GRASP Patterns

An introduction to object design in a easily understandable way.

These slides are motivated by “Applying UML and Patterns” by Craig Larman. Many of the pictures are taken from the web resources provided by the book web site.

Contents

GRASP Patterns

- GRASP stands for ***G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns*
- 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.
- GRASP patterns do not address architectural design

What is object design

- A simple definition (too simple☺):
  - 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.

A critical step in object design

- Central tasks in design are:
  - 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

What are 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.

Granularity of a responsibility

- 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.*
  - The *Sale* may collaborate with other objects, such as sending *a getSubtotal* message to each *SalesLineItem* object asking for its subtotal.

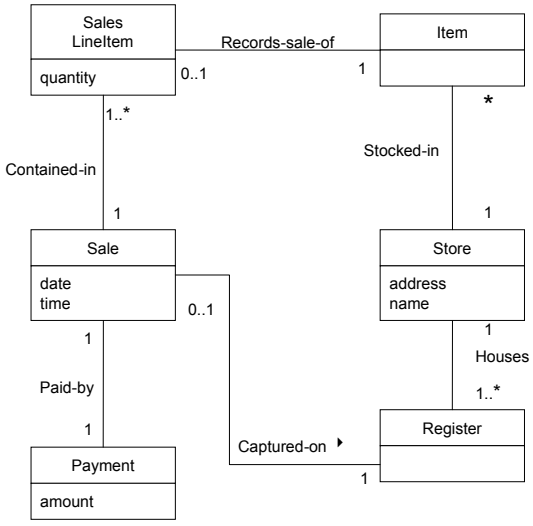
GRASP – learning and doing Basic Design

- 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 Interaction Diagrams

- Responsibilities are assigned to objects during object design while creating interaction diagrams.
  - Sequence diagrams
  - Collaboration diagrams
- Examples:
- "a *Sale* is responsible for creating *SalesLineItems"* (a doing), or
- "a *Sale* is responsible for knowing its total" (a knowing).

Example – domain model of a point of sale application

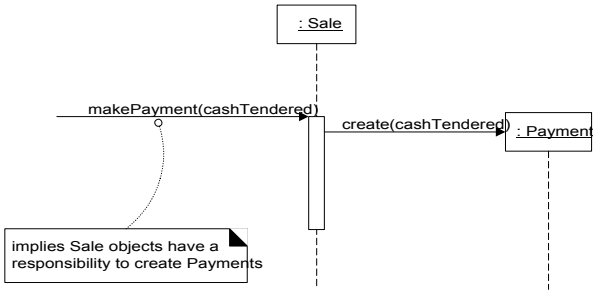


Responsibilities and Interaction Diagrams

- 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 GRASP Patterns

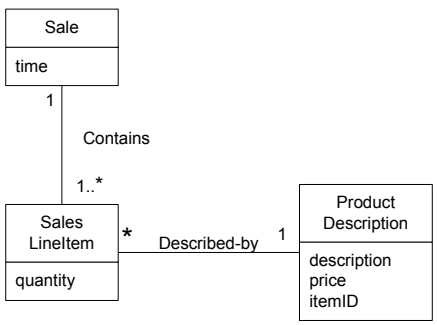
- **In this slideset: fourfirst GRASP patterns**
  - ***Creator***
  - ***Information Expert***
  - ***High Cohesion***
  - ***Low Coupling***
  - ***Controller***

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* that will be passed to A when it is created.

Example

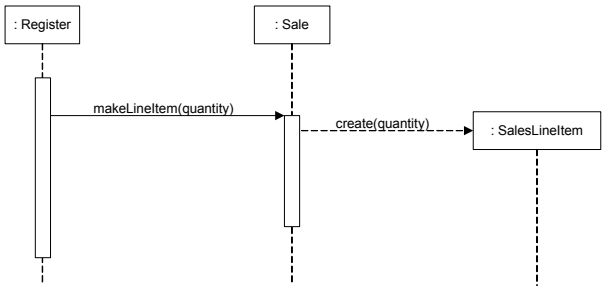
- Who should be responsible for creating a *SalesLineItem* instance?
- Applying Creator, we look for a class that aggregates, contains, and so on, *SalesLineItem* instances.
- Consider the following partial domain model:



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.



