Tecniche di Progettazione: Design Patterns

Design principles

General design principle: Encapsulation

- Aka Information Hiding
- While encapsulation is a fundamental attribute of the Object- Oriented paradigm, it also describes a fundamental principle of good class design; namely, *hide all implementation details from the user of the class.*
- The reason for doing this is so that you can change the underlying implementation without requiring user changes.
- A class that makes internal representations visible is usually poorly designed.

Accessors & Mutators (aka *getters and setters)*

- The usual way of accessing the properties (attributes) of a class.
  - Good encapsulation will hide the data representation.
  - The user of a class should be unaware of whether there is an actual field in the object for a property or if the property is calculated.
  - The accessors and mutators are the interfaces to the properties.

Accessors

- Accessors retrieve the property.
  - An accessor should not have any side effects. This means that an accessor should not change the state of the property's object.
  - Further, it is not good practice to return a property as a value that, if you change it, will be reflected in the original object.
    - For example, assume object A has a Vector, v, that it uses to store some set of items and provides an accessor method, getV(). Now, if getV() returns the reference to the actual vector v, the caller to getV() can modify the contents of the vector.
  - Unless there is a critical need to allow such modifications, you should return a clone of the vector.

Mutators

- Mutators (or setters) are methods that allow (controlled) modification of properties. In effect, the mutators change the state of the object.
- Mutators should also be very specific in their effect.
  - They should *only modify the property* specified and cause no other side effects to the state of the object.

Discussion

- Should you provide accessors and mutators for every property?
- There are several disadvantages in doing so.
  - First of all, you may not need them. Whenever you provide an accessor or mutator to a property, you are telling other programmers that they are free to use them. You have to maintain these methods from that point on.
  - Second, you may not want a property to change. If so, don't provide a mutator.

General design principle: Cohesion

- Cohesion examines how the activities within a module are related to one another. The cohesion of a module may determine how tightly it will be coupled to other modules.
- The objective of designers is to create highly cohesive modules where all the elements of a module are closely related.

General design principle: Decoupling

- Aka uncoupling, aka coupling
- The elements of one module should not be closely related to the elements of another module.
- Such a relationship leads to tight coupling between modules.
- Ensuring high cohesion within modules is one way of reducing tight coupling between modules.

SOLID

- Robert C. Martin.
- Aka uncle Bob
- Five basic principles of object-oriented programming and design.

- http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

SOLID

- Single Responsibility Principle
  - A class (or method) should only have one reason to change.
- Open Closed Principle
  - Extending a class shouldn't require modification of that class.
- Liskov Substitution Principle
  - Derived classes must be substitutable for their base classes.
- Interface Segregation Principle
  - Make fine grained interfaces that are client specific.
- Dependency Inversion Principle
  - Program to the interface, not the implementation.

SOLID 1: Single Responsibility Principle

- A class (or method) should only have one reason to change.
  - In this context a responsibility is considered to be one reason to change. This principle states that if we have 2 reasons to change for a class, we have to split the functionality in two classes. Each class will handle only one responsibility and on future if we need to make one change we are going to make it in the class which handle it. When we need to make a change in a class having more responsibilities the change might affect the other functionality of the classes.
  - Single Responsibility Principle was introduced by Tom DeMarco in his book Structured Analysis and Systems Specification, 1979. Robert Martin reinterpreted the concept and defined the responsibility as a reason to change.

SOLID 2: Open Closed Principle

- Extending a class shouldn't require modification of that class.
- Software entities like classes, modules and functions should be open for extension but closed for modifications.
  - OPC is a generic principle. You can consider it when writing your classes to make sure that when you need to extend their behavior you don’t have to change the class but to extend it. The same principle can be applied for modules, packages, libraries.
  - OPC can be ensured by use of Abstract Classes and concrete classes for implementing their behavior

SOLID 3: Liskov Substitution Principle

- The Liskov Substitution Principle was described by Barbara Liskov at MIT. Basically, the LSP says:

  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 o1 is substituted for o2 then S is a subtype of T.

- Derived classes must be substitutable for their base classes.

SOLID 4: Interface Segregation Principle

- Make fine grained interfaces that are client specific.
- *Clients should not be forced to depend upon interfaces that they don't use.*
  - This principle teaches us to take care how we write our interfaces. When we write our interfaces we should take care to add only methods that should be there. If we add methods that should not be there the classes implementing the interface will have to implement those methods as well. For example if we create an interface called Worker and add a method lunch break, all the workers will have to implement it. What if the worker is a robot?
  - As a conclusion Interfaces containing methods that are not specific to it are called polluted or fat interfaces. Avoid them!

SOLID 5: Dependency Inversion Principle

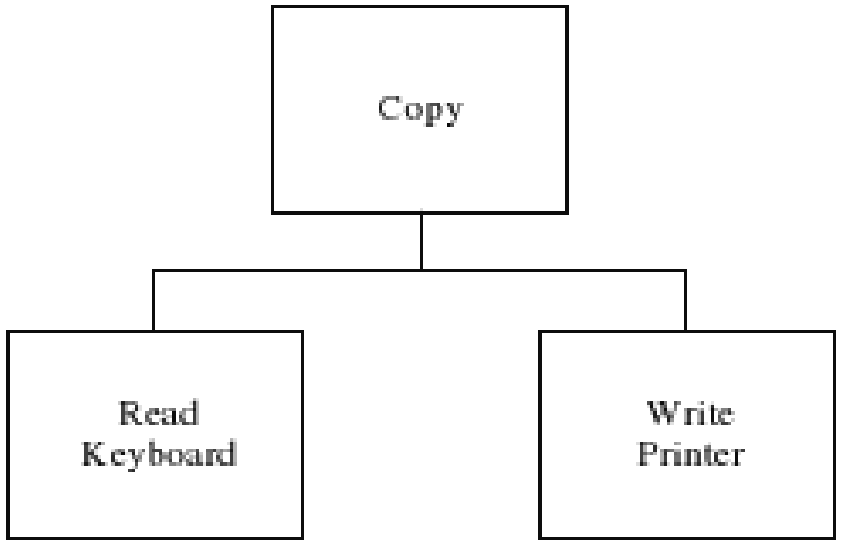
- Program to the interface, not the implementation.
  - *High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.*
  - DIP states that we should decouple high level modules from low level modules, introducing an abstraction layer between the high level classes and low level classes. Furthermore it inverts the dependency: instead of writing our abstractions based on details, the we should write the details based on abstractions.
  - Put simply, this says "depend only on things which are abstract", what I have called in the past, "interface programming" or "programming to the interface". In essence, you should not rely on any concrete implementations of any classes, be they your own or framework objects.

DIP

Example: the "Copy" program.

A simple example may help to make this point. Consider a simple program that is charged with the task of copying characters typed on a keyboard to a printer. Assume, furthermore, that the implementation platform does not have an operating system that supports device independence. We might conceive of a structure for this program that looks like Figure 1:

Figure 1. Copy Program.



DIP

The two low level modules are nicely reusable. They can be used in many other programs to gain access to the keyboard and the printer. This is the same kind of reusability that we gain from subroutine libraries.

