Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

An introduction to the object oriented design

Software Architecture

Juan Carlos Cruellas

Bibliography

Basic. Used throughout all the set of slides:

[1] Larman, Graig. ”Applying UML and Patterns”. Third edition. Prentice Hall PTR.

Introduction. Some reminders

- ULTIMATE GOAL: To achieve well-designed, robust, reusable, and maintainable software system, which brings value to the users.
- Activities for obtaining the OO software grouped in three main types:
  - **ANALYSIS** (DO THE RIGHT THING -INVESTIGATION): Activities leading to obtain a COMPLETE, ACCURATE, AND NON AMBIGUOUS DESCRIPTION OF THE PROBLEM TO BE SOLVED, as well as A CLEAR UNDERSTANDING AND DETAILED DESCRIPTION OF WHAT HAS TO DO THE SYSTEM FOR SOLVING IT, AND WHAT IS REQUIRED TO SOLVE IT.
  - **DESIGN** (DO THE THING RIGHT): Activities leading to define **HOW THE SYSTEM HAS TO SOLVE THE PROBLEM**. This includes to completely specify:
    - Classes that will be part of the software
    - Their relationships
    - The objects and their required interactions for solving the problem
  - **Implementation**: Activities leading to build the actual program using programming language(s).

Design. Responsibilities-Driven Design (RDD)

- Popular approach to OO Design: think in terms of responsibilities of the different objects, roles and collaborations between objects: Responsibilities-Driven Design (RDD).
- Under this approach, **objects have responsibilities** (contracts, or obligations), **and collaborate with other objects** for achieving goals that satisfy users needs.
- Identifying and distributing the responsibilities among the different classes is a key issue in the OO Design process.

Design. Responsibilities-Driven Design (RDD)

- Types of responsibilities of an object:
  - DOING:
    - something by itself, or
    - initiating actions in other objects, or
    - controlling/coordinating activities in other objects.
  - KNOWING either
    - about private encapsulated data, or
    - about related objects, or
    - about things that it can derive or compute.
- "Size" and complexity of responsibilities: from big and complex, which might involve complex classes with a significant number of methods, to small and simple, which may require few methods (even one in some cases).
- RESPONSIBILITY != METHOD !!!!

Design. Responsibilities-Driven Design (RDD)

- "KNOWING" responsibilities: can be discovered using the Model Domain (classes are connected by associations, for instance).
- ”DOING" responsibilities: more complex skills required.
- Our goal for this term is LEARNING RULES/PRINCIPLES FOR PROPERLY ASSIGNING RESPONSIBILITIES TO CLASSES.
- Larman: General Responsibilities Assigning Patterns (GRASP).
- Patterns or Rules/Principles?. This should not worry us too much. What is relevant is to use them for properly assigning responsibilities.

Design. General Responsibilities Assignment Patterns

- Pattern in OO Design is a **NAMED tuple** formed by:
  - One well known and general problem
  - One well known and good solution to that problem applicable in several contexts
  - Advice on how to apply the solution
  - Discussion on its trade-offs, implementations, and variations.
- Naming patterns helps in building design knowledge and terminology which helps designers to communicate that knowledge and saves lot of time in design working sessions.