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 excellent 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.*

Contradictions and Benefits

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

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: A key question

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

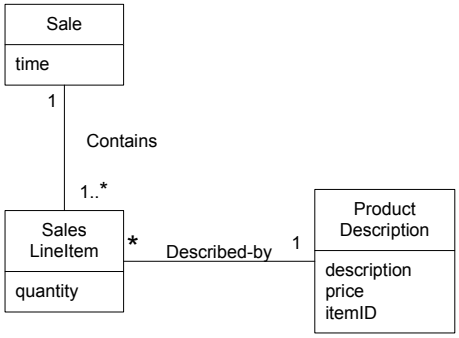
1. If there are relevant classes in the Design Model, look there first.
2. Otherwise, look in the Domain Model, and attempt to use (or expand) its representations to inspire the creation of corresponding design classes.

Example on applying

- 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 or a minimal Design Model, 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.*

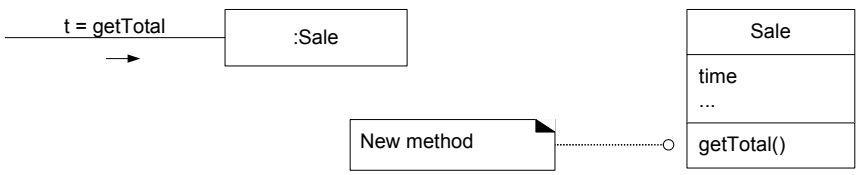
Example

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

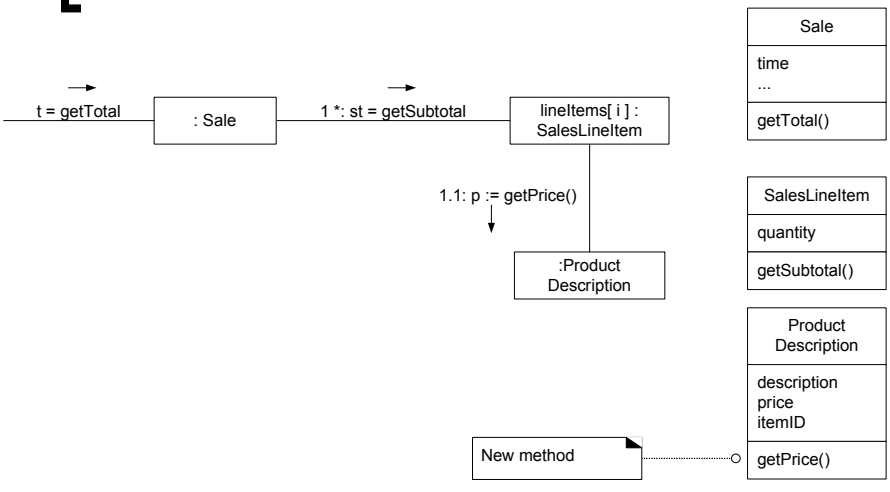


A cascade of responsibilities

- What information 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.



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 of objects.
  - 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.

Contradictions

- In some situations a solution suggested by Expert is undesirable, usually 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.

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.

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

High coupling – too much reliance on other classes

- 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.

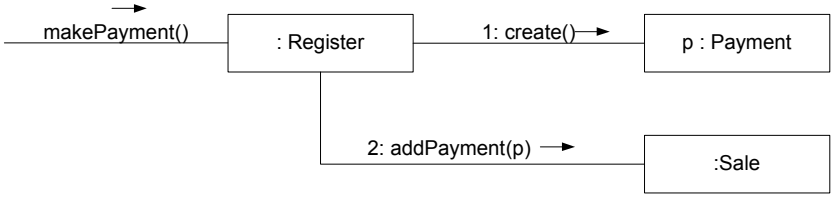
Example

- A partial class diagram:

- We need to create a *Payment* instance and associate it with *Sale.*
- What class should be responsible for this?
- 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

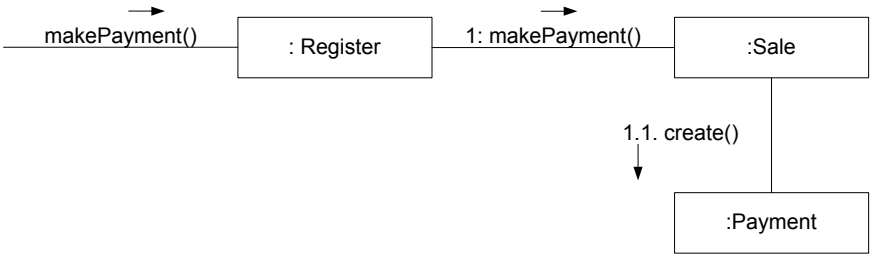
Corresponding collaboration diagram



- 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

A better solution (?) for creating the *payment* is shown below:



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 object-oriented languages common forms of coupling from *TypeX* to *TypeY* include:
  - *TypeX* has an attribute (data member or instance variable) that refers to a *TypeY* instance, or *TypeY* itself.
  - A *TypeX* object calls on services of a *TypeY* object.
  - *TypeX* has a method that references an instance of *TypeY,* or *TypeY* itself, by any means. These typically include a parameter or local variable of type *TypeY,* or the object returned from a message being an instance of *TypeY.*
  - *TypeX* is a direct or indirect subclass of *TypeY.*
  - *TypeY* is an interface, and *TypeX* implements that interface.
- A subclass is strongly coupled to its superclass. The decision to derive from a superclass needs to be carefully considered since it is such a strong form of coupling.

