



Today: From GRASP to Design Patterns

- Continue with *design*
- Reference: Chapter 16 (again)
- **Design Patterns** help make good designs
- Why make good designs?
and...
- What are **Design Patterns**?



Good Designs are Good

- Why make good designs?
- What are Design Patterns?
- Which of the following are advantages of good design?
 - Systems are easier to understand
 - Systems are easier to debug
 - Systems are easier to maintain
 - System components are easier to reuse



Design Patterns and GRASP

- Why make good designs?
- What are Design Patterns?
- **GRASP** stands for
General
Responsibility
Assignment
Software
Patterns
- Thus, design patterns help by guiding **assignment of responsibilities to objects**



Design Patterns: what are they?

- Recall some design patterns:
 - High cohesion, low coupling, information expert, creator, & controller
- Is a design pattern more like
 - an outline for designs, or
 - a constraint on designs?
- Is a design pattern more like
 - an abstract design, or
 - a strategy for designing?
- Is a design pattern more like
 - a rule for designs, or
 - a general framework that is fleshed out to give a particular design?



Design Patterns

- The purpose is to decide
 - What member variables and methods go with what classes
- How to decide?
 - Design patterns suggest **responsibilities** for classes
 - Responsibilities are implemented with
 - Member variables and methods (**know** and **do**)
 - Example: know about UP, run lecture() method each Tu/Th at 2
- Result: the responsibilities of each class are determined
- The **design** is the **description** of
 - What member variables and methods will meet the responsibilities
- (From a good description you can implement good code)



The Design Process (in Brief)

- Figure out the design process by ordering the following 3 things:
 - Determine member variables and methods for the responsibilities
 - Determine the responsibilities of the classes
 - Determine the classes



The Design Process (in Brief)

- Answer:
 - Determine the classes
 - Determine the responsibilities of the classes
 - Determine member variables and methods for the responsibilities



Role of *Design Patterns* in Design

- Determine the classes
- **Determine the responsibilities of the classes**
 - "GRASP" – see the connection?
- Determine member variables and methods for the responsibilities



How to Determine Class Responsibilities?

- Use Design Patterns
 - (Also use Class Diagrams and Interaction Diagrams)
- Some major design patterns appear on page *-i* (inside front cover)
- The most important are described in chapter 16
- We listed them last time
- Let's look at them more closely now...



"Information Expert" Design Pattern

- General idea of this principle:
 - If you know what the responsibility is...
 - you still need to figure out where it goes
 - Consider putting it in the *class that has the information needed* to meet the responsibility
- Other names for this pattern abound
 - ...they help get an idea of what it is...



Information Expert Pattern: Other Names for the Idea

- Information Expert
- Expert
- "Place responsibilities with data"
- "That which knows, does"
- "Do it myself"
- "Put services with the attributes they work on"



The Information Expert Pattern

- Recall: put the responsibility (e.g. method)
 - ...where the information it needs is
- Example:
 - Where should the `grandTotal()` method be defined (see Figure 16.3)?
 - Think-pair-share
 - (Figure it out, then explain to neighbor, then explain neighbor's answer to class)



The Information Expert Pattern II

- Recall: put the responsibility (e.g. method)
 - ...where the information it needs is
- Example:
 - Where should the subtotal() method be defined (see Figure 16.3)?
 - Note:
 - $\text{subtotal} = (\# \text{ of the same item}) * (\text{price for each one})$
 - Think-pair-share
 - (Figure it out, then explain to neighbor, then explain neighbor's answer to class)



The Information Expert Pattern III

- Recall: put the responsibility (e.g. method)
 - ...where the information it needs is
- Example:
 - Where should the giveProductPrice() method be defined (see Figure 16.3)?
 - Think, commune with your inner software engineer, share w. class



The Information Expert Pattern IV

- Suppose grandTotal() is in class Sale
 - The partial design is in Figure 16.4 (see part of 16.6)
- Suppose subTotal() is in class salesLineItem
 - Augment the design – then let's uncover more of 16.6
- Suppose giveProductPrice() is in class productSpec
 - Augment the design more – then uncover more



Information Expert Pattern – a Footnote

- Is the information expert pattern always right?



Pattern 2: the Creator Pattern

- Most classes will have objects **created** of them
- The creating must be done somewhere
- This creating is a responsibility
 - someone has to do it
- The **Creator Pattern** suggests what class(es) have this responsibility
 - (Is it class or classes?)



Creator Pattern II

- When should class B create objects of class A?
 - Everyone write something down, pass up

Creator Pattern IIa

- When should class B create objects of class A?
 - B "aggregates" objects of class A
 - B "contains" objects of class A
 - B "records" objects of class A
 - B heavily uses objects of class A
 - B has the information to pass to A's constructor
- What class should create salesLineItem instances? (See same Fig.)
 - Why? Write it down

The *Controller* Design Pattern

- What classes should handle **system events**?
 - (What is a *system event*?)
- What is a controller?
 - - A controller is a **non-UI** object that gets or handles some system event
 - UI objects get the events – but simply pass them on to controllers!
 - (UI objects really don't do much)

The *Controller* Design Pattern II

- The question:
 - How to assign responsibility for handling system events?
- The answers (2 of them):
 - 1) The class for the **overall (sub)system**
 - E.g. the class containing main()
 - 2) The class for the **overall use case scenario** that the system event happens in

The Controller Pattern III

- Example:
 - enterItem() event
 - Ok to handle it in POSSystem class?
 - Ok to handle it in ProcessSale class?
 - Ok to handle it in CashiersWindow class?
 - Which of those represents the entire system?
 - Which of those handles the relevant use case scenario?
 - Which do you think is the best choice?

Modularity, Coupling, and Cohesion

- Consider the concept of "modularity"
 - This is the idea that software should be made of good modules
 - What makes modules good?
 - Each module is **cohesive**
 - Modules interact in a **loosely coupled** way

Coupling and Cohesion

- A good design has (high, low) coupling?
- A good design has (high, low) cohesion?
- See notes in another file...