Listing 1. The Copy Program

```
void Copy()
{
  int c;
  while ((c = ReadKeyboard()) != EOF)
    WritePrinter(c);
}
```

However the "Copy" module is not reusable in any context which does not involve a keyboard or a printer. This is a shame since the intelligence of the system is maintained in this module. It is the "Copy" module that encapsulates a very interesting policy that we would like to reuse.

For example, consider a new program that copies keyboard characters to a disk file. Certainly we would like to reuse the "Copy" module since it encapsulates the high level policy that we need. i.e. it knows how to copy characters from a source to a sink. Unfortunately, the "Copy" module is dependent upon the "Write Printer" module, and so cannot be reused in the new context.

DIP

We could certainly modify the "Copy" module to give it the new desired functionality. (See Listing 2). We could add an 'if' statement to its policy and have it select between the "Write Printer" module and the "Write Disk" module depending upon some kind of flag. However this adds new interdependencies to the system. As time goes on, and more and more devices must participate in the copy program, the "Copy" module will be littered with if/else statements and will be dependent upon many lower level modules. It will eventually become rigid and fragile.

Listing 2. The "Enhanced" Copy Program

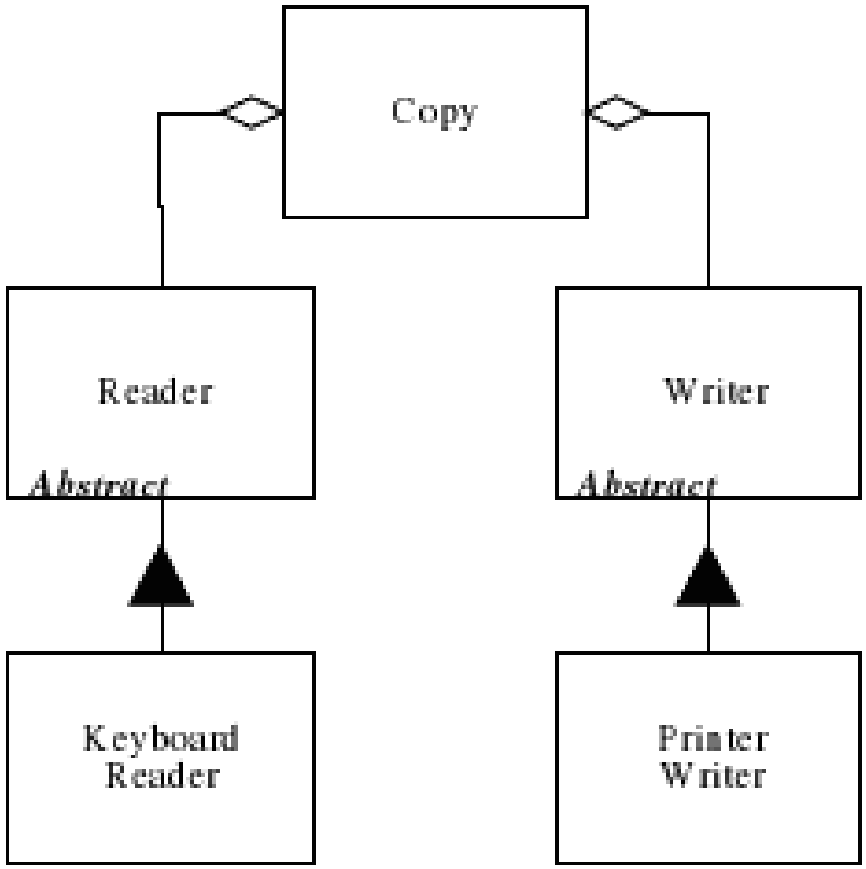
```
enum OutputDevice {printer, disk};
void Copy(outputDevice dev)
{
  int c;
  while ((c = ReadKeyboard()) != EOF)
    if (dev == printer)
      WritePrinter(c);
    else
      WriteDisk(c);
}
```

DIP

input device to any output device. OOD gives us a mechanism for performing this *dependency inversion*.

Consider the simple class diagram in Figure 2. Here we have a "Copy" class which contains an abstract "Reader" class and an abstract "Writer" class. One can easily imagine a loop within the "Copy" class that gets characters from its "Reader" and sends them to its "Writer" (See Listing 3). Yet this "Copy" class does not depend upon the "Keyboard Reader" nor the "Printer Writer" at all. Thus the dependencies have been *inverted*; the "Copy" class depends upon abstractions, and the detailed readers and writers depend

Figure 2: The OO Copy Program



DIP

upon the same abstractions.

Now we can reuse the "Copy" class, independently of the "Keyboard Reader" and the "Printer Writer". We can invent new kinds of "Reader" and "Writer" derivatives that we can supply to the "Copy" class. Moreover, no matter how many kinds of "Readers" and "Writers" are created, "Copy" will depend upon none of them. There will be no interdependencies to make the program fragile or rigid. And Copy() itself can be used in many different detailed contexts. It is mobile.

