Object-Oriented Software Analysis and Design

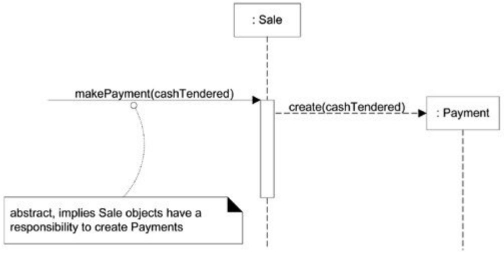
School of Computer Science
University of Windsor

GRASP: A Methodical Approach to Basic OO Design

- General Responsibility Assignment Software Patterns (or Principles)
- A **learning aid** for OOD with Responsibilities
- It is a **set of principles and patterns** to help us understand essential object design and apply design reasoning in a methodical, rational, explainable way.

Responsibilities, GRASP, and UML

- You can think about **assigning responsibilities to objects** while coding or while modeling.
- Within the UML, drawing interaction diagrams becomes the occasion for considering these responsibilities (realized as methods).


: Sale
makePayment(cashTendered)
create(cashTendered)
: Payment
abstract, implies Sale objects have a responsibility to create Payments

What are Patterns?

- Experienced OO developers (and other software developers) build up **a repertoire of both general principles and idiomatic solutions** that guide them in the creation of software.
- These **principles and idioms**, if codified in a structured format describing the problem and solution and named, may be called **patterns**.
- In OO design, a pattern is a **named description of a problem and solution** that can be applied to new contexts;

What are Patterns? (contd.)

- Ideally, a pattern advises us on **how to apply its solution in varying circumstances** and considers the forces and trade-offs.
- Many patterns, given a specific category of problem, guide the assignment of responsibilities to objects.

What are Patterns? An Example of a Sample Pattern

- **Pattern Name:**
  - Information Expert
- **Problem:**
  - What is a basic principle by which to assign responsibilities to objects?
- **Solution:**
  - Assign a responsibility to the class that has the information needed to fulfill it.

The Gang-of-Four Design Patterns Book

- The idea of named **patterns** in software comes from Kent Beck (also of Extreme Programming fame) in the mid 1980s.
- 1994 was a major milestone in the history of patterns, OO design, and software design books: The massive-selling and hugely influential book **Design Patterns** was published, authored by *Gamma, Helm, Johnson, and Vlissides*.
- The book considered the "Bible" of design pattern books, describes 23 patterns for OO design, with names such as Strategy and Adapter.

The Gang-of-Four Design Patterns Book (contd.)

- These 23 patterns, authored by four people, are therefore called the Gang of Four (or GoF) design patterns.
- However, **Design Patterns** isn't an introductory book; it assumes significant prior OO design and programming knowledge, and most code examples are in C++.
- From now on our goal is to learn both GRASP and essential GoF patterns.

Where are we now?

- The iterative **process background** - Prior artifacts? How do they relate to OO design models? How much time should we spend design modeling?
- **RDD** as a metaphor for object design - a community of collaborating responsible objects.
- **Patterns** as a way to name and explain OO design ideas - **GRASP** for basic patterns of assigning responsibilities, and **GoF** for more advanced design ideas. Patterns can be applied during modeling and during coding.
- **UML** for OO design visual modeling, during which time both GRASP and GoF patterns can be applied.
- With that understood, it's time to focus on some details of object design.

A Short Example of Object Design with GRASP

- There are nine GRASP patterns:
  - Creator
  - Information Expert
  - Low Coupling
  - Controller
  - High Cohesion
  - Pure Fabrication
  - Indirection
  - Polymorphism
  - Protected Variations

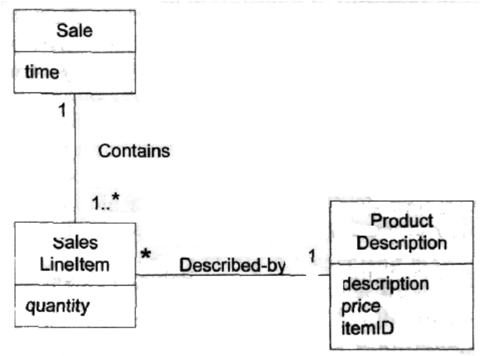
GRASP: Creator

GRASP: Creator

- **Problem:**
  - Who should be responsible for creating a new instance of some class?
  - Assigned well, the design can support low coupling, increased clarity, encapsulation, and reusability.
- **Solution:**
  - Assign class B the responsibility to create an instance of class A if one of these is true
    - B "contains" or compositely aggregates A.
    - B has the initializing data for A that will be passed to A when it is created. Thus B is an Expert with respect to creating A.