Design. General Responsibilities Assignment Patterns

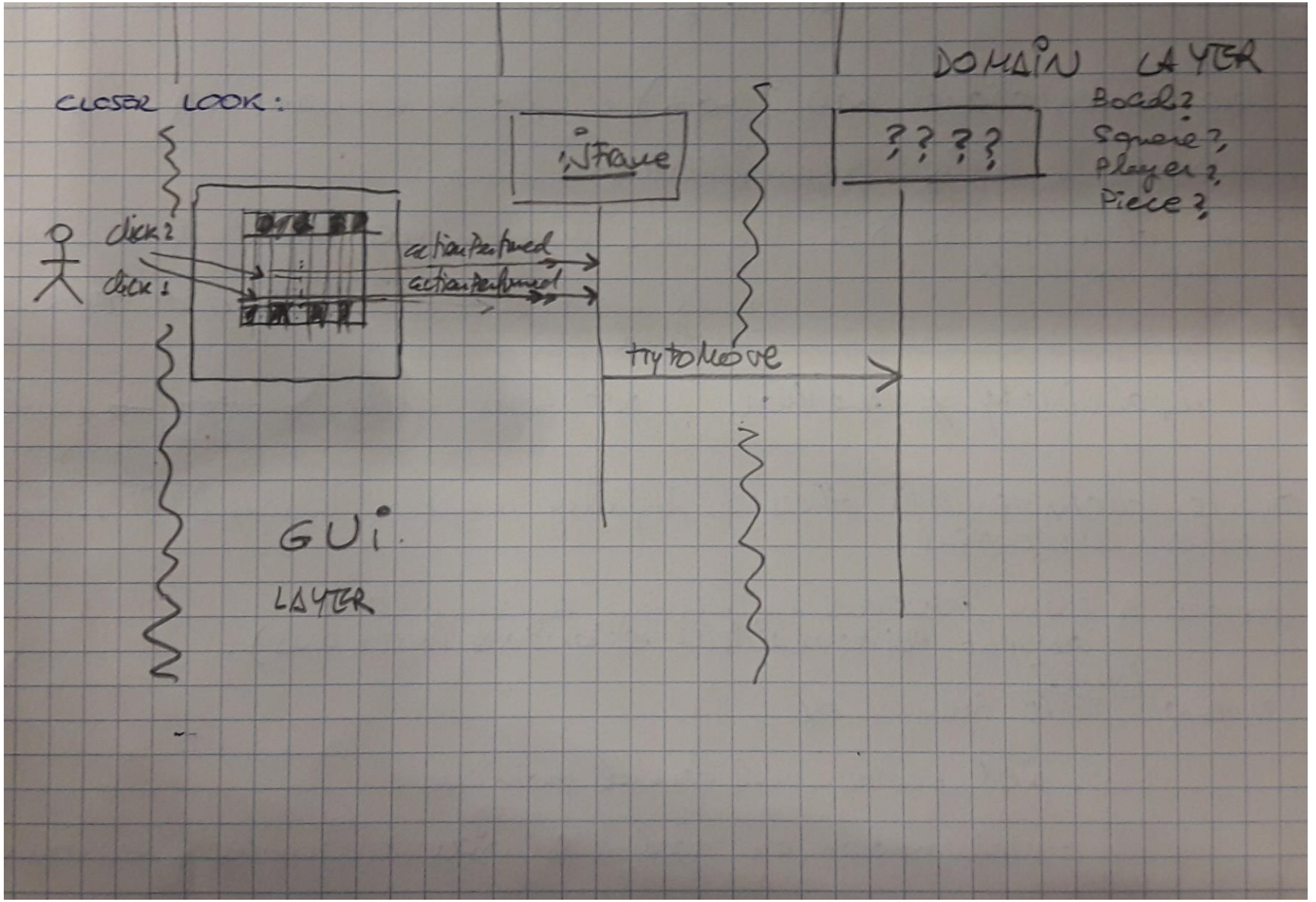
- Pattern in OO Design is a **NAMED tuple** formed by:
  - One well known and general problem
  - One well known and good solution to that problem applicable in several contexts
  - Advice on how to apply the solution
  - Discussion on its trade-offs, implementations, and variations.
- Naming patterns helps in building design knowledge and terminology which helps designers to communicate knowledge and saves lot of time in design working sessions.

GENERAL RESPONSIBILITIES ASSIGNMENT PATTERNS (GRASP)

GENERAL RESPONSIBILITIES ASSIGNMENT PATTERNS (GRASP)

- Controller
- Creator
- Information Expert (or Expert)
- Polymorphism
- Pure fabrication
- Low coupling
- High cohesion

First approach to the environment



GRASP: Controller

- **PROBLEM**: What object, beyond the UI, receives and coordinates (controls) a system operation?. The object with this responsibility is known as CONTROLLER.
- **ANSWER**: Assign the responsibility to a class that represents ONE OF THE FOLLOWING CHOICES:
  - IT REPRESENTS THE "OVERAL" SYSTEM, A ROOT OBJECT, A DEVIDE THAT THE SOFTWARE IS RUNNING WITHIN ("Phone", for instance) OR A MAJOR SUBSYSTEM, OR
  - IT REPRESENTS A USE CASE SCENARIO WITHIN WHICH THE SYSTEM EVENT OCCURS. If this is the case, add the suffix Handler, Coordinator, Session to the name of the class, AND use the same controller for ALL THE SYSTEM EVENTS IN THE SAME USE CASE.

