# GRASP Patterns

David Duncan

November 16, 2012

## Introduction

- GRASP (General Responsibility Assignment Software Patterns) is an acronym created by Craig Larman to encompass nine object-oriented design principles related to creating responsibilities for classes
- These principles can also be viewed as design patterns and offer benefits similar to the classic “Gang of Four” patterns
- GRASP is an attempt to document what expert designers probably know intuitively
- All nine GRASP patterns will be presented and briefly discussed

## What is GRASP?

- GRASP = General Responsibility Assignment Software Patterns (or Principles)
- A collection of general objected-oriented design patterns related to assigning defining objects
- Originally described as a collection by Craig Larman in *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, 1st edition, in 1997.

## Context (1 of 2)

- The third edition of *Applying UML and Patterns* is the most current edition, published in 2005, and is by far the source most drawn upon for this material
- Larman assumes the development of some type of analysis artifacts prior to the use of GRASP
  - Of particular note, a domain model is used
    - A domain model describes the subject domain without describing the software implementation
    - It may look similar to a UML class diagram, but there is a major difference between domain objects and software objects

# Context (2 of 2)

- Otherwise, assumptions are broad: primarily, the practitioner is using some type of sensible and iterative process
  - Larman chooses the Unified Process because it is:
    - Iterative
    - Flexible and open, integrates well with agile processes
    - Popular with OOAD projects

## Responsibility-Driven Design

- GRASP patterns are used to assign responsibility to objects
- As such, their use results in a Responsibility-Driven Design (RDD) for Object Orientation (OO)
  - Contrast to (the more traditional) Data-Driven Design
- With this point of view, assigning responsibilities to objects is a large part of basic object design

## Why GRASP?

- Traditionally in Object-Oriented Programming (OOP), object design is glossed over
  - E.g., think of nouns and convert to objects; think of verbs and convert to methods
  - Or even: After requirements analysis and creation of a domain model, just create objects and their methods to fulfill requirements
    - (Oh.  ...Ok.  Poor inexperienced OO developers.)
- UML is just a language—it expresses an OO design but for the most part does not provide guidance
- Per Larman, GRASP helps one "understand essential object design and apply reasoning in a methodical, rational, explainable way."

## Design Patterns

- Software design patterns were launched as a concept in 1987 by Kent Beck and Ward Cunningham, based upon Christopher Alexander’s application in (building) architecture
- Core definition: a named description of a problem and a corresponding reusable solution
- Ideally, the pattern advises on when it should be used and the typical trade-offs
- The most famous design patterns are the 23 described by the “Gang of Four” (GoF) book in 1993

## Design Pattern Advantages

- Both the GoF patterns and GRASP patterns have notable benefits:
  - Simplifying: provides a named, generally understood building block
    - Facilitates communication
    - Aids thinking about the design
  - Accelerates learning to not have to develop concepts from scratch

## GRASP vs. GoF Patterns

- GRASP patterns are in a way even more fundamental than the GoF patterns
  - GRASP patterns are equally well referred to as principles, while the GoF patterns are rarely referred to as such
- While the naming of both types of patterns is important, it’s less important for the GRASP patterns
  - The concepts are truly what are important

## About Responsibilities

- Two types of responsibilities for objects:
  - Doing
  - Knowing
- Knowing responsibilities are often easily inspired by software domain objects
  - E.g., domain class for a Sale has a time attribute → Sale class in OO design knows its time as well
  - Result of meeting the domain model aim to have a Low Representational Gap (LRG)
- Doing responsibilities often come from early modeling
  - E.g., each message in a UML interaction diagram is suggestive of something that must be done

## GRASP patterns

- There are nine GRASP patterns, likely some already recognizable and some not:
  - Creator
  - Information Expert (or just Expert)
  - Low Coupling
  - Controller
  - High Cohesion
  - Polymorphism
  - Pure Fabrication
  - Indirection
  - Protected Variations
- (NOTE: The problem and solution statements that follow are almost verbatim from Larman, except for a very few minor attempts at adding clarity.)

## Creator

- Problem: Who should be responsible for creating a new instance of some class?
  - If done poorly, this choice can affect coupling, clarity, encapsulation, and reusability.
- Solution: Assign class B the responsibility to create an instance of class A if one of the below is true (the more the better). If more than one option applies, usually prefer a class B which aggregates or contains A.
  - B contains or is composed of 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.

## Creator Discussion

- This pattern generally avoids adding coupling to a design (which is bad—see GRASP pattern #3).
- When creation is a complex process or varies depending upon an input, often you'll want to create using a different class implementing the GoF pattern Concrete Factory or Abstract Factory.

## Information Expert

- Problem: What is a general principle of assigning responsibilities to objects?
- Solution: Assign a responsibility to the information expert—the class that has the information necessary to fulfill the responsibility.

# Information Expert Discussion

- This is general principle and probably the most used of any GRASP pattern.
- This generally is key to loose coupling and high cohesion, but not always so.
  - Imagine a case where it is better to hand off data in order to preserve a large functional divide and aid cohesiveness.
- We are implicitly talking about info held by a software object, but if there are not relevant software classes, try the domain model.

## Low Coupling

- Problem: How to support low dependency, low change impact, and increased reuse?
- Solution: Assign a responsibility so that coupling remains low.  Use this principle to evaluate alternatives.
  - Coupling refers to any type of tangible dependency between elements—classes, subsystems, systems, etc.—and is referenced by its degree:
    - Weak (low) is good.
    - Strong (high) is bad.

