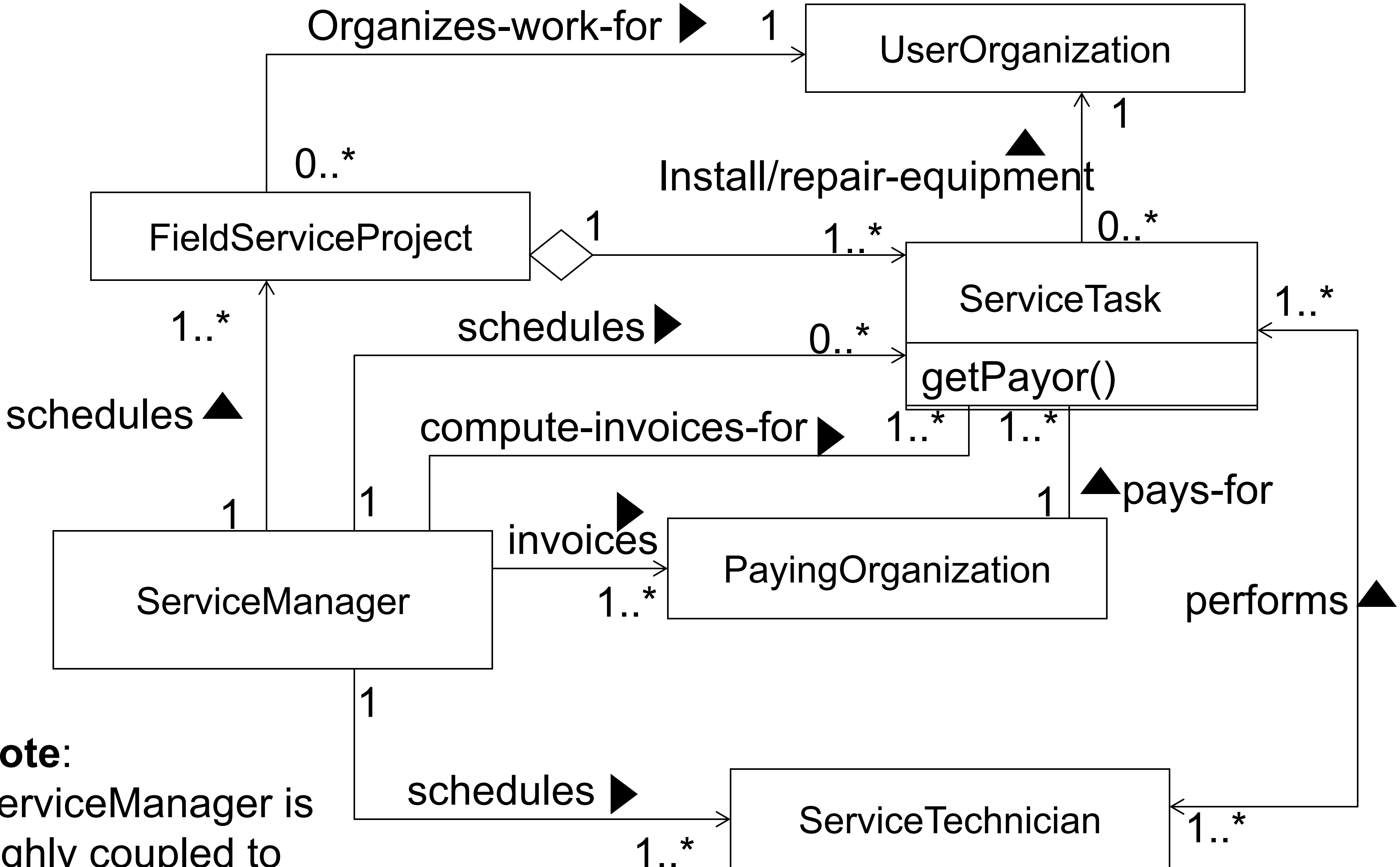
•Some service calls are paid by the organization that uses the equipment; equipment vendors pay from some service calls; others are paid for jointly.