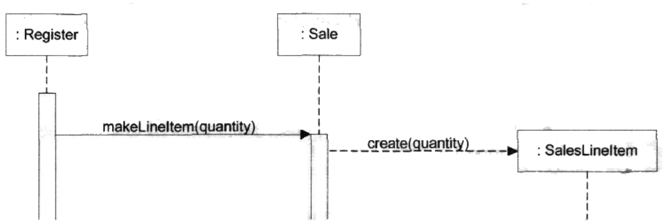
GRASP: Creator (contd.)

- **Example: NextGen POS**



- Who should be responsible for creating a `SalesLineItem` instance?
- By Creator, look for a class that aggregates, contains `SalesLineItem` instances.

GRASP: Creator (contd.)



- A `Sale` contains (aggregates) many `SalesLineItem` objects
  - Creator pattern suggests that `Sale` is a good candidate to have the responsibility of creating `SalesLineItem` instances.
  - This leads to the design of object interactions as shown in the diagram
  - This assignment of responsibilities requires that a `makeLineItem` method be defined in `Sale`

GRASP: Creator (contd.)

- **Benefits**
  - Low coupling is supported, which implies **lower maintenance dependencies** and **higher opportunities for reuse**.
    - Coupling is not increased because the created class is visible to the creator class, due to the existing associations.

GRASP: Creator (contd.)

- **Related Patterns**
  - Low Coupling
  - Concrete Factory and Abstract Factory
  - Whole-Part describes a pattern to define aggregate objects that support encapsulation of components.

GRASP: Information Expert

GRASP: Information Expert

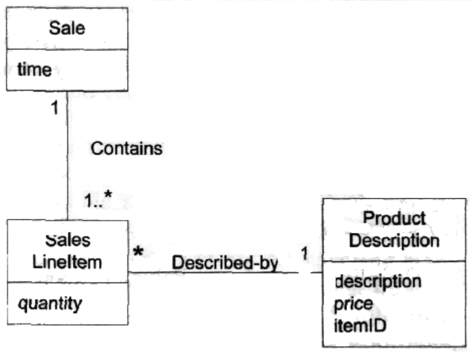
- **Problem:**
  - What is a general principle of assigning responsibilities to objects?
- **Solution:**
  - Assign a responsibility to the information expert—the class that has the information necessary to fulfill the responsibility.

GRASP: Information Expert (contd.)

- **Example: NextGen POS**
- Some class needs to know the grand total of all the `SalesLineItem` instances of a sale.
- Who should be responsible for **knowing** that?
  - If there are relevant classes in the Design Model, look there first.
  - Otherwise, look in the Domain Model, and attempt to use (or expand) its representations to inspire the creation of corresponding design classes.

GRASP: Information Expert (contd.)

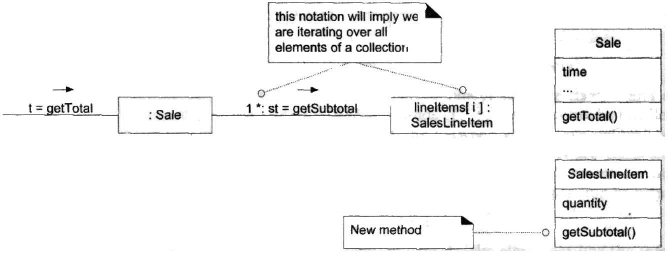
- **Example: NextGen POS**



- Information Expert of the Domain Model: look for class that has the information needed to determine the total—`Sale`.
    - Give the responsibility of knowing its total—method `getTotal`.
    - This approach supports low representational gap between software design of objects and the concepts of real domain.

GRASP: Information Expert (contd.)

