CSSE 374: More GRASP'ing and Use Case Realization

Shawn Bohner

Office: Moench Room F212

Phone: (812) 877-8685

Email: bohner@rose-hulman.edu

ROSE-HULMAN INSTITUTE OF TECHNOLOGY

Learning Outcomes: Patterns, Tradeoffs

Identify criteria for the design of a software system and select patterns, create frameworks, and partition software to satisfy the inherent trade-offs.

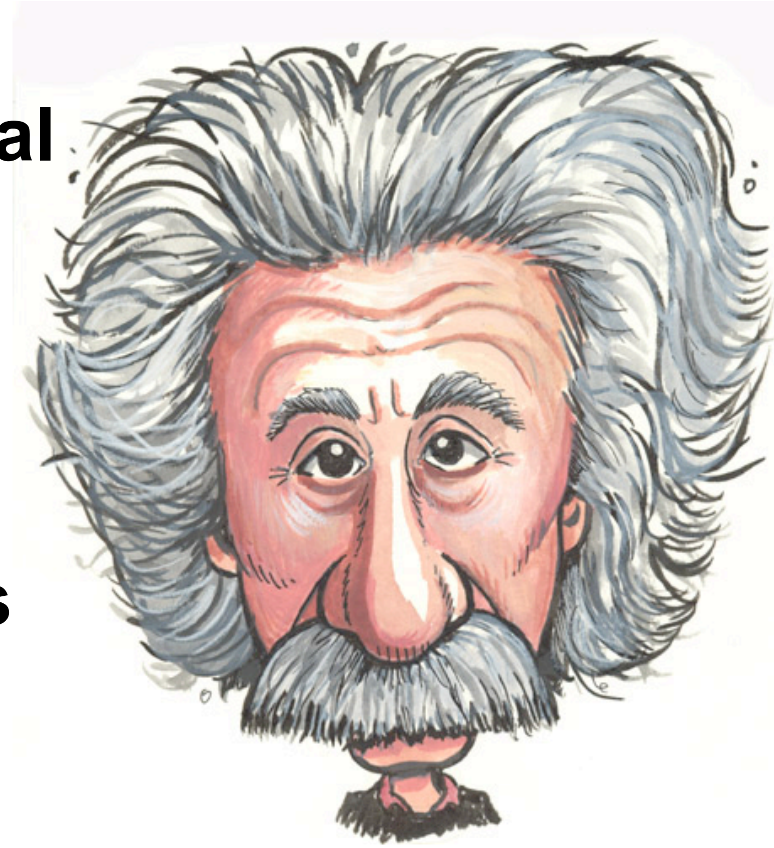
- Recap GRASP Patterns:
  - Creator
  - Information Expert
  - Controller
  - Low Coupling
  - High Cohesion
- Examine Use Case Realization

Recall GRASP: Creator

- **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*

Recall GRASP: Information Expert

- **Problem**: What is a general principle of assigning responsibilities?
- **Solution**: Assign a responsibility to the class that has the necessary information