•Service manager is given field service projects for a user organization

•Service manager is schedules service technicians to perform the tasks

•Service manager sends invoices for completed tasks to the paying organizations

Patterns in Java, Volume 2 Fig 4.14



Note:
ServiceManager is highly coupled to other classes

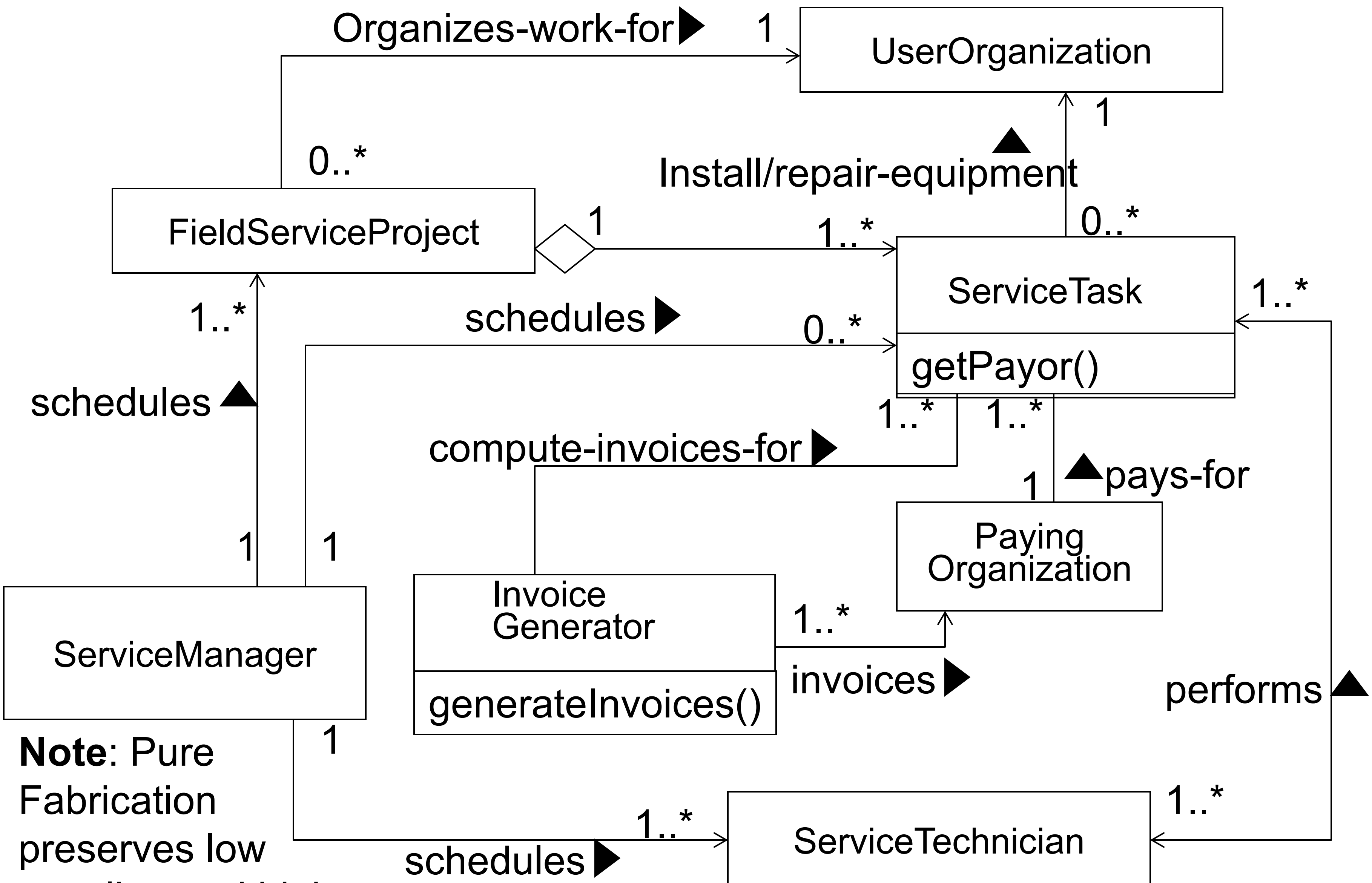
Consider the tasks assigned to the ServiceManager:

- scheduling tasks
- scheduling projects
- scheduling technicians
- generating invoices

These are central to the function of the service manager

no reasonable class in the domain to assign this to, so using Pure Fabrication, *fabricate* a new class for this purpose