Listing 3: The OO Copy Program

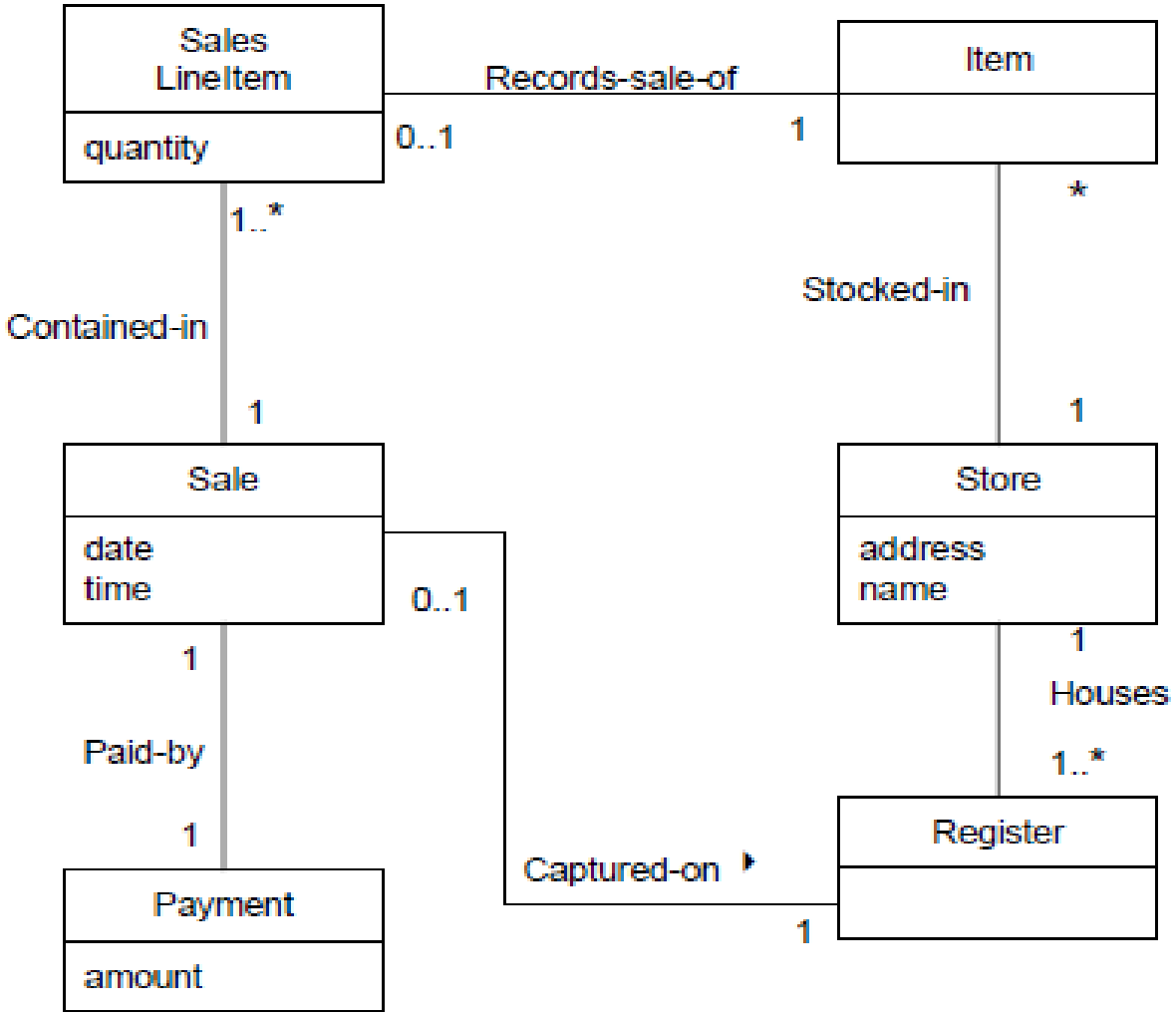
```
class Reader
{
  public:
    virtual int Read() = 0;
};

class Writer
{
  public:
    virtual void Write(char) = 0;
};

void Copy(Reader& r, Writer& w)
{
  int c;
  while((c=r.Read()) != EOF)
    w.Write(c);
}
```

GRASP

- General Responsibility Assignment Software Patterns
- "Applying UML and Patterns" by Craig Larman
- These are not 'design patterns', rather fundamental principles of object design: GRASP patterns focus on one of the most important aspects of object design, assigning responsibilities to classes
- Information Expert, Creator, Controller, Low Coupling, High Cohesion, Polymorphism, Pure Fabrication, Indirection, Protected Variations

Example – domain model

point of sale application



OO design

- A (too ☺) simple definition :
- In the analysis part of the current and previous iterations you have
  - Identified use cases and created use case descriptions to get the requirements
  - Created and refined the domain concept model
- Now in order to make a piece of object design you
  - Assign methods to software classes
  - Design how the classes collaborate (i.e. send messages) in order to fulfill the functionality stated in the use cases.

Central tasks in design

- Deciding what methods belong where
- How the objects should interact

- *A use-case realization*
  - *describes how a particular* use case is realized within the design model in terms of collaborating objects.
  - Use-case realization work is a design activity, the design grows with every new use case realization.
  - Interaction diagrams and patterns apply while doing use-case realizations

Def of responsibilities

- Responsibilities are related to the problem domain
- In design model, responsibilities are obligations of an object in terms of its behavior.
- There are two main types of responsibilities:
  - *Doing responsibilities:*
    - Doing something itself, such as creating an object or doing a calculation
    - Initiating action in other objects
    - Controlling and coordinating activities in other objects.
  - *Knowing responsibilities*
    - Knowing about private encapsulated data
    - Knowing about related objects.
    - Knowing about things it can derive or calculate.
  - *Knowing are often easy to infer from the domain model, where* the attributes and associations are illustrated.

Responsibility vs method

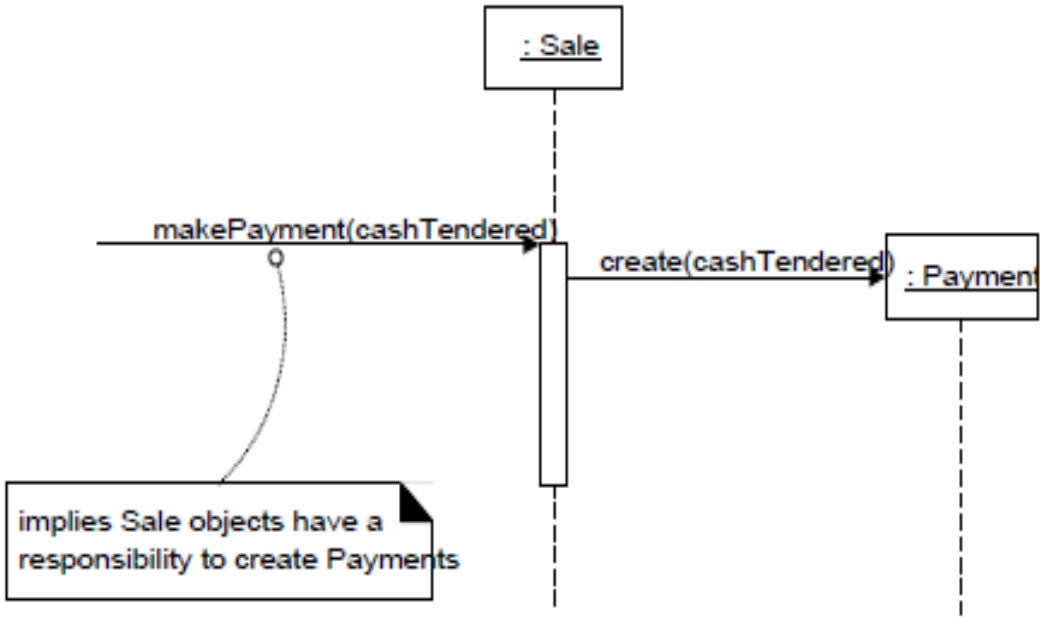
- The translation of problem domain responsibilities into classes and methods is influenced by the granularity of the responsibility.
  - A responsibility is not the same thing as a method, but methods are implemented to fulfill responsibilities.
- Example
  - The *Sale class might define a methods to know its total;* say, a method named *getTotal*.
  - To fulfill that responsibility, the *Sale may collaborate with other objects, such as* sending *a getSubtotal message to each SalesLineItem* object asking for its subtotal.