Recall GRASP: Controller

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

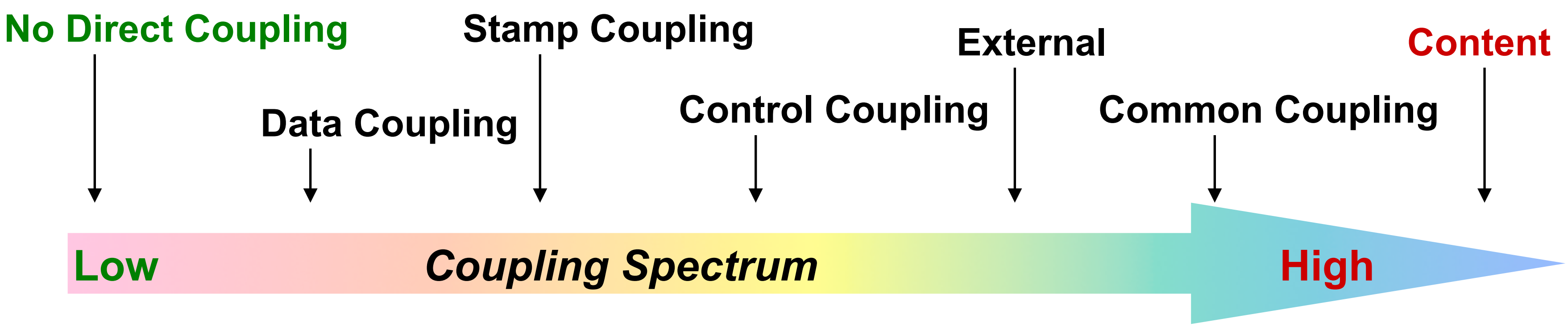
- **Solution: Assign the responsibility to either…**
  - **A façade controller, representing the overall system and handling all system operations, or**
  - **A use case controller, that handles all system events for a single use case**

Recall GRASP: Low Coupling

Problem: How do you support low dependency, low change impact, and increased reuse?

Solution: Assign a responsibility so that coupling remains low. Use this principle to evaluate alternatives.

Types of Coupling

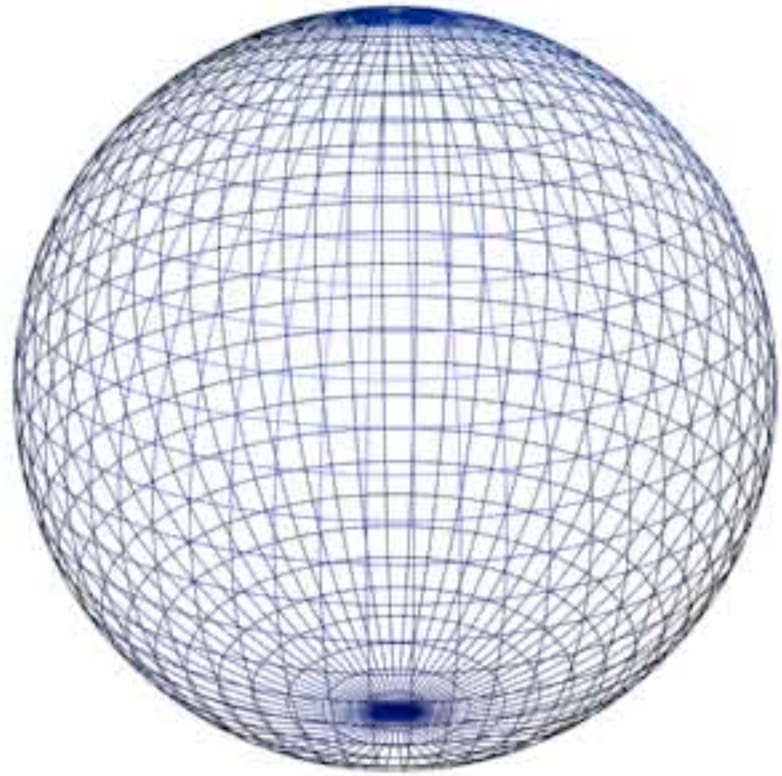


A measure of the interdependence among software components

Content: one component directly references the content of another
Common: both components have access to the same global data
Control: One component passes the element of control to another
Stamp: Two components modify or access data in the same object
Data: One component passes simple data to another as an argument