Patterns in Java, Volume 2 Fig 4.15



Note: Pure Fabrication preserves low coupling and high cohesion

Another Example

- In Monopoly, one could have a *Dice* object that has properties such as the number of dice and methods to roll them and retrieve the total
- Would rolling and getting the total be one method or two? Why?
- If you're creating games, dice are generally useful, so this class could be reused

Object Design

- By representational decomposition
- By behavioral decomposition
- Most objects represent things in the problem domain, and so are derived by the former
- Sometimes it is useful to group methods by a behavior or algorithm, even if the resulting class doesn't have a real-world representation

Object Design

- *TableOfContents* would represent an actual table of contents.
- *TableOfContentsGenerator* is a pure fabrication class that creates tables of contents.

Contraindications

- This can be overused. Information Expert is often a better choice, since it has the information. Use with caution.

Indirection Pattern

- Problem is how to de-couple objects so that low coupling is supported and the chance of reuse is increased? A related issue is to avoid writing special-purpose code too high up in your application.
- Solution is to create an intermediate object that “talks” to both sides.

Indirection

- Example is the TaxCalculatorAdapter. These provide a consistent interface to disparate inner objects and hide the variations
- “Most problems in computer science can be solved by adding another layer of indirection.”
- “Many performance problems can be solved by removing another layer of indirection.”