GRASP – learning and doing Basic Design

- The GRASP patterns are a learning aid to help one understand essential object design.
- Design reasoning is applied in a methodical, rational, explainable way.
- GRASP approach is based on assigning responsibilities, thus creating the basic object and control structures
  - *Guided by patterns of assigning responsibilities*

Responsibilities and Sequence Diagrams

- Responsibilities are illustrated and assigned to classes by creating mainly sequence diagrams.
- Note that during this design work you should stay at the *specification perspective, thinking about the service interfaces* of objects, not their internal implementation
- *Sale objects are* given a responsibility to create *Payments*.
- The responsibility is invoked with a *makePayment* message



The nine GRASP Patterns

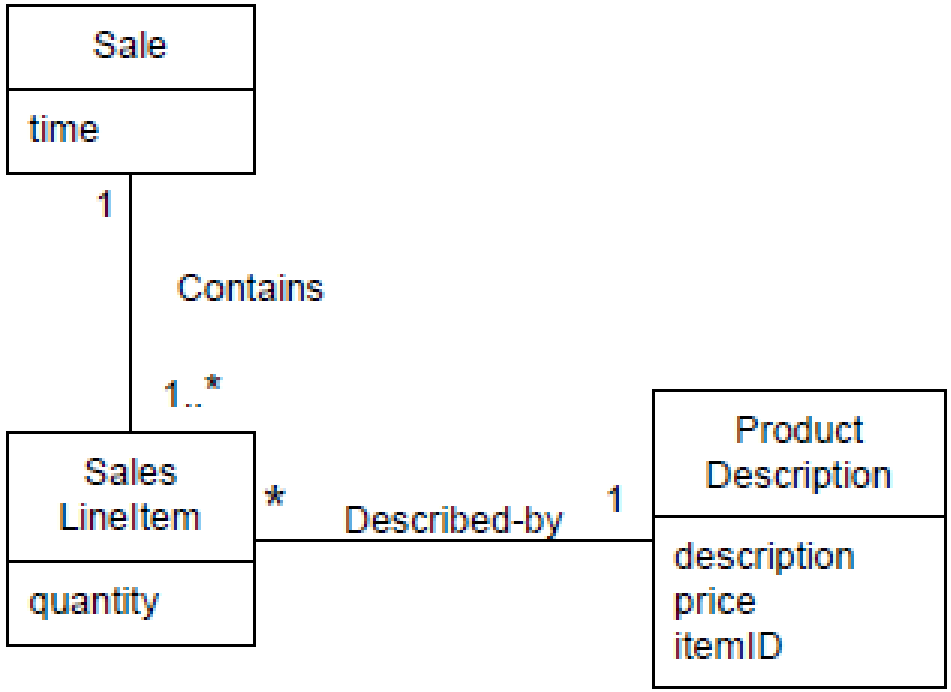
- Creator
- Information Expert
- High Cohesion
- Low Coupling
- Controller
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

Creator

- Problem
  - Who should be responsible for creating new instances of a class?
- Solution
- Assign class B the responsibility to create an instance of class A if one or more of the following is true:
  - B aggregates A objects.
  - B contains A objects.
  - B records instances of A objects.
  - B closely uses A objects.
  - B has the initializing data

Example

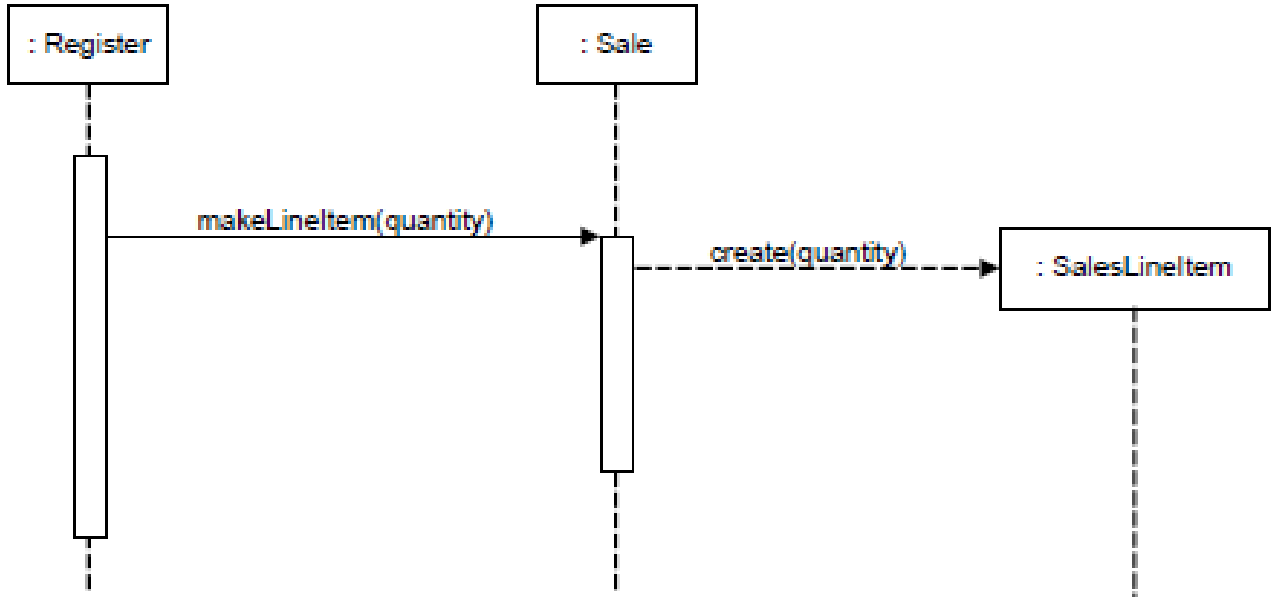
- Who should be responsible for creating a *SalesLineItem* instance?
- Applying Creator, we look for a class that aggregates, contains, and so on, *SalesLineItem instances*.
- Consider the following partial domain model:



Creating SalesLineItem

- a Sale contains many SalesLineItem objects, thus the Creator pattern suggests that Sale is a good candidate to have the responsibility of creating SalesLineItem instances.

This assignment of responsibilities requires that a *makeLineItem* method be defined in Sale.



Creator: discussion

- The 'basic rationale' behind Creator pattern is to find a creator that needs to be connected to the created object in any event.
- Thus assigning it 'creating responsibility' supports low coupling
- Composite objects are good candidates for creating their parts
- Sometimes you identify a creator by looking for the class that has the initialization data that will be passed to constructor during creation.
- This in fact is an application of Expert pattern.
- For example, *Payment instance* needs to be initialized, when created with the *Sale total.*
- Since *Sale knows the total, Sale is a candidate creator of the Payment.*

Creator: discussion (cont'd)

- Benefits
  - Low coupling is supported, which implies lower maintenance dependencies and higher opportunities for reuse.

- Contradictions
  - Often, creation is a complex design issue involving many contradicting forces
  - In these cases, it is advisable to delegate creation to a helper class called a *Factory*.
  - GoF patterns contain many factory patterns that may inspire a better design for creation.