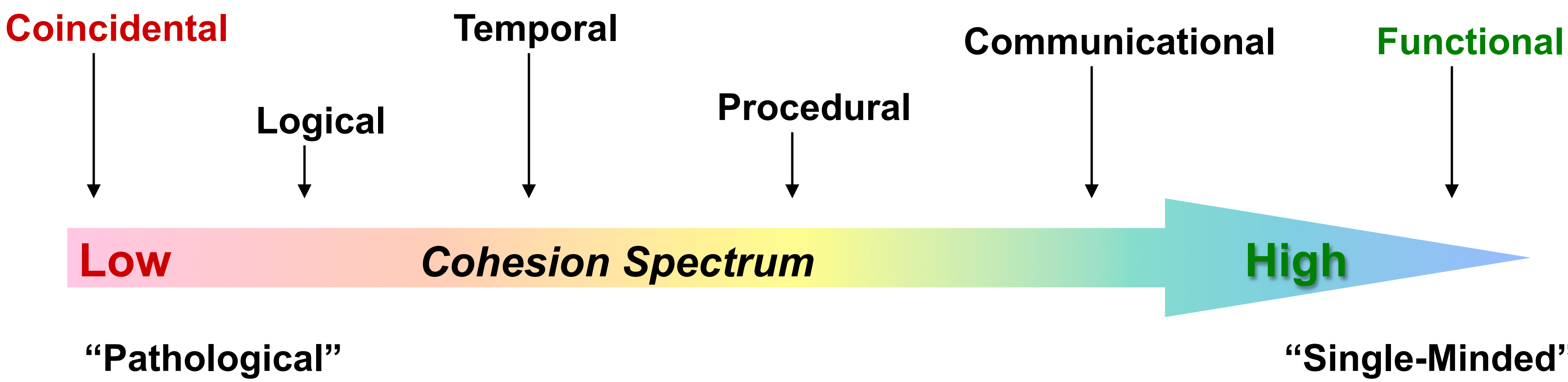
Recall GRASP: High Cohesion

Problem: How do you 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 principle to evaluate alternatives.