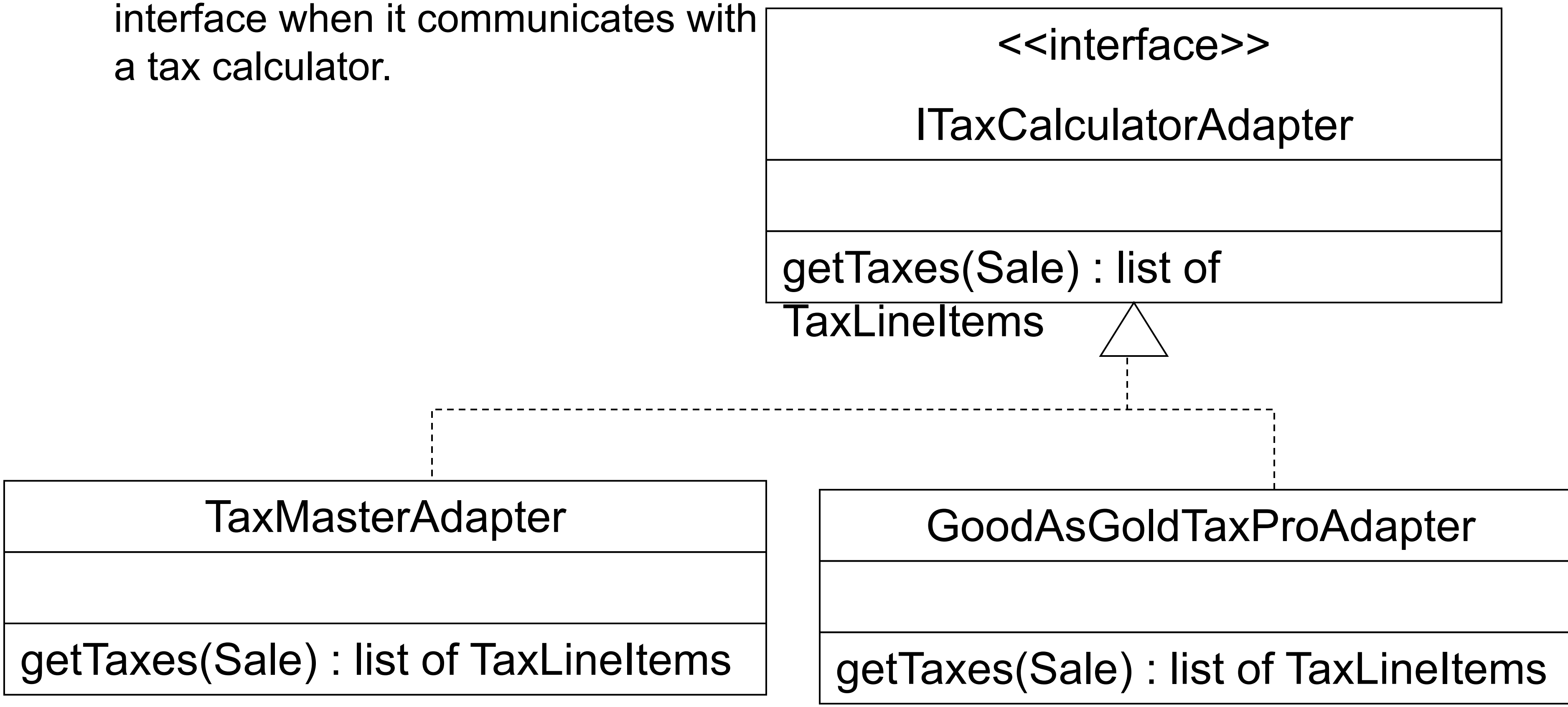
Adapter

Adapter converts the interface of a class into another interface (expected or needed by a client).

- The client sends a message to the adapter, and the adapter sends the appropriate message on to the adaptee.
- The structural aspect of the pattern:

