UML. Design Principles.

Arthur Molnar

Babes-Bolyai University

arthur@cs.ubbcluj.ro

January 20, 2018

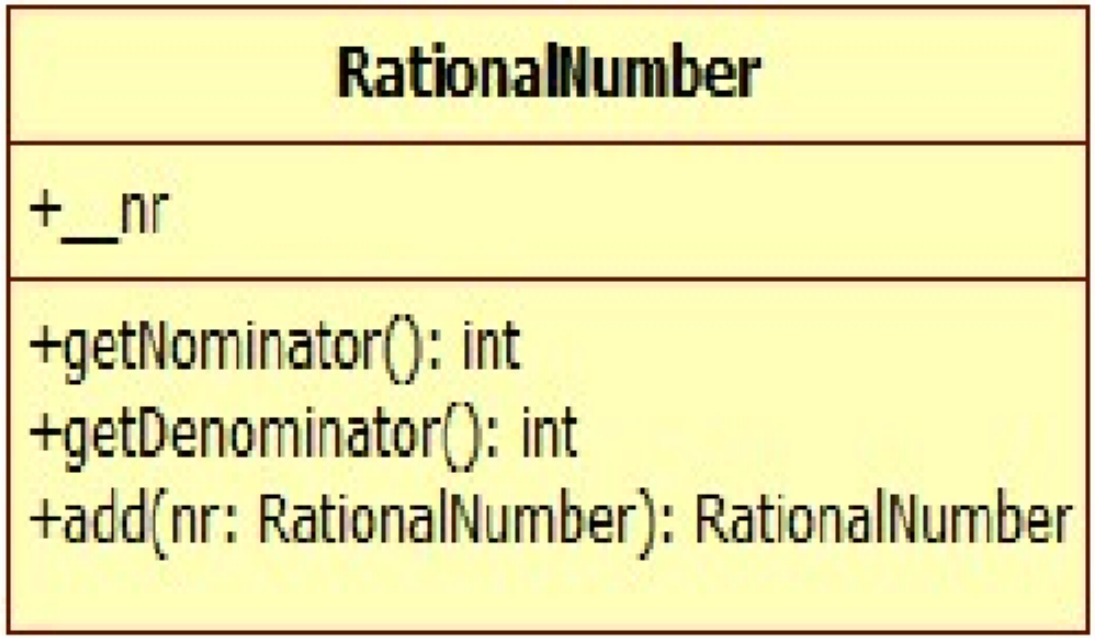
Overview

UML Diagrams

- **Unified Modeling Language (UML)** - a standardized general-purpose modeling language in the field of object-oriented software engineering.
- UML includes a set of graphic notation techniques to create visual models of object-oriented software.

Class Diagrams

UML Class diagrams - describe the structure of a system by showing the system's classes, their attributes, and the relationships between them.



Class Diagrams


```
class Rational:
    def __init__(self,a,b):
        '''
        Initialize a rational number
        a,b integers
        '''
        self.__nr = [a,b]
    def getDenominator(self):
        '''
        Denominator getter
        '''
        return self.__nr[1]
    def getNominator(self):
        '''
        Nominator getter
        '''
        return self.__nr[0]
```

Class Diagrams

In the diagram classes are represented using boxes which contain three parts:

- Upper part holds the name of the class
- Middle part contains the attributes of the class
- Bottom part contains the methods or operations

Relationships

- A relationship is a general term covering the specific types of logical connections found on class diagrams.
- A *Link* is the basic relationship among objects. It is represented as a line connecting two or more object boxes.

