More GRASP’ing and Use Case Realization
CSSE 574: Session 4, Part 1

Steve Chenoweth
Phone: Office (812) 877-8974
Cell (937) 657-3885 Email: chenowet@rose-hulman.edu
GRASP

- General Responsibility Assignment Software Patterns (or Principles)
  1. Low Coupling
  2. High Cohesion
  3. Information Expert
  4. Creator
  5. Controller
Coupling

- A measure of how strongly one element:
  - is connected to,
  - has knowledge of, or
  - relies on other elements

- Want low (or weak) coupling
Cohesion

- A measure of how strongly related and focused the responsibilities of a class (or method or package...) are
- Want high cohesion
Low Coupling and High Cohesion

- Inherent trade-offs of Cohesion and Coupling
  - To minimize coupling, a few objects have most of the responsibility
  - To maximize cohesion, a lot of objects have limited responsibility
  - Trade-off from alternative designs for best results

- Support both by
  - Evaluating alternatives to keep objects focused, understandable, and maintainable
  - Assigning so object’s responsibilities are closely related
  - Avoid spreading the responsible objects too thin
  - “Teamwork”
Some real coupling no-no’s

❖ “Coordinating coupling” – A passes a parameter to B that is used in B to determine the actions of B (like “flag passing”)

❖ “Common environment coupling” – A and B contain references to some shared data area that incorporates knowledge about its structure and organization. Any change to the format of the block requires that all the modules using it must also be modified.

There are cohesion no-no’s as well!

Don’t do the following:

- “Logical cohesion” – Operations put together because they are logically similar, but which involve very different actions internally.
  - Like system startup or shutdown operations
- “Coincidental cohesion” – Putting things together just to avoid more coding tasks, but the things have no conceptual link.
  - Often relates to filling up one large class with stuff, to avoid having to create new classes and call their methods.

From *Software Design*, by David Budgen, 2nd ed, 2003, p. 79.
Problem: What is a general principle of assigning responsibilities?

Solution: Assign a responsibility to the class that has the necessary information.
Problem: Who should be responsible for creating a new instance of some class?

Solution: Make $B$ responsible for creating $A$ if...
- $B$ contains or is a composition of $A$
- $B$ records $A$
- $B$ closely uses $A$
- $B$ has the data to initialize $A$
Project Creativity

- Consider a couple of classes from your project. “Who” (i.e., which other class) should be responsible for creating each of these classes?
- Justify your answer based on the Creator pattern.
Controller

- Problem: What first object beyond the UI layer receives and coordinates a system operation

- Solution: Assign the responsibility to either...
  - A façade controller, representing the overall system and handling all system operations, or
  - A use case controller, that handles all system events for a single use case
Controller Example

What domain layer class should own handling of the `enterItem` system operation?
Controller Guidelines

.clientWidth

- Controller should delegate to other domain layer objects

- Use façade controller when...
  - There are a limited number of system operations, or
  - When operations are coming in over a single “pipe”

- Use use case controller when a façade would be bloated (low cohesion!)
Controller Benefits

- Increased potential for reuse
- Can reason/control the state of a use case
  - e.g., don’t close sale until payment is accepted
Controller Issues

- Controller bloat—too many system operations
- Controller fails to delegate tasks
- Controller has many attributes

Switch from façade to use case controllers

Delegate!
Consider whether your project is more likely to benefit from using a façade controller or use case controllers (or perhaps no controllers at all).

Justify your answer.
Cartoon of the Day

Jan 4, 2010. Used by permission
Getting a GRASP on Design

- No ‘magic’ to assigning responsibilities

- If you don’t have a reason for placing a method in a class, it shouldn’t be there!
  - You should be able to say: ‘I placed method X in class Y based on GRASP Z’
Use Case Realization
Use Case Realization

The process of generating the design model from use cases and other requirements artifacts

- Use Cases drove the development of
  - Domain Model
  - System Sequence Diagrams
  - Operation Contracts
System Sequence Diagrams

- Helped us identify system operations
- Use these to begin interaction diagrams
  - System operations are the starting messages
  - Starting messages are directed at controller objects
Operation Contracts

- Defined **post-conditions** of system operations as changes to objects/associations in the domain model

- Use **post-conditions** to help determine...
  - What should happen in the interaction diagrams
  - What classes belong in the design class diagram

Also often discover classes that were missed in the domain model
Where to Begin

- In code, you begin at the beginning
- In design, you defer design of the Start Up UC
  - Start Up handles created and initializing objects
  - Discover necessary objects as we do the other Ucs
  - So defer Start Up design to avoid rework

Register (the controller) implements system operation makeNewSale()
### Example: Design `makeNewSale`

<table>
<thead>
<tr>
<th>Operation:</th>
<th><code>makeNewSale()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cross References:</td>
<td>Use Case: Process Sale</td>
</tr>
<tr>
<td>Preconditions:</td>
<td>none</td>
</tr>
<tr>
<td>Postconditions:</td>
<td>- A Sale instance <code>s</code> was created</td>
</tr>
<tr>
<td></td>
<td>- <code>s</code> was associated with the Register</td>
</tr>
<tr>
<td></td>
<td>- Attributes of <code>s</code> were initialized</td>
</tr>
</tbody>
</table>