Adapter

NextGenPOS uses a certain interface when it communicates with a tax calculator.



Each adapter is unique to a third-party product. To NextGenPOS, they all look alike

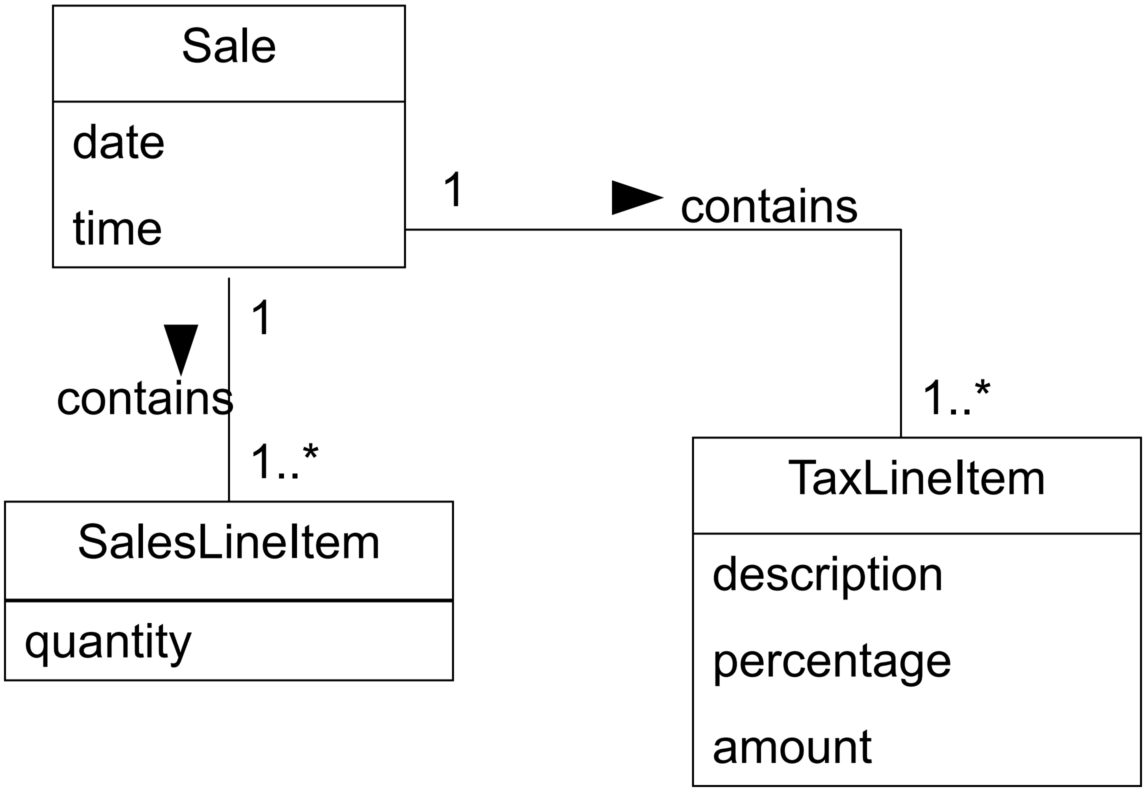
Adapter

How does Adapter relate to the Grasp patterns?