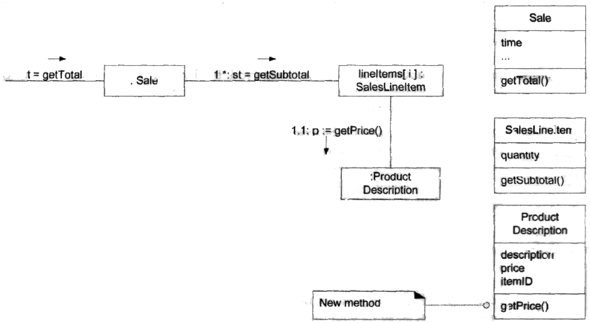
- **Example: NextGen POS**


this notation will imply we are iterating over all elements of a collection
t = getTotal
: Sale
1 *: st = getSubtotal
lineItems[i] : SalesLineItem
Sale
time
...
getTotal()
SalesLineItem
quantity
getSubtotal()
New method

- Who should know the line item `subtotal`?
  - The `SalesLineItem` knows its quantity and its associated `ProductDescription`; therefore, by Expert, `SalesLineItem` should determine the `subtotal`.
  - For interaction diagram, the `Sale` should send `getSubtotal` messages to each of the `SalesLineItems` and sum the results

GRASP: Information Expert (contd.)

- **Example: NextGen POS**



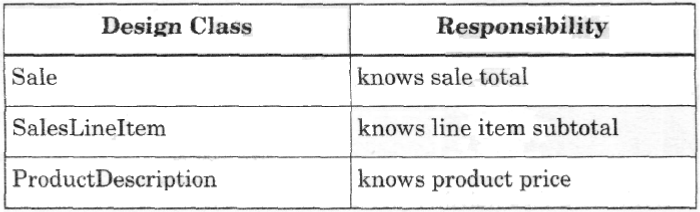
- To fulfill the responsibility of knowing its subtotal, a `SalesLineItem` has to know the product price.

- The `ProductDescription` is an information expert on answering its price;

- `SalesLineItem` sends a message asking for the product price

GRASP: Information Expert (contd.)

- **Example: NextGen POS**
- To fulfill the responsibility of knowing and answering the sale's total.
  - Assign three responsibilities to three design classes of objects in the interaction diagram.
  - Summarize the methods in the method section of a class diagram.



Design Class	Responsibility
Sale	knows sale total
SalesLineItem	knows line item subtotal
ProductDescription	knows product price

GRASP: Information Expert (contd.)

- **Benefits**
  - **Information encapsulation** is maintained since objects use their own information to fulfill tasks. This usually supports low coupling, which leads to more robust and maintainable systems.
  - **Behavior** is distributed across the classes that have the required information, encouraging more cohesive "lightweight" class definitions that are easier to understand and maintain. High cohesion is usually supported.

GRASP: Information Expert (contd.)

- **Related Patterns**
  - Low Coupling
  - High Cohesion

GRASP: Low Coupling

GRASP: Low Coupling

- **Problem:**
  - How to support low dependency, low change impact, and increased reuse?
- **Solution:**
  - Assign a responsibility so that coupling remains low. Use this principle to evaluate alternatives.

GRASP: Low Coupling (contd.)

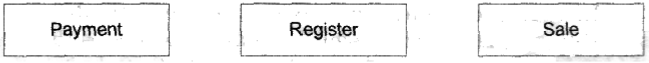
- **Coupling:**
  - A measure of how strongly one element is connected to, has knowledge of, or relies on other elements.
  - An element with low coupling is not dependent on too many other classes, subsystems, systems.
  - **High coupling problems:**
    - Forced local changes because of changes in related classes.
    - Harder to understand in isolation.
    - Harder to reuse because its use requires the additional presence of the classes on which it is dependent.

GRASP: Low Coupling (contd.)

- Low Coupling is a principle to keep in mind during all design decisions.
- It is an evaluative principle that you apply while evaluating all design decisions.
- A subclass is strongly coupled to its superclass.
- Classes that are **inherently generic** and **high reuse** should have low coupling.

GRASP: Low Coupling (contd.)