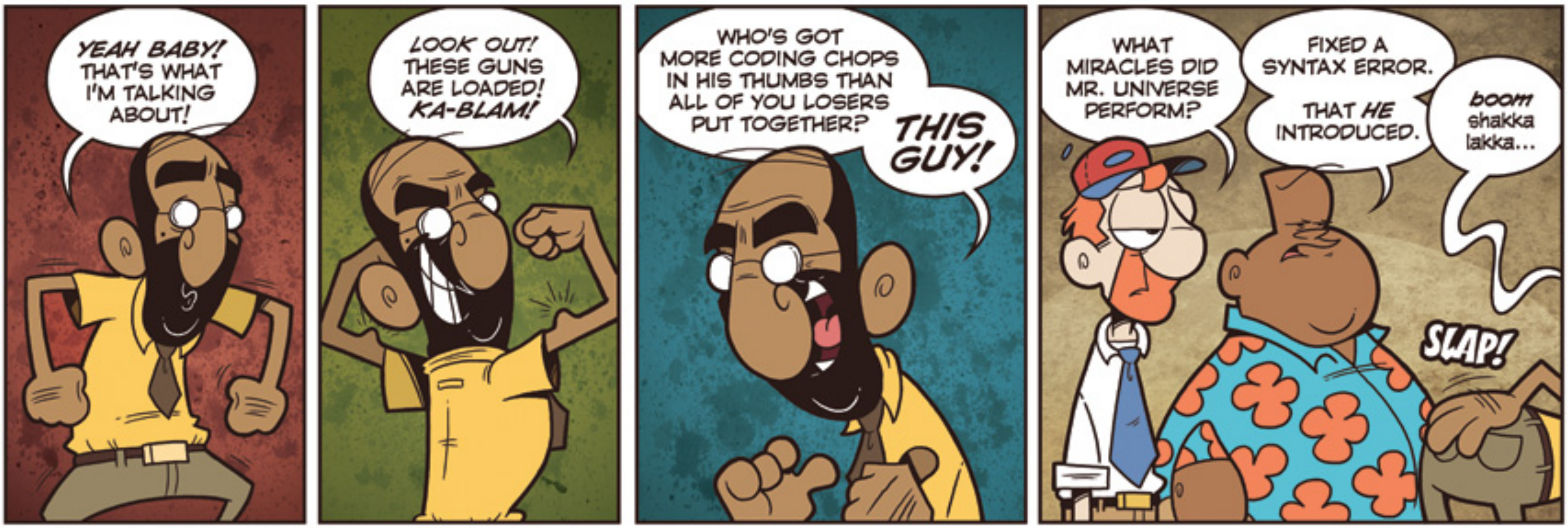
Types of Cohesion



A measure of the relative functional strength of a software component

<u>Coincidental</u>: multiple, completely unrelated actions or components
<u>Logical</u>: series of related actions or components (e.g. library of IO functions)
<u>Temporal</u>: series of actions related in time (e.g. initialization modules)
<u>Procedural</u>: series of actions sharing sequences of steps.
<u>Communicational</u>: procedural cohesion but on the same data.
<u>Functional</u>: one action or function

Legend in his own lunchtime…



NotInventedHere.com

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’

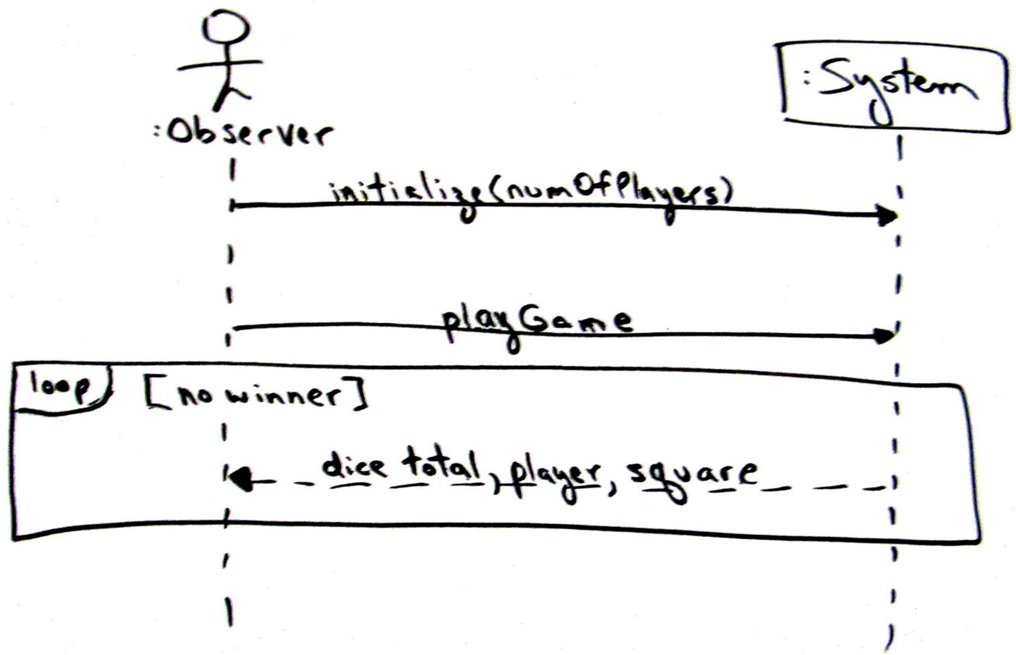
Use Case Realization

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 (SSD)

- Help us identify *system operations*
- Use these to begin interaction diagrams
  - System operations are the *starting* (AKA found) messages
  - Starting messages are directed at controller objects