- When the system is running, an adapter object provides indirection
- Because of the common interface each adapter implements the same methods. Each adapter implements a method differently from the others – we have polymorphism.
- The pattern isolates where changes need to occur if the interface of a third party system changes, and so it provides for protection from variations.

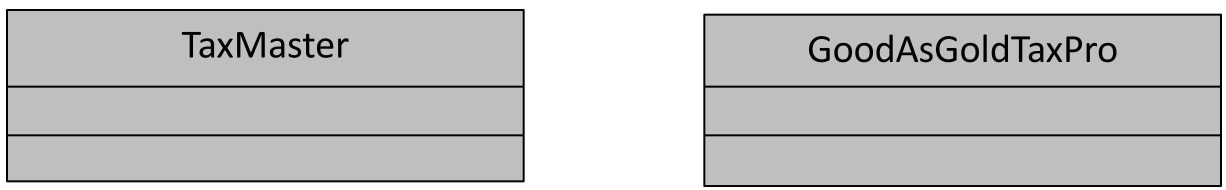
Adapter

Analysis & design leads to an enhancement to the model:



Assume:

- System uses external third-party tax calculator
  - Many different kinds, such as:

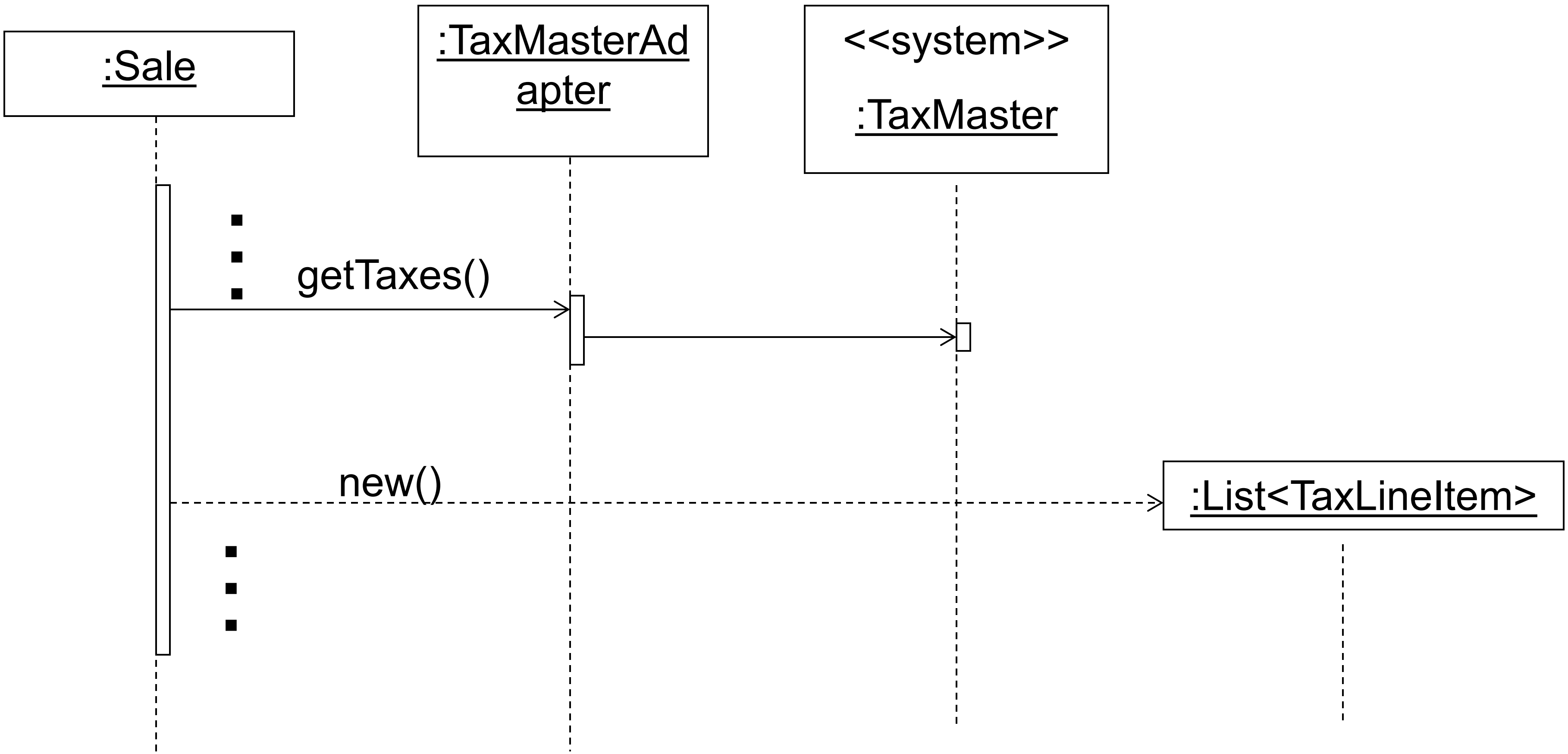


How to make different calculators pluggable?

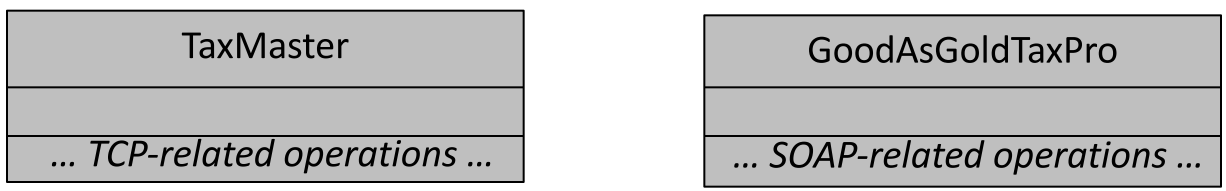
We’ll need to decouple tax calculator from Sale, but how?

Adapter

A Sale collaborates with an adapter to get the tax line items for the sale; the behavioural aspect of adapter:



- Tax calculators have similar but <u>varying</u> interfaces
  - One supports raw TCP socket protocol
  - Another has SOAP interface



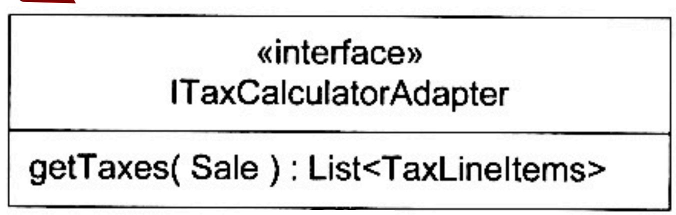
How do we make tax calculators interchangeable?

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: To reduce the potential drawbacks of design instability
  - Identify points of variation/instability
  - Create stable interface (in the broad sense) around such points

Example use of Protected Variations Pattern to decouple Sales and tax calculators

Wrap all the different calculator adapters with one interface



How to wrap tax calculators with interface?