- **Example: NextGen POS**



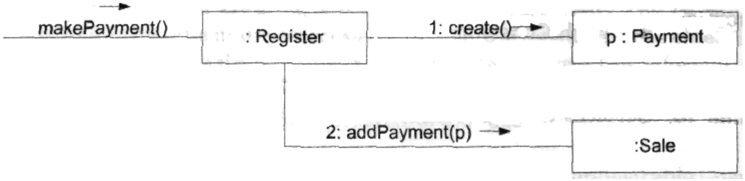
- To create a `Payment` instance and associate it with the `Sale`. What class should be responsible for this?

- Which design, based on assignment of responsibilities, supports Low Coupling?

- In both cases we assume the `Sale` must eventually be coupled to knowledge of a `Payment`.

GRASP: Low Coupling (contd.)

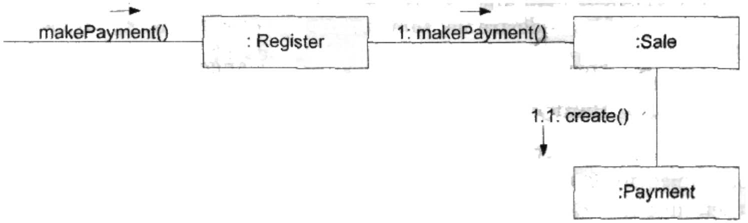
- **Example: NextGen POS**
- Design 1



  - Since a `Register` "records" a `Payment` in the real-world domain, the Creator pattern suggests `Register` as a candidate for creating the `Payment`.
  - The `Register` instance could then send an `addPayment` message to the `Sale`, passing along the new `Payment` as a parameter.
  - Adds coupling of `Register` to `Payment`

GRASP: Low Coupling (contd.)

- **Example: NextGen POS**
- Design 2



- the `Sale` does the creation of a `Payment`, does not increase the coupling.
- Purely from the point of view of coupling, **prefer Design 2** because it maintains overall lower coupling.
- In practice, the level of coupling alone can't be considered in isolation from other principles such as Expert and High Cohesion.

GRASP: Low Coupling (contd.)

- **Benefits**
  - **Not affected by changes** in other components
  - Simple to understand in **isolation**
  - Convenient to **reuse**

GRASP: Low Coupling (contd.)

- **Related Patterns**
  - Protected Variation

GRASP: Controller

GRASP: Controller

- **Problem:**
  - What first object beyond the UI layer receives and coordinates (“controls”) a system operation?
    - A **controller** is the first object beyond the UI layer that is responsible for receiving or handling a system operation message.
    - System operations were first explored during the analysis of SSD. These are the major input events upon our system.

      e.g.,

      When a cashier using a POS terminal presses the “end sale” button, he is generating a system event indicating “the sale has ended.”

      e.g.,

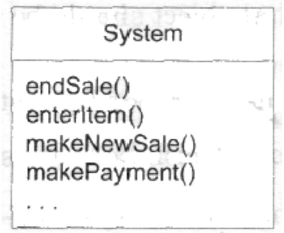
      When a writer using a word processor presses the “spell check” button, he is generating a system event indicating “perform a spell check.”

GRASP: Controller (contd.)

- **Solution:**
  - Assign the responsibility to a class representing one of the following choices:
    - Represents the overall "system," a "root object," a device that the software is running within, or a major subsystem - these are all variations of a facade controller.
    - Represents a use case scenario within which the system event occurs, often named <UseCaseName>Handler, <UseCaseName>Coordinator, or <UseCaseName>Session (use case or session controller).
    - Use the same controller class for all system events in the same use case scenario.
    - Informally, a session is an instance of a conversation with an actor. Sessions can be of any length but are often organized in terms of use cases (use case sessions).
  - *Note that "window," "view," and "document" classes are not on this list. Such classes should not fulfill the tasks associated with system events; they typically receive these events and delegate them to a controller.*

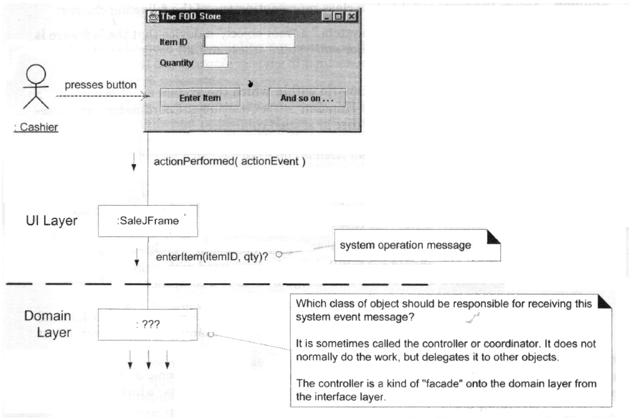
GRASP: Controller (contd.)