The nine GRASP Patterns

- Creator
- Information Expert
- High Cohesion
- Low Coupling
- Controller
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

Information Expert

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

Information Expert: discussion

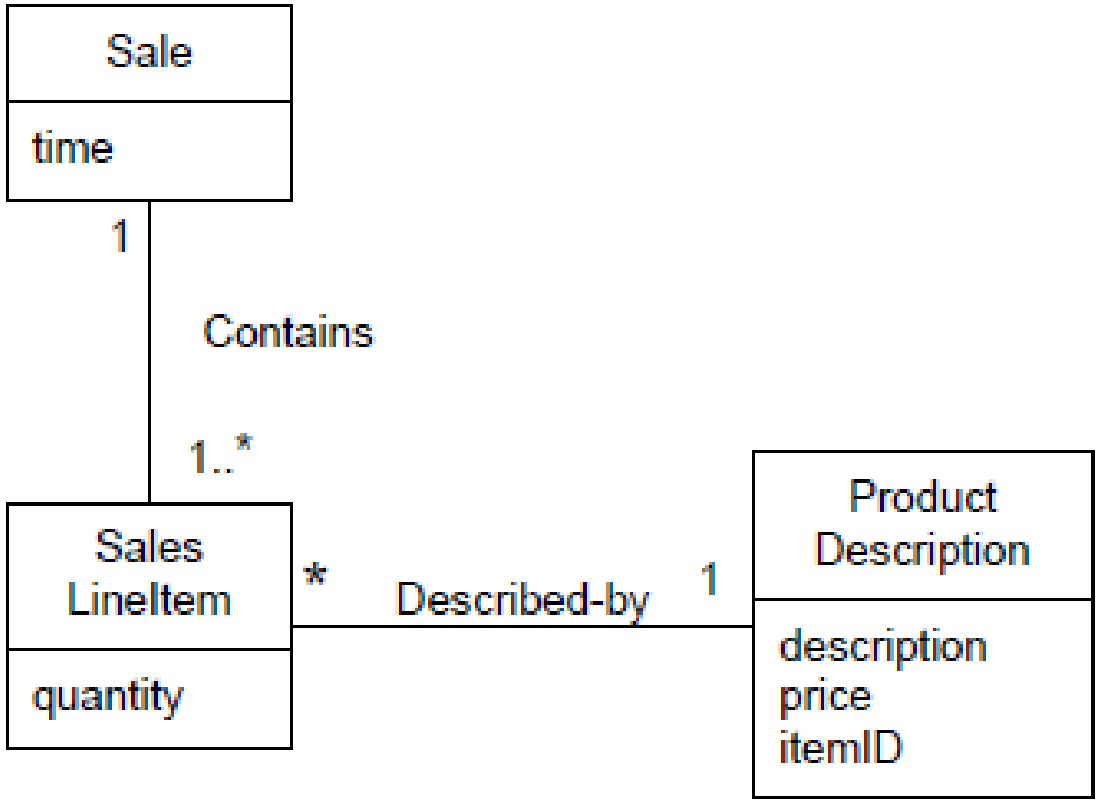
- Question
  - Do we look at the Domain Model or the Design Model to analyze the classes that have the information needed?
  - Domain model illustrates conceptual classes, design model software classes
- Answer
  - If there are relevant classes in the Design Model, look there first.
  - Otherwise, look in the Domain Model, and attempt to use (or expand) its representations to inspire the creation of corresponding design classes

How to apply the pattern

- Start by clearly stating the responsibility:
  - "Who should be responsible for knowing the total of a sale?
- Apply "Information Expert" pattern…
- Assume we are just starting design work and there is no Design Model or a minimal one, therefore
  - Search the Domain Model for information experts; the real-world Sale is a good candidate.
  - Then, add a software class to the Design Model similarly called *Sale, and give it the responsibility of knowing* its total, expressed with the method named *getTotal*.
- *CRC cards*

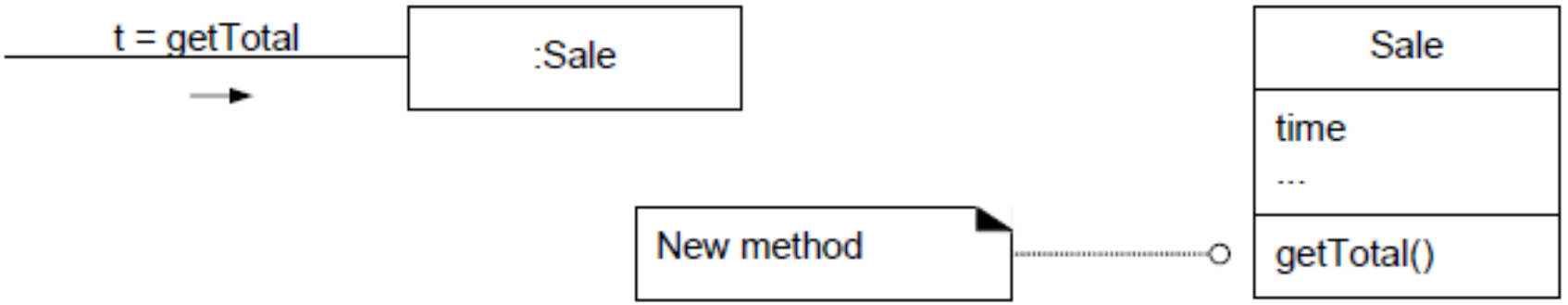
Example

- Consider the following partial Domain Model



Discussion

- What information is needed to determine the grand total?
  - It is necessary to know about all the SalesLineItem instances of a sale and the sum of their subtotals.
- A Sale instance contains these; therefore,
  - by the guideline of Information Expert, Sale is a suitable class of object for this responsibility.