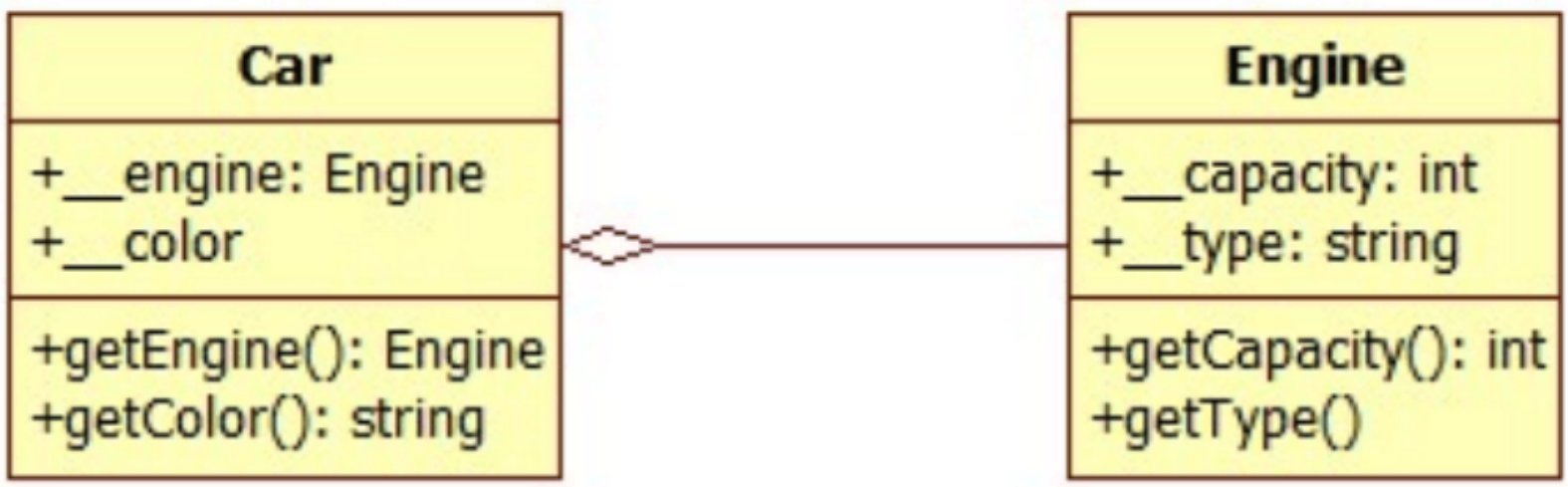
Associations

Binary associations (with two ends) are normally represented as a line, with each end connected to a class box.

An association can be named, and the ends of an association can be annotated with role names, ownership indicators, multiplicity, visibility, and other properties. Association can be Bi-directional as well as uni-directional.

Aggregation

Aggregation - an association that represents a part-whole or part-of relationship.



Aggregation

Aggregation - an association that represents a part-whole or part-of relationship.

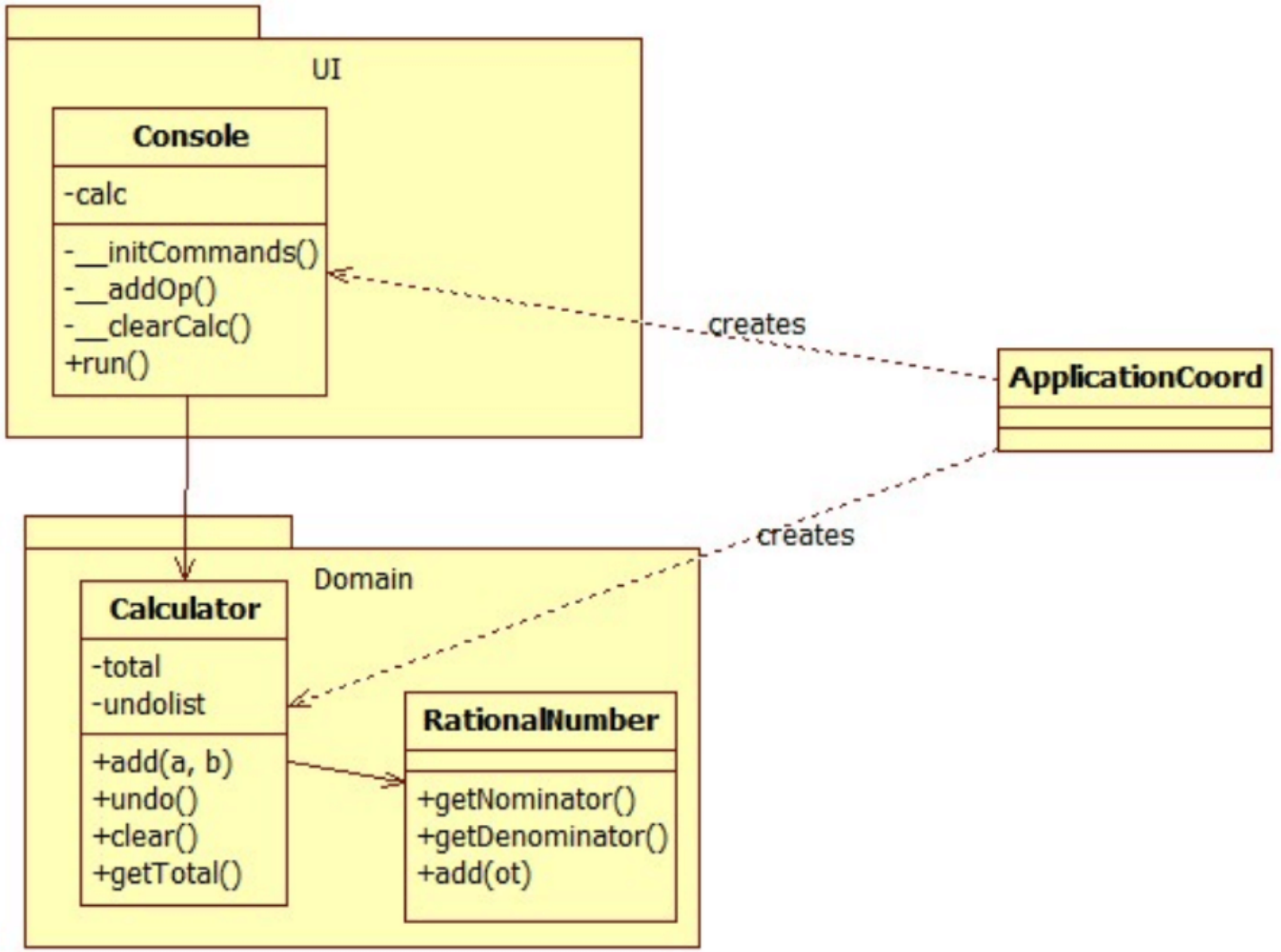
```
class Car:
    def __init__(self, eng, col):
        '''
        Initialize a car
        eng - engine, col - string, i.e 'white'
        '''
        self.__eng = eng
        self.__color = col
class Engine:
    def __init__(self, cap, type):
        '''
        Initialize the engine
        cap - positive integer, type - string
        '''
        self.__capacity = cap
        self.__type = type
```