# Low Coupling Discussion

- Higher coupling can lead to:
  - More difficulty in understanding
  - Changes propagating excessively
  - More obstacles to code reuse
- Lower coupling often goes hand-in-hand with higher cohesion (good—see GRASP pattern #5).
- Consider this principle with *every* design decision.
- Note that too little coupling would indicate something is wrong with the design, likely including low cohesion
  - In RDD (and really OO) , the aim is to have a broad network of focused objects using communication toward fulfilling the requirements in an organized fashion.
- The more unstable the class coupled to, the more concerning the connection
  - E.g., consider a language's standard library vs. a class a coworker just defined a couple days ago.

## Controller

- Problem: What first object beyond the UI layer receives and coordinates (“controls”) a system operation?
- Solution: Assign the responsibility to one of the following types of classes:
  - Façade Controller: represents the overall “system”, a “root object”, or a device that the software is running
  - Use case or session controller: represents a use case scenario within which a system event occurs (e.g., “add an address book entry”)
    - The class would typically be named <UseCaseName>Handler, <UseCaseName>Coordinator, or <UseCaseName>Session

## Controller Discussion

- A controller attempts to coordinate the work without doing too much of it itself (again, guided by the degrees of coupling and cohesion)
  - The keyword is *delegation*.
- An easy example of this is that UI objects shouldn't perform business logic; there are other classes for that.
- The controller in the Model-View-Controller (MVC) architecture is effectively the same thing.
  - This, or its variation Model-View-Presenter, is frequently used in web applications

## High Cohesion

- Problem: How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?
- Solution: Assign a responsibility so that cohesion remains high. Use this to evaluate alternatives.
  - Cohesion refers to the functional cohesion between elements (classes, subsystems, systems, etc.), which is a measure of how strongly focused the responsibilities of an element are.

## High Cohesion Discussion

- Very similar to Low Coupling
  - Often related (but not always)
  - Should be considered in *every* design decision.
- Lower cohesion almost always means:
  - An element more difficult to understand, maintain, or reuse
  - An element more likely to be affected by change
- Low cohesion suggests that more delegation should be used.

## Polymorphism

- Problem: How to handle alternatives based on type? How to create pluggable software components?
- Solution: When related alternatives or behaviors vary by type (class), assign responsibilities for the behavior—using polymorphic operations—to the types for which the behavior varies.
  - Polymorphic operations are those that operate on differing classes
  - Don’t test for the type of the object and use conditional logic to perform varying statements based on type.

## Polymorphism

- With respect to implementation, this usually means the use of a super (parent) class or interface
  – Coding to an interface is generally preferred and avoids committing to a particular class hierarchy.
- Code like the following should raise a red flag!

```
Switch creatureType
   Case batType: print “Screech!”
   Case cowType: print “Moooooo...”
   Case humanType: print “Let’s watch TV!”
   [...]
```

- Also see GRASP pattern #8, Protected Variations.

## Pure Fabrication

- Problem: What object should have the responsibility, 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?
- Solution: 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.

## Pure Fabrication Discussion

- In other words, getting class concepts from a good domain model or real-life objects won't always work out well!
- An example of a possible pure fabrication class: PersistentStorage
  - May very well not be in the domain model
  - May very well not map to a real-life object
  - But it might be the answer to achieve our goals of low coupling / high cohesion while still having a clear responsibility
- Observe that all of the GoF design patterns are pure fabrications (often of multiple classes)

## Indirection

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

## Indirection Discussion

- Often an indirection intermediary is also a pure fabrication.
  - The PersistentStorage example could very well be an indirection between a Sale class and the database.
- The GoF patterns Adapter, Bridge, Façade, Observer, and Mediator all accomplish this.
- The main benefit is lower coupling.

# Protected Variations

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

## Protected Variations Discussion

- In the solution “interface” is meant in the general sense; but you’ll often want to use an interface programming construct (in Java, for example) to implement the solution!
- Benefits:
  - Easy to extend functionality at PV points
  - Lower coupling
  - Implementations can be updated without affecting clients
  - Reduces impact of change
- Very similar to the open-closed principle or the concept of information hiding (not the same as data hiding)
- In Larman’s first edition, was the Law of Demeter, but Protected Variations is a more generalized expression
- “Novice developers tend toward brittle designs, intermediate developers tend toward overly fancy and flexible, generalized ones (in ways that never get used). Expert designers choose with insight.”

# For More Information…

- For more on GRASP, it's hard to beat the depth of Larman's text:
  - Larman, Craig. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Third edition, Prentice Hall, 2005.
    - It is well written and there is also substantial material on UML, agile, GoF design patterns, a project-level perspective, and more.
- If GRASP doesn't strike a chord, there are alternative approaches to the general question of "How do I create an OO design using objects?" For example:
  - Wirfs-Brock, Rebecca and McKean, Alan. *Object Design: Roles, Responsibilities, and Collaborations*. Addison-Wesley Professional, 2002.
  - Evans, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003.

## Conclusion

- GRASP provides a map of considerations to provide strong guidance for an OO designer
- But at the same time, GRASP still leaves a lot of room to the designer and creating a good design is still an art!
- Taking a look at GRASP—and really *Applying UML and Patterns*—is a good bet for OO designers who know the basics of OOP but are still inexperienced