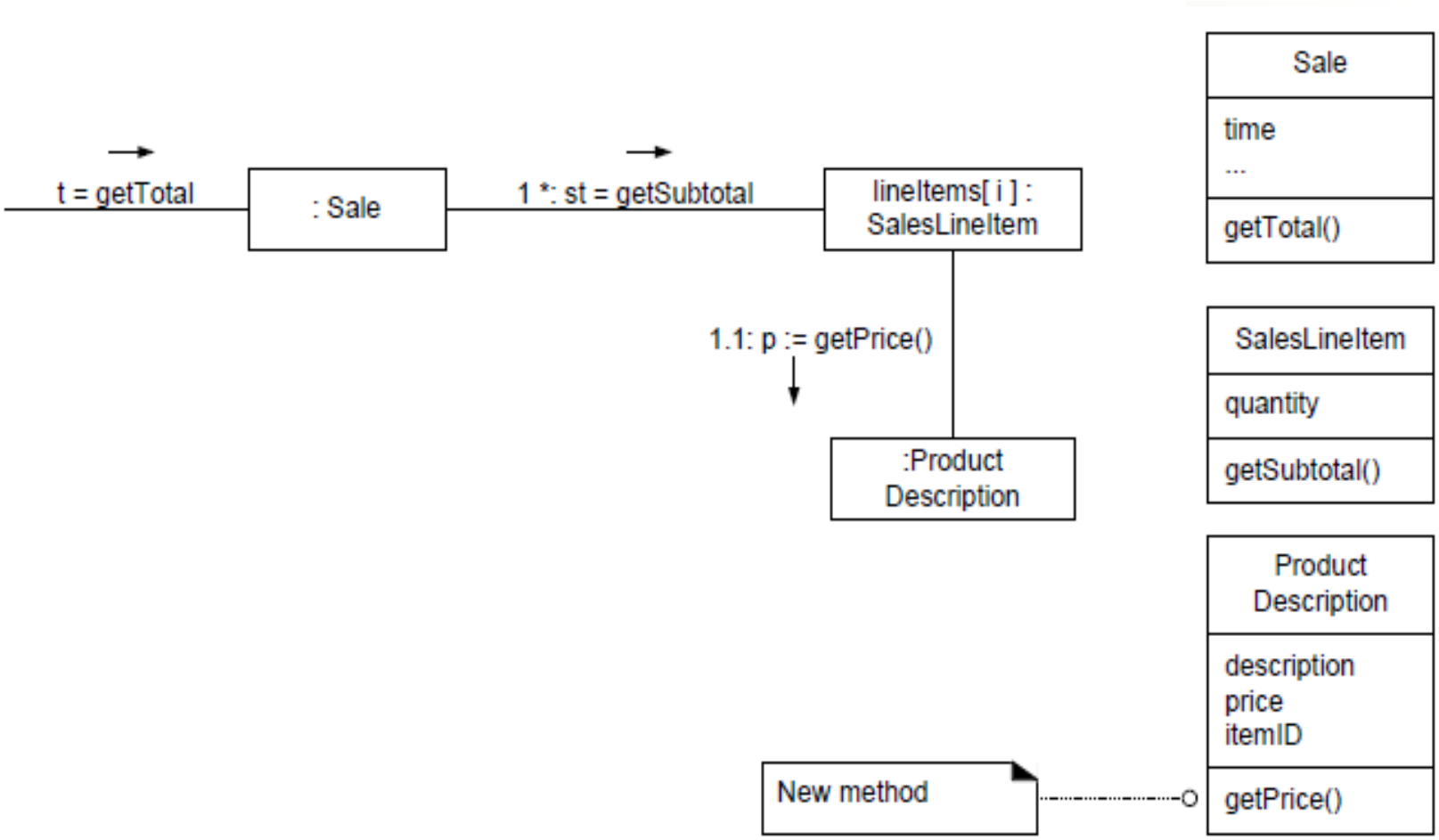


A cascade of responsibilities

- What is needed to determine the line item subtotal?
  - by Expert, *SalesLineItem should determine the subtotal*
  - To fulfill this responsibility, a *SalesLineItem needs to know the* product price.
  - By Expert, the *ProductDescription is an information expert on* answering its price
- In conclusion, to fulfill the responsibility of knowing and answering the sale's total, three responsibilities were assigned to three design classes:

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

The assigned responsibilities illustrated with a collaboration diagram.



Information Expert: discussion

- Information Expert is a basic guiding principle used continuously in object design.
- The fulfillment of a responsibility often requires information that is spread across different classes
  - This implies that there are many "partial" information experts who will collaborate in the task.
  - Different objects will need to interact via messages to share the work.
- The Information Expert should be an early pattern considered in every design unless the design implies a controller or creation problem, or is contraindicated on a higher design level.

Information Expert: Benefits

- Information encapsulation is maintained, since objects use their own information to fulfill tasks.
  - This usually supports low coupling.
- Behavior is distributed across the classes that have the required information,
  - thus encouraging cohesive "lightweight" class definitions that are easier to understand and maintain

Information Expert: Contradictions

- In some situations a solution suggested by Expert is undesirable, because of problems in coupling and cohesion.
- For example, who should be responsible for saving a *Sale in a* database?
  - If Sale is responsible, then each class has its own services to save itself in a database. The *Sale class must now contain* logic related to database handling, such as related to SQL and JDBC.
  - This will raises its coupling and duplicate the logic. The design would violate a separation of concerns – a basic *architectural design goal.*
- Thus, even though by Expert there could be justification on object design level, it would result a poor architecture design

The nine GRASP Patterns

- Creator
- Information Expert
- Low Coupling
- High Cohesion
- Controller
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

Low Coupling

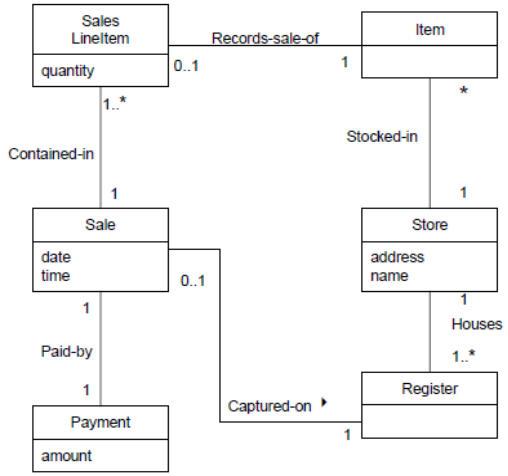
- Problem
  - How to support low dependency, low change impact, and increased reuse?
- Solution
  - Assign a responsibility so that coupling remains low.
- Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements.
- An element with low (or weak) coupling is not dependent on too many other elements

Low Coupling: discussion

- A class with high (strong) coupling suffers from the following 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.

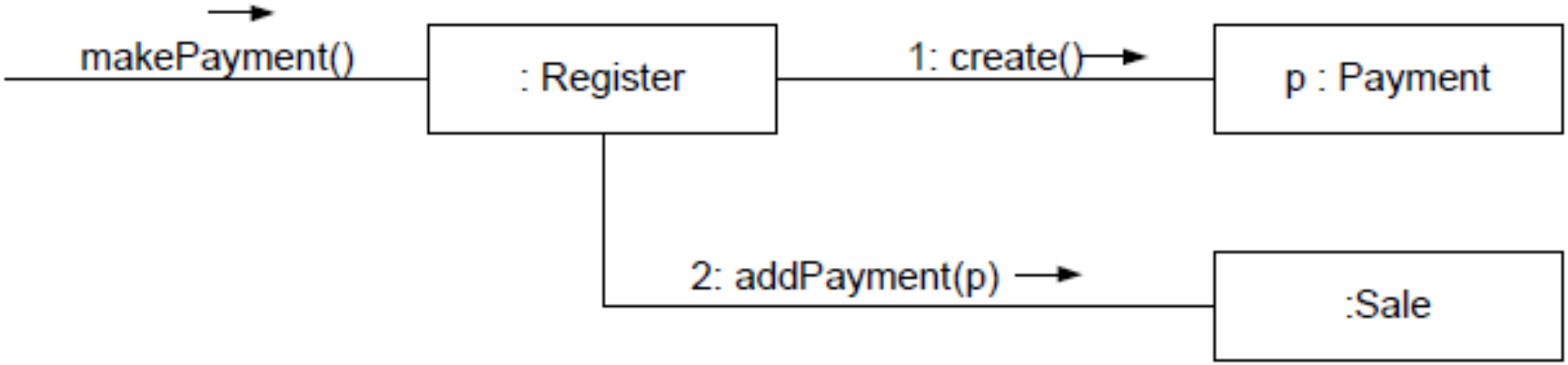
Example

- We need to create a *Payment instance and associate it with Sale*.
- What class should be responsible for this?
- Since *Register* "records" a *Payment*, the Creator pattern suggests *Register* as a candidate for creating the *Payment*.