Dependency, Package

Dependency - a relationship in which one element, the client, uses or depends on another element, the supplier

- Create instances
- Have a method parameter
- Use an object in a method

Dependency, Package



Design principles

Create software:

- Easy to understand, modify, maintain, test
- Classes - abstract, encapsulate, hide implementation, easy to test, easy to reuse

General scope: **managing dependency**

- Single responsibility
- Separation of concerns
- Low Coupling
- High Cohesion

Layered architecture

- **Layer** - a logical structuring mechanism for the elements that make up your software solution
- A multilayered software architecture is using different layers for allocating the responsibilities of an application.
- A layer is a group of classes (or modules) that have the same set of module dependencies to other modules and are reusable in similar circumstances.

Layered architecture

Layers our programs will use...

- **User Interface Layer** (aka View Layer, UI layer or Presentation layer)
- **Application Layer** (aka Service Layer or GRASP Controller Layer)
- **Domain layer** (Business Layer, Business logic Layer or Model Layer)
- **Infrastructure Layer** (data access or other persistence, logging, network I/O e.g. sending emails, and other kind of technical services)

GRASP patterns

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

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

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

Low Coupling dictates how to assign responsibilities to support:

- Low dependency between classes;
- Low impact in a class of changes in other classes;
- High reuse potential

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**.

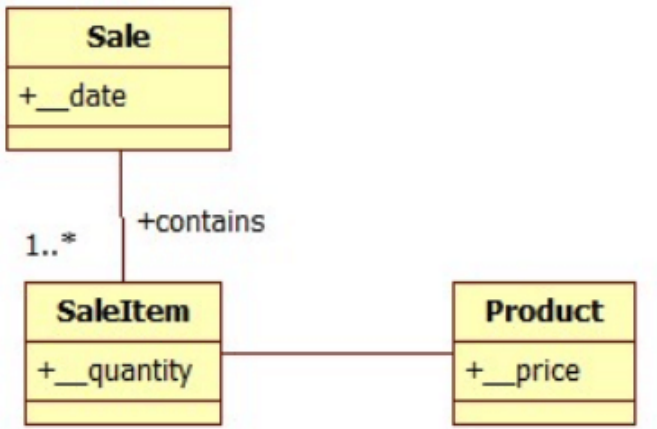
Information Expert

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

- **Information Expert** is a principle 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