Contraindications

- 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

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

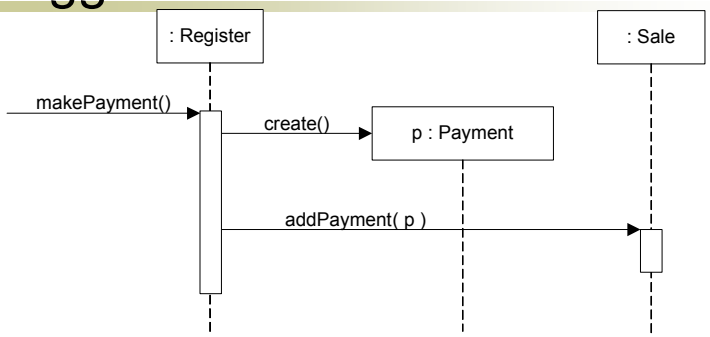
- A class with low cohesion suffer from the following problems:
  - Hard to comprehend and understand
  - Hard to comprehend (understand)
  - Hard to reuse
  - Delicate; constantly affected by change.
  - Hard to maintain
- Low cohesion classes have taken on responsibilities that should have been delegated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

Example

- Consider the same example used with coupling:

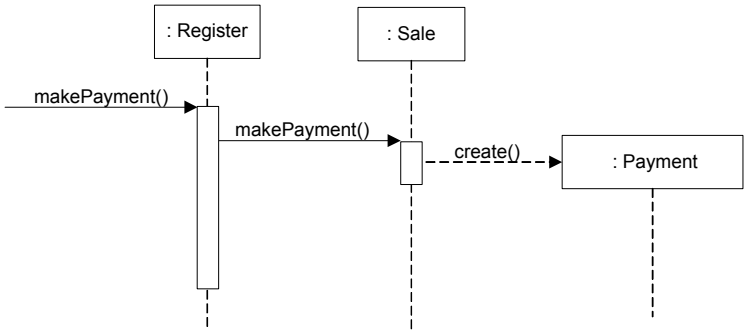
- We need to create a *Payment* instance and associate it with *Sale*.
- What class should be responsible for this?
- Since a *Register* "records" a *Payment* in the real-world domain,
  - the Creator pattern suggests *Register* as a candidate for creating the *Payment*.
- The *Register* instance could then send an *addPayment* message to the *Sale*, passing along the new *Payment* as a parameter.

Suggested Solution



- This places part of the responsibility for making a payment in the *Register*. This is acceptable in isolated example.
- 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.
- There are different degrees of cohesion:
- 1. *Very low cohesion*
- 2. *Low cohesion*
- *3. High cohesion*
- 4. *Moderate cohesion*

Very Low Cohesion

- A class is solely responsible for many things in very different functional areas.
- Assume a class exists called *RDB-RPC-Interface* which is completely responsible for interacting with relational databases and for handling remote procedure calls.

Low cohesion

- A class has sole responsibility for a complex task in one functional area.
- Assume a class called *RDBInterface* which is completely responsible for interacting with relational database**s**.
- The methods of the class are all related, but there are lots of them
- The class should split into a family of lightweight classes sharing the work to provide RDB access.

High cohesion

- A class has moderate responsibilities in one functional area and collaborates with other classes to fulfill tasks.
- Assume a class called *RDBInterface* which is only partially responsible for interacting with relational databases.

Moderate cohesion

- A class has lightweight and sole responsibilities in a few different areas that are logically related to the class concept, but not to each other.
- Assume a class called *Company* which is completely responsible for
  - knowing its employees and
  - knowing its financial information.
- These two areas are not strongly related to each other, although both are logically related to the concept of a company. In addition, the total number of public methods is small.

rule of thumb

When there are alternative design choices take a closer look at the cohesion and coupling implications of the alternatives and possibly a the future evolution pressures on the alternatives. Choose an alternative with good cohesion, coupling and stability.

Contraindications

- Bad cohesion usually begets bad coupling, and vice versa.
- There are cases in which accepting 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.

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 increased reuse