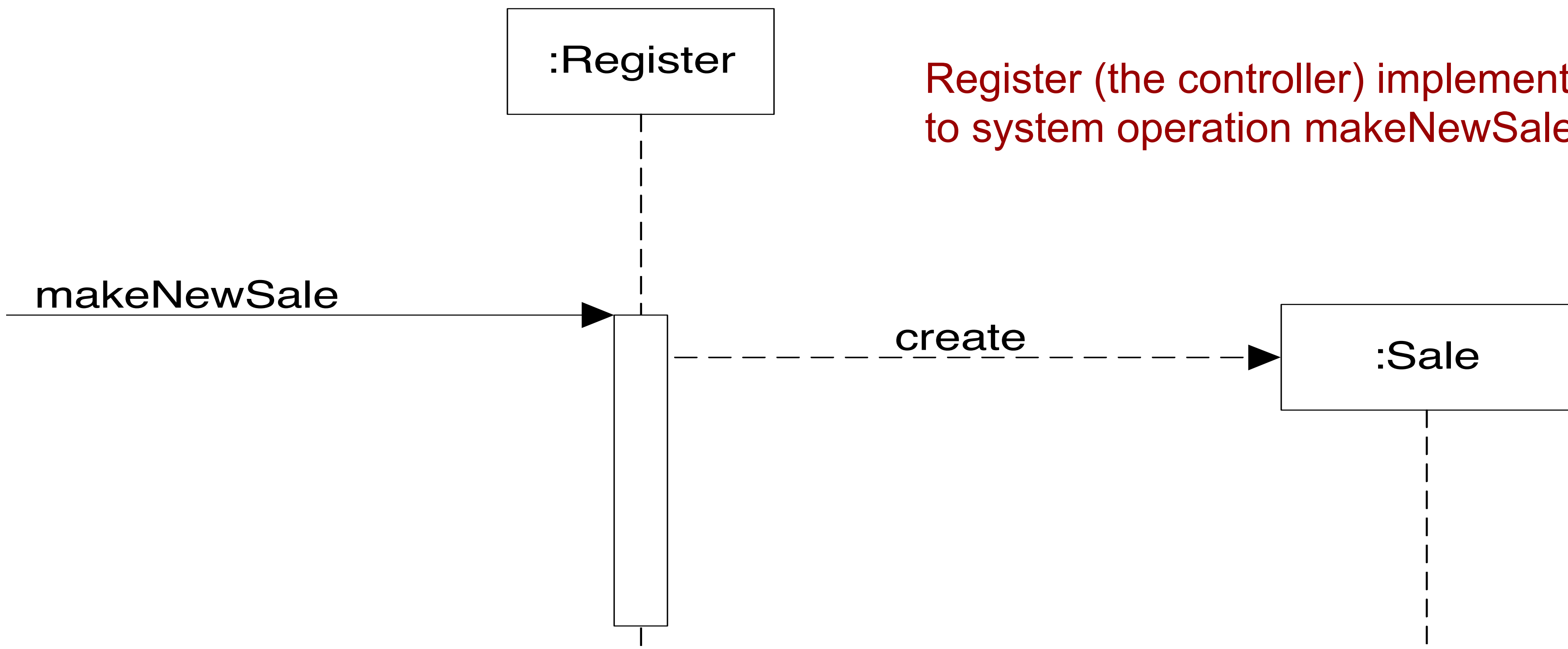
Operation Contracts (OC)

