GRASP Patterns (intro)

Arthur Molnar

Babes-Bolyai University

arthur@cs.ubbcluj.ro

November 21, 2020

Overview

1. GRASP Patterns
   - High Cohesion
   - Low Coupling
   - Information Expert
   - GRASP Controller
   - Protect Variation
   - Creator
   - Pure Fabrication
2. GRASP Patterns in layered architecture
3. Domain-driven Design (intro)
   - Entities
   - Value Objects
   - Aggregates
   - Data Transfer Objects

GRASP Patterns

What are they?

General Responsibility Assignment Software Patterns (or **Principles**) consists of guidelines for assigning responsibility to classes and objects in object oriented design.

- The answer to **how** is layered architecture that we've "encouraged ☺" you to use in your assignments.
- The answer to **why** are these patterns.
- Their main goal is to make software easy to understand (by humans) and easy to change (by humans).

High Cohesion

Main idea

Create functions, classes and modules so that they have a single, well defined responsibility.

High Cohesion

- **High Cohesion** - attempts to keep objects focused, manageable and understandable.
- High cohesion means that the responsibilities of a given element are strongly related and highly focused.
- Breaking programs into classes and subsystems is an example of activities that increase the cohesive properties of a system.
- Low cohesion is a situation in which an element has too many unrelated responsibilities. Elements with low cohesion often suffer from being hard to comprehend, hard to reuse, hard to maintain and adverse to change

Low Coupling

Main idea

Minimize the dependencies between functions, classes and modules. A function call is a good example of a dependency.

Low Coupling

Low Coupling dictates how to assign responsibilities to support

- Low impact in a class when the source code for other classes is changed
- Good potential for reusing already written and tested code
- Good code readability by avoiding spaghetti code

Low Coupling

Forms of coupling:

- **TypeX** has an attribute (field) that refers to a **TypeY** instance, or **TypeY** itself.
- **TypeX** has a method which references an instance of **TypeY**, or **TypeY** itself, by any means. (parameter, local variable, return value, method invocation)
- **TypeX** is a direct or indirect subclass of **TypeY**.

Information Expert

Main idea

How do I decide where the code for a functionality goes.

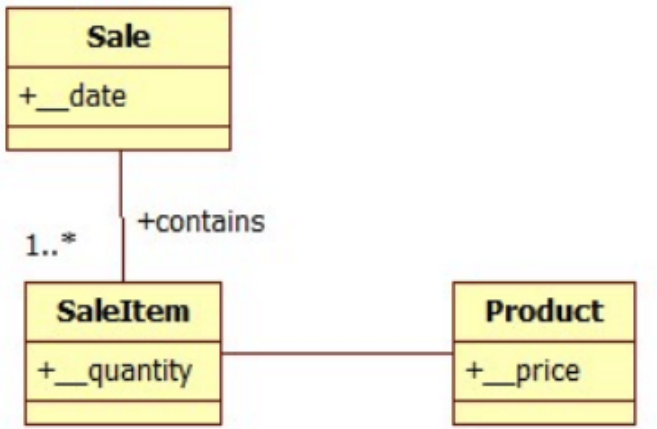
Information Expert

Assign a responsibility to the class that has the information necessary to fulfill the responsibility.

- Used to determine where to delegate responsibilities. These responsibilities include methods, computed fields and so on.
- Assign responsibilities by looking at a given responsibility, determine the information needed to fulfil it, and then figure out where that information is stored.
- Information Expert leads to placing responsibility on the class with the most information required to fulfil it

Information Expert

Point of Sale application



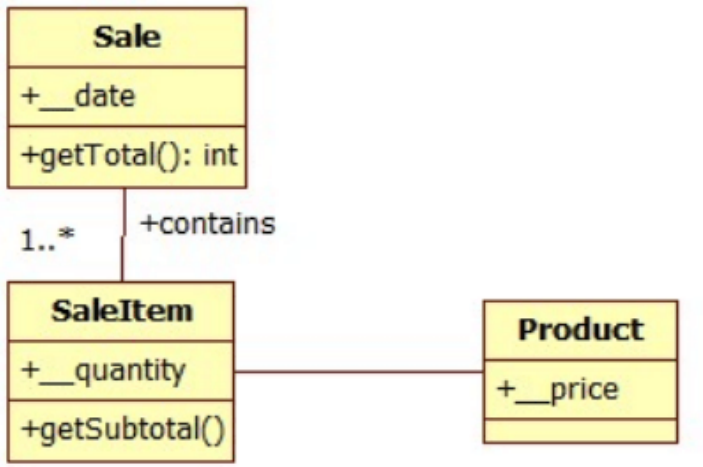
Who is responsible with computing the total?

Wee need all the SaleItems to compute the total.

Information Expert → **Sale**

Information Expert

Point of Sale application



According to the Expert

SaleItem should be responsible with computing the subtotal (quantity * price)

Information Expert

Point of Sale application

1. Maintain encapsulation of information
2. Promotes low coupling
3. Promotes highly cohesive classes
4. Can cause a class to become excessively complex

GRASP Controller

Main idea

Create a class that has a method for each user action. The first layer below the UI, its job is to *control* how the application fulfills required functionalities.