- **Example: NextGen POS**
  - Some system operations shown.
  - This model shows the system itself as a class (which is legal and sometimes useful when modeling).



GRASP: Controller (contd.)

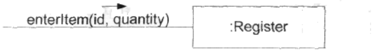
- During **analysis**, system operations may be assigned to the class System.
- During **design**, a controller class is assigned the responsibility for system operations

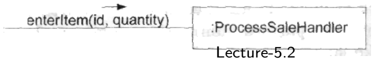


The FOO Store
Item ID
Quantity
Enter Item
And so on . . .
presses button
: Cashier
actionPerformed(actionEvent)
UI Layer
:SaleJFrame
enterItem(itemID, qty)?
system operation message
Domain Layer
: ???
Which class of object should be responsible for receiving this system event message?
It is sometimes called the controller or coordinator. It does not normally do the work, but delegates it to other objects.
The controller is a kind of "facade" onto the domain layer from the interface layer.

GRASP: Controller (contd.)

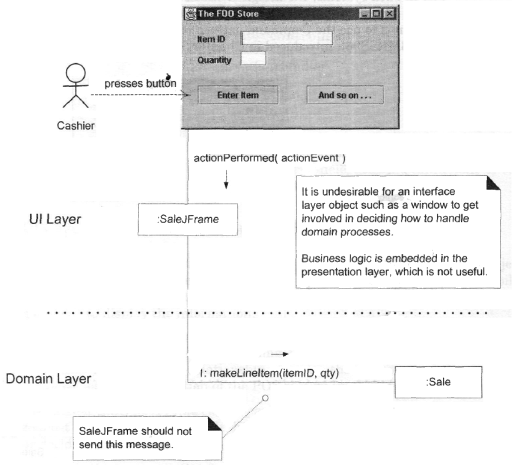
- Who should be the controller for system events such as `enterItem` and `endSale`?
- By the Controller pattern
  - Represents the overall "system," "root object," device, or subsystem. (`Register`, `POSSystem`)
  - A receiver or handler of all system events of a use case scenario. (`ProcessSaleHandler`, `ProcessSaleSession`)
- The domain of POS, a *Register* (POS Terminal) is a specialized device with software running in it.
- The interaction diagrams:





GRASP: Controller (contd.)

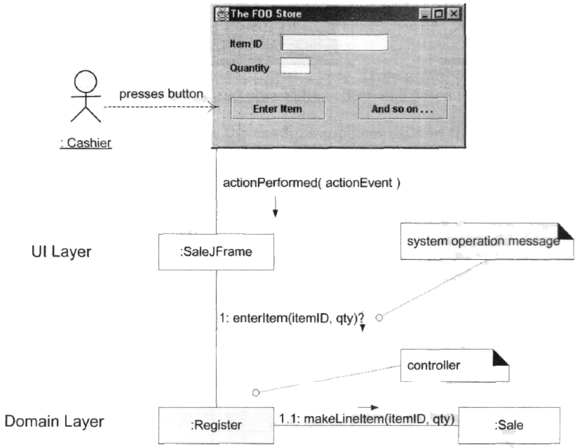
- **Less desirable coupling of interface layer to domain layer**



The FOO Store
Item ID
Quantity
Enter Item
And so on ...
presses button
Cashier
actionPerformed(actionEvent)
UI Layer
:SaleJFrame
It is undesirable for an interface layer object such as a window to get involved in deciding how to handle domain processes.
Business logic is embedded in the presentation layer, which is not useful.
Domain Layer
1: makeLineItem(itemID, qty)
:Sale
SaleJFrame should not send this message.

GRASP: Controller (contd.)

- **Desirable coupling of UI layer to domain layer**



GRASP: Controller (contd.)

- **Benefits**
  - Increased potential for **reuse and pluggable interfaces**
  - Opportunity to reason about the **state of the use case**.

GRASP: Controller (contd.)