- Define 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



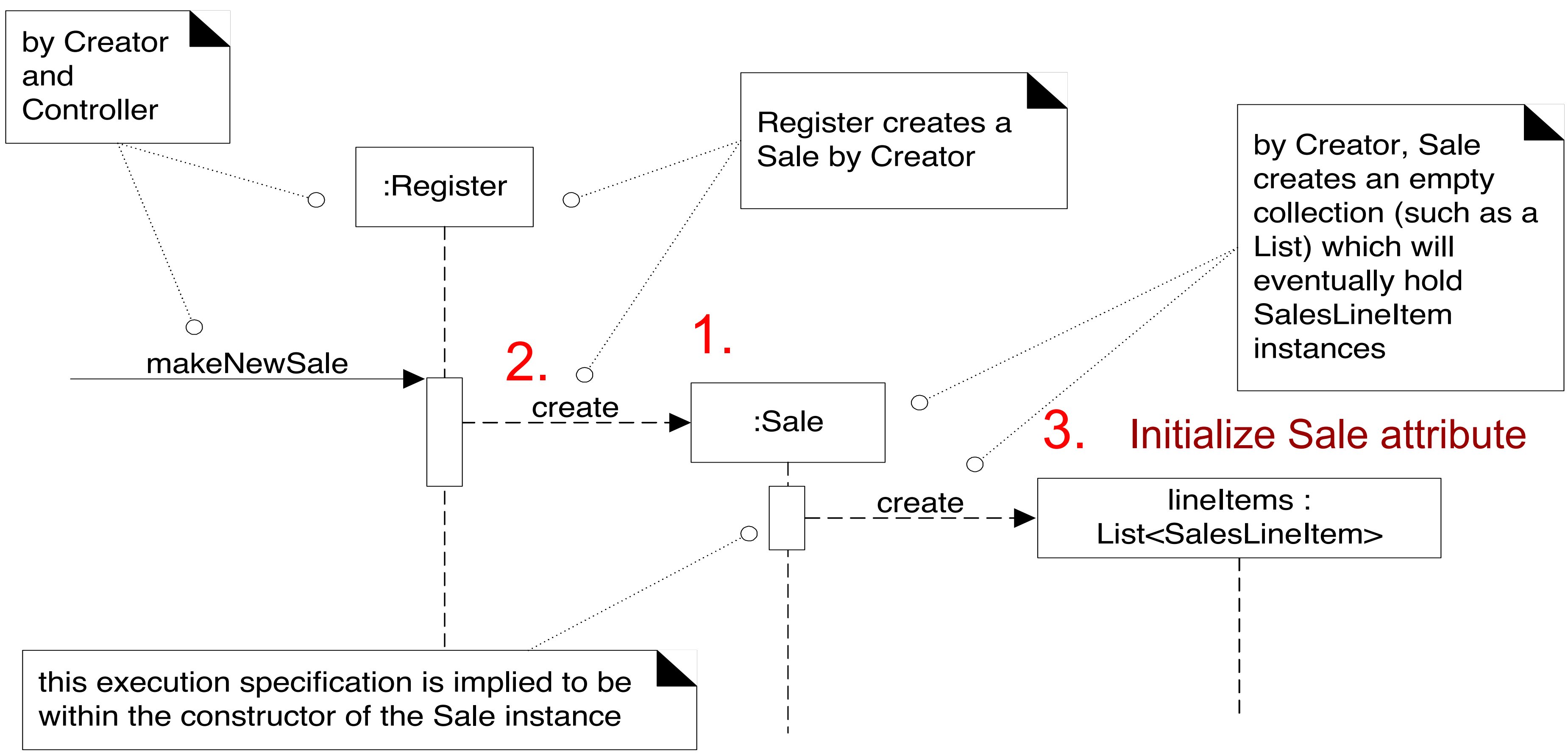
Register (the controller) implements to system operation makeNewSale()

- 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

Example: Design *makeNewSale*

Operation:	makeNewSale()
Cross References:	Use Case: Process Sale
Preconditions:	none
Postconditions:	○ A *Sale* instance *s* was created ○ *s* was associated with the *Register* ○ Attributes of *s* were initialized

