Ingegneria del Software
Corso di Laurea in Informatica per in Management

GRASP

Davide Rossi
Dipartimento di Informatica – Scienze e Ingegneria
Università di Bologna

Responsibility-driven design

- How to use OO principles to design a software system?
- RDD is a method to design software systems on the basis of responsibilities [Rebecca Wirfs-Brock and Brian Wilkerson].
- The UML defines a responsibility as "a contract or obligation of a classifier".

Responsibility

Doing responsibilities of an object include:

- 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 of an object include:

- knowing about private encapsulated data
- knowing about related objects
- knowing about things it can derive or calculate

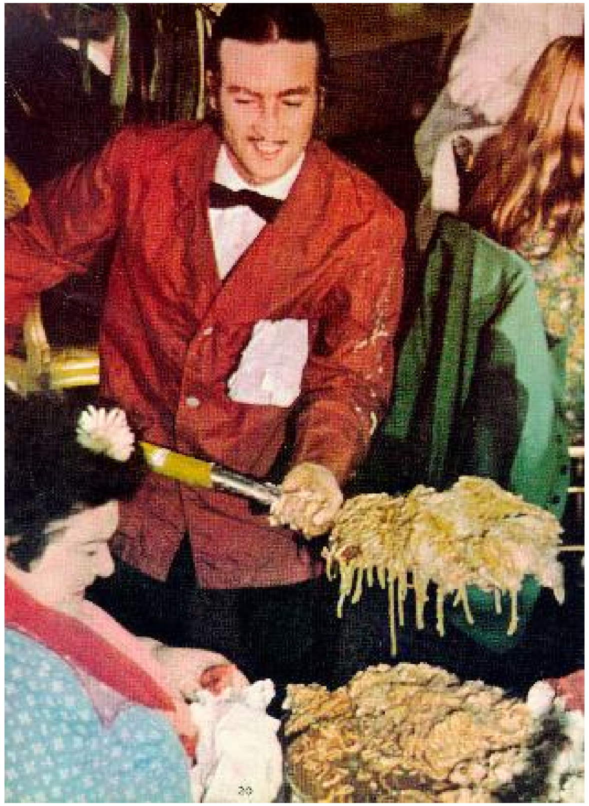
RDD

In RDD responsibilities are assigned to classes of objects during object design.

RDD

In RDD responsibilities are assigned to classes of objects during object design.

Beware: slavish assignment of responsibilities can easily lead to a design exhibiting the de-facto standard software architecture: the big ball of mud [Foote, Yoder – PLoP'97].



GRASP to the rescue

GRASP (General Responsibility Assignment Software Patterns) can be used to perform RDD while guaranteeing that solid construction principles are used.

To Larman, GRASP is a **learning aid** for OO design with responsibilities: it helps one understand essential object design and apply reasoning in a **methodical**, **rational**, **explainable** way.

Pattern

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice

[Alexander 77]

Design Pattern

- Patterns provide “workable solutions to all of the problems known to arise in the course of design”. [Beck, Cunningham - 87]
- A named description of a problem, solution, when to apply the solution, and how to apply the solution in new contexts. [Larman]
- A general reusable solution to a commonly occurring problem within a given context in software design. [wikipedia]

GRASP patterns

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

Information Expert (or Expert)

- P: How do we assigning responsibilities to objects so that our systems tend to be easier to understand, maintain, and extend, and our choices afford more opportunity to reuse components in future applications.
- S: Assign a responsibility to the information expert, the class that has the information necessary to fulfill the responsibility.

Creator

- P: who should be responsible for creating a new instance of some class?
- S: Assign class B the responsibility to create an instance of class A if one of these is true (the more the better)
  - B "contains" or compositely aggregates A
  - B records A
  - B closely uses A
  - B has the initializing data for A that will be passed to A when it is created. Thus B is an Expert with respect to creating A.

Controller

- P: Which object receives and coordinates ("controls") a system operation?
- S: Assign the responsibility to a class representing one of the following choices:
  - Represents the overall "system," a "root object," a device that the software is running within, or a major subsystem these are all variations of a facade controller.

Controller

- Represents a use case scenario within which the system event occurs, often named `<UseCaseName>Handler`, `<UseCaseName>Coordinator`, or `<UseCaseName>Session` (use case or session controller).
  - Use the same controller class for all system events in the same use case scenario.
  - Informally, a session is an instance of a conversation with an actor. Sessions can be of any length but are often organized in terms of use cases (use case sessions).

Low Coupling

- P: How to support low dependency, low change impact, and increased reuse?
- S: Assign a responsibility so that coupling remains low. Use this principle to evaluate alternatives.

High Cohesion

- P: How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?
- S: Assign a responsibility so that cohesion remains high. Use this to evaluate alternatives.

High Cohesion

A class with low cohesion does many unrelated things or does too much work. Such classes are undesirable; they suffer from the following problems:

- hard to comprehend
- hard to reuse
- hard to maintain
- delicate; constantly affected by change

Low cohesion classes often represent a very "large grain" of abstraction or have taken on responsibilities that should have been delegated to other objects.

Pure Fabrication

- P: What to do when you do not want to violate High Cohesion and Low Coupling, or other goals, but solutions offered by Expert (for example) are not appropriate?
- S: Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept something made up, to support high cohesion, low coupling, and reuse.

Indirection

- P: Where to assign a responsibility, to avoid direct coupling between two (or more) things? How to de-couple objects so that low coupling is supported and reuse potential remains higher?
- S: Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled.

Polymorphism

- P: Conditional variation by using control-flow statements produces code that is hard to extend.
- S: Use alternatives based on type. When related alternatives or behaviors vary by type (class), assign responsibility for the behavior using polymorphic operations to the types for which the behavior varies.

Protected Variations

- P: How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?
- S: Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them.

Protected Variations

- “OCP is essentially equivalent to [...] Protected Variation”. [C. Larman]
- LSP formalizes the principle of protection against variations in different implementations of an interface, or subclass extensions of a superclass.
- The “Law of Demeter” is also a special case of PV.

What is a dependency?

- A dependency signifies a supplier/client relationship between elements **where the modification of a supplier may impact the client**.
- A dependency implies that the semantics of the clients are not complete without the suppliers.

Adapted from UML specs. 2.5.1

Types of dependencies

- Use dependency
  - Example: client calls a method defined in supplies
  - Changes in supplier’s method name or parameters (number, types) impact the client
- Instantiate dependency
  - Special case of the former
- Parameter dependency
  - Example: client receives a supplier as a parameter and passes it when invoking another element’s method

Impacts of changes in the supplier

- Syntactic: the name of a called method is modified, the number of parameters is changed, …
- Semantic: the behavior activated by a method call is changed (not necessarily visible through return values, as in the case of side effects).

Dependencies and changes

- Dependencies are potential changes spreading paths
- The **criticality** of a dependency is related to the **chance** of it being a path to spread changes
  - Dependencies pointing at stable elements are less critical
  - Dependencies pointing at volatile elements are more critical

Stable and volatile elements

- All elements in a solution (classes, interfaces, …) can be stable or volatile depending on various factors
- However, we have empirical evidence that more abstract elements tend to be more stable than concrete, low level, ones

Dependencies management

- Minimization of dependencies is possible only to a given extent
- A solution with more “good” dependencies can be qualitatively much better than a solution with lesser “bad” dependencies
- We should organize our solution to minimize “bad” dependencies