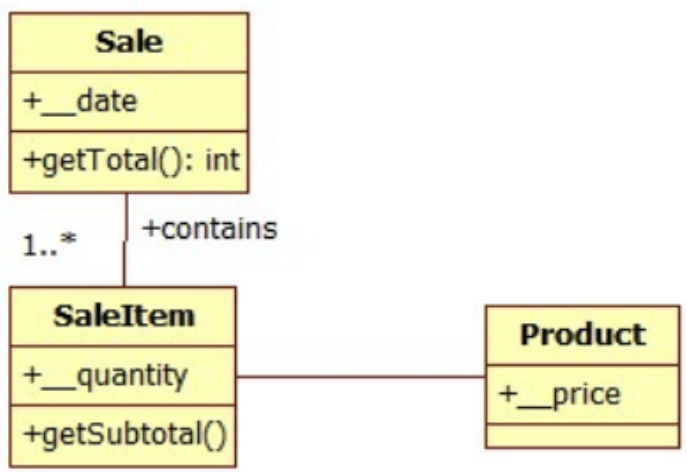
Who is responsible with computing the total?

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

Creator

- Creation of objects is one of the most common activities in an object - oriented system. Which class is responsible for creating objects is a fundamental property of the relationship between objects of particular classes.

Creator

- **Creator** pattern is responsible for creating an object of the class. In general, 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.

Protected Variations

- How responsibilities should be assigned in such a fashion that the current or future variations in the system do not cause major problems with system operation and/or revision?
- Create new classes to encapsulate such variations.
- The **protected variations** pattern protects elements from the variations on other elements (objects, systems, subsystems) by wrapping the focus of instability to a separate class. (with an interface and using polymorphism to create various implementations of this interface).

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

Protected Variations

Demo

Validate student - **ex19_studentValidator.py**

Pure Fabrication

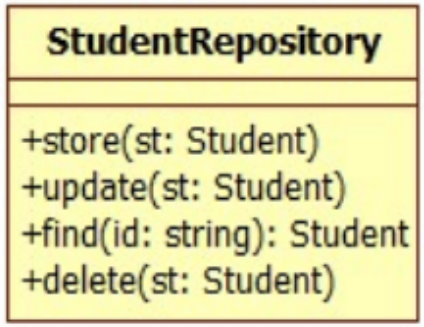
- When an expert violates high cohesion and low coupling
- Assign a highly cohesive set of responsibilities to an artificial class that does not represent anything in the problem domain, in order to support high cohesion, low coupling, and reuse
- **Pure Fabrication** - a class that does not represent a concept in the problem domain is specially made up to achieve low coupling, high cohesion

Pure Fabrication

- Problem: Store **Student** (in memory, file or database)
- Expert pattern - Student is the "expert" to perform this operation
- What about when changing the store (e.g from file to database)

Pure Fabrication - Repository

- Problem: Store **Student** (in memory, file or database)
- **Expert** pattern - Student is the "expert" to perform this operation. But putting this responsibility into the Student class will result in low cohesion, poor reuse
- Solution - **Pure Fabrication**



Class created with the responsibility to store Students

The Student class easy to reuse, has High cohesion, Low coupling

Repository will deal with the problem of managing a list o students (persistent storage)

Repository Pattern

A **repository** represents all objects of a certain type as a conceptual set. Objects of the appropriate type are added and removed, and the machinery behind the REPOSITORY inserts them or deletes them from persistent storage.

Repository Pattern

Demo

Student repository - **ex20_studentRepository.py**

GRASP Controller

- Decouple the event source(s) from the objects that actually handle the events.
- **Controller** is defined as the first object beyond the UI layer that receives and coordinates ("controls") a system operation.
- The controller should delegate to other objects the work that needs to be done; it coordinates or controls the activity. It should not do much work itself.
- Controller encapsulate knowledge about the current state of a use case presentation layer decoupled from problem domain

Task: create controller

Demo

Student controller - **ex21_studentController.py**

Application coordinator

- **Dependency injection** - a design pattern in object-oriented computer programming whose purpose is to reduce the coupling between software components.
- Frequently an object uses (depends on) work produced by another part of the system.
- With **DI**, the object does not need to know in advance about how the other part of the system works. Instead, the programmer provides (injects) the relevant system component in advance along with a contract that it will behave in a certain way

Assemble everything


```
validator = StudentValidator()
#Inject the validator into the repo
repo = InMemoryRepository(validator)
#Inject repository into controller
ctrl = StudentController(repo)
ui = Console(ctrl)
ui.showUI()
```