Fulfilling Duties per Ops. Contract



Example: Design *enterItem*

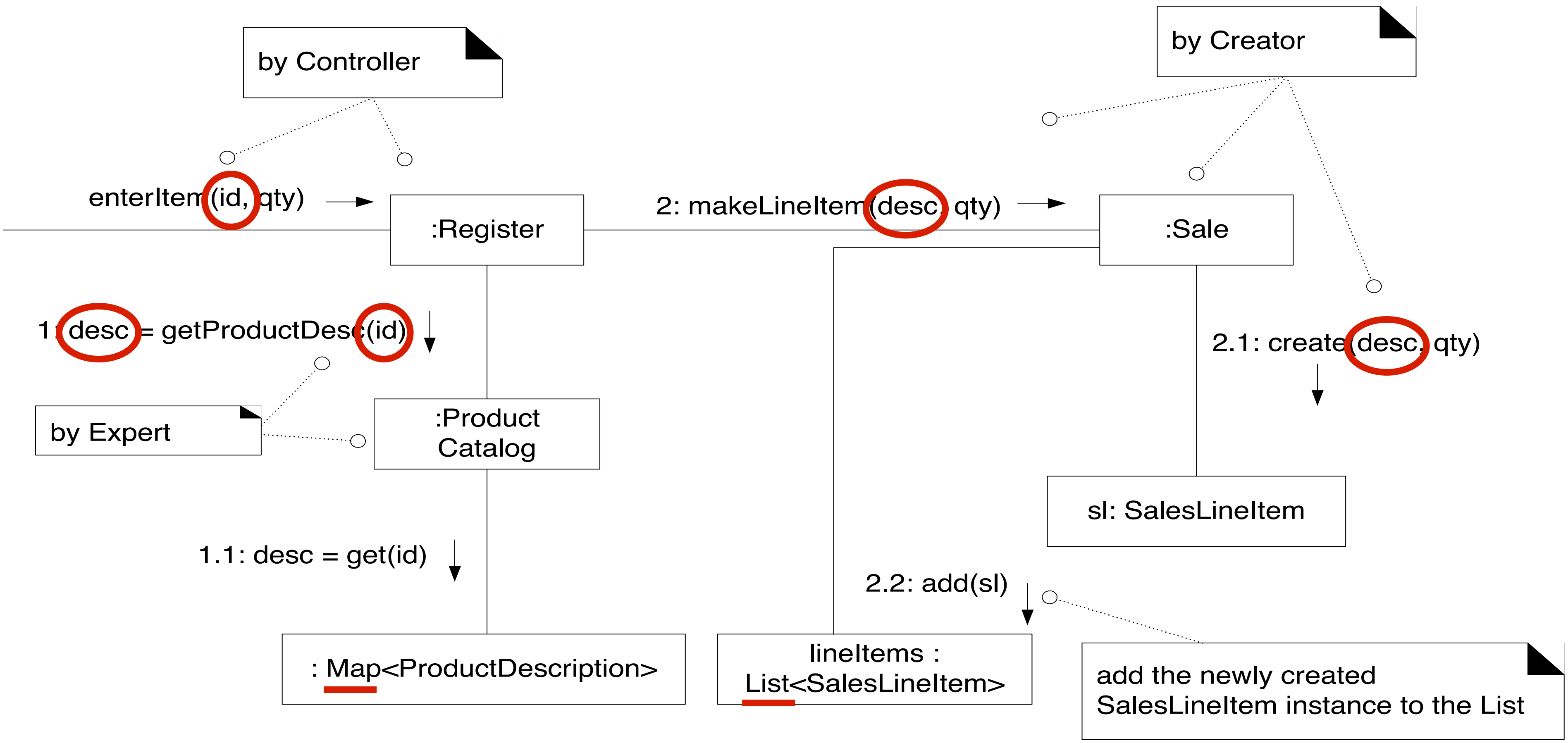
Operation:	enterItem(itemID: ItemID, quantity: integer)
Cross References:	Use Case: Process Sale
Preconditions:	Sale underway.
Postconditions:	○ A *SaleLineItem* instance *sli* was created ○ *sli* was associated with current Sale ○ sli.quantity became quantity ○ sli was associated with a ProductDescription, based on itemID match.

Assign *enterItem()* to a Controller

What must controller with enterItem() accomplish?

- Examine the operation contract…
  - Create a SalesLineItem (sli)
  - Associate it with current Sale
  - Set quantity attribute of sli
  - Associate sli with a ProductDescription …
- Whew! That's a lot of responsibility!
  - Requires careful analysis of operation contract to avoid missing any of these duties