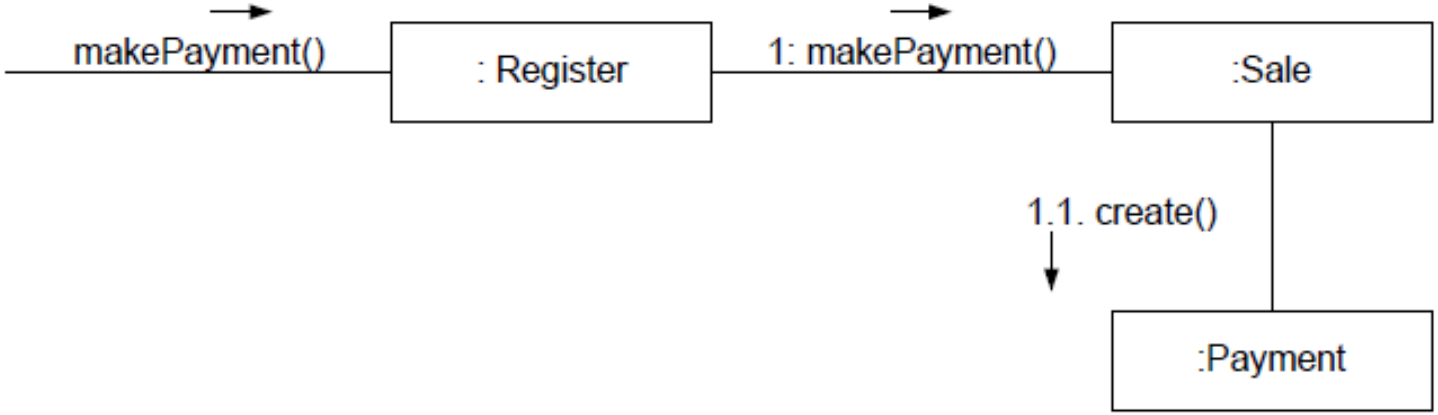
- The *Register instance could then send an addPayment* message to the *Sale, passing along the new Payment as a* parameter.

Unnecessary high coupling



- This assignment of responsibilities couples the *Register class to knowledge of the Payment* class.
- Register is also coupled to Sale, as it will be in any design solution. This hints us of another solution, according to low coupling pattern

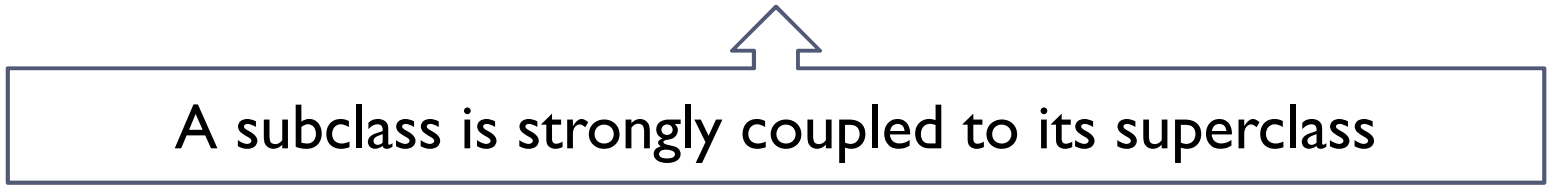
Low coupling solution



- Two patterns suggested different designs. This is very common. Creating a design is balancing contradicting forces.
- In practice, the level of coupling alone can't be considered in isolation from other principles such as Expert and Creator. Nevertheless, it is one important factor to consider in improving a design.

Discussion

- In OO languages common forms of coupling from *Type X to TypeY include:*
  - *X has an attribute (data member or instance variable) that* refers to a *Y instance, or Y itself.*
  - A *X object calls on services of a Y object.*
  - *X has a method that references an instance of Y, or Y itself, by any means. E.g. a* parameter or local variable of type *Y, or the object* returned from a message being an instance of *Y.*
  - *X is a direct or indirect subclass of Y.*
  - *Y is an interface, and X implements that interface.*



Discussion (cont'd)

- It is not high coupling per se that is the problem; the problem is high coupling to elements that are *unstable* in some dimension, such as their interface, implementation or presence.
- Coupling to stable or pervasive elements is seldom a problem
- Pick your battles
- Focus on the points of realistic high instability or future evolution
- Encapsulate the variability
- Low coupling between variable part and rest of the system

The nine GRASP Patterns

- Creator
- Information Expert
- Low Coupling
- High Cohesion
- Controller
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

High Cohesion

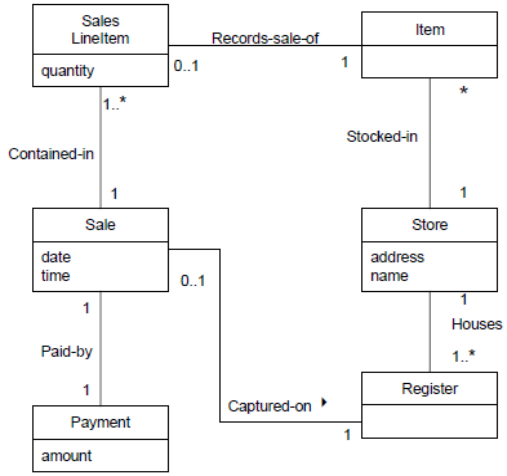
- Problem (one of them)
  - How to keep complexity manageable?
- Solution
  - Assign a responsibility so that cohesion remains high.
- Cohesion (or more specifically, functional cohesion)
  - Is a measure of how strongly related and focused the responsibilities of an element are.
  - An element with highly related responsibilities, and which does not do a tremendous amount of work, has high cohesion.
  - A class with low cohesion does many unrelated things, or does too much work.

Low Cohesion Problems

- A class with low cohesion suffers from the following problems:
  - Hard to comprehend (understand)
  - Hard to reuse
  - Delicate; constantly affected by change.
  - Hard to maintain
- Low cohesion classes have taken on responsibilities that should have been delegated to other objects.

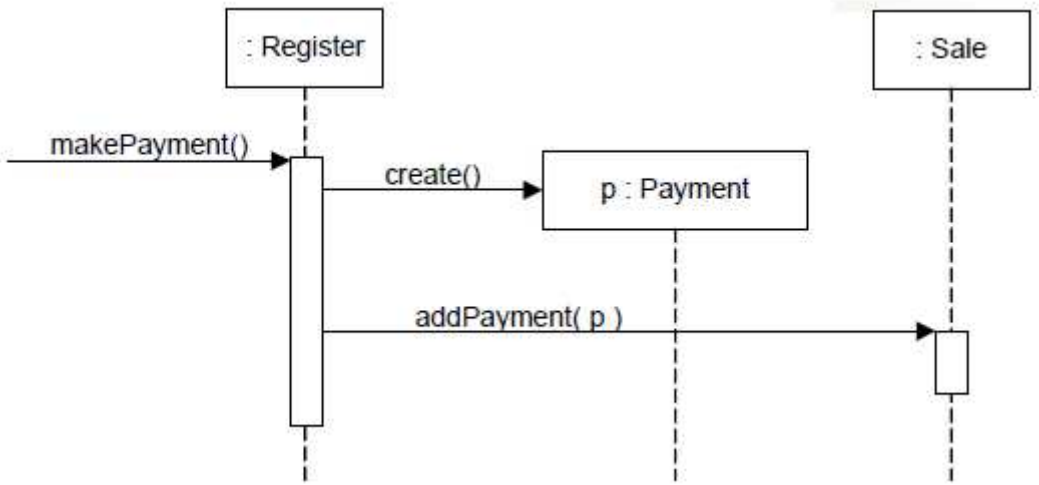
Example

- We need to create a *Payment instance and associate it with Sale*.
- What class should be responsible for this?
- Since *Register* "records" a *Payment*, the Creator pattern suggests *Register* as a candidate for creating the *Payment*.