GRASP: Controller

- FIRST ALTERNATIVE (only one controller) is suitable when there are not many system events or when the UI can not select among different controllers.

- SECOND ALTERNATIVE (one controller per use case) is suitable with the number and types of responsibilities is such that one controller becomes not cohesive and highly coupled (BLOATED).

GRASP: Controller

- Discussion:
  - UI SHOULD NOT contain application logic.
  - Controller acts as the façade into domain layer.
  - Using the same controller for all the system events of one use case allows to maintain information about the STATE of the use case.
  - Controller SHOULD DELEGATE to other objects in the domain layer the work to be done. IT COORDINATES THE ACTIVITY.
  - It SHOULD NOT HAVE TOO MANY RESPONSIBILITIES. If so, NOT COHESIVE AND HIGHLY COUPLED!!

GRASP: Controller

- Trade-offs.
  - Benefits:
    - allows reuse and pluggable UIs.
    - allows management of state of the use case.
  - Bloated controller symptoms:
    - One controller receiving all (being many) the system events.
    - Controller performing many tasks needed for fulfilling the system events.
    - Controller having many attributes and keeping big amount information of the system, which should be distributed in other objects.
- Chess program: Class Game

GRASP: Creator

- **PROBLEM**: Who should create the objects in the framework?. The object with this responsibility is known as CREATOR.
- **ANSWER**: Assign the responsibility of creating objects of a certain class A to a certain class B if one of the following situations is true:
  - B contains or aggregates A.
  - B records A.
  - B closely uses A
  - B has the initializing data for A.

GRASP: Creator

- Discussion:
  - The idea is assigning the responsibility of creating objects to other objects that are connected to the created objects. This decreases the coupling. Containers, recorders, owners of initializing data are examples of coupled objects to the created ones.
  - In a good range of real, complex problems, the creation of objects is a rather complex issue, which can not be properly solved using the Creator rule/pattern. Instead, more complex and sophisticated Design Patterns (Factory Method, Abstract Factory, for instance), are required.

GRASP: Creator

- Chess program. Initial proposal
  - Game creates:
    - The two players (closely uses Player)
    - The Board (the game is played on the Board)
    - Trace
  - Player creates the pieces (player owns the pieces)
  - Board creates Square (Board is an aggregation of squares)

GRASP: Information Expert (Expert)

- **PROBLEM**: What is a general principle for assigning responsibilities to objects?.
- **ANSWER**: Assign the responsibility to the class that has the information necessary to fulfil the responsibility.
- Discussion:
  - Rather intuitive: objects do things related to the information they have.
  - Not uncommon situation: there is no one single object with all the information required for a certain task; instead, it is distributed among several objects, which become "partial" experts. Then they will have to collaborate for performing the task.
  - Designs where objects do operations that are done to the things they represent!! ("animation" principle).

GRASP: Information Expert (Expert)

- Contraindications:
  - Sometimes this pattern/rule leads to bad solutions, which decrease cohesion and increase coupling.
    - Objects that have data that should be saved in a Data Base should not have the responsibility of doing it!! It would result in non cohesive objects and would be coupled to technical database services!!.
- Benefits:
  - Maintain information encapsulation
  - Distributes behaviour across the classes that have the required information.

GRASP: Polymorphism

- **PROBLEM**: How to assign responsibilities when behaviour depends of type?
- **ANSWER**: Assign the responsibility for the behaviour using polymorphic methods to the types for which the behaviour varies.
- Discussion:
  - Not using polymorphism leads to software full of if-then-else / switch-case based on type. **Bad choice** for managing new types with different behaviours. **Polymorphism only implies a new class** with an implementation of the polymorphic method: much easier to extend; better prepared for change.
  - Polymorphism implies NOT TESTING the type before running the behaviour.

GRASP: Polymorphism

- If there is not a behaviour by default, then make the polymorphic method in the superclass, abstract.

- Chess example: At least two type-dependant behaviours:
  - Check if a certain movement does correspond to a specific piece. No behaviour by default.
  - Check if between the origin square to the destination square there is a path free for the piece in the origin square.