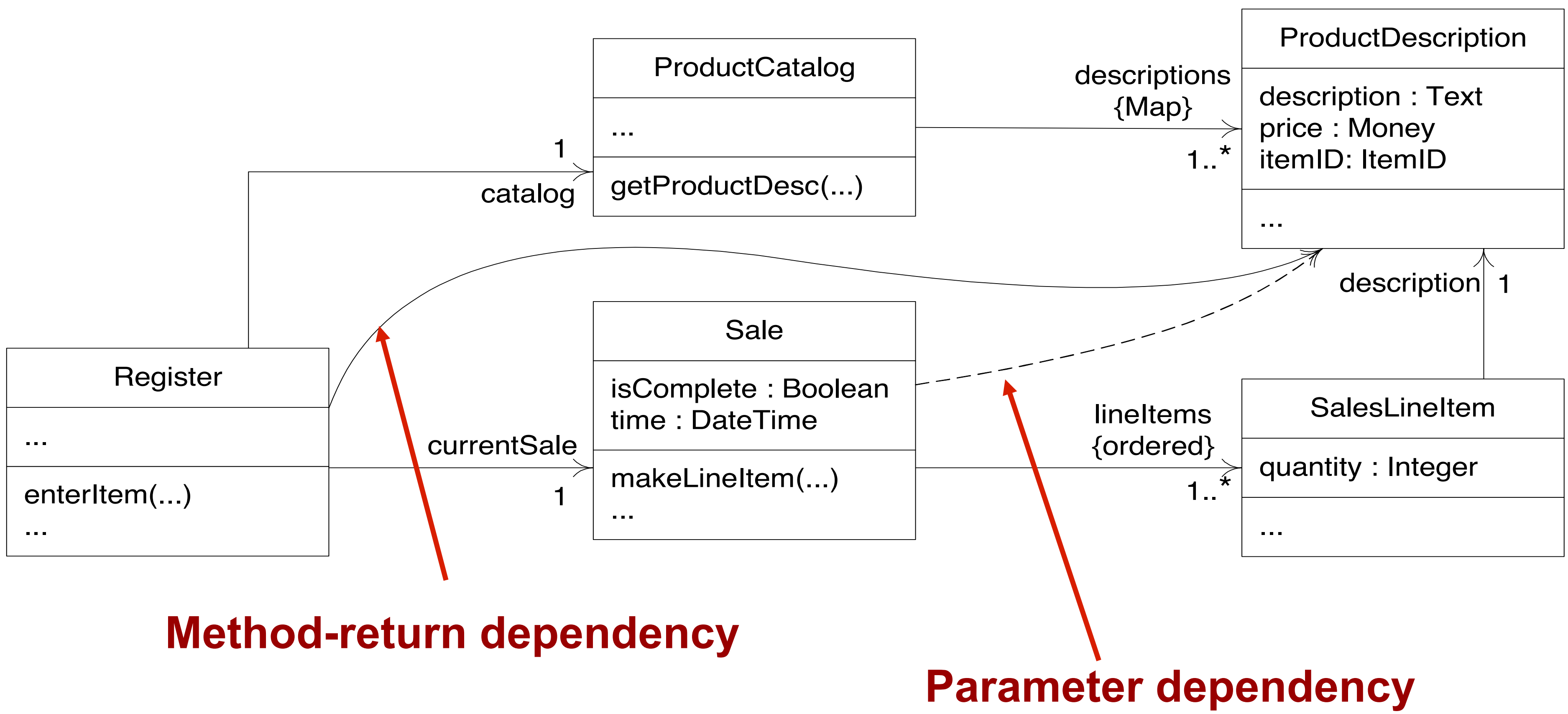
enterItem() Communication Diagram



Exercise on Design Examples

- Break up into your project teams
- Given the following:
  - The makeNewSale(…) and enterItem(...) OCs and SSDs
- Draw a partial Design Class Diagram to represent them.

Static View: Visibility



Homework and Milestone Reminders

- Read Chapter 19 on Visibility
- Milestone 3 – Junior Project SSDs, OCs, and Logical Architecture
  - Due by 11:59pm on Friday, January 7th, 2011
- Homework 4 – BBVS Design using GRASP and Midcourse Team Evaluation Exercise
  - Due by 11:59pm Tuesday, January 11th, 2011