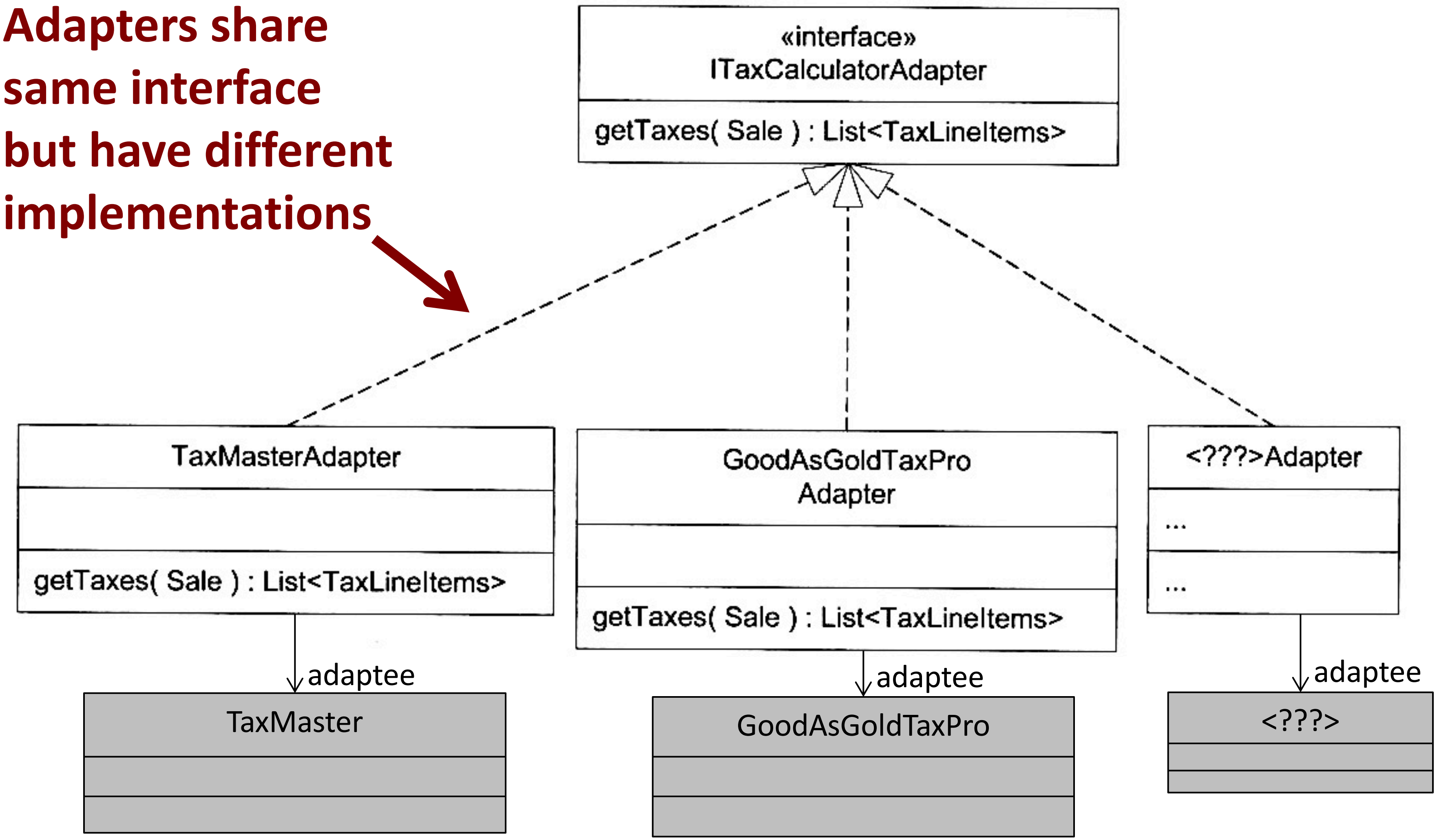
Polymorphism

When related behaviors vary by class, use polymorphic operations to handle the behaviors