GRASP: Pure fabrication

- **PROBLEM**: How to assign a certain responsibility when none of the solutions offered by Expert keeps coupling low and cohesion high enough?
- **ANSWER**: Assign the responsibility to an artificial convenience class, not representing any domain concept so that it is highly cohesive and as low coupled as possible.
- Discussion:
  - These classes are fabrication of the designers.
  - Programs where in addition to representational decomposition (Document, VAT, Postal address, Email address, bank account…) the behavioural decomposition (assign responsibilities by grouping behaviours/algorithms –ComponentChecker for instance, VATChecker, emailAddChecker, BankAccountChecker) are not uncommon.
  - Checkers are pure fabrication objects, proposed by behavioural decomposition. Components are domain objects.

GRASP: Indirection

- **PROBLEM**: How to assign a certain responsibility for avoiding direct coupling between two o more things?
- **ANSWER**: Assign the responsibility to an intermediate object (adapter) that will mediate between other components or services so that they are not directly coupled.
- Discussion:
  - Used in situations where there are interactions with external systems, for instance.
  - Other example: adapters for persistent storage (at the same time, pure fabrication)
  - Used with polymorphism provide consistent interface to inner objects and hide variations in external APIs.
  - Some design patterns are specializations of indirection: Adapter, Façade, Observer

GRASP: Protected Variations

- **PROBLEM**: How to design objects, subsystems, and systems so that the variations or instability in these elements does not impact on other elements?
- **ANSWER**: Identify points of predicted variations or instability and assign responsibilities to create a stable interface (this means functionality expressed in terms of set of methods, not necessarily a Java interface) around them.
- Discussion:
  - Protecting variations (managing well coupling) is a fundamental problem in OOD.
  - OOD principles and patterns deal with this problem.

GRASP: Low Coupling and High Cohesion

- Sometimes the solutions suggested by the Patterns/Rules for direct assignment of responsibilities lead to several alternatives.
- This does not mean that all of them are equally good.
- Two Patterns/Rules will help us in finding the most suited solution:
  - LOW COUPLING (with how many classes one class is connected). The less coupled classes are to others, the less they depend on the other, the less impact suffer when the other change, and the most reusable they are in other systems
  - HIGH COHESION (how many different responsibilities a class has). The most cohesive a class is, the most reusable it is, and the easier is to maintain.
- Low coupling and high cohesion are signs of WELL-DESIGNED SOFTWARE!.

GRASP: Low Coupling

- **PROBLEM**: How to support low dependency, low change impact, and increase reuse?.
- **ANSWER**: Assign the responsibility so that the coupling is kept as low as possible in elements that are considered to be unstable (subject to change).
- Discussion:
  - Common forms of coupling between classes A and B:
    - A has a reference to B
    - A calls services provided by B
    - A has methods that reference B (arguments, local variables, or returned result)
    - A is a direct or indirect subclass of B
    - A is an interface and B implements that interface

GRASP: Low Coupling

- It is high coupling among classes that are NOT STABLE (i.e. subjected to change) what is problematic, not high coupling per se.
- Coupling among very stable elements is OK.
  - For instance, our software is very strongly coupled to elements that are delivered in Java Development Kit (List, Map, Set, etc)!!!!, and there is no problem at all with this!!!.
- Warning about inheritance:
  - Inheritance implies a strong coupling between classes. It MUST BE CAREFULLY CONSIDERED during design. Bad designs abuse of inheritance.
  - Some Design Patterns, in fact, replace inheritance by composition.
  - BUT, yet, inheritance is certainly required and worth to use in OO software.

GRASP: High Cohesion

- **Cohesion of a class**: class whose responsibilities are highly related and does not do a very big amount of work, has high cohesion.
- **PROBLEM**: How to keep objects focused, understandable, and not very difficult to manage?.
- **ANSWER**: Assign responsibility so that cohesion is kept high.
- Discussion:
  - Low cohesion produces classes that are difficult to understand, reuse, and maintain. They are also very likely to be affected by change.
  - Some degrees of cohesion:
    - VERY LOW COHESION: One class is responsible for doing many things in many different functional areas.

GRASP: High Cohesion

- LOW COHESION: One class is responsible for doing one complex task in one functional area.
- MODERATE COHESION: One class has lightweight and responsibilities in few different areas, logically related to the class concept but not to each other.
- HIGH COHESION: One class has moderately complex responsibilities in one functional area and collaborates with other classes to fulfil tasks.

- Contraindications:
  - Sometimes is OK not to have high cohesion and group responsibilities in one class for simplifying maintenance by one person.
  - Distributed server objects. Remote methods that do several tasks for decreasing network traffic.