- **Related Patterns**
  - **Command:** In a message-handling system, each message may be represented and handled by a separate Command object.
  - **Facade:** A facade controller is a kind of Facade.
  - **Layers:** This is a POSA pattern. Placing domain logic in the domain layer rather than the presentation layer is part of the Layers pattern.
  - **Pure Fabrication:** This GRASP pattern is an arbitrary creation of the designer, not a software class whose name is inspired by the Domain Model. A use case controller is a kind of Pure Fabrication.

GRASP: High Cohesion

GRASP: High Cohesion

- **Problem:**
  - How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?
  - **Cohesion:** a measure of how strongly related and focused the responsibilities of a class or subsystem are.
  - Low cohesion class
    - Does many unrelated things or too much work.
    - Represent a very "large grain" of abstraction or have taken on responsibilities that should have been delegated to other objects
    - Hard to comprehend, reuse, maintain
    - Delicate (constantly affected by change)

GRASP: High Cohesion (contd.)

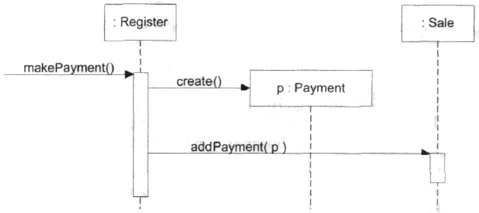
- **Solution:**
  - Assign a responsibility so that cohesion remains high.
  - Use this to evaluate alternatives.

GRASP: High Cohesion (contd.)

- **Example: NextGen POS**
  - Low Coupling pattern for High Cohesion.
  - To create a (cash) `Payment` instance and associate it with the `Sale`. What class should be responsible for this?

GRASP: High Cohesion (contd.)

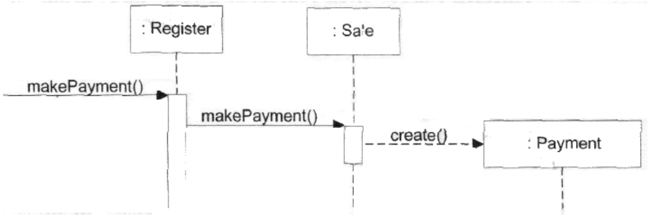
- Design 1



  - Since `Register` records a `Payment` in the real-world domain, the Creator pattern suggests *Register* for creating the Payment.
  - The `Register` instance could send an `addPayment` message to the *Sale*, passing along the new Payment as a parameter
  - To places the responsibility for making a payment in the `Register`.
- To continuous make the `Register` class responsible for doing most of the work related to more system operations
  - It will become increasingly with tasks and become incohesive.

GRASP: High Cohesion (contd.)

- Design 2



- The second design delegates the payment creation responsibility to the `Sale` supports higher cohesion in the `Register`.
  - The second design is desirable, since it supports both high cohesion and low coupling.
- In practice, the level of cohesion alone can't be considered in isolation from other responsibilities and other principles such as Expert and Low Coupling.

GRASP: High Cohesion (contd.)

- High cohesion class has a relatively small number of methods, with highly related functionality, and does not do too much work. It collaborates with other objects to share the effort if the task is large.
- This principle is an evaluative principle evaluating all design decisions.
- Real world
  - if a person takes on too many unrelated responsibilities, then the person is not effective.

GRASP: High Cohesion (contd.)

- **Benefits**
  - **Clarity and ease of comprehension** of the design is increased.
  - **Maintenance and enhancements** are simplified.
  - **Low coupling** is often supported.
  - **Reuse** of fine-grained, highly related functionality is increased because a cohesive class can be used for a very specific purpose.