GRASP Controller

- Decouple the event sources (usually the application UI) from the objects that actually handle the events.
- It is the first object beyond the UI layer that receives and *controls* system operation.
- The controller should delegate to other objects the work that needs to be done

Protect Variation

Main idea

Create a class where I can control allowed object variations.

Protected Variations

- Make sure current or future variations in the system do not cause major problems with system operation.
- Create new classes to encapsulate such variations.
- The protect variation pattern protects elements from the variations on other elements (objects, systems, subsystems) by wrapping the focus of instability to a separate class.

Protected Variations

Task: validate student, possible validation designs

- Class member function in Student that returns true/false
- Static function returning the list of errors
- Separate class that encapsulate the validation algorithm

Validator class

The protected variations pattern protects elements from variations on other elements (objects) by wrapping the focus of instability to a separate class

Creator

Main idea

How to decide who is responsible for creating domain entity objects.

Creator

- Object creation is one of the most common activities in an object-oriented system.
- Deciding who is responsible for this determines the relations between classes.
- Also, on non garbage-collected platforms, who is responsible for destroying objects? (object ownership)

Creator

- A class **B** should be responsible for creating instances of class **A** if one, or preferably more, of the following apply:
  - Instances of B contains or compositely aggregates instances of A
  - Instances of B record instances of A
  - Instances of B closely use instances of A
  - Instances of B have the initializing information for instances of A and pass it on creation.

Pure Fabrication

Main idea

Create a class that represents a persistent object store .

Pure Fabrication

- Assign a highly cohesive set of responsibilities to an artificial class that does not represent anything in the problem domain, in order to support high cohesion, low coupling, and reuse
- **Problem**: store **Student** (in memory, file or database)
  - Information expert says that *Student* is the "expert" to perform this operation
  - However, adding this functionality to *Student* means that we have to change domain entities when we want to update how we store domain entities.
  - We extract this functionality, breaking the information expert pattern to achieve good cohesion and coupling (and keep the force in balance ☺)

Layered architecture

Layer	**Main ideas**
All	High cohesion (each layer, class has a single, well-defined responsibility), low coupling (dependencies between layers and their classes are reduced and made clear)
User Interface	"Thin", contains as little functionality as possible
Controller	Application logic, the GRASP controller, uses the creator pattern
Domain	Entities from program domain
Validators	Protect variation, separate into its own class
Repository	Pure fabrication, responsible for storing domain entity objects

Entities

Entity

An object that is not defined by its attributes, but rather by a thread of continuity and its identity.

- If an object represents something with continuity and identity, it is something that is tracked through different states (or even across different implementations) it is an entity
- Usually has a correspondent in the real world

Entities

- Attributes of the entity may change but the identity remain the same
- Mistaken identity can lead to data corruption.
- Define what it means to be the same thing (e.g. if two objects have the same *type* and *id*)

Value Objects

Value Object

It describes a characteristic. It has no identity.

- An object that represents a descriptive aspect of the domain with no conceptual identity
- When you care only about the attributes of an element of the model:
  - *Address* is a good candidate for a value object; it is entirely determined by its attributes.
  - *Money* is another good example; each sum of equal value in the same currency are equal.
  - *Recipe ingredients* might be a good example; you can use any actual ingredients, as long as they are the right type and quantity.

Entities vs. Value Objects

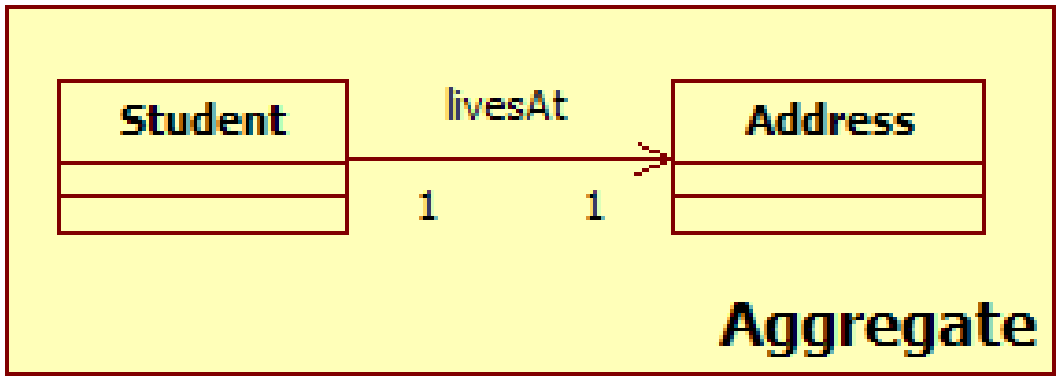
Discussion

- **Student** is an entity
- **Address** is a value object

Why isn't Student a value object?

Aggregates

- Cluster the **entities** and **value objects** into **aggregates** and define boundaries around them.
- Choose one **entity** to be the root of each aggregate, and control access to the objects inside the boundary using the root.
- Allow external objects to hold references to the root only.
- **e.g.** - only *StudentRepository*, **NOT** *AddressRepository*.



Data Transfer Objects

- **Data Transfer Objects (DTO)** are object used to carry data between processes.
- In the case where communication between processes is expensive (e.g. over the Internet), it makes sense to bundle up the data and send it in one go.
- DTO's have no behaviour, they only contain data, so should not require testing

NB!

Since our programs do not employ processes, we are not using DTO's exactly as intended. However, in real life you will find application layers on different machines/architectures.