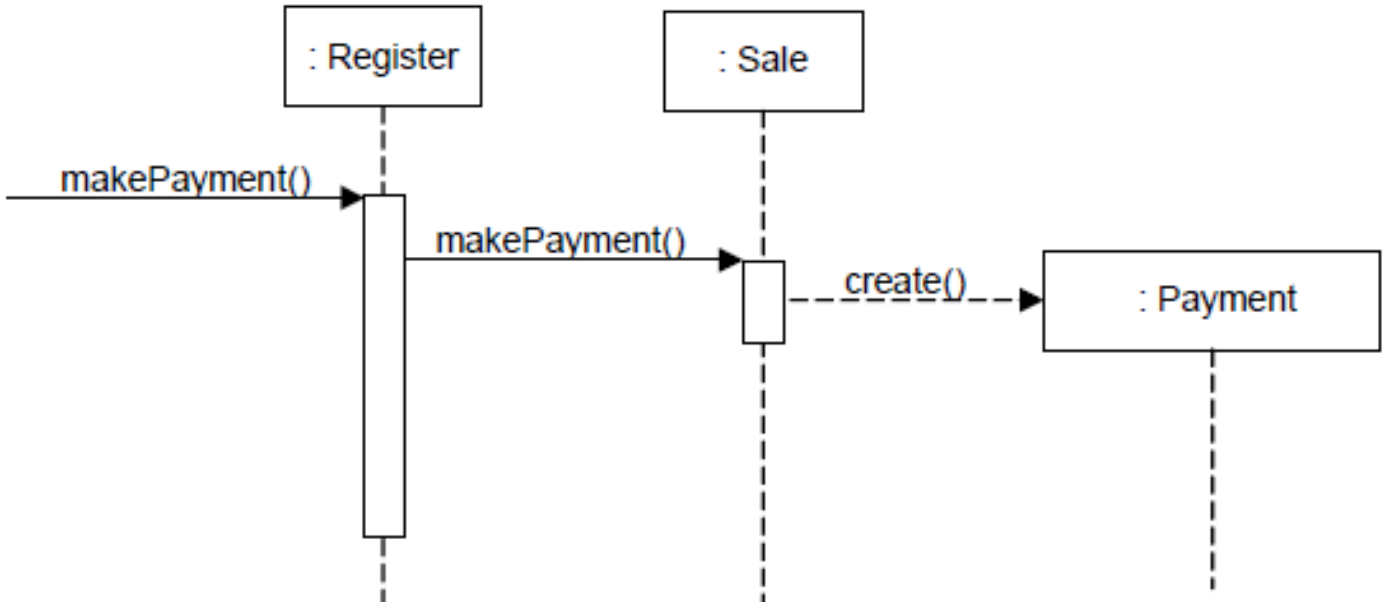
- The *Register instance could then send an addPayment* message to the *Sale, passing along the new Payment as a* parameter.

Suggested (wrong) Solution



- This places part of the responsibility for making a payment in the *Register. This is acceptable in isolated ex.*
  - However, if we continue to make the *Register class responsible* for doing some or most of the work, assigning it more system operations, it will become incohesive.

A better Solution



- This design delegates the payment creation responsibility to the *Sale, which supports higher cohesion in register.*
- This design supports both high cohesion and low coupling and is desireable.

Discussion

- Like Low Coupling, High Cohesion is a principle to keep in mind during all design decisions
  - It is important to evaluate design constantly with respect to these principles, regardless of the design result.
- Cohesion Benefits
  - Clarity and ease of comprehension of the design is increased.
  - Maintenance and enhancements are simplified.
  - Low coupling is often supported.
  - The fine grain of highly related functionality supports reuse

Discussion (cont'd)

- There are cases in which lower cohesion is justified.
  - to simplify maintenance by one person.
    - E.g. if there is only one or two SQL experts know how to best define and maintain this SQL.
  - If performance implications associated with remote objects and remote communication
  - As a simple example, instead of a remote object with three fine-grained operations *setName, setSalary, and setHireDate, there is one remote operation setData which* receives a set of data. This results in less remote calls, and better performance.

Suggested biblio

- Chapter 16 of Applying UML and Patterns, Craig Larman

Homework

Il sistema di fatturazione di una società di distribuzione di energia elettrica si basa sull'interazione tra diversi soggetti. Questo documento, destinato all'attenzione del responsabile sviluppo software, presenta un'analisi del problema basata su informazioni ottenute da interviste effettuate negli ultimi due mesi. La società di distribuzione (Società) eroga energia elettrica a utenti collegati all'impianto di distribuzione mediante un allacciamento controllato da un dispositivo elettronico di misura (contatore) in grado di registrare il consumo di energia, di fornire a richiesta della centrale di bassa potenza il valore della lettura corrente dei consumi, di segnalare eventuali malfunzionamenti nell'impianto elettrico installato presso l'utenza. La centrale di bassa potenza periodicamente raccoglie le letture e le invia al sistema di fatturazione della Società che provvede al calcolo dell'importo relativo al consumo registrato dal contatore e all'emissione di una bolletta o fattura. Il calcolo dell'importo è effettuato considerando il regime fiscale applicato a ogni utente (esente IVA o soggetto a IVA) che prevede, quando applicabile, un'imposizione determinata da un'aliquota (aliquota IVA, attualmente pari al 20%). Inoltre, nel rispetto della normativa vigente, il calcolo dell'importo deve considerare una soglia (cosiddetta di consumo sociale) sotto la quale deve essere applicata una tariffa ridotta (costo unitario sociale). Per consumi che superano questa soglia, l'importo deve essere calcolato applicando la tariffa piena (costo unitario). Una volta emessa, la bolletta è spedita al domicilio dell'utente che può provvedere al pagamento presso un qualsiasi ufficio postale (Posta). Nel caso in cui l'utente abbia domiciliato le bollette presso un istituto bancario, la bolletta è inviata anche all'istituto (Banca) che provvede automaticamente al suo pagamento alla data di scadenza indicata. In questo caso, la bolletta inviata all'utente è stampata in modo che non possa essere pagata presso un ufficio postale. Un utente può essere intestatario di più contratti, ognuno associato a un contatore. Periodicamente, la Società provvede alla verifica dei pagamenti delle bollette, mediante un processo di accreditamento. I pagamenti effettuati presso la Banca e la Posta sono incrociati con le bollette emesse. In casi di morosità grave, la Società si riserva il diritto di sospendere l'erogazione dell'energia elettrica.