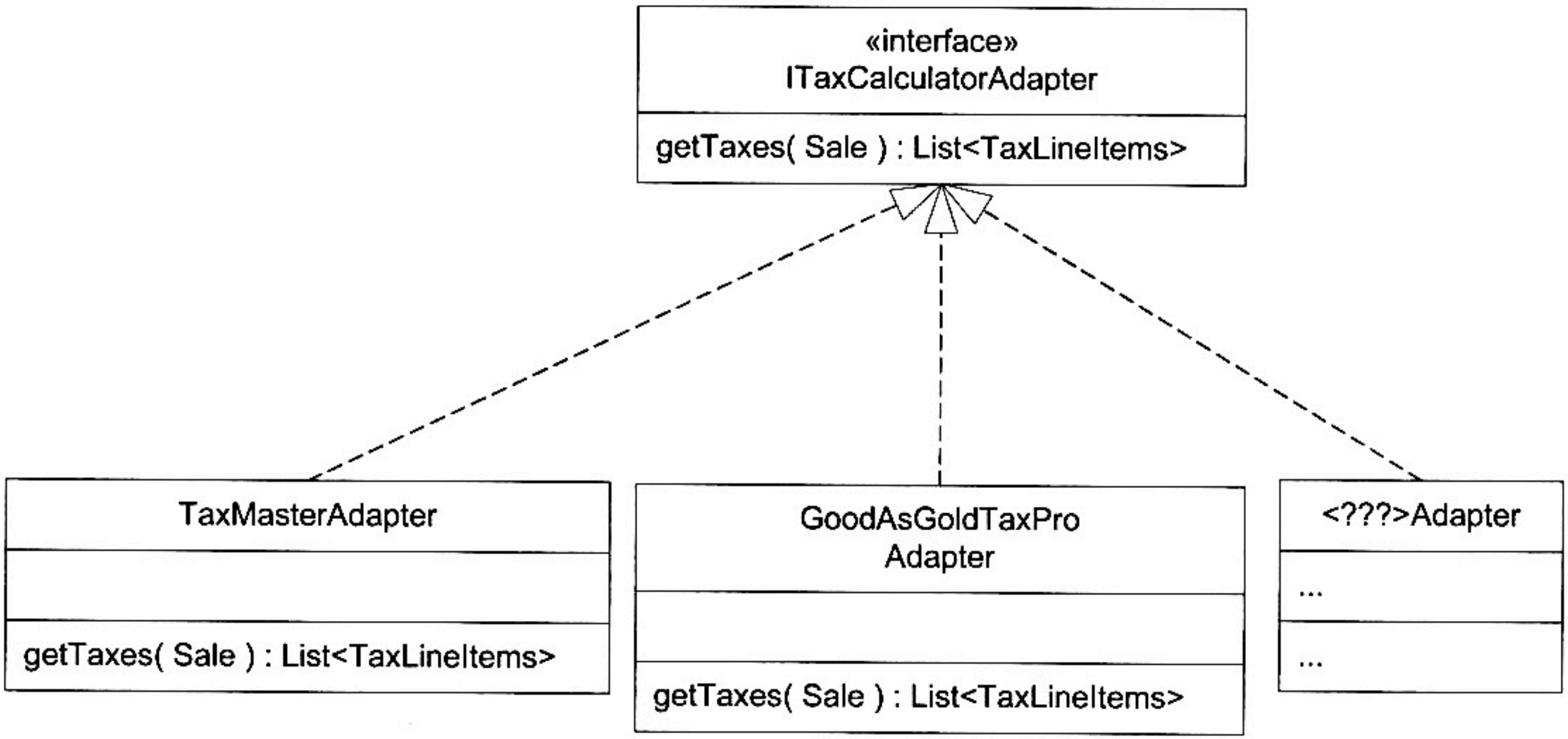
Polymorphic operations: Operations of different objects that have the same signature, making the objects/methods interchangeable

How to apply Polymorphism to tax calculators

Adapters share same interface but have different implementations



Thus, different tax calculators can be swapped in and out



http://flic.kr/p/9ksxQa

Protected Variations

- Many other patterns and concepts in software design derive from this, including data encapsulation, polymorphism, data-driven designs, interfaces, virtual machines, etc.

Data-Driven Designs

- These include techniques such as reading codes, values, class file paths, class names and so on from an external source to “parameterize” a system at run time
- Also includes style sheets, metadata, reading window layouts, etc.

Protected Variations example

- The tax calculator problem illustrates this. The point of instability is the different interfaces of different calculators
- This pattern protects against variations in external APIs

Service Lookup

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

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
- SQL stored functions; Excel formulas

Reflective or Meta-Level Designs

- Getting metadata from an external source.
- Special case of data-driven design

Liskov Substitution Principle

- “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 behavior of *P* is unchanged when *o1* is substituted for *o2* then *S* is a subtype of *T*

Liskov Substitution Principle

- Translating: software that refers to a type *T* should work properly with any subclass of *T*

Why is this design fragile (in the face of change)?

```
public void fragileMethod() {
  AccountHolder holder =
    sale.getPayment().getAccount().getAccountHolder();
  ...
}
```

Depends too much on object structure—a common point of instability

•Coupling: Couples "this" to many classes

•Cohesion: "This" has added responsibility of knowing how to use those classes

Law of Demeter
"Don't talk to strangers"

- Original version of Protected Variations.
- Within a method, only send messages to:
  - The "this" object
  - 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

Stranger and stranger…

```
public void fragileMethod() {
  AccountHolder holder =
    sale.getPayment().getAccount().getAccountHolder();
  ...
}
```

Possible Problems with PV

- Overgeneralization: trying to protect against future variations by writing code that can be extended, when these variations will never happen

Information Hiding

- Private variables
- Hide information about the design from other modules, at the points of difficulty or likely change. (David Parnas)

Open-Closed Principle

- Modules should be both open (for extension) and closed (to modification in ways that affect clients).
- OCP includes all software components, including methods, classes, subsystems, applications, etc.