GRASP: Polymorphism

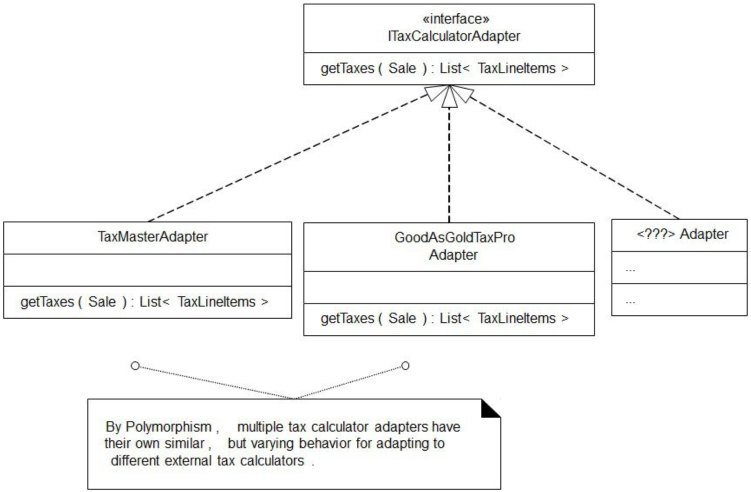
GRASP: Polymorphism

- **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 case logic—often in many places. 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?
- **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.

GRASP: Polymorphism (contd.)

- **Example: NextGen POS**
  - How to Support Third-Party Tax Calculators?
  - 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 TaxPro).
  - The system needs to be able to integrate with different ones.
  - Each tax calculator has a different interface, 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.

GRASP: Polymorphism (contd.)


«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 .

GRASP: Polymorphism (contd.)

- 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.

GRASP: Polymorphism (contd.)

- 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 `getTaxes` method will have minor impact on the existing design.

GRASP: Polymorphism (contd.)

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

GRASP: Polymorphism (contd.)

- **Related Patterns**
  - Protected Variations
  - A number of popular GoF design patterns rely on polymorphism, including Adapter, Command, Composite, Proxy, State, and Strategy.

GRASP: Pure Fabrication

GRASP: 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?
- **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 purehence a pure fabrication.

GRASP: Pure Fabrication (contd.)

- **Example: NextGen POS**
  - Saving a `Sale` Object in a database.
  - 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:

GRASP: Pure Fabrication (contd.)

- 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. (**cohesion**)
- 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. (**coupling**)
- 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. (**reuse**)

GRASP: Pure Fabrication (contd.)

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

GRASP: Pure Fabrication (contd.)

- **Notice the name: `PersistentStorage` 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.**

GRASP: Pure Fabrication (contd.)

- **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.
- 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.

GRASP: Pure Fabrication (contd.)

- **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.

GRASP: Pure Fabrication (contd.)

- **Related Patterns**
  - Low Coupling.
  - 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, such as Adapter, Command, Strategy, and so on, are Pure Fabrications.

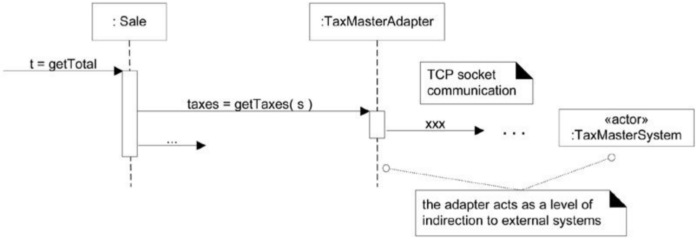
GRASP: Indirection

GRASP: Indirection (contd.)

- **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.

GRASP: Indirection (contd.)

- **Example: NextGen POS (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.

GRASP: Indirection (contd.)

- 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.

GRASP: Indirection (contd.)

- **Benefits**
    - Lower coupling between components.

GRASP: Indirection (contd.)

- **Related Patterns**
  - Protected Variations
  - Low Coupling
  - Many GoF patterns, such as Adapter, Bridge, Facade, Observer, and Mediator.
  - Many Indirection intermediaries are Pure Fabrications.

GRASP: Protected Variations

GRASP: Protected Variations (PV)

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

GRASP: Protected Variations (contd.)

- **Example: NextGen POS (TaxCalculatorAdapter)**
  - 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.

GRASP: Protected Variations (contd.)

- **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.

GRASP: Protected Variations (contd.)

- **Related Patterns**
  - 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.

- 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.

It's Quiz Time

1. The . . . . pattern answers this simple question: What first object after or beyond the UI layer should receive the message from the UI layer?
   1.1 Creator
   1.2 Information Expert
   1.3 Pure Fabrication
   1.4 Controller

2. . . . . focuses on complexity within a module.
   2.1 Cohesion
   2.2 Coupling
   2.3 Controlling
   2.4 Protected Variation

3. Low coupling supports the design of classes that are more independent, which reduces the impact of change. (True or False)